

John Ewart, Matthias Marschall, Earl Waud

Chef: Powerful Infrastructure Automation

Learning Path

Deploy software, manage hosts, and scale your infrastructure with Chef



Packt

Chef: Powerful Infrastructure Automation

Deploy software, manage hosts, and scale your infrastructure with Chef

A course in three modules



BIRMINGHAM - MUMBAI

Chef: Powerful Infrastructure Automation

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, 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 course.

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

Published on: May 2017

Production reference: 1080517

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

ISBN 978-1-78839-297-6

www.packtpub.com

Credits

Authors

John Ewart
Matthias Marschall
Earl Waud

Content Development Editor

Devika Battike

Reviewers

Joshua Black
Lauren Malhoit
Eric Maxey
Robert Curth
Kristian Hoffmann
Max Manders
Greg Swallow
Earl Waud
Federico Gimenez

Graphics

Kirk D'penha

Production Coordinator

Nilesh Mohite

Preface

Chef is a configuration management tool that turns IT infrastructure into code. Chef provides tools to manage systems at scale. This learning path takes you on a comprehensive tour of Chef's functionality, ranging from its core features to advanced development.

What this learning path covers

Module 1, Chef Essentials, will take you on a comprehensive tour of Chef's functionality, ranging from core concepts to advanced development. You will be brought up to speed with what's new in Chef and how to set up your own Chef infrastructure for individuals, or small or large teams. Once you have the core components, get to grips with bootstrapping hosts to then develop and apply cookbooks. If you want to fully leverage Chef, this module will show you advanced recipes to handle new types of data providers and resources. By the end of this module, you will be confident in how to manage infrastructure, scale using the cloud, and extend the built-in functionality of Chef itself.

Module 2, Chef Infrastructure Automation Cookbook Second Edition, will take you on a journey through the many facets of Chef. This module begins with explaining the 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 for automating your own infrastructure. Instead of wasting time trying to get existing community cookbooks running in your environment, you'll get ready made code examples to get you started. After describing how to use the basic Chef tools, the module 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 module concludes by providing you additional, indispensable tools and giving you an in-depth look into the Chef ecosystem.

Module 3, Mastering Chef Provisioning, will show you the best practices to describe your entire infrastructure as code. With the help of this module you can expand your knowledge of Chef and implement robust and scalable automation solutions. You can automate and document every aspect of your network, from the hardware to software, middleware, and all your containers. You will become familiar with the Chef's Chef Provisioning tool. You will be able to make a perfect model system where everything is represented as code beneath your fingertips.

What you need for this learning path

The primary requirements are as follows:

- Administrative access to a host that runs a modern version of Linux;
Ubuntu 13.10
- If you do not have access to a dedicated Linux host, a virtual host (or hosts), running inside of virtualization software, such as VirtualBox will work.
- An installation of the Ruby programming language Version 1.9 or higher.
- A computer running OS X or Ubuntu Linux 14.04.
- It helps if you have Ruby 2.1.0 with bundler (<http://bundler.io/>) installed on your system as well.
- ChefDK version 0.12.0

Who this learning path is for

If you are a system administrator, Linux administrator, a cloud developer or someone who just wants to learn and apply Chef automation to your existing or new infrastructure, then this learning path will show you all you need to know. In order to get the most out of this learning path, some experience of programming or scripting languages would be useful.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – 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 course'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 course, 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 course from your account at <http://www.packtpub.com>. If you purchased this course 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 course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course'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 course is also hosted on GitHub at :

<https://github.com/PacktPublishing/Chef-Powerful-Infrastructure-Automation>

We also have other code bundles from our rich catalog of books, videos, and courses 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 courses – 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 course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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 course 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 course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Table of Contents

Module 1

Chapter 1: Installing Chef	3
Terminology	3
Working with Chef	4
Installing chef-solo	5
The Ruby gem	5
Managing gems	5
Verifying that chef-solo works	6
Installing a Chef server	9
Requirements and recent changes	9
Installation requirements	9
What you will be installing	10
Getting the installer	11
Installation outline	11
Installing on Ubuntu	11
Downloading the package	12
Installing the package	12
Installing on Red Hat Enterprise Linux	12
Downloading the package	12
Configuring a Chef server	13
Understanding how chef-server-ctl works	14
What's happening on my server?	14
Verifying that the services are running	16
Validating that your service is working	17
Ensuring that your knife configuration works	19
Summary	21

Table of Contents

Chapter 2: Modeling Your Infrastructure	23
Getting to know Chef	23
Modeling your infrastructure	25
Roles	27
Defining roles	28
Implementing a role	30
Determining which recipes you need	31
Installing a cookbook	32
Applying recipes to roles	33
Mapping your roles to nodes	36
Converging a node	38
Environments	39
Organizing your configuration data	40
Example attribute data	42
Data bags	43
Knowing when to use data bags	43
Large-scale infrastructure	45
Summary	45
Chapter 3: Integrating with the Cloud	47
Leveraging the cloud	47
Amazon EC2	48
Installing the EC2 knife plugin	48
Setting up EC2 authentication	48
Provisioning an instance	49
Bootstrapping the instance	50
Terminating the instance	51
Removing the Chef node	52
Rackspace Cloud	52
Provisioning an instance	53
Terminating an instance	55
Removing the Chef node	57
Summary	57
Chapter 4: Working with Cookbooks	59
Attributes	62
Multiple attribute files	64
Supporting multiple platforms	65
Loading external attributes	65
Using attributes	66
Metadata	67
Recipes	69
Resources	70
Using resources	71
Overriding a default behavior	73

Table of Contents

Templates	74
Why use templates?	74
A quick ERB primer	76
The template resource	77
The template variables	78
Passing variables to a template	78
Accessing computed configurations	79
Searching for templates	80
Definitions	81
Recipes	83
Developing recipes	83
Writing recipes	84
Starting out small	84
Installing a simple service	85
Getting more advanced	86
Summary	88
Chapter 5: Testing Your Recipes	89
Testing recipes	90
RSpec	90
RSpec and ChefSpec	91
Testing basics	92
Comparing RSpec with other testing libraries	92
Using ChefSpec	93
Getting started with ChefSpec	93
Installing ChefSpec	94
Locking your dependencies in Ruby	94
Creating a simple recipe and a matching ChefSpec test	95
Writing a ChefSpec test	96
Building your recipe	97
Executing tests	98
Understanding failures	98
Expanding your tests	100
Multiple examples in a spec test	100
Testing for multiple platforms	102
Summary	104
Chapter 6: From Development to Deployment	105
Describing the setup	106
Deploying software with Chef	106
Configuring your local environment	107
Modeling a simple Python application	107
Managing the cookbooks	108
Downloading cookbooks	108

Table of Contents

Looking at the database recipe	109
Looking at your application deployment cookbook	110
Preparing the directories	112
Constructing your Python virtual environment	113
Checking the source code	114
Installing any extra dependencies	115
Managing dependencies in Chef	115
Managing dependencies elsewhere	116
Using Python's requirements file	116
Configuring your application	117
Keeping your application running	118
Defining roles	119
Creating the base server role	119
Creating the database server role	120
Creating the web server role	121
Adding users	122
Provisioning EC2 instances	122
Configuring the database host	123
Configuring the web server	123
Deploying your software	124
Manually deploying updates	125
Automating deployment	125
Summary	126
Chapter 7: Beyond Basic Recipes and Cookbooks	127
Managing users	127
Evolution of a shell user recipe	128
Storing data in data bags	129
Creating a data bag for users	129
Searching for data	131
Searching inside recipes	132
Enhancing your user cookbook	133
Distributing SSH keys	136
Templating the authorized keys	136
Adding deployment keys	138
Writing custom extensions	139
Developing a custom definition	139
Organizing your code	139
Writing a definition for using PIP	139
Defining a full application template	142
Building a resource	145
Defining the resource	145
Implementing the provider	146
Modifying resources	148

Table of Contents

Loading an existing resource	149
Declaring that a resource was updated	150
Working with data bags	150
Securing your data bags	150
Secret keys	150
Encrypting your data	151
Decrypting your data	152
Storing keys on nodes	152
Searching your data	153
Searching your data bags with knife	153
Searching your data bags from a recipe	153
Querying your data	153
Managing multiple machines with search queries	155
Summary	155
Chapter 8: Extras You Need to Know	157
Vagrant and Chef-solo	157
Installing Vagrant	158
Provisioning a new host with Vagrant	158
Booting your Vagrant image	159
Combining Vagrant with Chef-solo	161
Understanding the limitations of Chef-solo	161
Configuring Chef-solo	162
Telling Chef-solo what to run	163
Using roles and data bags with Chef-solo	163
Getting to know the Chef shell	165
Using the Chef shell	165
The standalone mode	165
The solo mode	165
The client mode	166
Interacting with the Chef server using the shell	166
Interacting with data	167
Searching your data	168
Editing your data	168
Transforming data	168
Executing recipes with Chef shell	169
Creating a recipe in the shell	170
Defining node attributes	171
Using configuration blocks	173
Interactively executing recipes	173
Debugging with the Chef shell	175
Using the breakpoint resource	175
Integration testing	177
Using Test Kitchen	177
Installing Test Kitchen	178
Testing with Test Kitchen	178

Table of Contents

Extending Chef	184
Writing an Ohai plugin	184
A note about writing Ohai plugins	186
Chef with Capistrano	187
Automation and integration	188
Automated updates and deployments	188
Summary	189

Module 2

Chapter 1: Chef Infrastructure	193
Introduction	194
Using version control	194
Getting ready	195
How to do it...	195
How it works...	196
There's more...	197
See also	197
Installing the Chef development kit on your workstation	197
How to do it...	198
How it works...	198
See also	198
Using the hosted Chef platform	199
Getting ready	199
How to do it...	199
How it works...	201
There's more...	201
See also	202
Managing virtual machines with Vagrant	202
Getting ready	202
How to do it...	202
How it works...	204
There's more...	206
See also	206
Creating and using cookbooks	206
Getting ready	206
How to do it...	207
How it works...	208
See also	208

Table of Contents

Inspecting files on your Chef server with knife	208
Getting ready	208
How to do it...	209
How it works...	210
There's more...	210
See also	210
Defining cookbook dependencies	211
Getting ready	211
How to do it...	211
How it works...	211
There's more...	212
See also	212
Managing cookbook dependencies with Berkshelf	213
Getting ready	213
How to do it...	213
How it works...	214
There's more...	216
See also	217
Downloading and integrating cookbooks as vendor branches into your Git repository	217
Getting ready	217
How to do it...	218
How it works...	220
There's more...	221
See also	222
Using custom knife plugins	222
Getting ready	222
How to do it...	222
How it works...	223
There's more...	223
See also	224
Deleting a node from the Chef server	224
Getting ready	224
How to do it...	224
How it works...	224
There's more...	225
See also	225
Developing recipes with local mode	225
Getting ready	226
How to do it...	226
How it works...	227

Table of Contents

There's more...	227
Running knife in local mode	227
Moving to hosted Chef or your own Chef server	227
See also	227
Using roles	228
Getting ready	228
How to do it...	228
How it works...	229
See also	229
Using environments	229
Getting ready	230
How to do it...	230
How it works...	232
There's more...	232
See also	233
Freezing cookbooks	233
Getting ready	233
How to do it...	233
How it works...	234
There's more...	234
See also	234
Running Chef client as a daemon	235
Getting ready	235
How to do it...	235
How it works...	235
There's more...	235
Using chef-shell	236
How to do it...	236
How it works...	237
There's more...	238
See also	238
Chapter 2: Evaluating and Troubleshooting Cookbooks and Chef Runs	239
 Introduction	240
 Testing your Chef cookbooks	240
Getting ready	240
How to do it...	240
How it works...	241
There's more...	242
See also	242
 Flagging problems in your Chef cookbooks	242
Getting ready	242

Table of Contents

How to do it...	242
How it works...	243
There's more...	244
See also	244
Test-driven development for cookbooks using ChefSpec	245
Getting ready	245
How to do it...	245
How it works...	247
There's more...	248
See also	249
Integration testing your Chef cookbooks with Test Kitchen	250
Getting ready	250
How to do it...	250
How it works...	252
There's more...	254
See also	254
Showing affected nodes before uploading cookbooks	255
Getting ready	255
How to do it...	255
How it works...	256
See also	256
Overriding a node's run list to execute a single recipe	256
Getting ready	257
How to do it...	257
How it works...	258
See also	258
Using why-run mode to find out what a recipe might do	258
Getting ready	258
How to do it...	258
How it works...	260
See also	260
Debugging Chef client runs	260
Getting ready	260
How to do it...	261
How it works...	262
There's more...	262
See also	262
Inspecting the results of your last Chef run	262
Getting ready	262
How to do it...	263
How it works...	264

Table of Contents

See also	264
Raising and logging exceptions in recipes	264
Getting ready	264
How to do it...	264
How it works...	266
There's more...	266
See also	266
Diff-ing cookbooks with knife	266
Getting ready	266
How to do it...	266
How it works...	267
There's more...	267
See also	267
Using community exception and report handlers	268
Getting ready	268
How to do it...	268
How it works...	270
There's more...	270
See also	270
Creating custom handlers	270
Getting ready...	270
How to do it...	271
How it works...	272
There's more...	273
See also	273
Chapter 3: Chef Language and Style	275
Introduction	276
Using community Chef style	276
Getting ready	276
How to do it...	276
How it works...	277
There's more...	278
See also	278
Using attributes to dynamically configure recipes	278
Getting ready	278
How to do it...	279
How it works...	279
There's more...	280
Calculating values in the attribute files	280
See also	281

Using templates	281
Getting ready	281
How to do it...	282
How it works...	283
There's more...	283
See also	284
Mixing plain Ruby with Chef DSL	284
Getting ready	284
How to do it...	285
How it works...	285
There's more...	286
See also	286
Installing Ruby gems and using them in recipes	287
Getting ready	287
How to do it...	287
How it works...	288
See also	288
Using libraries	288
Getting ready	288
How to do it...	288
How it works...	289
There's more...	290
See also	290
Using definitions	291
Getting ready	291
How to do it...	291
How it works...	292
There's more...	293
See also	293
Creating your own Lightweight Resource Providers (LWRP)	293
Getting ready	293
How to do it...	294
How it works...	295
There's more...	297
See also	298
Extending community cookbooks by using application wrapper cookbooks	298
Getting ready	298
How to do it...	298
How it works...	300
There's more...	300
See also	300

Table of Contents

Creating custom Ohai plugins	300
Getting ready	301
How to do it...	301
How it works...	303
There's more...	304
See also	304
Creating custom knife plugins	304
Getting ready	305
How to do it...	305
How it works...	306
There's more...	307
See also	308
Chapter 4: Writing Better Cookbooks	309
Introduction	310
Setting the environment variables	310
Getting ready	310
How to do it...	310
How it works...	311
There's more...	311
See also	312
Passing arguments to shell commands	312
Getting ready	312
How to do it...	312
How it works...	313
There's more...	314
See also	314
Overriding attributes	314
Getting ready	314
How to do it...	314
How it works...	315
There's more...	315
See also	317
Using search to find nodes	317
Getting ready	317
How to do it...	317
How it works...	318
There's more...	318
Using knife to search for nodes	319
Searching for arbitrary node attributes	319
Using Boolean operators in search	319
See also	319

Table of Contents

Using data bags	320
Getting ready	320
How to do it...	321
How it works...	322
See also	323
Using search to find data bag items	323
Getting ready	323
How to do it...	323
How it works...	324
There's more...	324
See also	324
Using encrypted data bag items	324
Getting ready	324
How to do it...	325
How it works...	326
There's more...	326
Using a private key file	326
See also	327
Accessing data bag values from external scripts	327
Getting ready	328
How to do it...	328
How it works...	329
There's more...	329
See also	329
Getting information about the environment	330
Getting ready	330
How to do it...	330
How it works...	331
There's more...	331
See also	331
Writing cross-platform cookbooks	331
Getting ready	332
How to do it...	332
How it works...	333
There's more...	333
Avoiding case statements to set values based on the platform	333
Declaring support for specific operating systems in your cookbook's metadata	334
See also	334
Finding the complete list of operating systems you can use in cookbooks	334
How to do it...	335
How it works...	335

Table of Contents

There's more...	335
See also	336
Making recipes idempotent by using conditional execution	337
Getting ready	337
How to do it...	337
How it works...	338
There's more...	338
See also	338
Chapter 5: Working with Files and Packages	339
Introduction	339
Creating configuration files using templates	340
Getting ready	340
How to do it...	340
How it works...	341
There's more...	342
See also	342
Using pure Ruby in templates for conditionals and iterations	342
Getting ready	342
How to do it...	343
How it works...	344
There's more...	345
See also	345
Installing packages from a third-party repository	345
Getting ready	346
How to do it...	346
How it works...	348
See also	349
Installing software from source	349
Getting ready	349
How to do it...	350
How it works...	351
There's more...	352
See also	353
Running a command when a file is updated	353
Getting ready	353
How to do it...	353
How it works...	354
There's more...	355
See also	355

Table of Contents

Distributing directory trees	355
Getting ready	355
How to do it...	355
How it works...	357
There's more...	357
See also	357
Cleaning up old files	358
Getting ready	358
How to do it...	358
How it works...	360
There's more...	360
See also	360
Distributing different files based on the target platform	360
Getting ready	361
How to do it...	361
How it works...	362
See also	362
Chapter 6: Users and Applications	363
Introduction	363
Creating users from data bags	364
Getting ready	364
How to do it...	364
How it works...	366
There's more...	367
See also	368
Securing the Secure Shell Daemon (SSHD)	368
Getting ready	368
How to do it...	369
How it works...	370
There's more...	370
See also	371
Enabling passwordless sudo	371
Getting ready	371
How to do it...	372
How it works...	373
There's more...	374
See also	374
Managing NTP	374
Getting ready	374
How to do it...	375

Table of Contents

How it works...	376
There's more...	376
See also	376
Managing nginx	377
Getting ready	377
How to do it...	377
How it works...	379
There's more...	380
See also	381
Creating nginx virtual hosts	381
Getting ready	381
How to do it...	382
How it works...	384
There's more...	384
See also	385
Creating MySQL databases and users	385
Getting ready	385
How to do it...	386
How it works...	388
There's more...	389
See also	389
Managing WordPress sites	389
Getting ready	389
How to do it...	390
How it works...	391
There's more...	391
See also	392
Managing Ruby on Rails applications	392
Getting ready	392
How to do it...	392
How it works...	394
There's more...	396
See also	397
Managing Varnish	397
Getting ready	397
How to do it...	398
How it works...	399
There's more...	399
See also	400

Table of Contents

Managing your local workstation	400
Getting ready	400
How to do it...	401
How it works...	402
There's more...	403
See also	403
Chapter 7: Servers and Cloud Infrastructure	405
Introduction	406
Creating your infrastructure using Chef Provisioning	406
Getting ready	406
How to do it...	406
How it works...	408
There's more...	409
See also	409
Creating cookbooks from a running system with Blueprint	409
Getting ready	410
How to do it...	410
How it works...	411
There's more...	412
See also	412
Running the same command on many machines at once	412
How to do it...	413
How it works...	413
There's more...	413
See also	414
Setting up SNMP for external monitoring services	414
Getting ready	414
How to do it...	414
How it works...	415
There's more...	416
See also	416
Deploying a Nagios monitoring server	416
Getting ready	416
How to do it...	417
How it works...	419
There's more...	420
See also	420
Building high-availability services using heartbeat	420
Getting ready	420
How to do it...	422

Table of Contents

How it works...	424
There's more...	425
See also	425
Using HAProxy to load-balance multiple web servers	425
Getting ready	425
How to do it...	426
How it works...	427
See also	428
Using custom bootstrap scripts	428
Getting ready	428
How to do it...	428
How it works...	429
There's more...	430
See also	430
Managing firewalls with iptables	430
Getting ready	430
How to do it...	431
How it works...	432
See also	433
Managing fail2ban to ban malicious IP addresses	433
Getting ready	433
How to do it...	434
How it works...	435
There's more...	435
See also	436
Managing Amazon EC2 instances	436
Getting ready	436
How to do it...	436
How it works...	438
There's more...	439
See also	440
Loading your Chef infrastructure from a file with spiceweasel and knife	440
Getting ready	440
How to do it...	441
How it works...	442
There's more...	442
See also	442

Module 3

Chapter 1: Setting Up a Development Environment on Your Workstation	445
Re-introducing Chef	446
What version of the Chef tools do I use?	446
References	447
Filling in the gaps in your Chef knowledge base	447
Major pieces of a Chef puzzle	447
The Chef client	448
The Ohai Chef tool	449
Recipes and cookbooks	449
Chef run lists	449
Chef roles	450
Chef resources	450
More things to consider	450
Data bags	451
Environments	451
The Chef supermarket	452
Chef Development Kit	452
Setting up an on-premise (private) Chef Server	452
References	456
Exploring the Chef Development Kit	457
The Chef Knife	457
Community tools	457
References	458
Installing Chef Development Kit	458
Downloading ChefDK	458
Installing ChefDK	459
Verifying ChefDK	460
References	464
Setting up a standard Chef repo	464
Using the Chef tool for Chef repo	465
The repo starter kit	466
Files for your .chef folder – a safer way	467
Growing your own tree	468
Self-signed certificates	469
References	471
What else do you need?	471
Using your favorite editor	471

Table of Contents

Version control systems	473
Virtualization hypervisor	474
Vagrant	474
References	475
What is chef-apply?	475
Why is that useful?	475
References	476
Summary	477
Chapter 2: Knife Tooling and Plugins	479
Knife help	479
References	481
Knife subcommands	481
Knife Bootstrap	481
Knife Node	483
Knife Diff	483
Knife Exec	484
Knife SSH	485
Knife Search	485
References	487
Knife to editor integration	487
Using the "wait" option	487
Does my editor require the wait option?	488
References	488
Knife Windows plugin	489
Installing the Knife Windows plugin	489
Bootstrapping a Windows node	489
Searching for Windows node data	490
References	490
Knife EC2 plugin	491
Installing the Knife EC2 plugin	491
Creating an EC2 instance	492
Bootstrapping an existing EC2 instance	493
Deleting EC2 instances	494
References	494
Creating new Knife plugins	495
The basic plugin file format	495
Example – A simpler EC2 Create Server plugin	496
References	497
Summary	497

Table of Contents

Chapter 3: Leveraging Roles, Environments, and Policies	499
Using Chef roles	499
How to create a role	500
Using per-environment run-lists in roles	502
Using nested roles	504
There is a downside to roles	505
References	505
Using Chef environments	506
The _default environment	506
Creating environments	506
Specifying cookbook versions in environments	507
References	508
Using Chef organizations	508
Creating organizations	508
Access control	509
References	509
Learning the attribute precedence hierarchy	510
Understanding attribute precedence factors	510
Resulting precedence hierarchy	512
Exploring policies	513
What exactly is a policy?	513
Why would I use a policy?	514
How do I use a policy?	514
Another example	516
How to specify the source options for cookbooks	519
What are the "Gotchas"?	520
References	521
Summary	521
Chapter 4: Custom Resources	523
Out with the old (LWRPs), in with the new (Custom Resources)	523
What is the difference between the old way and the new way of defining resources?	524
What is a Custom Resource?	525
References	525
Creating an example use case	526
What is the goal?	526
Creating the cookbook	526
Creating the default recipe	527
The recipe worked, but what is missing?	531
References	531

Table of Contents

Creating and using a Custom Resource	531
What makes up a Custom Resource?	532
Creating the "website" Custom Resource	532
Using our Custom Resource	535
References	537
Making a Custom Resource more reusable	537
Making a Custom Resource-only cookbook	538
Reuse the code from the "mywebapp" example	538
Using the reusable Custom Resource-only cookbook	538
Handling idempotence	539
References	541
Subclassing a Custom Resource	541
Convert our resource into a library module	541
Subclassing the library module Custom Resource	543
References	545
Summary	546
Chapter 5: Provisioning in the Traditional Data Center	547
VMware provisioning	547
Installing the knife-vsphere plugin	549
Provisioning VMs using the knife-vsphere plugin	551
Cleaning up our vSphere and Chef environments	554
Using Chef with VMware and Vagrant	555
References	558
OpenStack provisioning	559
Setting up a DevStack OpenStack all-in-one environment	560
Deploying the DevStack Host VM	560
Downloading and installing DevStack	563
Downloading the demo project rc file and applying it	567
Adding the security rules to allow ping and ssh traffic	567
Installing the knife OpenStack plugin	571
Setting up the DevStack public key	571
Allocating some floating IP addresses to the project	572
Configuring the knife-openstack plugin knife settings	573
Adding a CentOS instance image to provision	576
Provisioning a new instance	577
References	581
Network automation	581
Automating juniper devices	582
Automating Cisco systems	585
Automating F5 LTM	588

Table of Contents

References	591
Summary	592
Chapter 6: Provisioning in the Cloud	593
Provisioning AWS EC2 instances	594
Understanding EC2 authentication	594
Creating key pairs	594
Creating security groups	596
Installing Chef Server into an EC2 instance	598
References	602
Provisioning Microsoft Azure instances	603
Installing the knife-azure plugin	604
Configuring the plugin	605
Verifying the configuration	606
Generating public/private key pairs	607
Provisioning virtual machines	608
Provisioning a new Chef server into Azure	610
References	612
Provisioning in the Google Cloud platform	613
Installing and configuring the knife-google plugin	613
Creating and registering your key pairs	615
Provisioning virtual machines with the GCE	616
Deploying Chef server into a GCE instance	619
References	622
Looking at an honorable mention – Linode	622
References	623
Provisioning containers in Docker	624
Setting up a Docker server to host our containers	624
Installing and using Docker on our Chef Workstation	627
Using the knife-docker plugin	628
References	629
Summary	630
Chapter 7: Test-Driven Development	631
Analyzing cookbooks with RuboCop	632
Using RuboCop	632
Examining the RuboCop output	633
Customizing RuboCop	634
References	635
Analyzing cookbooks with Foodcritic	636
Using Foodcritic	636
Examining the Foodcritic output	637

Table of Contents

Customizing Foodcritic	638
References	638
Unit testing with ChefSpec	639
Using ChefSpec	639
Expanding our tests	643
References	646
Integration testing with Kitchen and ServerSpec	646
Preparing our tests	647
Kitchen create	650
Kitchen converge	651
Kitchen verify	652
References	655
Summary	656
Chapter 8: Using Chef Provisioning	657
What is Chef provisioning?	658
It's highly idempotent	658
A hypothetical example	658
References	659
How to use Chef Provisioning	659
Chef Provisioning resources	660
A Vagrant example	660
A cloud example with AWS	664
A Docker example	667
References	671
Summary	672
Index	673

Module 1

Chef Essentials

Discover how to deploy software, manage hosts, and scale your infrastructure with Chef

1

Installing Chef

Before you can start using Chef, you will need to install it. Here you will find a guide to install Chef, and because Chef requires Ruby, some Ruby concepts as well. This chapter discusses the following:

- Key terminology and concepts related to Chef
- An overview of Chef's architecture
- Working with Ruby gems
- Installing chef-solo (a local-only engine to use Chef)
- A brief example on using chef-solo
- Installing the Chef server on your own host
- Verifying your Chef installation

Terminology

As with any other technology, Chef has its own terminology. As you will see, Chef's nomenclature is a mix of technological terms (nodes, workstations, servers, roles, and so on) and cooking terms (cookbooks, recipes, and so on). There are three primary actors that we are concerned with at this point: nodes, the Chef service, and workstations.

- **Node:** A node is a client that applies roles and recipes, as described by the administrator in the Chef service (that is, a server in your environment that is being configured via Chef). These are the consumers of the configuration, the elements of your infrastructure. They can be physical or virtual machines and can run on Linux, Windows, or technically any other system that is capable of running Ruby (some systems may not be supported out of the box by Chef).

- **Chef service:** The Chef service is a multicomponent system that combines several services to provide its functionality. The primary functional components are an API service, full-text searching via Solr, persistent storage using PostgreSQL, and RabbitMQ for interservice communication. Additionally, there is a web interface that provides a graphical tool to manage system data. Clients (nodes) use the API service to determine which roles and recipes to apply, and `knife` (a command-line tool) uses the API to allow an administrator to edit and manage their Chef configuration.
- **Workstation:** A workstation is a host that is used to issue commands. A workstation can be a separate host outside of your Chef service installation, a managed node, or the server that the Chef components are running on. There are a variety of command-line tools that are provided to interact with the service, which will be installed onto your workstation(s).
- **Recipe:** A recipe is a script that describes a set of steps to take to achieve a specific goal. As an example, a recipe might describe how to deploy your custom software, provision a database, or add a host to a load balancer.
- **Cookbook:** A cookbook is a collection of recipes that are used to collectively describe how to install, configure, and manage various aspects of your infrastructure. For example, a cookbook might describe how to provision MySQL, PostgreSQL or Apache, manage users, install printers, or perform any other system tasks.

Working with Chef

For single user setups, `chef-solo` is a version of the `chef-client` that allows you to use Chef without requiring access to a server. `Chef-solo` runs locally and requires that a cookbook and any of its dependencies be present on the node being managed. As a result, `chef-solo` provides a limited subset of the full `chef server` mode of operation. Most of the features that `chef-solo` is missing revolve around search and centralized data management, which are not critical for managing virtual machines or a small collection of nodes. The installation and maintenance is simple, but the feature set is smaller.

Installing the Chef server will give you access to the full set of Chef functionality. This mode requires access to a Linux-based host that is network-accessible by the nodes and workstations that will interact with the system. Thanks to the recent effort from the folks at Chef (formerly Opscode), the process of installing Chef has been greatly simplified. The benefits of this installation model are that you get centralized management, search, user authentication, and such, but at the cost of managing your own service.

If you need the features of Chef but do not want to maintain your own server, hosted Chef is a great option for you. Hosted Chef (<https://manage.opscode.com/signup>) gives you all the features of a self-hosted Chef but without having to worry about upgrades, extra hardware, or system availability. For a small infrastructure (up to five hosts), hosted Chef is free and a great way to get started. Beyond this, plans have a monthly fee, and the price will vary according to the number of hosts you want to manage.

Installing chef-solo

Chef-solo is designed for individuals who do not need a hosted installation for a large-scale infrastructure management. Typical use cases of chef-solo include developers managing virtual machines, test installations, or small-scale infrastructure management. The installation of chef-solo is as simple as installing a single Ruby gem.

The Ruby gem

For those who are not intimately familiar with Ruby, a Ruby gem is a mechanism to package, deliver, and manage Ruby code. These packages may be libraries that provide functionality for developers, or they may be composed only of scripts and tools. Chef-solo is, like many things in life, somewhere in the middle. The gem contains a set of libraries that make up the core functionality as well as a suite of scripts that are used by end users. Before you install Chef, you should consider installing **Ruby Version Manager (RVM)**, rbenv, chruby, or another Ruby manager of your choice to keep your gem collections isolated.

Managing gems

A great tool to manage your gems is RVM. The simplest way to install RVM is to use the installation script provided by the development team on the RVM website (<http://rvm.io>). The following command will download the script and pipe it through bash:

```
curl -sSL https://get.rvm.io | bash -s stable
```

Once it is installed, you will initially need to include RVM's functionality in your shell:

```
source ~/.rvm/scripts/rvm
```

Additionally, you might need to add the previous command line to your shell's startup scripts (such as `~/.bashrc` or `~/.zshrc`, depending on which shell you use). Once RVM is installed, you will want to install a recent version of Ruby, for example, Ruby 1.9.3:

```
rvm install 1.9.3
```

Once Ruby 1.9 is installed, you will want to create a **gemset**. A gemset is RVM's way of isolating gems inside a container, and it will provide you with a place to install gems in such a way that they will not conflict with other gems. This has the benefit of allowing you to install anything you want, without requiring administrative privileges and keeping gems from conflicting with each other. A gemset can be created using the following command:

```
rvm use 1.9.3@chef --create
```

The previous command will simultaneously create the gemset named `chef` (if it does not exist) for your installation of Ruby 1.9.3 and then set it as the active gemset. Once you start using this new gemset, you will want to install the Chef gem—this contains `chef-solo` and all the command-line tools you need to work with Chef—using the `gem` command-line tool:

```
gem install chef
```

Verifying that chef-solo works

Now that the Chef gem is installed, it is time to verify that everything is working fine. In order to use `chef-solo`, you need to give the following information to it:

- What recipes to apply by providing a run list in a file named `node.json`
- What the recipes are—these are stored in cookbooks that are found in the `cookbooks` directory
- How to find the cookbooks and the run list via a file named `solo.rb`

For simplicity, we will store all of these files inside of the `chef` directory in your home directory. You are free to put things where you see fit as you become more comfortable working with Chef.

In order to exercise our new tool, we will do something simple: we'll write a recipe that will create an `example.txt` file in your home directory. The recipe we create will be called `create_file`, and we'll put that recipe inside a cookbook, which will be named `demo`.

First, create the directory that will contain the `demo` cookbook's recipes (and any in between):

```
user@host:~ $ mkdir -p ~/chef/cookbooks/demo/recipes
```

Next, add the following code to a file, `create_file.rb`, located in the `demo` cookbook directory you created at `~/chef/cookbooks/demo/recipes`:

```
file "#{ENV['HOME']}/example.txt" do
  action :create
  content "Greetings #{ENV['USER']}!"
end
```

This tells Chef that we want to create a file, `$HOME/example.txt`. Its contents should be `Greetings $USER`, where `$USER` will be replaced with the value of `$USER`, typically the login name of whoever is executing the recipe.



For those unfamiliar, UNIX (and Windows as well) uses environment variables as a mechanism to exchange data between processes. Some environment variables are set when the user logs in to the system such as `HOME`, `USER`, and a variety of others. These variables are available in Ruby using the `ENV` hash, where the keys are the variable names. In a UNIX shell, these are accessed using the `$` prefix. So, the user's home is referred to as `$HOME` in the shell and `ENV['HOME']` inside Ruby.

Now we will need to create a JSON document that describes what `chef-solo` should execute. JSON is an acronym for JavaScript Object Notation, and Chef uses JSON extensively because it is easy to parse, human readable, and easy to generate from all sorts of tools and languages. Create a file, `node.json`, located in our work directory (`~/chef/` in this case) and add the following content in order to tell Chef that we want to execute the newly created `create_file` recipe in the `demo` cookbook:

```
{
  "run_list": [
    "recipe[demo::create_file]"
  ]
}
```

Here, we are defining the node as having a run list, which is just an array of things to do, and that the run list contains one recipe, `create_file`, which it can find in the `demo` cookbook (the general form of a recipe being `cookbook::recipe`).

Installing Chef

Finally, we'll tell Chef where to find the files we just created using a `solo.rb` file that we will store in our working directory (`~/chef` in our case):

```
CHEF_ROOT="#{ENV['HOME']}/chef"
file_cache_path "#{CHEF_ROOT}"
cookbook_path "#{CHEF_ROOT}/cookbooks"
json_attribs "#{CHEF_ROOT}/node.json"
```

Now that you have populated the required configuration files, you can run `chef-solo` and execute the run list specified. In our case, the run list is defined as only one recipe, `create_file`, but can be as simple or as complex as needed. The previous configuration tells Chef to load the node configuration from the file `node.json` to look for cookbooks in `~/chef/cookbooks/` and to store any state data in `~/chef/`. In order to execute these commands, you will want to run `chef-solo`:

```
chef-solo -c ~/chef/solo.rb
```

The `-c` option tells `chef-solo` which script contains the configuration. Once you do this, you will see the actions that your recipe is performing:

```
Starting Chef Client, version 11.8.2
Compiling Cookbooks...
Converging 1 resources
Recipe: demo::create_file
  * file[/home/user/example.txt] action create
    - create new file /home/user/example.txt
    - update content in file /home/user/example.txt from none to b4a3cc
      --- /home/user/example.txt      2014-01-20 23:59:54.692819000
      -0500
      +++ /tmp/.example.txt20140122-13411-1vxtg7v  2014-01-20
      23:59:54.692819000 -0500
    @@ -1 +1,2 @@
    +Greetings user!
```

```
Chef Client finished, 1 resources updated
```

Once it is completed, you will see that `~/example.txt` contains the greeting that you defined in the recipe. Now that you have successfully used `chef-solo`, let's move on to the Chef service.

Installing a Chef server

If your team needs to have centralized infrastructure management and does not want to use a hosted platform, then a self-installed Chef server is a perfect fit. This installation guide assumes that you will be running the Chef server on a supported Linux-based system.

The Chef service components can be installed on a single machine without any issue. Installing it on a single host will limit your ability to scale or be highly available, but will provide a very simple path to getting started with the Chef service.

Requirements and recent changes

Since the Chef service is designed to be a multiuser platform and provides functionalities that chef-solo does not offer, the installation is more complex and involves more software to achieve this functionality. Services such as Solr for full-text indexing and PostgreSQL for data storage can be a significant resource for consumers, so you will want to install Chef on a host with sufficient memory and disk space. A system with 2 GB of memory and 5-10 GB of disk space available will be plenty for a small to medium sized installation. You will need more resources as your requirements for data storage and indexing increase over time, so plan accordingly.

Additionally, for those who have installed the Chef server before, the installation path has been greatly simplified. In addition to replacing CouchDB with PostgreSQL as the primary data storage engine, there is now a single **omnibus** installation package for Chef that installs all of the requirements for Chef at a single location so that it operates in isolation and does not require dependencies to be installed separately.

Installation requirements

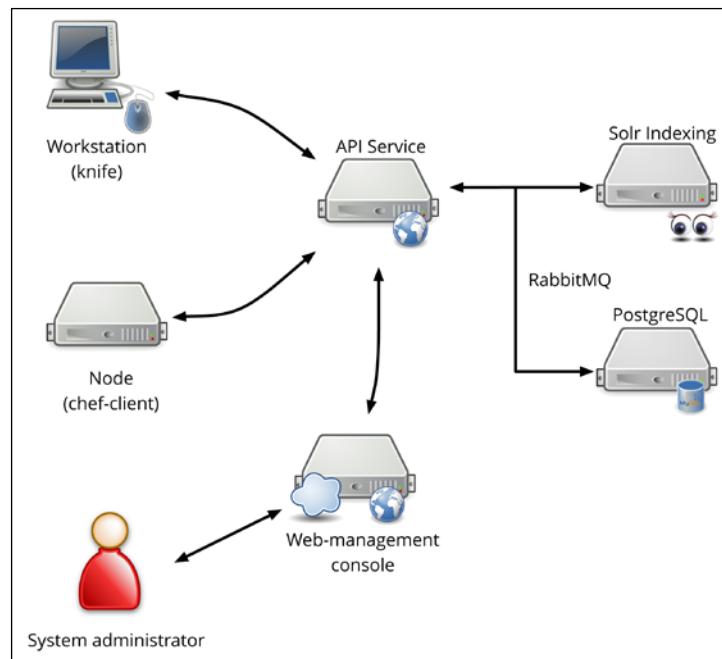
In order to install the Chef service, you will need to have the following:

- A system running a supported Linux variant (64 bit Ubuntu Linux 10.04 through 12.10 or 64 bit Red Hat Enterprise Linux 5 or 6) – this can be physical or virtual. If you do not have the local resources for this, AWS or RackSpace cloud servers are good options.
- A network connection to the host in order to download the installer.
- Administrative privileges (using `sudo` or direct root access) on the host where you are installing the services.
- Enough free space on the host to perform the download and installation (minimum 500 MB, including the download, but 1 GB to 2 GB is preferred).

What you will be installing

At the end of this section, you will have a fully functional Chef service installed and ready to work with. Before you get started, let's look at what you will be installing on your system so that you know what to expect. The components that make up a Chef service are as follows:

- The Chef API service
- Message queue (AMQP)
- Data storage
- Search service
- Web-based management console



The Chef API service is responsible for delivering run lists and receiving information from nodes as well as providing a way for a system administrator to configure recipes, run lists, data bags, and the like. In order to generate this data, the API service relies on its persistent data storage engine, in this case PostgreSQL, to store its data. The option to search for data is provided by the Solr search engine, and RabbitMQ is responsible for gluing them all together. Together, these components provide Chef with the ability to distribute, store, index, and manage your infrastructure's configuration data.

Getting the installer

The easiest way to install Chef is through a single downloadable package, which is provided for Ubuntu 10.04 through 12.10 and Red Hat Enterprise Linux Versions 5 and 6. This package, referred to as the **omnibus** installer, contains everything you need to get a server up and running. You can find it on <http://www.getchef.com/chef/install/>.

At the time of writing this, 11.0.10 is the latest version and is the one that will be used for this book. The newer version of the 11.x series of Chef should have a very similar, if not identical, configuration. Note that these installers are somewhat large, being that they contain all of the dependencies needed. For example, the Ubuntu 12.10 package for Chef 11.0.10 is approximately 200 MB in size.



Although these are the officially supported distributions and releases, it is entirely possible that these installers will work on different but compatible distributions. It may be possible, for example, to use CentOS instead of Red Hat Enterprise Linux or Debian instead of Ubuntu. However, these will most likely require some manual dependency resolutions and may not work without a lot of effort (and even then, possibly not at all).

Installation outline

Installation on all supported platforms is relatively similar. The only key differences are the names of the package files that you will download and the commands you will use to install Chef.

The high-level steps you will take are as follows:

1. Downloading the Chef installer for your platform.
2. Installing the package as an administrative user.
3. Configuring the Chef service.
4. Testing the server using command-line tools.

Because steps 3 and 4 will be the same for both Ubuntu and Red Hat installation procedures, the instructions will be in a section following the Red Hat installation guide.

Installing on Ubuntu

The following are instructions for an Ubuntu-based system; they were performed on an Ubuntu 12.04 host, but should be identical for all supported Ubuntu distributions. For Red Hat-based installation instructions, see the next section.

Downloading the package

You can download the package by returning to the download page referenced previously (<http://www.getchef.com/chef/install/>), or you can download Version 11.0.10 directly from https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64/chef-server_11.0.10-1.ubuntu.12.04_amd64.deb.

Installing the package

In order to perform the installation, open a terminal on your Ubuntu host (either locally or connect via SSH) as a user who has administrative privileges. This can be done directly either as the root or any user who has permission to execute arbitrary commands via `sudo`.

Once you log in to the host, navigate to where you want to store the package (remember it's quite large, approximately 200 MB) and download the file using `curl`:

```
user@ubuntu:~ $ curl -O https://opscode-omnibus-packages.s3.amazonaws.com/ubuntu/12.04/x86_64/chef-server_11.0.10-1.ubuntu.12.04_amd64.deb
```

Once the file is downloaded, the `dpkg` tool will be used to perform the package installation:

```
user@ubuntu:~ $ sudo dpkg -i chef-server_11.0.10-1.ubuntu.12.04_amd64.deb
```

Once this is finished, the Ubuntu-specific portion of the setup is complete, and you will need to configure Chef using the `chef-server-ctl` command, which we will discuss in the *Configuring Chef Server* section, following the *Installing on Red Hat Enterprise Linux* section.

Installing on Red Hat Enterprise Linux

Installation on a Red Hat Enterprise Linux distribution is as straightforward as installing any other package. You download the package to the local disk and install it using RPM tools.

Downloading the package

You can download the latest version of the package by returning to the download page referenced previously (<http://www.getchef.com/chef/install/>), or you can download Version 11.0.10 directly from https://opscode-omnibus-packages.s3.amazonaws.com/el/6/x86_64/chef-server-11.0.10-1.el6.x86_64.rpm.

In order to perform the installation, open a terminal on your Red Hat host (either locally or connect via SSH) as a user who has administrative privileges. This can be done directly either as the root or any user who has permission to execute arbitrary commands via `sudo`.

Once you log in to the host, navigate to where you want to store the package (remember it's quite large, approximately 200 MB) and download the file using `curl`:

```
user@rhel:~ $ curl -O https://opscode-omnibus-packages.s3.amazonaws.com/el/6/x86_64/chef-server-11.0.10-1.el6.x86_64.rpm
```

How long this takes will vary according to the available bandwidth but should take somewhere between 5 and 20 minutes on a reasonably fast connection.

Once the file is downloaded, the `rpm` tool will be used to perform the package installation:

```
user@rhel:~ $ sudo rpm -ivh chef-server-11.0.10-1.el6.x86_64.rpm
```

Once this is finished, the Red Hat-specific portion of the setup is complete, and you will need to configure Chef using the `chef-server-ctl` command, which we will discuss in the following section.

Configuring a Chef server

Historically, installing Chef requires manual editing of configuration files, choosing RabbitMQ credentials, installing CouchDB, and a handful of other tasks. Now, with the omnibus installer, all of this is taken care of for you. If you have been following along, your system has the Chef server and all of its dependencies installed on the system in the `/opt/chef-server` directory.

Included with the installation of the package is a shell script, `chef-server-ctl` (located at `/opt/chef-server/bin`), which is responsible for configuring your newly installed Chef server. In order to configure your services, you will need to run it as `root` because the scripts will need to modify your system in ways that your regular account may not be able to. Initializing the configuration tool is as simple as issuing the following command:

```
sudo chef-server-ctl reconfigure
```

Running this script may take a few minutes, and it will produce a lot of output while it is doing its work. While it is running, let's take a few minutes to discuss how it works and what it is doing.

Understanding how chef-server-ctl works

Earlier in this chapter, you were briefly introduced to the `chef-solo` tool. You saw how it can be used to manage your local server using on-disk recipes and configuration data. The Chef team has leveraged this ability to do just that with the Chef server using `chef-solo` to bootstrap the server configuration. If you were to look at the code for the `/opt/chef-server/bin/chef-server-ctl` script, you would see that the last line in the script executes the following command:

```
/opt/chef-server/embedded/bin/omnibus-ctl chef-server /opt/chef-server/
embedded/service/omnibus-ctl $@
```

If you follow the trail and dig into the `omnibus-ctl` script, you will find that it is just a wrapper around the `omnibus-ctl` Ruby gem. Digging into the `omnibus-ctl` gem, you will see that in the end, the `reconfigure` command you pass on the command line is a Ruby method that makes the following call:

```
run_command("chef-solo -c #{base_path}/embedded/cookbooks/solo.rb -j
#{base_path}/embedded/cookbooks/dna.json")
```

This tells us that the Chef omnibus package uses `chef-solo` to configure itself—a pretty clever trick indeed! You can see just how powerful a tool `chef-solo` can be, being able to configure and reconfigure the Chef service.

What's happening on my server?

What you will probably notice right away is that a lot of text is being scrolled past in your terminal window. If you were to look at the contents, you would see that it shows you the actions that are being taken by `chef-solo` to provision your new services. As there is a lot of information going past (thousands of lines), here is a high-level overview of what is happening on your host:

1. A new user, `chef_server`, and its corresponding group are being provisioned.
2. Chef services are being set up, and startup scripts for upstart are being placed in the appropriate system directories. The Run scripts for Chef services are located at `/opt/chef-server/sv`.
3. Chef state directories are being created in `/var` including `/var/opt/chef-server` and `/var/log/chef-server`.
4. RabbitMQ is being configured to store data in `/var/opt/chef-server` and log the output to `/var/log/chef-server` as well as its startup scripts in `/opt/chef-server/sv/rabbitmq/run`.

5. PostgreSQL is being configured with its data in `/var/opt/chef-server/postgresql/data` along with a user, `opscode-pgsql`, to run the service. Some system-level changes to share memory sizes are being set via `sysctl` to make PostgreSQL work as well as persisted in `sysctl.conf`.
6. Solr is being set up to work with the configuration and data rooted in `/var/opt/chef-server/chef-solr/`, with the run script being placed in `/opt/chef-server/sv/chef-solr/run`.
7. Chef-expander (the data-indexing service) is being configured for `/var/opt/chef-server/chef-expander` as its working directory with Solr and RabbitMQ endpoints on the localhost. The run script is located at `/opt/chef-server/sv/chef-expander/run`.
8. The Chef bookshelf metadata service is readied in `/var/opt/chef-server/bookshelf/` with its run script at `/opt/chef-server/sv/bookshelf/run`.
9. Erchef, the Erlang Chef service, is installed and pointed at the local Solr, RabbitMQ, bookshelf, and PostgreSQL services.
10. The system is then bootstrapped using the bootstrap recipe. This recipe verifies that the system is running (by checking that the `http://localhost:8000/_status` returns an HTTP 200 response) and installs the SSL certificate for the web-based UI in `/etc/chef-server/chef-webui.pem`.
11. The web-based UI configuration files are generated and placed in `/var/opt/chef-server/chef-server-webui/`.
12. A copy of nginx to host the web UI is placed in `/var/opt/chef-server/nginx`, and the initial self-signed SSL certificates as well as the static assets are installed in `/var/opt/chef-server/nginx/html`.
13. The Chef API testing framework, `chef-pedant`, is installed.
14. Finally, `/etc/chef-server/chef-server-running.json` is generated with the current configuration settings for your Chef services.

Clearly, there is a lot happening here; if you have any outstanding concerns about what is being done, be sure to read through the output. One of the great things about Chef is that the recipes are just a set of scripts that you can open and view the contents of, and the output shows you what is happening during the execution. Everything it does is transparent and manageable by you.

Verifying that the services are running

Once the configuration of your services is complete, you will want to validate that the required services are running. Again, the `chef-server-ctl` script will be used, but we will invoke the `status` subcommand instead of the `reconfigure` subcommand, as shown in the following code:

```
user@host:~ $ sudo chef-server-ctl status
run: bookshelf: (pid 3901) 3123s; run: log: (pid 3900) 3123s
run: chef-expander: (pid 3861) 3129s; run: log: (pid 3860) 3129s
run: chef-server-webui: (pid 4053) 3095s; run: log: (pid 4052) 3095s
run: chef-solr: (pid 3819) 3135s; run: log: (pid 3818) 3135s
run: erchef: (pid 4230) 3062s; run: log: (pid 3937) 3117s
run: nginx: (pid 4214) 3064s; run: log: (pid 4213) 3064s
run: postgresql: (pid 3729) 3146s; run: log: (pid 3728) 3146s
run: rabbitmq: (pid 3423) 3172s; run: log: (pid 3422) 3172s
```

The `status` subcommand will show you the process ID of each component, how long it has been running for, the PID of the logging process associated with that service, and how long the logging service has been running. For example, we can see that `chef-server-webui` has a PID of 4053 and has been running for close to an hour, and the logger has a PID of 4052, having been running for just as long as the service.

As you can see, the installation of Chef yields a number of components that will need to be up and running in order to successfully use Chef. You should have the following components running and listening on the following network ports:

Component	What to look for in the process list	Port(s)	Public?
Chef API server	Erchef and nginx	80, 443	Yes
Web management console	chef-server-webui and nginx	80, 443	Yes
Data indexer	chef-expander	N/A	N/A
Solr	java (running start.jar in the Chef directory)	8,983	No
PostgreSQL	postmaster	5,432	No
RabbitMQ	beam.smp running rabbit	5,672	No

Public components need to be made available to any clients, nodes, or end users that expect to use the Chef service over the network. Configuring your infrastructure to ensure that your services are available via the network is outside of the scope of this book as there are a near-infinite number of possible network configurations.

At a higher level, make sure that any firewall devices or packet-filtering systems are not preventing traffic from reaching these services if you see that they are running, but are having difficulties in connecting to them. If any of these services are not running, you will need to consult the logfiles generated by the service to determine what might be preventing them from starting up.

Validating that your service is working

In order to work with Chef, you will need a way to interact with it. Fortunately, Chef provides a suite of command-line utilities, which we will discuss at length as the book progresses. There is one primary tool, `knife`, that allows an administrator to interact with the service in the command line. The `knife` tool is run from a workstation and provides many commands to view, search, and modify data maintained by the Chef service. Once you have installed and verified that all the services are running, we can move on to setting up `knife`.



You will see that the standard place to store your Chef configuration data is in `$HOME/.chef` (on a UNIX-like system.) This is not mandatory, and these files can be stored anywhere you like.

The `knife` tool communicates with the Chef server via HTTP and uses certificates for authentication between the workstation and the server. In order to get started with `knife`, we will need to do two things: gain access to the certificates that were generated during the installation of Chef and then use those credentials to set up a new user in the system.

In the following examples, we will be using the host that the Chef services were installed on as our workstation (where we will use `knife`). If you want to use a different host, you will need to get the required certificate (`.pem`) files to your local machine using `scp` or some other mechanism. By using the following commands, we can get the required authentication materials into our work directory:

```
mkdir $HOME/.chef
sudo cp /etc/chef-server/admin.pem $HOME/.chef
sudo cp /etc/chef-server/chef-validator.pem $HOME/.chef
sudo cp /etc/chef-server/chef-webui.pem $HOME/.chef
sudo chown -R $UID $HOME/.chef
```



Chef uses a signed header authentication for requests to the API, which means there must be a shared key that is present on both the client and the server. Chef-server will generate the `chef-validator.pem` file when it is configured. New nodes or clients use the `chef-validator.pem` file to sign the requests used to register themselves with the system.

Once you have these files copied into your Chef work directory, it is time to configure `knife` itself. Fortunately, `knife` has an interactive configuration mode that will walk you through the process of generating a configuration file. First, ensure that you are using your Chef gemset (if you are using RVM as we discussed earlier) and then run `knife` on your workstation (again, in this example, we are using our Chef service host for both purposes):

```
user@chef:~$ rvm use 1.9.3@chef
user@chef:~$ knife configure -i
```

When you run `knife` with the `-i` flag, you will be prompted by the following questions, which you can answer with the defaults for almost everything (non-default answers are in bold):

```
WARNING: No knife configuration file found
Where should I put the config file? [/home/user/.chef/knife.rb]
Please enter the chef server URL: [https://localhost:443]
Please enter a name for the new user: [user]
Please enter the existing admin name: [admin]
Please enter the location of the existing admin's private key: [/etc/
chef-server/admin.pem] ~/.chef/admin.pem
Please enter the validation clientname: [chef-validator]
Please enter the location of the validation key: [/etc/chef-server/chef-
validator.pem] ~/.chef/chef-validator.pem
Please enter the path to a chef repository (or leave blank):
Creating initial API user...

Please enter a password for the new user:
Created user[user]
Configuration file written to /home/user/.chef/knife.rb
user@chef:~$
```

As mentioned earlier, this does two things:

- First, it uses the validation key and client name specified at the prompts to contact the API service and register a new client (user) with the service
- Secondly, it generates a configuration file for `knife` that has the settings needed to connect to the service from now on

Since Chef and its components are written in Ruby, the resulting configuration file is a Ruby script, which contains some code that configures `knife` so that it knows what API server to connect to, which key files to use, what client name to use, and so on.

An inspection of the configuration file that was generated by the previous command will look like the following:

```
log_level          :info
log_location       STDOUT
node_name          'user'
client_key         '/home/user/.chef/user.pem'
validation_client_name 'chef-validator'
validation_key      '/home/user/.chef/chef-validator.pem'
chef_server_url    'https://localhost:443'
syntax_check_cache_path '/home/user/.chef/syntax_check_cache'
```

Because we are using the service host as our workstation, the Chef server URL points to the localhost. If your workstation were to be a different system such as your laptop, then this URL would be the IP or hostname of the host running the Chef service.

Ensuring that your knife configuration works

After setting up `knife`, we can use it to validate that it was configured correctly by querying the Chef server using some simple commands. The `knife` commands follow the format `knife <command> <subcommand>`, where `command` is either a client, configuration, cookbook, cookbook site, data bag, environment, exec, help, index, node, recipe, role, search, ssh, status, or tag. Subcommands will vary with the command, but they typically include things such as show, create, list, and delete (among others).

As there will initially be no nodes, cookbooks, recipes, roles, data bags, and such, we will query the list of clients that the server knows about. This should be a list of two clients: `chef-webui` (as it is a consumer of the API itself) and `chef-validator` (without it, it wouldn't be possible to register a new client).

Installing Chef

The `client` command, with the `list` subcommand, prints a list of clients that the server knows about. At this point, running the command would look like this:

```
user@chef:~$ knife client list
chef-validator
chef-webui
user@chef:~$
```



If you do not get the previous output, but get an error instead, you will need to go back and make sure that all the previous steps are completed and verified.



Once you know that it works, you can use `knife` to interact with the API. Unfortunately, we do not have much data in the system just yet, but we can use the `show` subcommand in conjunction with the `client` command and a client name to display more detailed information about a client:

```
user@chef:~$ knife client show chef-webui
admin:      true
chef_type:   client
json_class:  Chef::ApiClient
name:        chef-webui
public_key:  -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBCgKCAQEaos5cQ1NxP7zKf1zRM33g
YeVvHNOO5NcICjSIvqQ5A37wwLfgtPLJQqboW7ZcNL3xYcKOlfYSEK7xha3ss8tT
A+XMifaFp3JsdheyPeIJir2bc9iltUUcbpw9PJ2aQKTBlFNx23A7ag+zBfxcDjbY
7RkdcziwB74ynd6e/K8c0JTRnA5NxoHkFc6v8a/itwujGwugWJXDQunWfCmAvjws
JgDOUu2aHOCVIVkc8it51Sc7Anx0YnCjNmhdz1xIo0MOVNOEmC9ypP0Z7mVv1C69
WWBOEvS9zimjXo4rxBwFmWkPEIG6yPQjhnuNmFd69K14vZQtAsH07AZFRSS7HLWnZ
WQIDAQAB
-----END PUBLIC KEY-----

validator:  false
user@chef:~$
```

Summary

Congratulations! If you have gotten this far, you now have a fully functional Chef service and a copy of the command-line utilities, including `chef-solo`. You now have covered the following:

- Using RVM
- Installing `chef-solo`
- Creating a simple recipe
- Running recipes with `chef-solo`
- Installing the Chef service
- Getting started with the `knife` utility
- Verifying that your Chef service is operating correctly

Now that you are able to use your Chef service, we can begin to investigate how to model our environment with Chef and see what it can do for us.

2

Modeling Your Infrastructure

Now that you have set up your server or are using the hosted offering, let's discuss how to model your infrastructure with Chef. Chef allows you to do this using building blocks that should be familiar to any system administrator.

This chapter will cover how to model your infrastructure with Chef. This will involve the following:

- Learning some terminologies and concepts relevant to Chef
- Analyzing a simple application infrastructure and seeing how it can be modeled with Chef
- Decomposing our architecture into the various components to be modeled
- Examining how data is stored and configurations are generated with Chef
- Bootstrapping and provisioning hosts with cloud providers such as AWS and Rackspace Cloud

Getting to know Chef

As with any other tool or system, there are new concepts and terminologies to be learned. Here are some terms that you may have seen or will see in this chapter:

- **Node:** A node is a system that is managed by Chef. These can be servers, desktop systems, routers, and anything else that is capable of running the Chef client and has a supported operating system.
- **Workstation:** A workstation is a special node that is used by a system administrator to interact with the Chef server and with nodes. This is where the command-line tools are executed, specifically the knife command-line tool.

- **Bootstrap:** This is a process of setting up a node to be used as a Chef client. This involves performing any work required to install the dependencies for Chef as well as Chef itself.
- **Bootstrap Script:** There are a number of possible ways to install Chef, Ruby, other core requirements, as well as any additional configuration that is needed for your specific systems. To provide this level of flexibility, the bootstrap process is scripted; on Windows, this is a batch file.
- **Recipe:** Recipes provide the instructions required to achieve a goal, such as installing a software package, configuring a firewall, provisioning users, installing printers, or managing other system resources. These are written in Ruby and executed on the nodes specified by the system administrator through the Chef console.
- **Cookbook:** A cookbook is a collection of recipes; typically, a cookbook provides one specific group of actions such as installing Apache or MySQL, providing Chef resources for a specific software tool, and so on.
- **Attributes:** Various components of the system have their own attributes and properties that describe how the software is to be configured. These properties are defined at various levels, ranging from node-specific settings to general defaults for a cookbook or a role.
- **Role:** A role is a collection of recipes and configuration data that describe how a resource should be configured in order to play that role in your overall system architecture. Examples of roles might include MSSQL Servers, Exchange Servers, IIS Servers, file servers, and so on. A role does not contain any knowledge of resources (systems) to apply the role to, only the configuration data.
- **Run List:** A run list is a list of recipes to be applied to a given node in a certain order. A run list can be composed of zero or more roles or recipes, and the order is important as the run list's items are executed in the order specified. Therefore, if one recipe is dependent upon the execution of another, you need to ensure that they run in the correct order.
- **Resource:** Resources are a way of describing what a recipe is performing. Some examples of resources would include files, directories, printers, users, packages, and so forth. A resource is an abstraction of something that is concretely implemented in a provider.

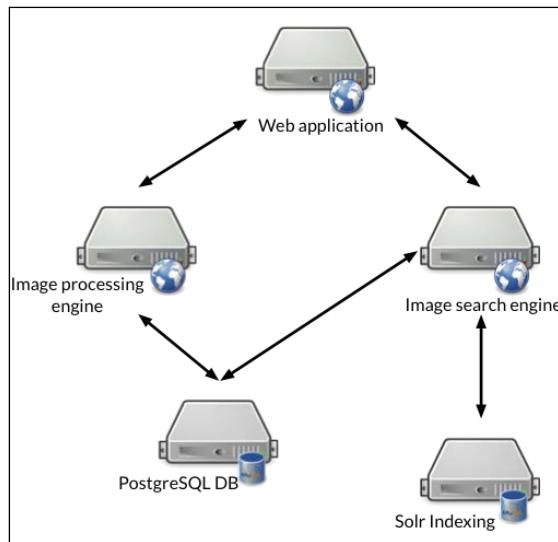
- **Provider:** A provider is a concrete implementation of a resource. For example, a user is a generic resource, but LDAP users or Active Directory users are concrete implementations of a user resource. The type of provider being selected will depend on some factors, such as the platform.
- **Data bags:** Data bags contain shared data about your infrastructure. Information that is not specific to a role or a resource, such as firewall rules and user accounts, will be stored in data bags. This is a good place to store system-wide configuration data.
- **Environments:** Environments provide a level of encapsulation for resources. For example, you may have two identical environments, one for testing and one for production. Each of these may have similar setups but different configurations, such as different IP addresses and users.

Modeling your infrastructure

Now that you're more familiar with some of the terms you need to know, let's take a look at a sample model and map it to Chef's components. At a higher level, the approach we will take is as follows:

1. Define an overview of your infrastructure that is decomposed into roles to be performed within the model (web servers, firewalls, database servers, and so on).
2. Collect or develop recipes that describe the configuration, software, and actions to be applied for those roles.
3. Bootstrap hosts with the Chef client so that they can participate in being managed.
4. Add any required configuration data into data bags to be used by nodes when running recipes such as IP address ranges, hostnames, users, software keys, or anything else that is specific to the active configuration.
5. Segregate hosts and configurations into different environments to provide a replicated infrastructure (development, staging, production, and so on). This step is optional.

In this chapter, we will be using Chef to build the infrastructure for a multi-tiered, photo-sharing application whose components are diagrammed in the following image:



Building an architecture diagram gives us a good overview of our system so that we can have a map of the system before we start building it. It is important to note that a model of our infrastructure doesn't need to be mapped directly to resources (physical or virtual); rather, it provides an abstract, high-level overview of your systems. Once you have the model, you can apply it to the resources available as you see fit.

Our sample service-oriented web application is composed of the following software services:

- A frontend web application service
- An image-processing engine
- An image search engine

Each of these components is a **role** that is being played in the system. These roles may coexist on shared resources or may be applied to dedicated resources. A service-oriented architecture is a good example to work with for several reasons:

- It is flexible and scalable
- It will provide us with a complete system that is composed of multiple independent components to model, making it more interesting as an example

In this example, in addition to these roles, we might want to further configure our infrastructure to provide two different environments: one for staging and integration testing and one for production. Again, because this is a model, our staging environment and production environment will be composed of the same roles and have the same overall architecture; however, each will have different resources and configuration data associated with them. You may choose, for example, to consolidate resources in a test environment in order to keep costs down.

For this initial overview, we will assume that we have an account with a popular cloud-server-hosting company, that the network and operating systems are installed and operational, and that we have a functional and configured Chef service and workstation.

In our hypothetical system, each service can be mapped to a specific role in Chef. To model the infrastructure described, we will have a number of roles, one per element in our architecture. In this case, we will build one role for each service in our stack as they provide very specific features.

Roles

A role describes a part that a system plays in your infrastructure through a combination of recipes to execute and configure data. These roles can be fine-grained or broadly described, depending on your needs. There are benefits and drawbacks to both the approaches: fine-grained roles are smaller and easier to work with but require a larger number of roles to manage, whereas broadly scoped roles are less flexible and not as reusable.

For example, consider a typical LAMP (Linux, Apache, MySQL, and PHP) stack. The stack could be represented by three roles: an Apache web service with PHP, a MySQL database service, and an OpenSSH service for administration. Alternatively, you could define one role that describes the installation of the MySQL database service, the SSH service, and the Apache service.

Roles themselves know nothing about resources; instead, they are a description of how to configure a system in order to fill that role. The system administrator, via the chef console, assigns roles to the node(s) that they will be applied to. This may be a one-to-one, one-to-many, or many-to-one mapping, depending upon your capacity planning. At any time, an administrator can change the list of roles that are applied to a node, adding or removing them as needed. For example, you might decide to apply all your roles to one host today for cost savings, but scale them out in the future as your budget and needs grow.

Defining roles

Let's take a look at some roles we might define to model our SOA application on as described earlier in the chapter. Here, we will define fine-grained roles as they are easier to dissect and deploy onto separate nodes later. At a higher level, the following roles are what our services need to provide.

A web application service role

When defining what a web application server will need to do, we will need the following:

- nginx HTTP service
- Ruby 2.0
- Memcached service
- PostgreSQL client libraries
- Open TCP ports on the external networks: 80 for HTTP and 443 for HTTPS

An image-processing role

This role requires some image-processing libraries and custom software to be installed:

- ImageMagick libraries
- Git (to check out the source code)
- Build tools (to compile our source)
- The latest version of our image-processing software

An image search role

A service that provides image searching through perceptual hashing will provide an image search role functionality. This role will require the following:

- A Java runtime environment (JRE or JDK)
- Our custom-built service that is developed in Java
- TCP port 8999 open to internal hosts

A PostgreSQL service role

For the PostgreSQL database service role, the list is as follows:

- PostgreSQL 9.x server
- TCP port 5432 open to internal network clients
- Database backup software to back up data to an external cloud data storage service such as S3

A Solr service role

A system that provides the Apache Solr service will need the following:

- A compatible Java runtime (Oracle JRE or OpenJDK)
- TCP port 8993 open to internal servers
- Apache Solr itself

An OpenSSH service role

An OpenSSH service role will need the following:

- OpenSSH server
- TCP port 22 open on all the interfaces

Notice that these roles have no specific host information, such as IP addresses or servers to install the software on to; instead, they are blueprints for the packages we need to install and the configuration that those roles will provide, such as open ports. In order for these role definitions to be made as reusable as possible, we will write our recipes to use node- and role-specific configuration or data from our data bags to provide the required configuration data.

In order to define these roles, you will need recipes that describe the sets of steps and operations that will be applied to hosts in order to fulfill each role. For example, the PostgreSQL database server will require you to install PostgreSQL, open the firewall, and so on. These definitions are created by developing recipes that contain the necessary information to perform the tasks required, such as installing packages, generating configuration files, executing commands, and so on. Most of the services mentioned here (our custom imaging software being the likely exception) have cookbooks that already exist and are available for download.

Implementing a role

Now that you have seen what our infrastructure might look like at a higher level, let's take a look at how we will go about implementing one of our roles in Chef. Here, we will implement the PostgreSQL server role as it is simple to configure and has a very robust cookbook available already.

As mentioned before, you will need to either develop your own cookbooks or download existing ones in order to build your systems. Fortunately, there are thousands of cookbooks already written (over 1,500 as of this writing in the Chef Supermarket) and, as we will see in further chapters, developing new cookbooks is a straightforward process.

In order to define a role, we need to create it; this can be accomplished through a web interface or by using knife. Here, and elsewhere in this book, we will use knife as the way to interact with the Chef service because it provides a consistent experience across self-managed and hosted Chef. So let's get started!

The first thing you will need to do is create a new role with knife, which is as simple as executing the following:

```
knife role create -d postgresql_server
```

This will tell knife to connect to your Chef server's API and create a new role named `postgresql_server`. The `-d` flag tells knife to skip opening an editor and instead accept the default values. If you want to see what the underlying JSON looks like, omit the `-d` flag and make sure you have an appropriate `EDITOR` environment variable set. Once you run this, you can verify that your role was created with the following command:

```
knife role list
```

This will show you that you have a single role in the system, `postgresql_server`. Currently, however, this role is empty and has no information associated with it, just a name and an entry in the system. Now that we have a role in the system, let's look at how we can work with some recipes to make our role do something useful, such as install the PostgreSQL service.

Determining which recipes you need

Recipes are how Chef knows how to make sure that the correct packages are installed, what commands need to be executed in order to open ports on the firewall, which ports need to be opened, and so on. Like any good cook, Chef has a wide array of cookbooks at its disposal, each of which contains recipes relevant to that particular set of functionality. These cookbooks can either be developed by the system administrator or downloaded from a variety of places such as GitHub, BitBucket, or from a collection of cookbooks maintained by the Chef community on the Chef Supermarket (<http://supermarket.getchef.com>). We will discuss how to download and get started with some simple recipes and then further discuss how to develop and distribute our own recipes in later chapters.

Considering how we have arranged our roles, we would need recipes to install and configure the following:

- nginx
- A PostgreSQL server
- A PostgreSQL client
- Ruby 2.0
- Solr
- Java
- OpenSSH
- A Memcached server
- Memcached client libraries
- ImageMagick
- Git
- A Custom imaging software (we will call it **Image-O-Rama**)

Here, we will take an in-depth look at the recipe required for our PostgreSQL server and how we can leverage that to install the service on a host.

Installing a cookbook

Installing a cookbook for use on our clients is quite simple and involves only two steps:

1. Developing a cookbook, or downloading the cookbook from somewhere.
2. Uploading the cookbook to the Chef service using `knife`.

To get started, we will download an existing PostgreSQL cookbook from the Chef cookbook collection and upload it to our Chef service. Note that in order to install the PostgreSQL cookbook, you will also need to install any dependencies that are required. For simplicity, they are provided here as part of the instructions; however, you may find that when you experiment with other cookbooks in the future, you will need to download a few cookbooks before all of the dependencies are met, or use a tool such as Berkshelf for managing them.

To download a cookbook from Chef's provided collection of cookbooks, we will use `knife` with the following command:

```
knife cookbook site download <cookbook_name>
```

In this case, we will need to download five different cookbooks:

- `postgresql`
- `build-essential`
- `apt`
- `chef-sugar`
- `openssl`

For each of the items in the list, we will download them using the following command:

```
knife cookbook site download postgresql
knife cookbook site download build-essential
knife cookbook site download apt
knife cookbook site download chef-sugar
knife cookbook site download openssl
```

Each download will result in an archive being downloaded to your workstation. These archives contain the cookbooks, and you will want to decompress them after downloading them. They can be downloaded anywhere, but it would probably be a good idea to keep them in a common cookbooks directory, something like `chef/cookbooks` inside your home directory would be a good idea if you need one.

Once they are downloaded and decompressed, you will need to upload them to the Chef service. This can be done with only one command using `knife cookbook upload` as follows; they are uploaded from the directory in which you stored your decompressed cookbooks:

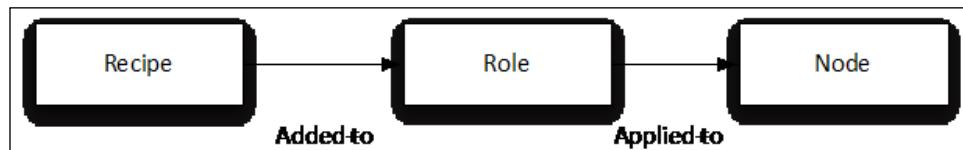
```
knife cookbook upload -o . apt build-essential postgresql chef-sugar  
openssl
```

This will upload the five cookbooks we downloaded and tell knife to search the current directory by way of the `-o .` directive. Once this is done you can verify that they have been installed using the `knife cookbook list` command.

Once they are installed, your cookbooks are registered with the Chef service, and we can take a look at how we can configure and apply the PostgreSQL server to a new Ubuntu host.

Applying recipes to roles

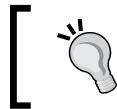
Now that you have some cookbooks registered with your Chef service, you need to add them to a role's run list in order for their behavior to take effect on any end hosts. The relationship between a recipe and any given node is shown in the following diagram:



Because of the nature of this relationship, recipes deliberately have no knowledge of individual nodes. Just as a recipe for chocolate chip cookies has no idea about who manufactured the rolling pin and spatula; a Chef recipe is simply a set of instructions on what to do and in what order to perform those actions.

Because we have uploaded our cookbooks to the system, we have already added the recipes contained inside of those cookbooks to our system; therefore, we can now associate a recipe with our recently created role. If you look at the contents of the `recipes` directory inside of the `postgresql` cookbook, you will see that there is a `server.rb` file. This describes a recipe to install the PostgreSQL server and is what we will be adding to our `postgresql_server` role in order to perform the actual installation.

To do this, we need to edit our role and add the recipe to its run list; we will do this using `knife`.



Ensure that you have a valid text editor in your EDITOR environment variable; otherwise, you will have difficulty editing your entities with knife.

In order to edit our role, we can use the knife role edit command:

```
knife role edit postgresql_server
```

This will open the JSON file that represents the postgresql_server role stored in the Chef server in a text editor where you should see the following content:

```
{
  "name": "postgresql_server",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {
  },
  "override_attributes": {
  },
  "chef_type": "role",
  "run_list": [
    ],
  "env_run_lists": {
  }
}
```

The most important section of this JSON blob at this moment is the run_list key—this is an array of all the things we want to run. This can be a list of recipes or roles, and each of those has the following naming structure:

- recipe [cookbook::recipe] for recipes
- role [role_name] for roles

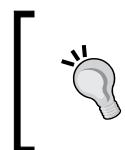
So our server recipe inside our postgresql cookbook would therefore be named "recipe [postgresql::server] ". This is exactly what we will be adding to our role's run list JSON. Update the run_list entry from the original value of an empty array:

```
"run_list": [
  ],
```

To include our PostgreSQL server recipe, use the following code:

```
"run_list": [
  "recipe[postgresql::server]"
],
```

This is all we need to change now in order to apply the PostgreSQL server role to our node.



Notice that we have not added any values to the role's attributes; this means that our recipe will be executed using its default attributes. Most recipes are written with some set of acceptable default values, and the PostgreSQL server recipe is no different.

For now, there is no need to modify anything else, so save the JSON file and exit your editor. Doing this will trigger knife to upload your modified JSON in place of the previous JSON value (after doing some validation on your JSON contents), and the role will now have the `postgresql::server` recipe in its run list. You should see an output from knife indicating that the role was saved, and you can verify that this is the case with a simple knife role show command:

```
knife role show postgresql_server
```

This will show you an overview of the role in a more human-readable format than the source JSON. For example, our role should now have one entry in the run list as shown in the following output:

```
chef_type:          role
default_attributes:
description:
env_run_lists:
json_class:        Chef::Role
name:              postgresql_server
override_attributes:
run_list:          recipe[postgresql::server]
```

Once this is complete, our role is now ready to be applied to one of our nodes. At this point, we have uploaded our cookbooks, defined a role, and associated a recipe with our newly created role. Now let's take a look at the final step: applying our new role to a node.

Mapping your roles to nodes

As has been discussed, roles are a definition of what components need to be brought together to serve a particular purpose; they are independent of the hardware they are applied to. This helps to separate concerns and build reusable components to accelerate the configuration of infrastructure in new arrangements. In order to manifest a role, it must have a node that the role is applied to; in order to manage a node, it must have the Chef client and its dependencies installed and be registered with the Chef service.

Once a node is registered with Chef, you can set node-specific properties, assign roles and run the `chef-client` tool on the host in order to execute the generated run lists. For our sample application stack, we may have the following hosts running Ubuntu Linux 14.04:

- cayenne
- watermelon
- kiwi

Once they are bootstrapped and registered with the Chef service, we will then decide which roles are to be applied to which nodes. This could yield a configuration that looks like the following:

- cayenne
 - Web application service role
- watermelon
 - A PostgreSQL database role
 - A Solr search engine role
- kiwi
 - An image-processing role
 - An image search role

Without any hardware, roles are just an abstract blueprint for what needs to be configured together to provide a particular type of functionality. Here, we have combined our resources (cloud instances or dedicated hardware) and our recipes to build a concrete instance of our services and software.

In order to apply our newly created role to our host, watermelon, we will need to bootstrap that host, which will install the Chef client on the host and then register it with the Chef service. This is really a simple process, as we will see here, and is achieved using the `knife bootstrap` command:

```
knife bootstrap -x root -d ubuntu14.04 <ip address>
```

 For our example, the node will use an Ubuntu 14.04 host created on DigitalOcean, an inexpensive cloud-hosting provider; you can bootstrap just about any modern Linux distribution, but if you are following along with the commands in the book, you will get the best results by using an Ubuntu 14.04 machine.

This process will go through the steps required to install the Chef client on the node and register it with your Chef service. Once it is complete, you will see that the Chef client has finished with an output similar to the following:

```
Chef Client finished, 0/0 resources updated in 4.264559367 seconds
```

If you want to verify that the host has been added, a simple `knife node list` command will show you that it has been registered with the Chef service. If you don't see the client output above, or you don't see the newly bootstrapped node in your list, make sure that the output of `knife bootstrap` doesn't indicate that anything went wrong along the way.

Once our node is registered, we can add our `postgresql_server` role to our node's run list using the following `knife` command:

```
knife node run_list set watermelon role[postgresql_server]
```

This command will set the run list on our new host, watermelon, to contain the `postgresql_server` role as its only entry. This can be verified using the `knife node show` command:

```
knife node show watermelon
```

```
Node Name:    watermelon
Environment: _default
FQDN:        watermelon
IP:          162.243.132.34
Run List:    role[postgresql_server]
Roles:
Recipes:
Platform:   ubuntu 14.04
Tags:
```

Now that the node has a run list with entries, it's time to actually converge the node. Converging a node means that the Chef server will compile the configuration attributes and then provide the end host with a complete list of recipes to run along with the required cookbook data and then execute them on the node.

Converging a node

Converging a node is done by executing the `chef-client` command-line utility on the host; this can be done in one of two different ways. The simplest way is to SSH into the host using an SSH client and then execute `chef-client` as the root; another way is to use `knife` to execute a command on a set of hosts in parallel, which will be discussed in later chapters. For now, simply SSH into your server and run `chef-client` as the root:

```
root@watermelon:~# chef-client
```

The Chef client will connect to the Chef service and download any information needed to execute its complete run list. A node's run list is determined by expanding every entry in the node's run list until it is a list of recipes to execute. For example, our node contains one element in its run list, the `postgresql_server` role. This role, in turn, contains one entry, the `postgresql::server` recipe, which means that the fully expanded run list for our node contains only one entry. In this simple case, we could have just added the recipe directly to our node's run list. However, this has a number of shortcomings, including not being able to add extra configuration to all the PostgreSQL servers in our infrastructure, as well as a number of other reasons that will be discussed in the future.

In addition to computing the run list, the Chef service will also determine what the final set of configuration data for our node will look like and then deliver it to the client. This is computed according to a set of rules shown later in this chapter. Once that is delivered, the client will also download all the cookbooks needed in order to know how to execute the recipes specified in the final run list. In our case, it will download all the five cookbooks that we uploaded previously, and then, when the client run is complete, the result will be presented in a running PostgreSQL server.

Once the client run is complete, it will report on how long the run took and how many resources were modified. The output should look something like the following:

```
Chef Client finished, 8/10 resources updated in 61.524995797 seconds
```

Assuming that nothing went wrong, you should now have a fully functional PostgreSQL server deployed to your new host. This can be verified by looking at the process list for a PostgreSQL process:

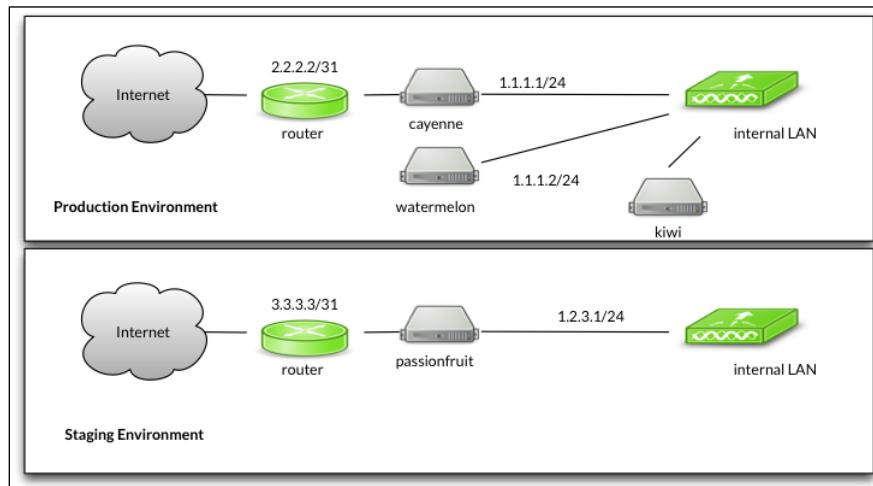
```
root@watermelon:~# ps ax |grep post
11994 ? S 0:00 /usr/lib/postgresql/9.3/bin/postgres -D
```

There you have it, with only one command; your node has now been provisioned as a PostgreSQL database server. Now let's take a look at how we can use some other features of Chef to model our infrastructure.

Environments

Beyond creating roles and having resources to apply them to, there are often requirements around grouping certain resources together to create a distinct **environment**. An example of this might include building a staging environment that functions exactly like a production environment for the purposes of preproduction testing and simulation. In these cases, we would have an identical set of roles but would very likely be applying them to a different set of nodes and configuration values. Chef achieves this separation through the environment primitive, which provides a way to group separate configurations and hosts together so that you can model similar, yet separate, portions of your infrastructure.

In our hypothetical infrastructure, the three hosts in our production environment may be condensed down to one server in preproduction in order to save money on resources (or for other reasons). To do this, we would need to bootstrap a node, perhaps named `passionfruit` and then configure it to have all of the roles applied to it, rather than spreading them out across systems, as shown in the following figure:



Here, in the previous image, you can see that each environment has a very similar setup but a different set of IP addresses and resources. Even though we have a heterogeneous hardware scale in our environments (production has three nodes, and preproduction has only one), any changes we make will be applied to all of the systems in a consistent manner.

In order to achieve this type of functionality, Chef needs a way to organize and compile the configuration data in order to provide it to the end host when the time comes to configure the host. Now that we understand how to model our systems with Chef, let's take a look at how Chef handles the configuration data to make all of this happen.

Organizing your configuration data

Chef runs on configuration data—this data can be stored in a variety of different locations, and with a variety of purposes. When computing the final configuration for a given node, the sources of configuration data are then "squashed" into a single, authoritative configuration to be deployed. Those sources include the following:

- **Cookbooks:** To provide reasonable defaults for recipes
- **Nodes:** Node-level overrides and defaults
- **Roles:** Per-role configuration data
- **Data bags:** System-wide configuration data storage

Data from these locations is combined to produce a final hash of attributes when a client requests its run list from the server. Cookbooks provide a baseline set of attributes that the recipes inside rely on. These attributes act as "sane defaults" for the recipes that, in the absence of overriding values, are sufficient to execute the recipes without extra work. Other sources, including the environment, role and node itself, may in turn override these attributes in order to provide the final configuration.

When developing recipes, these attributes can be accessed through the `node` hash and are computed by Chef using a set of rules to determine precedence. The order of precedence when computing this hash is broken down into the following levels (lowest to highest priority):

- Default
- Normal (also `set`)
- Override

Within each level, the sources of attribute data in order of increasing precedence are as follows:

- The attributes file inside of a cookbook
- Environment
- Role
- Node

This means that a node-specific override attribute takes precedence over all others, which in turn is more important than the role, environment and cookbook override attributes, and so down the chain of precedence. As your scope becomes narrowed from the most generic description of a role—the recipes—to the most specific component in the system—the actual node itself—these settings override the more generic values. A node knows best what the authoritative configuration should be as opposed to a recipe, which does not know anything about resources on the host. For example, consider the following scenario in which you have two hosts, *potassium* and *chromium*. For some legacy reason, their disks are configured slightly differently, as follows:

Potassium:

- 16 GB root partition
- 250 GB SSD data partition in /opt

Chromium:

- 32 GB root partition
- 400 GB EBS disk mounted at /usr

In order to install the PostgreSQL database server, you need to make sure you install it at a location that provides enough storage space for the data. In this example, there will be more data than either root disks can contain. As a result, the data directory will need to reside in /opt on potassium and /usr on chromium. There is no way that the PostgreSQL recipe can account for this, and the `postgresql_server` recipe does not know anything about its resources. Subsequently, the logical place to configure the data directory is at the node level. If the default location according to the recipe were /usr/local, then a node-level override may not be needed for chromium; however, in the case of potassium, it could be directed to store data in /opt/data instead.

What all this means is that as you develop recipes, any `default` attribute set by your cookbook will be the lowest priority. You can safely set some reasonable defaults in your cookbook knowing that they will only be used as a fallback if nobody overrides them further down the chain.

Example attribute data

A simple default attributes file for PostgreSQL cookbook might look like the following:

```
default['postgresql']['port'] = "5432"
default['postgresql']['data_dir'] = "/usr/local/pg/data"
default['postgresql']['bind_address'] = "127.0.0.1"
```

Notice that the attributes for a cookbook are a Ruby hash. Typically, good practice dictates that the namespace (first key in the hash) is the same name as the cookbook (in this case, `postgresql`), but this does not need to be the case. Due to cookbooks often containing multiple recipes, a cookbook's attributes file will often have per-recipe default configurations. Consider a further evolution of the PostgreSQL attributes file if it were to contain recipes for both the server and the client installation:

```
default[:postgresql][:client][:use_ssl] = true
default[:postgresql][:server][:port] = "5432"
default[:postgresql][:server][:log_dir] = "/var/log/pglog"
```

There are times when just a simple attributes file doesn't make sense because the configuration may be dependent on some property of the node being managed. The fact that the attributes file is just a Ruby script allows us to implement some logic inside our configuration (though you should avoid being overly clever). Consider a recipe where the default group for the `root` user depends on the platform you are using: `"user d` on BSDs, `"n` on Ubuntu Linux, and `"n Ubu` elsewhere. Chef provides a method, `value_for_platform`, that allows the attribute to be changed depending on the platform the recipe is being executed on, as the following example demonstrates:

```
default[:users][:root][:primary_group] = value_for_platform(
  :openbsd => { :default => "wheel" },
  :freebsd => { :default => "wheel" },
  :ubuntu => { :default => "admin" },
  :default => "root"
```

Where it makes sense, attributes can also be shared between cookbooks. There are limited uses for this, and it should be used with care as it blurs the boundaries between cookbooks and causes them to become too tightly coupled with one another.

Data bags

There are times when configuration data transcends a recipe, role, environment, or node. This type of configuration data tends to be system-wide data such as the following:

- Firewall rules for various types of hosts
- User accounts
- SSH keys
- IP address lists (white lists and black lists)
- API keys
- Site configuration data
- Anything that is not unique to a specific entity in your infrastructure

Data bags are very free-form, as the name implies; recipes that rely on data from data bags will impose their own expectations of the organization within a data bag, but Chef itself does not. Data bags can be considered, like all other Chef configuration data, to be one large hash of configuration data that is accessible to all the recipes across all the nodes in the system.

Knowing when to use data bags

Building firewall rules are a good use case for data bags. A good cookbook is an island by itself; it makes as few assumptions about the world as possible in order to be as flexible and useful as it can be. For example, the PostgreSQL cookbook should not concern itself with firewall rules, that is, the realm of a `firewall` cookbook. Instead, an administrator would leverage a generic firewall model and a cookbook with a specific firewall implementation such as the UFW cookbook to provide those features. In this case, if you were to look at the UFW cookbook, you would see the `ufw::databag` recipe making use of data bags to make the firewall rules as flexible as possible.



UFW stands for uncomplicated firewall, a popular iptables-based firewall rule generation package for Linux that comes with many modern distributions and eases the management of a firewall configuration.

In this case, `ufw::databag` expects that there is a specific data bag named `firewall` and inside of it are items that share names with roles or nodes; this is in line with the notion that data bags are free-form, but certain cookbooks expect certain structure. If our infrastructure model had two roles, `web_server`, and `database_server`, then our firewall data bag would have contained two items named accordingly. The `web_server` item could look like the following hash:

```
{  
  "id": "web_server",  
  "rules": [ {  
    "HTTP": {  
      "dest_port": "80",  
      "protocol": "tcp"  
    },  
    "HTTPS": {  
      "dest_port": "443",  
      "protocol": "tcp"  
    }  
  ]  
}
```

Here, `id` of the item maps to the name of the item, which is also the name of the role, so that the `ufw::databag` recipe knows where to fetch the data it needs to build its internal firewall rules. To compute its list of firewall rules to apply, the `ufw::databag` recipe examines the list of roles that the node is configured with and then loads the corresponding data from the items in the `firewall` data bag.

As you can see, data bags allow you to store centralized, free-form configuration data that does not necessarily pertain to a specific node, role, or recipe. By using data bags, cookbooks for configuring users, databases, firewalls, or just about any piece of software that needs shared data can benefit from the data stored in a data bag.

One might wonder why we have data bags when we already have attribute data, and that would be a good question to ask. Attributes represent the state of a node at a particular point in time, meaning that they are the result of a compaction of attribute data that is being supplied to a node at the time the client is being executed. When the Chef client runs, the attribute data for all the components contributing to the node's run list is evaluated at that time, flattened according to a specific priority chain, and then handed to the client. In contrast, data bags contain arbitrary data that has no attachment to a specific node, role, or cookbook; it is free-form data that can be used from anywhere for any purpose. One would not, for example, be likely to store user configuration data in a cookbook or on a specific node because that wouldn't make much sense; users exist across nodes, roles, and even environments. The same goes for other data such as network topology information, credentials, and other global data that would be shared across a fleet.

Large-scale infrastructure

One of the many benefits of Chef is the power to apply roles to nodes at scale. This means that once you define a set of roles and some supporting recipes, you can apply them to one host just as easily as any other. There are many organizations that manage very large infrastructure using Chef, including companies such as Facebook, Ancestry, and Riot Games. With Chef, configuring one hundred hosts is as straightforward as configuring one host. Being able to achieve scalability goals while remaining cost-effective is a critical part of running a technology business. To this end, Chef provides tools to automate the creation, provisioning, maintenance and termination of virtual hosts using the provided tools, which can help achieve both scalability and conservation of resources. The next chapter discusses how to use Chef to extend your infrastructure into the cloud.

Summary

Now that you've learned the key terminology that Chef uses and dissected an example infrastructure a bit, you can see the following:

- Infrastructure can be decomposed into the various roles that resources (nodes) play within that infrastructure
- A combination of recipes and configuration data provide us with roles that describe a part of our overall infrastructure
- Chef analyzes the roles applied to hardware resources (nodes) and generates a run list that is specific to the node that those roles are being applied to
- A run list is then combined with the cookbooks, recipes, templates, and configuration data to build a specific set of scripts that are executed on the node when the chef-client is run
- We can apply these methodologies of automated configuration to cloud servers and physical systems alike.

Now that you understand how Chef models interact, let's take a look at how we can get started with cloud services using Chef.

3

Integrating with the Cloud

Being able to configure new hosts automatically means that if you outgrow your existing resources, you can easily bring up new servers to increase your capacity with very little effort. The Chef command-line tool, `knife`, provides the ability to provision new hosts with cloud services automatically from the command line if configured correctly. This chapter introduces you to using Chef with two popular cloud platform providers: Amazon EC2 and Rackspace Cloud. Here, you will learn to use `knife` with both of these providers in order to perform the following:

- Provision new hosts according to your hardware needs
- Bootstrap the Chef client and register hosts with the Chef service
- List your existing capacity
- Terminate unneeded capacity

You will see that all of this can be done using the command-line tools provided without ever having to log in to the provider's web interface, and all of this within a few minutes of time.

Leveraging the cloud

Cloud computing providers such as Rackspace Cloud and Amazon EC2 provide on-demand computing power at the push of a button, a feature that has become immensely popular with developers and systems administrators alike. One of the most touted benefits of cloud computing is cost savings; however, these on-demand instances can become very expensive if they are left running. Often the capacity of time will be configured in order to handle large-scale events and then left online because of the time required to reconfigure the systems if they are needed again. As underutilized capacity ends up costing money rather than saving it, being able to reduce or expand the capacity quickly and easily will help you match your computing needs while saving both time and money.

This section specifically looks at two of the more popular cloud providers: Amazon EC2 and Rackspace Cloud; however, there are others, and the techniques described here will be broadly applicable to any other supported cloud provider.

Amazon EC2

Amazon EC2 is a very popular cloud-computing platform, and `knife` has support to manage EC2 instances from the command line through the `knife-ec2` plugin. The following steps demonstrate how you can work with EC2:

1. Install the EC2 `knife` plugin.
2. Set up your SSH keys for use with EC2.
3. Configure `knife` with your AWS credentials.
4. Find the desired AMI.
5. Provision a new host with `knife`.
6. Bootstrap the newly created host.
7. Configure the new host with a role.

Installing the EC2 knife plugin

As of Chef 0.10, the `ec2` subcommands have been moved from being built in `knife` to an external gem, `knife-ec2`. In order to use EC2 commands, you will need to install the gem, which can be done via the following command:

```
gem install knife-ec2
```

This will install all of the gem dependencies that the EC2 plugin requires.



Some of the cloud provider plugins have conflicting dependencies, so it may be best to leverage a gem manager in order to isolate them. For example, using RVM or rbenv, you might create one Rubygem environment per provider so that you could switch back and forth with a simple command such as `rvm gemset use chef-ec2`.

Setting up EC2 authentication

In order to manage your EC2 hosts, you will need your EC2 key-pair properly registered with SSH and your AWS access keys set in your `knife` configuration file.

To do the first, make sure you have your EC2 SSH keys downloaded and registered with your SSH agent. One way to do this is to add the following to your SSH configuration file, typically, `$HOME/.ssh/config`:

```
Host *.amazonaws.com
  ForwardAgent yes
  CheckHostIP no
  StrictHostKeyChecking no
  UserKnownHostsFile=/dev/null
  IdentityFile ~/.ssh/ec2_keypair.pem
```

In order to configure your AWS keys, you will need to add some information to your `knife.rb` configuration file (`$HOME/.chef/knife.rb`):

```
knife[:aws_access_key_id] = "YOUR ACCESS KEY"
knife[:aws_secret_access_key] = "SECRET KEY"
```

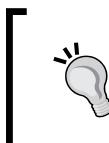
These keys tell `knife` which AWS credentials to use when making API calls to perform actions such as provision new hosts and terminate instances. Without this, `knife` will be unable to make API calls to EC2. With these required changes made, let's look at how to create a new EC2 instance with `knife`.

Provisioning an instance

Initially, we will look at provisioning an instance using one of the Ubuntu AMIs. With `knife`, we can specify the AMI to use, the availability zone to target, and the size instance to be created. For example, to create an `m1.large` size in the `us-east-1e` availability zone with Ubuntu 12.04.3 LTS, we would need to use the AMI with `ami-23447f4a` as its identifier.

In order to determine the AMI ID, you will need to look it up at the following URL:

<http://uec-images.ubuntu.com/>



Remember that when deciding which AMI to use, some of the EC2 instances will be 32 bit and some 64 bit; choose the appropriate AMI based on the instance type, region, and storage method you want to use.

The progress of provisioning can be seen using the following command:

```
$ knife ec2 server create -I ami-23447f4a -f m1.large -Z us-east-1e -N
<node name> -x ubuntu --sudo
```

The output from the previous command will show you the progress of the provisioning (this may take a minute or two, depending on the region, instance size, how long status checks take, and so on):

```
[user] % knife ec2 server create -I ami-23447f4a -f m1.large -S ec2-keypair -Z us-east-1e -N <node name> -x ubuntu --sudo  
Instance ID: i-0dfec92d  
Flavor: m1.large  
Image: ami-23447f4a  
Region: us-east-1  
Availability Zone: us-east-1e  
Security Groups: default  
Tags: Name: i-0dfec92d  
SSH Key: ec2-keypair  
  
Waiting for instance.....  
Public DNS Name: ec2-54-80-59-97.compute-1.amazonaws.com  
Public IP Address: 54.80.59.97  
Private DNS Name: ip-10-157-31-234.ec2.internal  
Private IP Address: 10.157.31.234
```

Bootstrapping the instance

As you can see, knife will tell you the public IP and public DNS name of the new instance along with the instance ID, tags, and so forth. Once the instance is provisioned and is online, it will need to be bootstrapped. Remember that bootstrapping will install the Chef client and register the instance with the Chef service, which we can do in the same way we bootstrap any other host:

```
$ knife bootstrap <instance-public-ip-address> -N <node-name> -x ubuntu --sudo
```

As EC2 provisions each instance with an `ubuntu` user that has `sudo` privileges, we provide the bootstrap command with `-x ubuntu` and `--sudo` to ensure we have the required privileges to perform the bootstrapping. Additionally, as you more than likely do not want the AWS-provided DNS name as the node name, the Chef node name is set through the `-N <node-name>` command line flag. Once the bootstrap step is finished, assuming that there are no errors, verify that your newly provisioned host is listed in your chef service:

```
$ knife node list
```

The output will contain your newly bootstrapped node ID, as specified by you in the command line or the DNS name, if you don't specify a node name. You have now provisioned a new EC2 instance and registered it with your Chef service with only two commands!

Terminating the instance

Once you are done with testing, you may not want to leave the EC2 instance running, as it will incur costs if it remains idle. To ensure this doesn't happen, perform the following four steps:

1. List your EC2 instances
2. Delete the server from EC2
3. Remove the server from Chef
4. Verify that the instance no longer exists in Chef or EC2

To list our EC2 instances, use the `server list` subcommand of the `ec2` command, which will list all of the EC2 instances in the specified region. If you do not specify a region, `us-east-1` is the default region. The full command to list EC2 servers is as follows:

```
$ knife ec2 server list
```

As an example, executing this command after provisioning the first host will show a table of one instance as follows:

Instance ID	Name	Public IP	Private IP
i-0dfec92d	i-0dfec92d	54.80.59.97	10.157.31.234

For most `knife` commands, you will need the instance ID so the previous table can be truncated to fit in print.



Listing EC2 nodes will result in a table that contains all the currently provisioned EC2 instances in the region by means of the EC2 API, which is separate from the Chef service API. This means you will get a list of all the instances in EC2 whether or not they are registered with Chef. The full table will contain most of the information you can see on the EC2 control panel, including the public and private IP, flavor, AMI, SSH key, and so on.

Deleting an instance is just as easy as creating or listing them. Here, the `server delete` subcommand is invoked with the instance identifier to be terminated. This will use the EC2 API to issue a terminate command—this is not reversible and so the command will prompt you to ensure that you really did want to delete the instance:

```
[user] % knife ec2 server delete i-0dfec92d
Instance ID: i-0dfec92d
Flavor: m1.large
Image: ami-23447f4a
Region: us-east-1
Availability Zone: us-east-1e
Security Groups: default
SSH Key: ec2-keypair
Root Device Type: instance-store
Public DNS Name: ec2-54-80-59-97.compute-1.amazonaws.com
Public IP Address: 54.80.59.97
Private DNS Name: ip-10-157-31-234.ec2.internal
Private IP Address: 10.157.31.234
Do you really want to delete this server? (Y/N)
WARNING: Deleted server i-0dfec92d
WARNING: Corresponding node and client for the i-0dfec92d server
were not deleted and remain registered with the Chef Server
```

Removing the Chef node

At this point, the EC2 instance is being terminated and removed from your account. However, it is not removed from the Chef service that needs to be done separately with the `node delete` command. Here, the Chef node name is specified, not the instance identifier:

```
$ knife node delete my-first-ec2-instance
```

Verify that the node was removed from Chef using `node list`:

```
$ knife node list
```

The output should show you that your EC2 instance is no longer registered with Chef.

Rackspace Cloud

Rackspace Cloud is another popular cloud-computing provider that is well supported by Chef. Similar to EC2, there is a `knife` plugin for Rackspace Cloud:

```
gem install knife-rackspace
```

In the same way that AWS requires a set of credentials to interact with the API to create and terminate instances, Rackspace Cloud has its own configuration. However, the Rackspace Cloud API is a little simpler; you will need to provide `knife` with your Rackspace Cloud's username and API key. For those who do not already have their API key, it can be found in your Rackspace Cloud control panel. The appropriate configuration to add to your `knife.rb` file is as follows:

```
knife[:rackspace_api_username] = "Your Rackspace API username"
knife[:rackspace_api_key] = "Your Rackspace API Key"
```

This data can be hard coded into your configuration file, or since the `knife` configuration file is just Ruby, it can be generated by evaluating environment variables or looking at a local file. This is useful if you are submitting your `knife.rb` file into a source repository so that credentials are not leaked.

Provisioning an instance

Rackspace Cloud server provisioning is just as straightforward as it is with EC2. There is some variation in the command-line options passed to `knife` because of the way Rackspace provides images for systems. Instead of using the instance size and an AMI, you can specify the flavor of the system to provision (the node's CPU, memory, and disk allocation) and the operating system to image the instance with. In order to determine what flavors are available, the `knife-rackspace` plugin provides the `rackspace flavor list` subcommand:

```
$ knife rackspace flavor list --rackspace-region=IAD
```

As it is possible that there are different capacities in different regions, it is a good idea to check what is available in the region where you want to provision a node. This will result in a list of flavors and their specifications; as of now, some of the current offerings in IAD are as follows:

ID	Name	VCPUs	RAM	Disk
2	512MB Standard Instance	1	512	20 GB
3	1GB Standard Instance	1	1024	40 GB
4	2GB Standard Instance	2	2048	80 GB
performance1-1	1 GB Performance	1	1024	20 GB
performance1-2	2 GB Performance	2	2048	40 GB
performance2-120	120 GB Performance	32	122880	40 GB
performance2-15	15 GB Performance	4	15360	40 GB

In addition to knowing which flavor to provision, you need an image identifier (similar to an AMI) to apply to the new host. Again, this list may vary with region and possibly change over time so there is a command, `rackspace image list`, to list the various images:

```
$ knife rackspace image list --rackspace-region=IAD
```

The output here is quite long, so it has been sampled to show enough to be useful:

ID	Name
ba293687-4af0-4ccb-99e5-097d83f72dfe	Arch 2013.9
41e59c5f-530b-423c-86ec-13b23de49288	CentOS 6.5 (PVHVM)
857d7d36-34f3-409f-8435-693e8797be8b	Debian 7 (Wheezy)
896caae3-82f1-4b03-beaa-75fbdde27969	Fedora 18 (Spherical Cow)
fb624ffd-81c2-4217-8cd5-da32d32e85c4	FreeBSD 9.2
1705c794-5d7e-44d6-87da-596e3cf92144	Red Hat Enterprise Linux 6.5
df27d481-63a5-40ca-8920-3d132ed643d9	Ubuntu 13.10
d88188a5-1b02-4b37-8a91-7732e42348c1	Windows Server 2008 R2 SP1

As you can see, there are a number of Linux, BSD, and Windows distributions available to provision. In order to provision a new host, you will use the `server create` command, similar to the EC2 command. The following `knife` command will provision a 512 MB host with Ubuntu 13.10 in the IAD datacenter:

```
$ knife rackspace server create -I df27d481-63a5-40ca-8920-3d132ed643d9  
-f 2 --rackspace-region=IAD
```

As soon as the API responds to the request to provision a new host, you will see the Rackspace metadata for the host, such as the instance ID, name, flavor, and image:

```
Instance ID: 993d369f-b877-4f0f-be4b-cfc45c240654  
Name: rs-21230044929009695  
Flavor: 512MB Standard Instance  
Image: Ubuntu 13.10 (Saucy Salamander)
```

Shortly after this—once the system has been provisioned, the network interfaces have been configured, and the root password has been set—the IP and root password will be displayed:

```
Public DNS Name: 162.209.104.248.rs-cloud.xip.io  
Public IP Address: 162.209.104.248  
Private IP Address: 10.176.65.92  
Password: yZ3D3Tck8uGm
```

After SSH becomes available, `knife` will initiate the process of bootstrapping the host. By default, `knife` will use the `chef-full` template, which will install Chef via the omnibus installer for the platform you are bootstrapping. This can be altered by providing `knife` with the `-d` command-line option. Assuming that the host is bootstrapped properly, the system data will be displayed once again for your information:

```
Instance ID: 993d369f-b877-4f0f-be4b-cfc45c240654
Host ID: c478865ebb70032120024a9a2c8c65b9bb0913087991d4bab5acde00
Name: rs-21230044929009695
Flavor: 512MB Standard Instance
Image: Ubuntu 13.10 (Saucy Salamander)
Public DNS Name: 162.209.104.248.rs-cloud.xip.io
Public IP Address: 162.209.104.248
Private IP Address: 10.176.65.92
Password: yZ3D3Tck8uGm
Environment: _default
```

Once the bootstrap step is finished, assuming that there are no errors, verify that your newly provisioned host is listed in your chef service:

```
$ knife node list
```

The output will contain your newly bootstrapped node ID as specified by you in the command line (via `-N`) or the name generated by Rackspace (in this example, it will be `rs-21230044929009695`). Congratulations! You have provisioned a new Rackspace instance with a single command.

Terminating an instance

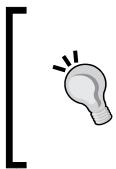
Once you are done with testing, you may not want to leave the EC2 instance running, as it will incur costs if it remains idle. To ensure this doesn't happen, perform the following four steps:

1. List your Rackspace servers.
2. Delete the server from Rackspace.
3. Remove the server from Chef.
4. Verify that the instance no longer exists in Chef or Rackspace.

To list your Rackspace instances, use the `server list` subcommand of the `rackspace` command, which will list all of the Rackspace instances in the specified region. Similar to the output from the EC2 `server list` command, the output will look like the following:

```
$ knife rackspace server list --rackspace-region=IAD
```

Instance ID	Name
993d369f-b877-4f0f-be4b-cfc45c240654	rs-21230044929009695



Similar to the EC2 output, the resulting table is too wide for print so only the instance ID and node name is shown. You should expect to see public and private IP addresses, instance types, and some other data that you will be able to see on the Rackspace Cloud control panel as well.



You can delete an instance using a single command; the `server delete` subcommand is invoked with the Rackspace instance identifier to be terminated. Remember that this is not reversible, so the command will prompt you to ensure that you really do want to delete the instance:

```
$ knife rackspace server delete 993d369f-b877-4f0f-be4b-cfc45c240654  
--rackspace-region=IAD
```

```
Instance ID: 993d369f-b877-4f0f-be4b-cfc45c240654
```

```
Host ID: c478865ebb70032120024a9a2c8c65b9bb0913087991d4bab5acde00
```

```
Name: rs-21230044929009695
```

```
Flavor: 512MB Standard Instance
```

```
Image: Ubuntu 13.10 (Saucy Salamander)
```

```
Public IP Address: 162.209.104.248
```

```
Private IP Address: 10.176.65.92
```

```
Do you really want to delete this server? (Y/N) y
```

```
WARNING: Deleted server 993d369f-b877-4f0f-be4b-cfc45c240654
```

```
WARNING: Corresponding node and client for the 993d369f-b877-4f0f-be4b-cfc45c240654 server were not deleted and remain registered with the Chef Server
```

Removing the Chef node

At this point, the EC2 instance is being terminated and removed from your account. However, it is not removed from the Chef service; this needs to be done separately with the `node delete` command. Here, the Chef node name is specified, not the instance identifier:

```
$ knife node delete rs-21230044929009695
```

Verify that the node was removed from Chef with `node list`:

```
$ knife node list
```

The output should show you that your recently created Rackspace instance is no longer registered with Chef.

Summary

The ability to scale your infrastructure through a combination of on- and off-site hosts is incredibly powerful. If you need more capacity, you can easily bring up new hosts on EC2, Rackspace Cloud, or any similar platform. Additionally, these techniques apply to not only public cloud providers but also to private cloud platforms such as VMWare vSphere and others (provided, a suitable plugin for `knife` exists).

As you have seen, with Chef you can spin up and spin down the server capacity to meet your needs with very little effort. Once your infrastructure management is automated, you can focus on higher level problems such as building scalable services and scaling to meet your customers' demands.

Expanding on this capability, you could use these tools to perform the following:

- Manually increase or decrease the capacity in order to match the demand
- Write a tool to analyze the current resource load and react accordingly
- Predict the future capacity and scale appropriately on a given schedule

Now that we have the ability to bring up some hosts to work with, we can take a look at how to work with cookbooks to learn how they work and how to build new ones.

4

Working with Cookbooks

Cookbooks are one of the fundamental components of the Chef system. They are containers for recipes, providers, resources, templates, and all the logic and information required to manage your infrastructure. This chapter covers the following:

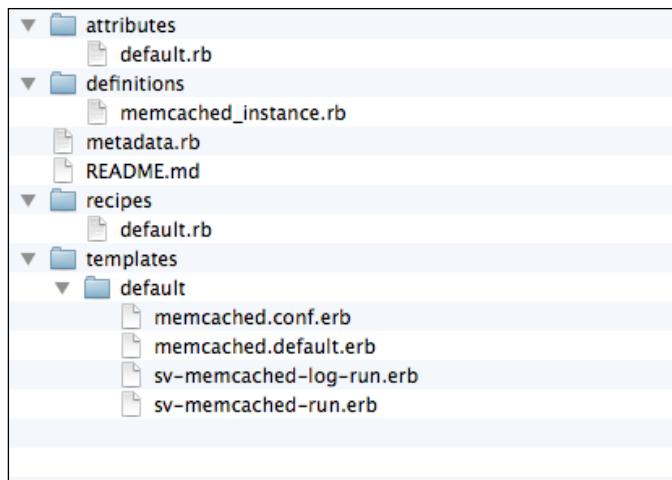
- Organization of cookbooks
- Building cookbooks
- Developing recipes
- Handling multiple platforms for a cookbook organization

Cookbooks are one of the core components of Chef. They are, as their name suggests, a collection of recipes and other data that when combined provide a specific set of functionality to a system administrator. In each cookbook, you will find a collection of directories and files that describe the cookbook and its recipes and functionality. The core components of a cookbook are as follows:

- Cookbook metadata
- Attributes
- Recipes
- Templates
- Definitions
- Resources
- Providers
- Ruby libraries
- Support files

A cookbook is a collection of files and directories with a well-known structure. Not every cookbook has all of these components. For example, there may be no need to develop custom resources or providers in a cookbook that only uses Chef-supplied resources. However, every cookbook does need to have a metadata file that provides various bits of information such as its name, version, dependencies, and supported systems.

Let's take a look at the memcached cookbook as it is a reasonably simple cookbook that is capable of installing and configuring the memcache server, which is a distributed memory-backed cache service:



Here, you can see that this cookbook contains **attributes**, **definitions**, **recipes**, and **templates**, as well as a file named `metadata.rb` (the metadata file) and a `README.md` file. It is a good idea to provide examples of how to use your cookbook and recipes in some sort of documentation, such as a `README.md` file. When you look at the preceding screenshot, you will see that the directory names map to the component names and each contains some files or subdirectories with files. We will discuss the organization of the specific Chef components in greater detail further on as we dive into more details on each type later. For now, it is sufficient to know that the directory structure is designed to group together files for each type of component. Also notice that, as mentioned earlier, this cookbook is an example of one that does not have all the components, as there are no new **resources** or **providers** in this cookbook.

Some of these files are purely informational and have no effect on your recipes or Chef itself, such as the `README.md` file. This file, and others such as `CHANGELOG`, `LICENSE`, or `DEVELOPMENT` files, is included to convey information to you about how to participate, license, or otherwise use the cookbook.

There is a lot of information that is stored inside a cookbook – this information includes the steps to take in order to achieve a desired effect such as the installation of a service or provisioning of users on a host. A high-level overview of the content that we will be learning about in this chapter, so that you have an idea of how the components work together before you learn about them in depth, is shown as follows:

- **Attributes:** These are attributes that the cookbook's recipes rely on. A well-defined cookbook should contain some sane defaults for the recipes such as installation directories, usernames, downloadable URLs, and version numbers. Anything a recipe expects the node to have defined should be given a default value so that the recipe will behave as expected.
- **Recipes:** Ruby scripts define the recipes in the cookbook. A cookbook can contain as few as one or as many recipes as its author would like to put into it. Most package-specific cookbooks only contain a few recipes, while some cookbooks, such as your organization's private cookbook, may have dozens of recipes for internal use.
- **Templates:** These are Ruby ERB files that are used to describe any file that needs to have some dynamic data in it; often, these are used for startup scripts or configuration files.
- **Resources:** These describe a resource that can be used in a recipe. Resources are Ruby scripts that use Chef's resource **domain-specific language (DSL)** to describe various actions, attributes, and other properties of the resource.
- **Providers:** These describe an implementation of a resource; in the case of the supervisord cookbook, the service provider file outlines the actual implementation-specific logic of the actions that a resource can perform. There are many types of services that you could have: supervisord, runit, monit, bluepill, and so on.

Additionally, cookbooks may include a variety of support files that are not directly part of the recipes, such as the following:

- **Definitions:** These are used to build reusable templates for resources. Perhaps you want to define the structure of a user account, a background worker, or a runnable process. These are a way to programmatically describe what these look like and implement any logic they might need.
- **Ruby libraries:** Any reusable code that your recipes need can be included in the cookbook. Things that go in here are accessible by your recipes and automatically loaded for you.

- **Support Files:** These are arbitrary data files that don't fall into any of the other categories.
- **Tests:** Recipes, composed of Ruby code, can include unit tests or cucumber tests to verify that the logic works. Note that these tests are unit tests, not integration tests; they are not designed to ensure that you have configured your nodes properly or that there are no conflicts or other issues when applying these recipes.

Attributes

Attributes are Chef's way of storing configuration data and can be thought of as a large, but disjointed, hash structure. Chef pulls data from various locations and combines that data in a specific order to produce the final hash of attributes. This data is computed when a client requests its run list from the server (such as when you execute `chef-client` on a node). This mechanism allows you to describe data with a higher level of specificity at each step of the process, decreasing in scope going from the cookbook attributes files down to node-specific configuration data.

For example, imagine you are deploying PostgreSQL onto the hosts in your infrastructure. With PostgreSQL, there are a very large number of configuration options that can be tuned, ranging from open ports and number of concurrent connections down to memory used for key caches and other fine-grained configuration options. The cookbook's attributes files should provide enough configurations for PostgreSQL to work without making any modification to a host and without other things being deployed; also, they would most likely contain a pretty vanilla set of configuration values, which at a high level might look like the following:

- Install Version 9.3 from the source code
- Listen on Port 5432 on IP 0.0.0.0/0
- Store data in `/var/lib/postgresql`
- Create and use a `pgsql` user

Attribute data has not only a number of sources that it can be pulled from but also a set of priorities: default, normal, and override (in increasing order). Within each level, data is pulled from the cookbook attributes files and then from the environment, role, and node configuration data stored in the Chef server (in that order). Combined, this provides a comprehensive mechanism to define and customize the behavior of your recipes as they are applied to the nodes.

Now, as you can imagine, this is fine for a number of installations where the server has all of the space allocated on the root mount point, or doesn't have security restrictions about which IP addresses should be listened on, and so on. It would be nice to be able to say that in production, we want to use Version 9.3, but in a test environment, we want to install Version 9.4 in order to perform some tests that we don't want to run in production. We may also want to specify that in production, our hosts are EC2 instances with a customized EBS RAID for our PostgreSQL data and so the data should be stored in /vol/ebs00/postgresql. Using this multilayered approach for configuration data, this is entirely possible.

Attribute files contain Ruby code that stores the configuration data. In this case, to achieve our described default behavior, we could have a file, `attributes/default.rb`, that contains the following text:

```
default['postgresql']['port'] = "5432"
default['postgresql']['listen_address'] = "*"
default['postgresql']['data_dir'] = "/var/lib/postgresql"
default['postgresql']['install_method'] = "source"
default['postgresql']['version'] = "9.3"
```

The hash that this describes will look like the following JSON dictionary:

```
'node' : {
  'postgresql': {
    'port': '5432',
    'listen_address': '*',
    'data_dir': '/var/lib/postgresql',
    'install_method': 'source',
    'version': '9.3'
  }
}
```

Now, as described, we want to override the version in our staging environment to install Version 9.4; this means that in our staging environment, the configuration (exactly how to make this change will be discussed later) will need to have the following information:

```
'node' : {
  'postgresql': {
    'version': '9.4',
  }
}
```

When the Chef client runs on a node in the staging environment, the Chef server knows that the node is in the staging environment and will take the stage configuration above and overlay it on top of the defaults specified in the cookbook. As a result, the final configuration dictionary will look like the following:

```
'node' : {
  'postgresql': {
    'port': '5432',
    'listen_address': '*',
    'data_dir': '/var/lib/postgresql',
    'install_method': 'source',
    'version': '9.4'
  }
}
```

Notice that the version has been changed, but everything else remains the same. In this way, we can build very specific configurations for our hosts that pull information from a variety of places.

It is important to note that because these are interpreted Ruby scripts, their contents can range from simple attribute-setting statements to complex logic used to determine an appropriate set of default attributes. However, it's worth remembering that the more complicated your configuration is, the harder it may be to understand.

Multiple attribute files

Chef loads attribute files in alphabetical order and cookbooks typically contain only one attribute file named `default.rb`. In some cases, it makes sense to separate some of the attributes into separate files, particularly when there are a lot of them. As an example, the community-maintained MySQL cookbook has two attribute files: `server.rb` for the server attributes and `client.rb` with client-specific attributes. Each file contains anywhere between 50 and 150 lines of Ruby code, so it makes sense to keep them separate and focused.

Supporting multiple platforms

There are times when a simple attributes file doesn't make sense, so being able to dynamically define the defaults is very useful. Consider a multiplatform cookbook that needs to know which group the root user is in. The name of the group will vary according to the operating system of the end host. If you are provisioning a FreeBSD or OpenBSD host, then the group for the root will be `wheel`, but on an Ubuntu machine, the group is named `admin`. The attributes file can use plain old Ruby code or optional Chef-provided convenience methods such as `value_for_platform`, which is a glorified but compact switch statement:

```
default[:users]['root'][:group] = value_for_platform(
  "openbsd"    => { "default" => "wheel" },
  "freebsd"    => { "default" => "wheel" },
  "ubuntu"     => { "default" => "admin" },
  "default"     => "root"
)
```

Loading external attributes

Sometimes it is useful to load attributes from another cookbook; if your cookbook is tightly coupled to another cookbook or provides some extended functionality, it may make sense to use them. This can be achieved in the attributes file by using the `include_attribute` method (again, this is a Chef-specific convenience method).

Let's consider that you want to know the port that Apache is configured to use. You could use the `port` key from the apache configuration section, but it is not guaranteed that it exists (it may not have been defined or the recipe that contains it may not have been loaded yet). To address this, the following code would load the settings from `attributes/default.rb` inside of the `apache` cookbook:

```
include_attribute "apache"
default['mywebapp']['port'] = node['apache']['port']
```

If you need to load an attributes file other than `default.rb`, say `client.rb`, inside the `postgresql` cookbook, you can specify it in the following manner:

```
include_attribute "postgresql::client"
```

Make sure that any cookbooks you rely on are listed as a dependency in your cookbook's metadata. Without this, the Chef server will have no way of knowing that your recipes or configuration data depend on that cookbook, and so your configuration may fail as a result of this.

Using attributes

Once you have defined your attributes, they are accessible in our recipes using the **node hash**. Chef will compute the attributes in the order discussed and produce one large hash, which you will have access to.

 Chef uses a special type of hash, called a **Mash**. Mashes are hashes with what is known as indifferent access—string keys and symbol keys are treated as the same, so `node[:key]` is the same as `node["key"]`.

If we had the PostgreSQL attributes and user attributes as specified previously, without any overrides, then the resulting configuration will look like the following:

```
'node' : {
  'postgresql': {
    'port': '5432',
    'listen_address': '*',
    'data_dir': '/var/lib/postgresql',
    'install_method': 'source',
    'version': '9.3'
  },
  'users' : {
    'root' : { 'group' => 'wheel' },
  }
}
```

This data could then be accessed anywhere in our recipes or templates through variables such as `node[:postgresql][:port]` or `node[:users][:root][:group]`. Remember that the final version of the node's configuration data is determined at the time the client makes the request for its configuration. This means that Chef generates a snapshot of the current state of the system, collapsed according to its rules of precedence, for that node and passes it to the host to perform its operations.

Metadata

Each cookbook contains a `metadata.rb` file in the root directory of the cookbook that contains information about the cookbook itself, such as who maintains it, the license, version, contained recipes, supported platforms, and the cookbook's dependencies. The contents of this script are used to generate a JSON file that describes the cookbook, which is used by the Chef server for dependency resolution, searching and importing into run lists.

This is a required file for a cookbook, and here is an example `metadata.rb` file from the PostgreSQL database server, which is slightly modified to fit the following:

```

name          "postgresql"
maintainer    "Opscode, Inc."
maintainer_email "cookbooks@opscode.com"
license       "Apache 2.0"
description   "Installs and configures PostgreSQL"
long_description IO.read(File.join(
  File.dirname(__FILE__), 'README.md'
))
version       "3.3.4"
recipe        "postgresql", "Includes postgresql::client"
recipe        "postgresql::ruby", "Installs Ruby bindings"
recipe        "postgresql::client",
             "Installs client package(s)"
recipe        "postgresql::server", "Installs server packages"
recipe        "postgresql::server_redhat",
             "Installs RedHat server packages"
recipe        "postgresql::server_debian",
             "Installs Debian server packages"

%w{ubuntu debian fedora suse amazon}.each do |os|
  supports os
end

%w{redhat centos scientific oracle}.each do |el|
  supports el, ">= 6.0"
end

depends "apt"
depends "build-essential"
depends "openssl"

```

Because the `metadata.rb` file is a Ruby script, it allows you to use arbitrary Ruby code inside of it. Here, for example, the `long_description` entry is generated programmatically by reading in the contents of the supplied `README.md` file:

```
long_description IO.read(File.join(File.dirname(__FILE__),  
'README.md'))
```

Here, the PostgreSQL cookbook supports multiple platforms, so instead of writing each platform that is supported on a line of its own, you could use a loop similar to the one used in the `metadata.rb` file:

```
%w{ubuntu debian fedora suse amazon}.each do |os|  
  supports os  
end
```

Additionally, if it only supports certain platforms with a minimum version, you could write something similar to the following, which declares support for RedHat-based distributions greater than (or equal to) Version 6.0:

```
%w{redhat centos scientific oracle}.each do |el|  
  supports el, ">= 6.0"  
end
```

In this cookbook, the dependencies are listed line by line but could be represented similarly if you have a large number:

```
depends "apt"  
depends "build-essential"  
depends "openssl"
```

Dependencies could also be rewritten as follows:

```
%w{apt build-essential openssl}.each do |dep|  
  depends dep  
end
```

Obviously, in this case, you aren't saving any room; however, if you had ten or more dependencies, it could make it more compact.

As long as your Ruby code produces something that is a compatible argument or configuration, you can be as clever as you want. Take advantage of your ability to dynamically generate a configuration.

Recipes

Recipes are the core component of getting things done. They are scripts, written in Ruby, that provide the instructions to be executed on end hosts when the Chef client is run. Recipes are placed in the `recipes` directory inside of a cookbook, and each recipe is designed to achieve a specific purpose, such as provisioning accounts, installing and configuring a database server, and custom software deployments.

Recipes combine configuration data with the current state of the host to execute commands that will cause the system to enter a new state. For example, a PostgreSQL database server recipe would have the goal of installing and starting a PostgreSQL server on any host that runs the recipe. Let's look at a few possible starting states and the expected behavior:

- A host without PostgreSQL installed would begin at the state of not having the service; then, it will execute the commands required to install and configure the service
- Hosts with an existing but outdated PostgreSQL service would be upgraded to the latest version of the database server
- Hosts with a current installation of PostgreSQL would have its PostgreSQL installation untouched
- In all cases, the configuration on the disk would be updated to match the configuration stored in the Chef server

To achieve these goals, recipes use a combination of resources, configuration data, and Ruby code to determine what to execute on the end host. Each host-level resource—files, configuration files, packages, users, and so on—is mapped to a Chef resource in a recipe. For example, consider the recipe that we saw earlier in the book that was used to demonstrate that the Chef-solo installation was functional:

```
file "#{ENV['HOME']}/example.txt" do
  action :create
  content "Greetings #{ENV['USER']}!"
end
```

This is a complete recipe; it has one step to create a file, and that is all it does. The file being created on the end host needs a name; here, it will be named `ENV['HOME'] / example.txt`, which is Ruby's way of representing `$HOME/example.txt`. In addition to a name, we are instructing Chef to create the file (we could just as easily instruct Chef to delete the file) and to put the contents `Greetings $USER` into the file, replacing what is in there.

We could extend our recipe to ensure that a specific user existed on the host and create a file with the owner set to the newly created user:

```
user "smith" do
  action :create
  system true
  comment "Agent Smith"
  uid 1000
  gid "agents"
  home "/home/matrix"
  shell "/bin/zsh"
end

file "/tmp/agent.txt" do
  action :create
  content "Hello from smith!"
  owner "smith"
  group "agents"
  mode "0660"
end
```

Each recipe is a script that is run from beginning to end, assuming that nothing causes it to abort. Also, each recipe can access the node's attribute data and leverage resources to compile templates, create directories, install packages, execute commands, download files, and do just about anything that you can do from a shell as an administrator. If you can't accomplish what you need to do using the existing Chef resources, you can either create your own custom resources, or you can execute an arbitrary user-defined shell script.

Resources

Resources are programmatic building blocks in Chef; they are a declarative mechanism to manipulate a resource on a host. Resources deliberately hide the underlying implementation that is left to a provider. It is important to recognize that a resource describes what is being manipulated, not how it is being manipulated; this is by design, as it provides a high level of abstraction for Chef recipes to be as platform-neutral as possible.

For example, Chef has built-in resources that include the following:

- Cron jobs
- Deployments
- File system components (mount points, files and directories, and so on)

- Source code repositories (Git and svn)
- Logs
- PowerShell scripts (Windows targets)
- Shell scripts
- Templates
- Packages
- Users and groups

Resources combined with providers (discussed shortly) are collectively referred to as LWRPs or lightweight resource providers; they make up a large portion of the functionality within a Chef recipe.

Resources are composed of a resource name (package name, file path, service name, and so on), an action, and some attributes that describe that resource.

Using resources

Resources, as we have seen in some examples, take the following form:

```
resource_name <name parameter> <ruby_block>
```

In the preceding code, `resource_name` is the registered name of the resource, such as `file`, `package`, and `user`. The name parameter is a special argument to the resource that gives this resource a unique name. This is often also used by the resource as a default value for an attribute that naturally maps to the name of the resource such as `filename`, `username`, and `package name` (you can see a pattern here); however, you can use an arbitrary name attribute and manually set the attribute. The Ruby block is just a block of code in Ruby; this is how Chef's DSL works. In Chef, each resource expects some specific things inside its configuration block. You will find that many resources have different expectations, but in general, a resource block in a recipe will be of the following form:

```
resource_name "name attribute" do
  attribute "value"
  attribute "value"
end
```

The previous example, which created a new user, was the following:

```
user "smith" do
  action :create
  system true
  comment "Agent Smith"
```

```
    uid 1000
    gid "agents"
    home "/home/matrix"
    shell "/bin/zsh"
end
```

Here, the resource name is `user`, the name attribute is `smith`, and the block of code being passed to the resource has seven attributes: `action`, `system`, `comment`, `uid`, `gid`, `home`, and `shell`. Each of these attributes has a value associated with it; internally, the Ruby code for the resource will store these in some variables to be used when manipulating the specified resource. In this case, constructing a user on the end host through the correct provider will be helpful.

One of these attributes, `action`, is a bit unique; its job is to tell the resource what action to take on the resource. The list of available actions will be different with each resource, but typically, a resource will have actions such as `create`, `delete`, or `update`. Have a look at the documentation for the resource you are working with; the documentation will describe the available actions and what they do separately from the other attributes.

To demonstrate how the name attribute is used as a default value for the `user` resource, the following recipe has the same behavior as the previous one, but has one minor change:

```
user "agent_smith" do
  username "smith"
  action :create
  system true
  comment "Agent Smith"
  uid 1000
  gid "agents"
  home "/home/matrix"
  shell "/bin/zsh"
end
```

Here, you can see that an additional attribute, `username`, has been added to the resource block with the value that was previously in the name attribute. Additionally, the name attribute has been changed to `"agent_smith"`. If you were to execute this recipe or the previous example, it would have the same effect: to create a local system user, `smith`, with the UID, GID, home, and other attributes specified.

Overriding a default behavior

In addition to properties, resources also have a default **action**. More often than not, the default action is `create`, but again, you will want to consult the documentation for the resource you are working with to make sure that you know what the default behavior is for a resource. You don't want to accidentally destroy something you thought you were creating!

A concrete example might be installing the `tcpdump` package on your system. To install the default version with no customization, you could use a resource description such as the following:

```
package "tcpdump"
```

This works because the default action of the package resource is to perform an installation. If you look at the source code for the package resource, you will see the following at the beginning of the constructor:

```
def initialize(name, run_context=nil)
  super
  @action = :install
  @allowed_actions.push(:install, :upgrade,
                        :remove, :purge, :reconfig)
  @candidate_version = nil
  @options = nil
  @package_name = name
```

This tells us that the default action, if unspecified, is to install the package and to use the `name` attribute as the package name. So, the previous simple one-line resource is the same as writing out the following block:

```
package "tcpdump" do
  action :install
  package_name "tcpdump"
end
```

Here, however, `package_name` will default to the `name` attribute, so we do not need to provide it if the resource name is the same as the package you wish to install. Additionally, if you wanted to be more verbose with your resource description and install a specific version of the `tcpdump` package, you could rewrite the package resource to look something like the following:

```
package "tcpdump" do
  action :install
  version "X.Y.Z"
end
```

If you read the documentation for the package resource or examine the full constructor for the package class, you will see that it has a number of other attributes as well as what they do and where their default values come from. All the resources follow this form; they are simply Ruby classes that have an expected structure, which they inherit from the base resource class.

Templates

Chef dynamically configures software and hosts, and a large part of configuring UNIX-based systems and software involves configuration files. Chef provides a straightforward mechanism to generate configuration files that make it easy to combine configuration data with template files to produce the final configuration files on hosts. These templates are stored in the `templates` directory inside of a cookbook and use the ERB template language, which is a popular and easy-to-use Ruby-based template language.

Why use templates?

Without templates, your mechanism to generate configuration files would probably look something like this:

```
File.open(local_filename, 'w') do |f|
  f.write("<VirtualHost *:#{node['app']['port']}>")
  ...
  f.write("</VirtualHost>")
end
```

This should be avoided for a number of reasons. First, writing configuration data this way would most likely make your recipe very cluttered and lengthy. Secondly, and more importantly, it violates Chef's declarative nature. By design, Chef provides you with the tools to describe what the recipe is doing and not how it is doing it, which makes reading and writing recipes much easier. Simpler recipes make for simpler configuration, and simpler configuration scales better because it is easier to comprehend. Handrolling a configuration file is the opposite approach; it very specifically dictates how to generate the file data. Consider the following code:

```
config_file = "#{node['postgresql']['dir']}/postgresql.conf"
pgconfig = node[:postgresql]
File.open(config_file 'w') do |f|
  f.write("port = #{pgconfig[:port]}")
  f.write("data_dir = #{pgconfig[:data_dir]}")
```

```
f.write("listen_address = #{pgconfig[:listen_address]}")  
end  
File.chown(100,100,config_file)  
File.chmod(0600, config_file)
```

This code generates a PostgreSQL configuration file from the attribute hash, one line at a time. This is not only time-consuming and hard to read but potentially very error-prone. You can imagine, even if you have not previously configured any PostgreSQL servers, just how many `f.write(...)` statements could be involved in generating a full `postgresql.conf` file by hand. Contrast that with the following block that leverages the built-in `template` resource:

```
template "#{$node['postgresql']['dir']}/postgresql.conf" do  
  source "postgresql.conf.erb"  
  owner "postgres"  
  group "postgres"  
  mode 0600  
end
```

The preceding block could be combined with a template file that contains the following content:

```
<% node['postgresql'].sort.each do |key, value| %>  
<% next if value.nil? -%>  
<%= key %> = <%=  
  case value  
  when String  
    "'#{value}'"  
  when TrueClass  
    'on'  
  when FalseClass  
    'off'  
  else  
    value  
  end  
%>  
<% end %>
```

If we take our template and then apply the following attribute data as we had shown previously, then we would have generated the exact same configuration file:

```
'node' : {  
  'postgresql': {  
    'port': '5432',  
    'listen_address': '*' ,
```

```
'data_dir': '/var/lib/postgresql',
'install_method': 'source',
'version': '9.3'
},
'users' : {
  'root' : { 'group' => 'wheel' },
}
}
```

Only now we can use a template that is highly flexible. Our template uses the key-value combinations stored in the configuration hash to dynamically generate the `postgresql.conf` file without being changed every time a new configuration option is added to PostgreSQL.

Chef uses ERB, a template language, that is provided by the core Ruby library. ERB is widely available and requires no extra dependencies; it supports Ruby inside of templates as well as some ERB-specific template markup.

A quick ERB primer

As ERB is very well documented and widely used, this portion of the chapter serves only as a quick reference to some of the most commonly used ERB mechanisms. For more information, see the official Ruby documentation at <http://ruby-doc.org/stdlib-2.1.1/libdoc/erb/rdoc/ERB.html>.

Executing Ruby

To execute some arbitrary Ruby code, you can use the `<% %>` container. The `<%` part indicates the beginning of the Ruby code, and `%>` indicates the end of the block. The block can span multiple lines or just one single line. Examples of this are as follows:

ERB code	Output
<code><%</code>	1
<code>[1,2,3].each do index </code>	2
<code> puts index</code>	
<code>end</code>	3
<code>%></code>	
<code><% users.collect{ u </code>	Bob Smith
<code> puts u.full_name } %></code>	Sally Flamingo

You can mix Ruby and non-Ruby code (useful to repeat blocks of non-Ruby text) as follows:

```
<% [1,2,3].each do |value| %>
  Non-ruby text...
<% end %>
```

This would yield the following:

```
Non-ruby text...
Non-ruby text...
Non-ruby text...
```

Variable replacement

ERB has a syntax to replace a section of the template with the results of some Ruby code rather than relying on print statements inside your Ruby. That container is similar to the last one, with the addition of the **equal sign** inside the opening tag. It looks like `<%= %>`. Any valid Ruby code is acceptable inside this block, and the result of this code is put into the template in place of the block. Examples of this are as follows:

```
<%= @somevariable %>
<%= hash[:key] + otherhash[:other_key] %>
<%= array.join(", ") %>
```

This can be combined with the previous example to produce complex output:

```
<% [1,2,3].each do |value| %>
  The value currently is <%= value %>
<% end %>
```

This would yield the following:

```
The value currently is 1
The value currently is 2
The value currently is 3
```

Using just these basic features of ERB, combined with Chef's configuration data, you can model just about any configuration file you can imagine.

The template resource

Chef provides a template resource to generate files via templates. There are three key attributes of the template resource, which are as follows:

- `path`: This specifies where to put the generated file
- `source`: This tells the resource which template file to use
- `variables`: This specifies what data to populate the template with

The path attribute uses the name attribute as its default value and populates the template specified by a source file with the data passed to it through the variables attribute. Templates are contained inside of the templates directory, which is placed inside of a cookbook; if a source is not specified, it will be expected that a file exists inside the directory with the same name as the path, which is only rooted in the templates directory with a .erb extension. Here is a simple template resource example:

```
template "/etc/motd" do
  variables :users => ["Bart", "Homer", "Marge"]
end
```

This resource will expect that a file exists in the template's search path (more on how that is determined in a bit) as etc/motd.erb, and it then exposes an array of three strings as a users variable and writes the results out as /etc/motd on the host. The corresponding MOTD template could look like the following:

```
Welcome to crabapple.mydomain.com! Our newest users are:
```

```
<% @users.each do |user| %>
  * <%= user %>
<% end %>
```

The template variables

There are two primary sources of data for a template: data passed explicitly through the resource block attributes and node configuration data. Explicit variables are user defined in the recipe and may be used to override some settings or pass in configuration that is dynamically generated inside the recipe. The node configuration data is computed by Chef at runtime and represents a snapshot of the current configuration that will be applied to the node.

Passing variables to a template

Sometimes you will need to pass data to a template from inside your recipe instead of relying on the global node attributes. Perhaps you have some logic that computes some variable data, but it doesn't belong in the node hash; Chef supports doing just this in the template resource. The data passed explicitly is available to the ERB template as an instance variable, prefixed in Ruby with the @ symbol. For example, consider the following recipe snippet:

```
config_hash = {
  :food => "asparagus",
```

```
:color => "blue"
}

template "/etc/myapp.conf" do
  source "myapp.conf.erb"
  owner "root"
  mode "0664"
  variables(
    :install_path => "/opt/#{hostname}/myapp",
    :config => config_hash
  )
end
```

The `:install_path` and `:config` keys are accessible in the template as instance variables with the same name. They will be prefixed by the `@` character and could be used in a template similar to the following:

```
database_path = "<%= @install_path %>/db"
storage_path = "<%= @install_path %>/storage"
<% config.each do |key,value| %>
<%= key %> = "<%= value %>"
<% end %>
```

Here, the template expects a specific key, `install_path`, to determine where to store the database; the key-value hash specified by `config` is then used to generate some arbitrary configuration settings in the template.

Accessing computed configurations

In addition to data passed via the `variables` attribute, a template can access a node's computed configuration data through the `node` local variable. This is accessed as a Ruby hash, which will be structured similarly to a dictionary or a hash in any other language. In our previous PostgreSQL attribute's data example, the following values were defined:

```
default['postgresql']['port'] = "5432"
default['postgresql']['listen_address'] = "*"
```

Even if no other configuration data supersedes these configuration values, there will be a `postgresql` key in the node's configuration data that contains the key's port and `listen_address`. Using this information, we can write a recipe that uses a template resource and a matching template like the following:

```
template "/etc/postgresql/postgresql.conf" do
  source "postgresql.conf.erb"
  owner "pgsql"
  mode "0600"
end

listen_addresses = '<%= node[:postgresql][:listen_address] %'
port = '<%= node[:postgresql][:port] %>
```

When the default attributes data is combined with the example template, the resulting `/etc/postgresql/postgresql.conf` file will have the following content:

```
listen_addresses = '*'
post = 5432
```

As previously discussed, the computed attributes hash for a given node comes from a variety of sources. Those sources include attributes files in the cookbook, role, environment, and node-level configuration values stored in Chef, each with its own level of precedence.

Searching for templates

As you have likely noticed, Chef attempts to allow you as much, or as little, specificity as you want when defining your configuration, and templates are no different. Just as the final node configuration is computed from a variety of locations, the `templates` directory has a specific search order. This allows the author of the cookbook to provide a set of default templates as well as support platform and host-specific overrides.

The `default` template directory should be used to provide default versions of the templates. Any platform- or host-specific directories are placed alongside it and will be used when appropriate. The search order for a template is as follows:

- Hostname
- Distribution version
- Distribution
- Default location

As an example, let's consider a scenario in which we applied a recipe with the `postgresql.conf.erb` template resource to a node, `db1.production.mycorp.com`, which is running Debian 6.0. Chef will then look for the following files inside of the `templates` directory:

- `host-db1.production.mycorp.com/postgresql.conf.erb`
- `debian-6.0/postgresql.conf.erb`
- `debian/postgresql.conf.erb`
- `default/postgresql.conf.erb`

The search is performed in that order with the first match being the chosen template, applying the highest level of specificity before the lowest (as is the pattern with other Chef mechanisms, including configuration data).

This differentiation of configuration files by host, platform, and even version is very useful. It allows you to provide a sane set of defaults while supporting host- or system-specific quirks simultaneously.

Definitions

Sometimes, you find that you are creating something repeatedly and, similar to a configuration template, you need a template to generate objects of a given type. Some examples of this might be Apache virtual hosts, a specific type of application, or anything else that is repeated a lot. This is where definitions come in, and they are stored in the `definitions` directory inside of a cookbook.

Definitions are loaded and available as named resources just as other resources such as packages, files, and so on are; the only difference is that there is no provider. You can think of them as resources and providers all in one. Subsequently, they are much more rigid and limited in scope than a normal resource would be. Here is an example definition to install Python libraries using `pip` and a `requirements.txt` file:

```
define :pip_requirements , :action => :run do
  name = params[:name]
  requirements_file = params[:requirements_file]
  pip = params[:pip]
  user = params[:user]
  group = params[:group]

  if params[:action] == :run
    script "pip_install_#{name}" do
      interpreter "bash"
      user "#{user}"
    end
  end
end
```

```
group "#{group}"
code <~-EOH
#{pip} install -r #{requirements_file}
EOH
only_if { File.exists?("#{requirements_file}") and File.
exists?("#{pip}") }
end
end
end
```

Here, we are declaring a new type of definition, a `pip_requirements` object. This looks and behaves similarly to a resource, except that it is much simpler (and less flexible) than a resource. It has some attributes, which are loaded via the special `params` argument, and contains a little bit of logic wrapped around a `script` resource. Let's take a look at how it would be used and then see how it works:

```
pip_requirements "my_requirements" do
  pip "#{virtualenv}/bin/pip"
  user node[:app] [:user]
  group node[:app] [:group]
  requirements_file "#{node[:app][:src_dir]}/requirements.txt"
end
```

Here you see what looks like a resource, but is in fact a definition. As mentioned earlier, these look very similar because they behave alike. However, you must have likely noticed that the definition of `pip_requirements` itself did not have any sort of abstraction; there is no pluggable provider, no validation, it doesn't subclass the `Resource` class, among other differences. Definitions provide you with a mechanism to declare reusable chunks of code that your recipes would otherwise duplicate so that your recipe can again describe the *what*, not the *how*.

The previous example tells us that we have a `pip_requirements` object and that we want to pass some parameters to it, namely, the path to `pip`, the user and group to run `pip` as, and the `requirements.txt` file to load. These are brought into the definition through the `params` argument and can be accessed as any other variable data. In this case, the definition says to run `bash` as the specified user and group and that the script should run the equivalent of the following:

```
pip install -r /path/to/requirements.txt
```

This will happen only if `pip` and the `/path/to/requirements.txt` file exist (as indicated by the `only_if` guard). By creating such a definition, it can be reused any time you need to install Python modules from a specific `requirements.txt` file on your host.

Recipes

Recipes are where all the magic happens with Chef; they are the secret sauce, the man behind the mask. They are the workhorses of configuring hosts with Chef. Recipes are scripts written in Ruby using Chef's DSL that contain the instructions to be executed on end hosts when the Chef client is run. Every time the client is executed on the end host, a few things happen:

1. The end host makes a request to the Chef server saying, "I need to do some work".
2. The Chef server looks at the requesting host's identity and determines:
 - Which recipes need to be run and in what order (the run list)
 - The computed configuration data for that host
3. This information is passed back to the end host along with the necessary artifacts it needs (recipes, templates, and so on).
4. The client then combines the configuration data with the recipes and begins to execute its run list.

Developing recipes

As a developer, you will be placing your recipes inside the `recipes` directory of your cookbook. Each recipe is designed to perform a specific action or set of actions to achieve a goal such as provisioning accounts, installing and configuring a database server, custom software deployments, or just about any other action that you could perform on a server.

A key concept when developing recipes is that they should be idempotent. For those unfamiliar with the term, an idempotent operation is an operation that can be applied multiple times and have the same outcome. Consider the following recipe:

```
user "smith" do
  action :create
  system true
  comment "Agent Smith"
  uid 1000
  gid "agents"
  home "/home/matrix"
  shell "/bin/zsh"
end
```

One would expect, from looking at this recipe, that Chef should be able to execute it once, five times, or one thousand times, and it would have the same effect as the initial application of the recipe. There would not be five or one thousand users on the host with the login name `smith`; there would be only one single user with the login name `smith`. Also, in all the runs, it would be constructed with the same UID 1000, the same group name, and so on.

Similarly, given a particular state of the system and assuming nothing has changed in between runs, subsequent client executions should produce the same, consistent ending state. In short, the Chef client should be able to run two times in a row, and if the configuration has not been updated, the system should look exactly the same after the second run as after the first run.

Recipes use provided configuration data along with the current state of the host to determine the flow and actions taken by the script. The execution of a recipe will take the system from its initial state, Sinitial, to its new state, Sfinal. Well-written recipes should be idempotent such that if they're executed immediately afterwards any number of times with no configuration or stat changes, then the system should go from Sfinal to the same Sfinal with no new changes to the system. This allows you to keep your systems in a consistent state at all times, assuming that nothing goes wrong during the execution of those operations; if something does go wrong, you should be able to revert to a previously known good state.

Writing recipes

As you have already seen, cookbooks provide a way to combine relevant pieces of configuration data such as attributes, templates, resources, providers, and definitions in one place. The only reason these components exist is to support our recipes. Recipes combine resources in a certain order to produce the desired outcome; much in the same way a chef would combine ingredients according to his or her recipe to produce some delicious food. By putting all of these resources together, we can build our own recipes that range from very simple single-step recipes to multistep, multiplatform recipes.

Starting out small

A very basic recipe, as we have discussed before, might only leverage one or two resources. One of the simplest conceivable recipes is the one we used earlier to verify that our Chef-solo installation was working properly:

```
file "#{ENV['HOME']}/example.txt" do
  action :create
  content "Greetings #{ENV['USER']} !"
end
```

Here again, we are combining a single resource, the `file` resource, specifying that we want to create the file named `$HOME/example.txt`, and store the string "Greetings `$USER`" in that file. `$USER` and `$HOME` will be replaced by the environment variables, most likely the login name of the user that is executing `chef-client` and their home directory respectively (unless the environment variables have been tampered with).

Following our goal of idempotence, executing this recipe multiple times in a row will have the same effect as only running it once.

Installing a simple service

Now that we've covered a simple recipe, let's take a look at one that configures the Redis engine and uses *supervisord* to run the daemon. This recipe doesn't install Redis; instead, it defines how to configure the system to start and manage the service. It does not have any advanced logic, but merely constructs some required directories, builds a configuration file from a template, and then uses the `supervisor_service` resource to configure the daemon to run and be monitored, as shown in the following code:

```
redis_user      = node[:redis][:user]
redis_group     = node[:redis][:group]
environment_hash = {"home" => "#{node[:redis][:home]}"

# Create the log dir and data dir
[node[:redis][:datadir], node[:redis][:log_dir]].each do |dir|
  directory dir do
    owner      redis_user
    group      redis_group
    mode       "0750"
    recursive   true
  end
end

# Generate the template from redis.conf.erb
template "#{$node[:redis][:config_path]}" do
  source "redis.conf.erb"
  owner redis_user
  group redis_group
  variables({:data_dir => "#{$node[:redis][:data_dir]}"})
  mode 0644
end

# Convenience variables for readability
stdout_log = "#{$node[:redis][:log_dir]}/redis-stdout.log"
```

```
stderr_log = "#{node[:redis][:log_dir]}/redis-stderr.log"
redis_bin  = "#{node[:redis][:install_path]}/bin/redis-server"
redis_conf = "#{node[:redis][:config_path]}"

# Tell supervisor to enable this service, autostart it, run it as
# the redis user, and to invoke:
#   /path/to/redis-server /path/to/redis.conf
supervisor_service "redis_service" do
  action          :enable
  autostart       true
  user            "#{redis_user}"
  command         "#{redis_bin} #{redis_conf}"
  stdout_logfile  "#{stdout_log}"
  stderr_logfile  "#{stderr_log}"
  directory       "#{node[:redis][:install_path]}"
  environment     environment_hash
end
```

You will notice that in order to keep the configuration consistent, we reuse a lot of attributes. For example, the beginning of the recipe uses `node[:redis][:datadir]` and `node[:redis][:log_dir]` to ensure that the directories exist by making use of a directory resource inside of a loop; then, these are used later on to define the supervisor configuration variables (where to write logs) and the template for the config file (where to store the data). In all, this recipe is composed of four resources: two directories in the loop, one template, and one supervisor service. By the end of this run, it will have ensured the critical directories exist, Redis is configured, and a supervisor configuration file is generated (as well as poked supervisord to reload the new configuration and start the service). Again, running this multiple times, assuming no configuration changes in between runs will put the system in the exact same state. Redis will be configured according to the host properties, and supervisor will run the service.

Getting more advanced

Let's move up and take a look at a slightly more complicated, yet fairly simple, recipe from the `git` cookbook that installs the `git` client on the host. The cookbook is multiplatform, so let's talk about what it will be doing before it shows you the source. This recipe will be performing the following actions:

1. Determine which platform the end host is running on (by inspecting the `node[:platform]` attribute).
2. If the host is running a Debian-based distribution, it will use the package resource to install `git-core`.

3. If the host is a RHEL distribution, it will perform the following:
 1. Include the EPEL repository by pulling in the `epel` recipe from the `yum` cookbook if the platform version is 5.
 2. Use the package resource to install `git` (as that is the RHEL package name).
4. If the host is Windows, it will install `git` via the `windows_package` resource and instruct it to download the file located at `node[:git][:url]` (which in turn pulls from the default attributes or overridden configuration), validate that the checksum matches the one specified by `node[:git][:checksum]`, and then install it; however, this is only if the EXE is not already installed.
5. If the host is running OS X, it will leverage the `dmg_package` resource to install a `.pkg` file from a `.dmg` image. Here, the download URL, volume name, package file, checksum, and app name are all attributes that need to be provided.
6. Finally, if none of the conditions are met, it falls back to the package resource to install the `git` package in the hope that it will work.

Here is the code for this recipe:

```
case node[:platform]
when "debian", "ubuntu"
  package "git-core"
when "centos", "redhat", "scientific", "fedora"
  case node[:platform_version].to_i
  when 5
    include_recipe "yum::epel"
  end
  package "git"
when "windows"
  windows_package "git" do
    source node[:git][:url]
    checksum node[:git][:checksum]
    action :install
    not_if { File.exists? 'C:\Program Files (x86)\Git\bin\git.exe' }
  end
when "mac_os_x"
  dmg_package "Git OSX-Installer" do
    app node[:git][:osx_dmg][:app_name]
    package_id node[:git][:osx_dmg][:package_id]
    volumes_dir node[:git][:osx_dmg][:volumes_dir]
    source node[:git][:osx_dmg][:url]
    checksum node[:git][:osx_dmg][:checksum]
```

```
  type "pkg"
    action :install
  end
else
  package "git"
end
```

One thing we haven't seen yet is the use of the `not_if` qualifier. This is exactly what it looks like; if the block supplied to `not_if` returns a `true` value, the resource will not be processed. This is very useful to ensure that you don't clobber important files or repeat expensive operations such as recompiling a software package.

Summary

This chapter introduced you to the critical components of a cookbook that are used to write recipes. It also showed you some example recipes to get you started; there are a number of advanced actions that can be accomplished in your recipes, such as searching the Chef server for data, loading data from data bags, or using encrypted data. Additionally, you can add more components to your cookbooks such as custom resources and providers, tests, and arbitrary Ruby libraries. All of these will be discussed in detail in later chapters, but first let's take a look at writing some complete cookbooks. We'll then learn how to test them before we move on to looking at some cookbooks for common system administration tasks, and then we'll progress on to advanced topics.

5

Testing Your Recipes

So far, you have seen how to model your infrastructure, provision hosts in the cloud, and what goes into a cookbook. One important aspect of developing cookbooks is writing tests so that your recipes do not degrade over time or have bugs introduced into them in the future. This chapter introduces you to the following concepts:

- Understanding test methodologies
- How RSpec structures your tests
- Using ChefSpec to test recipes
- Running your tests
- Writing tests that cover multiple platforms

These techniques will prove to be very useful to write robust, maintainable cookbooks that you can use to confidently manage your infrastructure. Tests enable you to perform the following:

- Identify mistakes in your recipe logic
- Test your recipes against multiple platforms locally
- Develop recipes faster with local test execution before running them on a host
- Catch the changes in dependencies that will otherwise break your infrastructure before they get deployed
- Write tests for bugs to prevent them from happening again in the future (regression)

Testing recipes

There are a number of ways to test your recipes. One approach is to simply follow the process of developing your recipes, uploading them to your Chef server, and deploying them to a host; repeat this until you are satisfied. This has the benefit of executing your recipes on real instances, but the drawback is that it is slow, particularly if you are testing on multiple platforms, and requires that you maintain a fleet of hosts. If your cookbook run times are reasonably short and you have a small number of platforms to support them, then this might be a viable option. There is a better option to test your recipes, and it is called ChefSpec. For those who have used RSpec, a Ruby testing library, these examples will be a natural extension of RSpec. If you have never used RSpec, the beginning of this chapter will introduce you to RSpec's testing language and mechanisms.

RSpec

RSpec is a framework to test Ruby code that allows you to use a domain-specific language to provide tests, much in the same way Chef provides a domain-specific language to manipulate an infrastructure. Instead of using a DSL to manage systems, RSpec's DSL provides a number of components to express the expectations of code and simulate the execution of portions of the system (also known as mocking).

The following examples in RSpec should give you a high-level idea of what RSpec can do:

```
# simple expectation
it 'should add 2 and 2 together' do
  x = 2 + 2
  expect(x).to eq 4
end

# Ensure any instance of Object receives a call to 'foo'
# and return a pre-defined value (mocking)
it 'verifies that an instance receives :foo' do
  expect_any_instance_of(Object)
    .to receive(:foo).and_return(:return_value)

  o = Object.new
  expect(o.foo).to eq(:return_value)
end

# Deep expectations (i.e client makes an HTTP call somewhere
# inside it, make sure it happens as expected)
it 'should make an authorized HTTP GET call' do
```

```
expect_any_instance_of(Net::HTTP::Get)
  .to receive(:basic_auth)
  @client.make_http_call
end
```

RSpec and ChefSpec

As with most testing libraries, RSpec enables you to construct a set of expectations, build objects and interact with them, and verify that the expectations have been met. For example, one expects that when a user logs in to the system, a database record is created, tracking their login history. However, to keep tests running quickly, the application should not make an actual database call; in place of the actual database call, a mock method should be used. Here, our mock method will catch the message in the database in order to verify that it was going to be sent; then, it will return an expected result so that the code does not know the database is not really there.



Mock methods are methods that are used to replace one call with another; you can think of them as stunt doubles. For example, rather than making your code actually connect to the database, you might want to write a method that acts as though it has successfully connected to the database and fetched the expected data.

This can be extended to model Chef's ability to handle multiple platforms and environments very nicely; code should be verified to behave as expected on multiple platforms without having to execute recipes on those platforms. This means that you can test the expectations about Red Hat recipes from an OS X development machine or Windows recipes from an Ubuntu desktop, without needing to have hosts around to deploy to for testing purposes. Additionally, the development cycle time is greatly reduced as tests can be executed much faster with expectations than when they are performing some work on an end host.

You may be asking yourself, "How does this replace testing on an actual host?" The answer is that it may not, and so you should use integration testing to validate that the recipes work when deployed to real hosts. What it does allow you to do is validate your expectations of what resources are being executed, which attributes are being used, and that the logical flow of your recipes are behaving properly before you push your code to your hosts. This forms a tighter development cycle for rapid development of features while providing a longer, more formal loop to ensure that the code behaves correctly in the wild.

If you are new to testing software, and in particular, testing Ruby code, this is a brief introduction to some of the concepts that we will cover. Testing can happen at many different levels of the software life cycle:

- Single-module level (called unit tests)
- Multi-module level (known as functional tests)
- System-level testing (also referred to as integration testing)

Testing basics

In the **test-driven-development** (TDD) philosophy, tests are written and executed early and often, typically, even before code is written. This guarantees that your code conforms to your expectations from the beginning and does not regress to a previous state of non-conformity. This chapter will not dive into the TDD philosophy and continuous testing, but it will provide you with enough knowledge to begin testing the recipes that you write and feel confident that they will do the correct thing when deployed into your production environment.

Comparing RSpec with other testing libraries

RSpec is designed to provide a more expressive testing language. This means that the syntax of an RSpec test (also referred to as a spec test or spec) is designed to create a language that feels more like a natural language, such as English. For example, using RSpec, one could write the following:

```
expect(factorial(4)).to eq 24
```

If you read the preceding code, it will come out like *expect factorial of 4 to equal 24*. Compare this to a similar JUnit test (for Java):

```
assertEquals(24, factorial(4));
```

If you read the preceding code, it would sound more like *assert that the following are equal, 24 and factorial of 4*. While this is readable by most programmers, it does not feel as natural as the one we saw earlier.

RSpec also provides `context` and `describe` blocks that allow you to group related examples and shared expectations between examples in the group to help improve organization and readability. For example, consider the following spec test:

```
describe Array do
  it "should be empty when created" do
    Array.new.should == []
  end
end
```

Compare the preceding test to a similar NUnit (.NET testing framework) test:

```
namespace MyTest {
    using System.Collections;
    using NUnit.Framework;
    [TestFixture]
    public class ArrayTest {
        [Test]
        public void NewArray() {
            ArrayList list = new ArrayList();
            Assert.AreEqual(0, list.size());
        }
    }
}
```

Clearly, the spec test is much more concise and easier to read, which is a goal of RSpec.

Using ChefSpec

ChefSpec brings the expressiveness of RSpec to Chef cookbooks and recipes by providing Chef-specific primitives and mechanisms on top of RSpec's simple testing language. For example, ChefSpec allows you to say things like:

```
it 'creates a file' do
  expect(chef_run).to create_file('/tmp/myfile.txt')
end
```

Here, `chef_run` is an instance of a fully planned Chef client execution on a designated end host, as we will see later. Also, in this case, it is expected that it will create a file, `/tmp/myfile.txt`, and the test will fail if the simulated run does not create such a file.

Getting started with ChefSpec

In order to get started with ChefSpec, create a new cookbook directory (here it is `$HOME/cookbooks/mycookbook`) along with a `recipes` and `spec` directory:

```
mkdir -p ~/cookbooks/mycookbook
mkdir -p ~/cookbooks/mycookbook/recipes
mkdir -p ~/cookbooks/mycookbook/spec
```

Now you will need a simple `metadata.rb` file inside your cookbook (here, this will be `~/cookbooks/mycookbook/metadata.rb`):

```
maintainer      "Your name here"
maintainer_email "you@domain.com"
license         "Apache"
description     "Simple cookbook"
long_description "Super simple cookbook"
version         "1.0"
supports        "debian"
```

Once we have this, we now have the bare bones of a cookbook that we can begin to add recipes and tests to.

Installing ChefSpec

In order to get started with ChefSpec, you will need to install a gem that contains the ChefSpec libraries and all the supporting components. Not surprisingly, that gem is named `chefspec` and can be installed simply by running the following:

```
gem install chefspec
```

However, because Ruby gems often have a number of dependencies, the Ruby community has built a tool called **Bundler** to manage gem versions that need to be installed. Similar to how RVM provides interpreter-level version management and a way to keep your gems organized, Bundler provides gem-level version management. We will use Bundler for two reasons. In this case, we want to limit the number of differences between the versions of software you will be installing and the versions the author has installed to ensure that things are as similar as possible; secondly, this extends well to releasing production software—limiting the number of variables is critical to consistent and reliable behavior.

Locking your dependencies in Ruby

Bundler uses a file, specifically named `Gemfile`, to describe the gems that your project is dependent upon. This file is placed in the root of your project, and its contents inform Bundler which gems you are using, what versions to use, and where to find gems so that it can install them as needed.

For example, here is the Gemfile that is being used to describe the gem versions that are used when writing these examples:

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

gem 'chef',      '11.10.0'
gem 'chefspec',   '3.2.0'
gem 'colorize',    '0.6.0'
```

Using this will ensure that the gems you install locally match the ones that are used when writing these examples. This should limit the differences between your local testing environments if you run these examples on your workstation.

In order to use a Gemfile, you will need to have Bundler installed. If you are using RVM, Bundler should be installed with every gemset you create; if not, you will need to install it on your own via the following code:

```
gem install bundler
```

Once Bundler is installed and a Gemfile that contains the previous lines is placed in the root directory of your cookbook, you can execute `bundle install` from inside your cookbook's directory:

```
user@host:~/cookbooks/mycookbook $> bundle install
```

Bundler will parse the Gemfile in order to download and install the versions of the gems that are defined inside. Here, Bundler will install `chefspec`, `chef`, and `colorize` along with any dependencies those gems require that you do not already have installed.

Creating a simple recipe and a matching ChefSpec test

Once these dependencies are installed, you will want to create a spec test inside your cookbook and a matching recipe. In keeping with the TDD philosophy, we will first create a file, `default_spec.rb`, in the `spec` directory. The name of the spec file should match the name of the recipe file, only with the addition of `_spec` at the end. If you have a recipe file named `default.rb` (which most cookbooks will), the matching spec test would be contained in a file named `default_spec.rb`. Let's take a look at a very simple recipe and a matching ChefSpec test.

Writing a ChefSpec test

The test, shown as follows, will verify that our recipe will create a new file, /tmp/myfile.txt:

```
require 'chefspec'

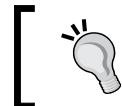
describe 'mycookbook::default' do
  let(:chef_run) {
    ChefSpec::Runner.new.converge(described_recipe)
  }

  it 'creates a file' do
    expect(chef_run).to create_file('/tmp/myfile.txt')
  end
end
```

Here, RSpec uses a `describe` block similar to the way Chef uses a resource block (again, blocks are identified by the `do ... end` syntax or code contained inside curly braces) to describe a resource, in this case, the `default` recipe inside of `mycookbook`. The described resource has a number of examples, and each example is described by an `it` block such as the following, which comes from the previous spec test:

```
it 'creates a file' do
  expect(chef_run).to create_file('/tmp/myfile.txt')
end
```

The string given to the `it` block provides the example with a human-readable description of what the example is testing; in this case, we are expecting that the recipe creates a file. When our recipes are run through ChefSpec, the resources described are not actually created or modified. Instead, a model of what would happen is built as the recipes are executed. This means that ChefSpec can validate that an expected action would have occurred if the recipe were to be executed on an end host during a real client run.



It's important to note that each example block resets expectations before it is executed, so any expectations defined inside of a given test will not fall through to other tests.



Because most of the tests will involve simulating a Chef client run, we want to run the simulation every time. There are two options: execute the code in every example or use a shared resource that all the tests can take advantage of. In the first case, the test will look something like the following:

```
it 'creates a file' do
  chef_run = ChefSpec::Runner.new.converge(described_recipe)
  expect(chef_run).to create_file('/tmp/myfile.txt')
end
```

The primary problem with this approach is remembering that every test will have to have the resource running at the beginning of the test. This translates to a large amount of duplicated code, and if the client needs to be configured differently, then the code needs to be changed for all the tests. To solve this problem, RSpec provides access to a shared resource through a built-in method, `let`. Using `let` allows a test to define a shared resource that is cached for each example and reset as needed for the following examples. This resource is then accessible inside of each block as a local variable, and RSpec takes care of knowing when to initialize it as needed.

Our example test uses a `let` block to define the `chef_run` resource, which is described as a new ChefSpec runner for the described recipe, as shown in the following code:

```
let(:chef_run) {
  ChefSpec::Runner.new.converge(described_recipe)
}
```

Here, `described_recipe` is a ChefSpec shortcut for the name of the recipe provided in the `describe` block. Again, this is a DRY (don't repeat yourself) mechanism that allows us to rename the recipe and then only have to change the name of the description rather than hunt through the code. These techniques make tests better able to adapt to changes in names and resources, which reduces code rot as time goes by.

Building your recipe

The recipe, as defined here, is a very simple recipe whose only job is to create a simple file, `/tmp/myfile.txt`, on the end host:

```
file "/tmp/myfile.txt" do
  owner "root"
  group "root"
  mode "0755"
  action :create
end
```

Put this recipe into the `recipes/default.rb` file of your cookbook so that you have the following file layout:

```
mycookbook/
  |- recipes/
  |   |- default.rb
  |- spec/
  |   |- default_spec.rb
```

Executing tests

In order to run the tests, we use the `rspec` application. This is a Ruby script that comes with the RSpec gem, which will run the test scripts as spec tests using the RSpec language. It will also use the ChefSpec extensions because in our spec test, we have included them via the line `require 'chefspec'` at the top of our `default_spec.rb` file. Here, `rspec` is executed through Bundler to ensure that the desired gem versions, as specified in our `Gemfile`, are used at runtime without having to explicitly load them. This is done using the `bundle exec` command:

```
bundle exec rspec spec/default_spec.rb
```

This will run RSpec using Bundler and process the `default_spec.rb` file. As it runs, you will see the results of your tests, a `.` (period) for tests that pass, and an `F` for any tests that fail. Initially, the output from `rspec` will look like this:

```
Finished in 0.17367 seconds
1 example, 0 failures
```

RSpec says that it completed the execution in 0.17 seconds and that you had one example with zero failures. However, the results would be quite different if we have a failed test; RSpec will tell us which test failed and why.

Understanding failures

RSpec is very good at telling you what went wrong with your tests; it doesn't do you any good to have failing tests if it's impossible to determine what went wrong. When an expectation in your test is not met, RSpec will tell you which expectation was unmet, what the expected value was, and what value was seen.

In order to see what happens when a test fails, modify your recipe to ensure that the test fails. Look in your recipe for the following file resource:

```
file "/tmp/myfile.txt" do
```

Replace the file resource with a different filename, such as `myfile2.txt`, instead of `myfile.txt`, like the following example:

```
file "/tmp/myfile2.txt" do
```

Next, rerun your spec tests; you will see that the test is now failing because the simulated Chef client execution did something that was unexpected by our spec test. An example of this new execution would look like the following:

```
[user@host] $ bundle exec rspec spec/default_spec.rb
```

```
F
```

Failures:

```
1) my_cookbook::default creates a file
   Failure/Error: expect(chef_run).to create_file('/tmp/myfile.txt')
     expected "file[/tmp/myfile.txt]" with action :create to be in Chef
run. Other file resources:

   file[/tmp/myfile2.txt]

# ./spec/default_spec.rb:9:in `block (2 levels) in <top (required)>'
```

```
Finished in 0.18071 seconds
```

```
1 example, 1 failure
```

Notice that instead of a dot, the test results in an F; this is because the test is now failing. As you can see from the previous output, RSpec is telling us the following:

- The `creates a file` example in the '`my_cookbook::default`' test suite failed
- Our example failed in the ninth line of `default_spec.rb` (as indicated by the line that contained `./spec/default_spec.rb:9`)
- The file resource `/tmp/myfile.txt` was expected to be operated on with the `:create` action
- The recipe interacted with a file resource `/tmp/myfile2.txt` instead of `/tmp/myfile.txt`

RSpec will continue to execute all the tests in the files specified on the command line, printing out their status as to whether they passed or failed. If your tests are well written and run in isolation, then they will have no effect on one another; it should be safe to execute all of them even if some fail so that you can see what is no longer working.

Expanding your tests

ChefSpec provides a comprehensive suite of tools to test your recipes; you can stub and mock resources (replace real behavior with artificial behavior, such as network or database connections), simulate different platforms, and more. Let's take a look at some more complex examples to see what other things we can do with ChefSpec.

Multiple examples in a spec test

Spec tests do not need to contain only one example; they can contain as many as you need. In order to organize them, you can group them together by what they describe and some shared context. In RSpec, context blocks contain examples that are relevant to the recipe or script being tested. Think of them as self-contained test suites within a larger test suite; they can have their own resources as well as setup and tear-down logic that are specific to the tests that are run in that context.

As an example, let's look at part of the spec test suite from the `render_file` example inside of ChefSpec itself. Consider this portion of the recipe:

```
file '/tmp/file' do
  content 'This is content!'
end

cookbook_file '/tmp/cookbook_file' do
  source 'cookbook_file'
end

template '/tmp/template' do
  source 'template.erb'
end
```

The recipe being shown has three resources: a template, a `cookbook_file`, and an ordinary file resource. A sample of the matching spec test (tests removed for formatting and ease of reading) contains an outer `describe` block, which tells us that we are executing tests for the `render_file`:`:default` recipe and three separate context blocks. Each context describes a different portion of the recipe that is being tested and the expectations of that particular type of resource. Together, they are all part of the default recipe, but they behave very differently in what content they render as well as where and how they store files on the system.

In this example, the `file` context contains tests that pertain to the expected results of the `file` resource, the `cookbook_file` context is concerned with the `cookbook_file` resource, and so on:

```
describe 'render_file::default' do
  let(:chef_run) {
    ChefSpec::Runner.new.converge(described_recipe)
  }

  context 'file' do
    it 'renders the file' do
      expect(chef_run).to render_file('/tmp/file')
      expect(chef_run).to_not render_file('/tmp/not_file')
    end
  end

  context 'cookbook_file' do
    it 'renders the file' do
      expect(chef_run).to render_file('/tmp/cookbook_file')
      expect(chef_run).to_not
        render_file('/tmp/not_cookbook_file')
    end
  end

  context 'template' do
    it 'renders the file' do
      expect(chef_run).to render_file('/tmp/template')
      expect(chef_run).to_not render_file('/tmp/not_template')
    end
  end
end
```

Contexts can be used to group together a set of examples that are related, not just ones that are specific to a particular resource. Consider the following example:

```
describe 'package::install' do
  context 'when installing on Windows 2012' do
  end
  context 'when installing on Debian' do
  end
  context 'when installing on FreeBSD' do
  end
end
```

In the previous example, our spec test contained tests that are grouped together by the platform being executed on. Inside of each context, the Chef run will be constructed with a `platform` argument instead so that the expectations being tested will be considered against a run of the recipe on the platform in question rather than the host's operating system. This is incredibly useful, as we will see in the next section on testing for multiple platforms.

Testing for multiple platforms

One of the more non-trivial uses of ChefSpec is to simulate executing recipes on multiple platforms. This is useful for developers who are building recipes that need to support more than one operating system. Software packages such as PostgreSQL, MySQL, Java, PHP, Apache, and countless other applications can be installed on many different platforms. Because each platform varies in its installation mechanism, user creation, and other core features, being able to test recipes against all the supported platforms is incredibly useful.

Let's look at a hypothetical example to develop a recipe to install MySQL on Windows 2012 and some things we might want to validate during such a run:

```
context 'when run on Windows 2012' do
  let(:chef_run) do
    # construct a 'runner' (simulate chef-client) running
    # on a Windows 2012 host
    runner = ChefSpec::ChefRunner.new(
      'platform' => 'windows',
      'version' => '2012'
    )
    # set a configuration variable
    runner.node.set['mysql']['install_path'] = 'C:\\\\temp'
    runner.node.set['mysql']['service_user'] = 'SysAppUser'
    runner.converge('mysql::server')
  end

  it 'should include the correct Windows server recipe' do
    chef_run.should include_recipe 'mysql::server_windows'
  end

  it 'should create an INI file in the right directory' do
    ini_file = "C:\\\\temp\\\\mysql\\\\mysql.ini"
    expect(chef_run).to create_template ini_file
  end
end
```

By constructing the `runner` with the `platform` and `version` options, the test will exercise running the `mysql::server` recipe and pretend as though it were running on a Windows 2012 host. This allows us to set up expectations about the templates that will be created, recipes that are being executed, and more on that particular platform.

Presuming that the `mysql::server` recipe was able to delegate to the OS-specific recipe on a given platform, we could write another test:

```
context 'when run on Debian' do
  let(:chef_run) do
    runner = ChefSpec::ChefRunner.new(
      'platform' => 'debian'
    )
    runner.node.set['mysql']['install_path'] = '/usr/local'
    runner.node.set['mysql']['service_user'] = 'mysql'
    runner.converge('mysql::server')
  end

  it 'should include the correct Linux server recipe' do
    chef_run.should include_recipe 'mysql::server_linux'
  end

  it 'should create an INI file in the right directory' do
    ini_file = "/usr/local/mysql/mysql.ini"
    expect(chef_run).to create_template ini_file
  end

  it 'should install the Debian MySQL package' do
    expect(chef_run).to install_package('mysql-server')
  end
end
```

In this way, we can write our tests to validate the expected behavior on platforms that we may not have direct access to in order to ensure that they will be performing the expected actions for a collection of platforms.

Summary

RSpec with ChefSpec extensions provides us with incredibly powerful tools to test our cookbooks and recipes. You have seen how to develop basic ChefSpec tests for your recipes, organize your spec tests inside of your cookbook, execute and analyze the output of your spec tests, and simulate the execution of your recipes across multiple platforms.

In future chapters, we will learn some more advanced testing mechanisms such as mocking and stubbing external services such as search and data bags. Adding testing to your development cycle allows you to feel confident in the correctness of your recipes, which is a critical step towards automating the management of your infrastructure.

Now, let's take a look at how we will build a cookbook to complement a web application so that we can see the full cycle of developing an application and deploying it using Chef.

6

From Development to Deployment

This chapter covers end-to-end software deployment of a Python-based web application. It will also introduce you to some common cookbooks and how to put them all together to create a fully automated deployment mechanism.

We will walk through the following topics:

- Configuring your local settings to work with AWS
- Modeling a simple `web.py` application with Chef
- Installing the cookbooks you need
- Provisioning EC2 instances for web and DB servers
- Defining your roles
- Adding users to hosts
- Installing the required software
- Configuring an application using Chef
- Deploying the application

Describing the setup

From a high level, here is what needs to happen in order to take an application that we have developed from a desktop to deployment. In order to deploy your application, you will provision two hosts, `web` and `db` (each with one user account) and `webapp`, whose home directory will be in `/home/webapp`. The source code will be hosted on GitHub and deployed using Git onto the web server. We will create a database, provision an account to access that database, and then configure and deploy a `web.py` application in a virtual Python environment that will be started and monitored by supervisord. This is a fairly common pattern for modern web applications, regardless of the framework and language being used. The demonstration application used in this chapter consists of only a handful of files making it easy to deploy and understand, but this will give you the concepts and tools to expand this example for use with future applications you might develop or need to deploy.

Deploying software with Chef

There are numerous benefits to deploying your software using Chef; the primary benefit is automation – the chef-client can be run periodically, and it can execute fully-automated deployments whenever changes are made to the source code repository. Additionally, Chef stores all your configuration data, so you can avoid storing sensitive secrets and hard-coding URLs or other dynamic data in your configuration. For example, if you have an application with a database pool, and you add a new database host to your pool, Chef can use a simple search to populate the list of hosts to include in the connection pool so that it is always up to date with your infrastructure.

However, deploying software with Chef does require some coordination between your application and Chef. You will need to maintain recipes required for deploying your application, and you will also want to use Chef as the authoritative source for your configuration data, which involves writing configuration templates. By using Chef to manage your deployments, you can also generate any configuration data needed to run your software based on your infrastructure configuration; in our case, a simple `config.py` file for your `web.py` application. This method can also be used to manage the `database.yml` file (and any other YAML files) for Rails applications, the `server.yml` file for a Dropwizard application, or any other configuration files needed to run your service.



YAML is a simple markup language to store configuration data. It is popular with modern developers because it is easy to parse and is very expressive, similar to JSON.

Configuring your local environment

By now, you should have access to a Chef server of some sort; here, we will be using the hosted Chef service, but the work (aside from configuring your `knife.rb` file) will remain the same across self-managed and hosted Chef instances. In order to follow along with the examples, you will need to configure your workstation with an appropriate `knife.rb` file and certificates. These files can be downloaded from the hosted Chef console and modified as needed.

Additionally, you will need to have installed `knife` through the `chef` gem and have the `knife-ec2` gem installed in order to interact with EC2. If you prefer to use a different provider, then you can refer to the previous chapter on how to provision cloud hosts with this provider when you get to the provisioning step.

In our example, using hosted Chef and EC2, our `knife.rb` file will contain content similar to the following code:

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

knife[:aws_access_key_id] = "YOUR AWS ACCESS KEY"
knife[:aws_secret_access_key] = "YOUR AWS SECRET KEY"
knife[:region] = "AWS REGION"
```

Again, the base files can be downloaded from the hosted Chef console, or if you are using a self-managed Chef installation, this can be found on your Chef server.

Modeling a simple Python application

Here we will consider a `web.py` application that has two primary components: a web server and a database server. We will provision one host for each role, bootstrap them, and deploy the software onto our new hosts.

Our application stack will consist of the following:

- `web.py` as our web framework
- PostgreSQL for data storage
- EC2 for virtual hosts

We want to define two primary roles that represent our web server and our database server. In addition, we will construct a **baseline** role for all our servers that will supply any common data we need such as user accounts, SSH keys, network configuration data, shells, common utilities, libraries, and so on.

We will need to find or write cookbooks for the following components we will use:

- Python
- supervisord
- PostgreSQL
- User accounts
- Our custom web application

Managing the cookbooks

The cookbooks that we will be using are all available at the following URL: https://github.com/johnewart/simplewebpy_app. Because a number of cookbooks used in this example are under active development, the ones required for the examples have been frozen (as of writing of this book) to ensure compatibility with the examples; it is better to have them slightly out of date than broken in this case.

However, when you write your own cookbooks and deploy your own software beyond this example, you will find that there are a large number of cookbooks that can be found through the Chef community site—<http://supermarket.getchef.com/>—or by searching the Web for cookbooks; many of these will be hosted on GitHub, BitBucket, or similar source code-hosting sites.

Downloading cookbooks

Here in the following code, we will simply download the cookbook collection as a whole:

```
http://github.com/johnewart/chef_essential_files
```

To install the collection, we can do the following from the `chef_essential_files/cookbooks` directory:

```
knife cookbook install -o . *
```

This will install all of the cookbooks that are provided. The provided cookbooks are all that is required for the examples in this chapter to be successful. Let's take a look at our custom cookbook, the `pythonwebapp` cookbook, as all of the others are off-the-shelf cookbooks that are designed to provide some general support functionalities.

Looking at the database recipe

We will do a few things here, so let's look at our database recipe. In order for our web application to be useful, it needs a database to connect to. Typically, this involves installing the database server software, constructing a database, and granting access to that database by a specified user (or users). Our application is no different, so we will leverage the `database` cookbook in order to accomplish this.

First, in our recipe, we need to include the PostgreSQL-specific resources from the `database` cookbook, which we will do using the following code:

```
include_recipe "database::postgresql"
```

You will need to know what database you will be creating and to which user you will be granting access to along with the password that will be used to identify them:

```
dbname = node[:webapp][:dbname]
dbuser = node[:webapp][:dbuser]
dbpass = node[:webapp][:dbpass]
```

In order to create a database and user as well as grant access, you will need to establish a connection to the database server with a user that has permission to do so. You will see that this user has also been granted access in your role's `pg_hba` settings so that PostgreSQL knows to allow the `postgres` user to connect to the database locally, as shown in the following code:

```
postgresql_connection_info = {
  :host      => 'localhost',
  :port      => node['postgresql']['config']['port'],
  :username  => 'postgres',
  :password  => node['postgresql']['password']['postgres']
}
```

Using this connection information, you can construct a database and a user (if they don't already exist), and then grant that user full access to our new database:

```
# Construct an actual database on the server
postgresql_database 'webapp' do
  connection postgresql_connection_info
  action      :create
end

# Create a database user resource using our connection
postgresql_database_user dbuser do
  connection postgresql_connection_info
  password    dbpass
  action      :create
end

# Grant all privileges on all tables in 'webapp'
postgresql_database_user dbuser do
  connection postgresql_connection_info
  database_name dbname
  privileges   [:all]
  action       :grant
end
```

This high-level language allows us to easily manipulate the database without the need to know any database-specific SQL or commands. If you want to convert your application to use MySQL, for example, provisioning a new MySQL database would largely be as easy as converting the word **postgresql** to **mysql** in our recipe, and the database-specific adapter in the database cookbook will be responsible for the implementation details.

Looking at your application deployment cookbook

Once our database has been provisioned, you can look at how you can install our web application. In the `pythonwebapp::webapp` recipe, you have all the information you need to do this. The way that you define a recipe for deploying an application will vary wildly among applications, as each application is unique. However, this particular example was designed to be a representative of most web applications (reasonably) and should present you with a good starting point to understand the basics of deploying a web application with Chef.

Modern web applications typically follow the same pattern: provision a user, install an interpreter, or other engine (such as Python, Ruby, and Java), create directories if needed, check out the source code, run any data migrations (if needed) to update your database, and then make sure that your service is up and running; this is no different. The more complicated your application, the more infrastructure you may need to model, such as job queue engines, asynchronous workers, and other libraries.

If you look at the web application cookbook located at `cookbooks/pythonwebapp`, you will see that it has the following: two recipes, a template, and a PIP-requirement definition inside it. The recipes included are for the web application itself and to manage the creation of the PostgreSQL database and user on the database host.

Most of the interesting work is in the application recipe, `cookbooks/pythonwebapp/recipes/webapp.rb`; so, let's start by taking a look at that. All applications are going to have a slightly different deployment logic, but modern web applications usually follow a pattern that looks like the following:

- Install any system-wide packages required
- Construct the directories needed for the software
- If this is Python or Ruby, possibly install a new `virtualenv` tool or RVM gemset
- Install the libraries needed to run the application
- Check out the application's source code
- Build and configure the application as needed
- Create or update the database schema
- Stop the web application services
- Start the web server or process manager that monitors the application

This example application is no different, so let's look at the steps needed to deploy this `web.py` application. First, declare any application configuration data needed with the following command:

```
app_root = node[:webapp][:install_path]
python_interpreter = node[:webapp][:python]
virtualenv = "#{app_root}/python"
virtual_python = "#{virtualenv}/bin/python"
src_dir = "#{app_root}/src/"
# Grab the first database host
```

```
dbhost = search(:node, "role:postgresql_server") [0] ['ipaddress']

environment_hash = {
  "HOME" => "#{app_root}",
  "PYTHONPATH" => "#{app_root}/src"
}
```

In this snippet, we used the computed attributes to tell our recipe where to install the application; in this case, the default is `/opt/webapp` but this can be overridden for flexibility. Additionally, we set the path to the Python interpreter we want to use for our Python virtualenv. However, you can just as easily specify a Ruby or Java path if your application used one of those languages. There is a path to the source code and a database host address. This path is determined by searching the Chef data for all nodes with the `postgresql_server` role, taking the first one, and using its IP. This allows us to replace the database server and not have to worry about updating our configuration data, which we'll see in a bit.

Preparing the directories

In order to deploy our application, and for it to run, we need to have a location to put our data. In this application, we have defined a need for: a configuration directory, a log directory, and a place to view the source code. In our recipe, we will create these directories and set proper ownership to our deployment user and group. Note that you do not need to create the application root directory if it already exists, and you do not need to set special ownership or permissions on the root directory. Because we are leveraging the recursive property of the `directory` resource, the root application directory will be implicitly created; however, we are constructing it here for the sake of completeness.

It is critical that our directories have the correct ownership and permissions; without this, the application will be unable to interact with those directories to store log data or read-and-write any configuration data. The following code constructs these directories for us and changes the ownership and permissions:

```
directory "#{app_root}" do
  owner node[:webapp][:user]
  group node[:webapp][:group]
  mode "0755"
  action :create
  recursive true
end

# Create directories
["src", "logs", "conf"].each do |child_dir|
```

```
directory "#{app_root}/#{child_dir}" do
  owner node[:webapp][:user]
  group node[:webapp][:group]
  mode "0755"
  action :create
  recursive true
end
end
```

One thing to note here is that we are using a loop to construct our directories. This is a good way to manage multiple resources of the same type that have the same set of configuration parameters. Here we are saying that we have three subdirectories, `src`, `log`, and `conf`. Also, we want to construct a `directory` resource inside of our application's root directory for each subdirectory with proper ownership and permissions. The recursive flag is similar to the `-p` option on `mkdir`, which tells it to create any directories that are missing in between the root and the directory being created.

Constructing your Python virtual environment

This may be new to non-Python developers but should be fairly straightforward. A virtual environment operates in a similar way to RVM or rbenv for Ruby, or a self-contained JAR file for Java. In that, it isolates the Python interpreter and installed libraries to a specific location on the system. In our case, we will use the following code to achieve this:

```
python_virtualenv "#{virtualenv}" do
  owner node[:webapp][:user]
  group node[:webapp][:group]
  action :create
  interpreter "#{python_interpreter}"
end
```

This `python_virtualenv` resource comes from the `python` cookbook and will construct a virtual environment in the location named by the resource (in our case, the directory stored in `virtualenv`, which as we saw previously, is defined as though in a `python` directory inside our application root) using the specified interpreter and ownership properties.

A virtual environment will be created, which contains a minimal installation of the Python interpreter as well as any Python libraries that are installed into the virtual environment. Think of it as your application's own installation of Python that is unaffected by, and subsequently does not affect, any other Python installation on the system.

This is a very useful technique to install and manage Python applications, and the same concept can be extended to the Rails application using any similar technology from the Ruby world such as RVM or rbenv, as mentioned earlier.

Checking the source code

One interesting thing in this recipe, which has been included for future reference, is the usage of a cookbook, `ssh_known_hosts`, that grabs a host's SSH key and adds it to the system's list of known SSH keys. This is extremely useful to deploy software via GitHub or BitBucket, where you are using SSH to pull down the source code, especially as their host keys might change:

```
# Need to install SSH key for github.com
ssh_known_hosts_entry 'github.com'
```

Note that it is also somewhat insecure as you are blindly accepting the host's fingerprint—if you are concerned about security, you can provide the known fingerprints manually using the `:key` attribute. Supplying the fingerprint is done through the following code:

```
ssh_known_hosts_entry 'github.com' do
  key  'github.com ssh-rsa AAAAB3NzaC1yc....'
end
```

If there are a large number of host fingerprints that you need to manage, or if they change frequently, you can use a data bag to store them. If you are interested, look at the README for the `ssh_known_hosts` cookbook for more examples.

Once the SSH keys are registered, you can now clone the source from a `git+ssh` URL such as GitHub's authenticated SSH endpoint.

In this example, we are using a publicly available HTTPS source code repository; if you were to replace this with your own SSH-enabled repository, you would need to change the `repository` attribute and also make sure to store your deployment key on the endhost:

```
# Clone our application source
git "#{src_dir}" do
  repository "https://github.com/johnewart/simplewebpy_app.git"
  action :sync
  branch 'master'
  user node[:webapp] [:user]
end
```

By using the `git` resource, the repository will be cloned into the designated source directory on the endhost. Here, we will also be pulling data from the `master` branch and performing this action as our `webapp` user.

Installing any extra dependencies

There are two ways to model dependencies for your application: in your cookbook and recipe, or through an external mechanism such as Bundler, pip, or other dependency resolution, and the downloading tool depending on the language of your choice. As with everything, there are both inherent drawbacks and benefits to each of these methods.

Managing dependencies in Chef

By modeling your dependencies in Chef, you have a consistent model that you can look to in a centralized location. This means that your application needs a new Ruby gem, or a Python library that someone must update a cookbook or Chef configuration with that information in order for the deployment to be successful. This can limit your ability to continuously deploy based solely on the contents of a source code repository. In effect, this requires you to model the following in Chef:

- Dependent libraries
- Library versions
- Possibly, the dependencies of any declared dependencies (which can spiral quickly)

However, modeling it this way does ensure that Chef has an accurate picture of all the information associated with your application. This solution does offer some other benefits:

- Dependencies are precisely modeled in Chef and can be queried by other tools
- Any system-specific packages that are needed for your interpreted libraries are going to be modeled by Chef anyway, so it's all in one place (examples can include native XML or database libraries)
- Developers can't arbitrarily change dependencies and accidentally break deployments because the underlying libraries have not been installed in production

Let's look at some things to think about when using tools external to Chef for this task.

Managing dependencies elsewhere

Using an external tool such as Bundler or pip has some advantages, including flexibility and ease of use by developers who may not be involved in infrastructure configuration. It also introduces the possibility of misconfigured dependencies and underlying libraries. The primary advantage of this mechanism is that it provides a simpler dependency management model for developers—simply add a requirement to the `Gemfile`, `requirements.txt`, or other metadata file, and Chef will automatically install them during the next run. You also now have to look in two different places to determine what is being installed on endhosts. This also means that you are now configuring dependencies in multiple places, increasing the possibility of making a wrong configuration change in one place.

It's important to take away that there is not always only one tool for the job, and depending on how your organization or team operates, you may choose to mix and match how you model the application-level dependencies. For the sake of demonstrating them both to you, the application cookbook models the dependencies in the recipe as well as through a `requirements.txt` file using pip. Additionally, you may find that initially your team uses one way and then moves to another as your requirements stop changing so frequently, or you are able to combine them to your advantage.

Using Python's requirements file

Our `webapp` cookbook has a custom `pip_requirements` definition that provides an easy way to install any requirements stored inside a `requirements.txt` file into a specified virtual environment using the copy of pip provided by that virtual environment. In the following code, you will see how we can achieve this:

```
pip_requirements "webapp" do
  action :run
  pip "#{virtualenv}/bin/pip"
  user node[:webapp][:user]
  group node[:webapp][:group]
  requirements_file "#{src_dir}/requirements.txt"
end
```

In this example, we are telling pip to run as our application's user and group and to install the dependencies in our `requirements.txt` into the virtual environment specified by `virtualenv`. Again, a similar resource can be created (if one does not already exist) to execute Bundler for Ruby, CPAN for Perl, or PEAR to manage PHP dependencies.

Configuring your application

Now that you have prepared your system for your application, you need to configure it. In order for the application to talk to our database, you must provide the required database connection information that we have stored in Chef. Here, we will use a template that is stored in `templates/default/config.py.erb`, and inject it with our database configuration. The resource for this looks like the following code:

```
template "#{src_dir}/config.py" do
  source "config.py.erb"
  user node[:webapp][:user]
  group node[:webapp][:group]
  mode "0600"
  variables({
    :dbname => node[:webapp][:dbname],
    :dbuser => node[:webapp][:dbuser],
    :dbpass => node[:webapp][:dbpass],
    :dbhost => dbhost
  })
end
```

Here, we load our database information onto our template and store it in our application's install directory (where we checked out the source for simplicity), and set some sane file permissions. Were this a Rails application, we can use a similar template to generate `database.yml` and matching `settings.yml`, or if it were a Dropwizard application, a `service.yml` file, a PHP INI file, or any other type of configuration data that were needed. In our case, we are simply populating the following Python code so that we have a database connection object:

```
import web
db_params = {
  'dbn': 'postgres',
  'db': '<%= @dbname %>',
  'user': '<%= @dbuser %>',
  'pw': '<%= @dbpass %>',
  'host' : '<%= @dbhost %>'
}
DB = web.database(**db_params)
cache = False
```

The previous example uses the `web.py` database module to construct a new database connection using the hash, which can then be imported and used in the other portions of the application. Again, this is a good starting example for our `web.py` application that can be used as a model for whatever framework or application server you are using in your systems.

Keeping your application running

All applications need to be started and kept running in some manner. If you are using Rails with `mod_passenger`, then the Apache daemon will be the primary entry point for your application, and this software will need to be installed and configured. In our example, we will be using the `supervisord` service from <http://supervisord.org>, which is written in Python and serves as a very configurable, lightweight, and reliable process manager. You can configure an entry in the `supervisord` system configuration using a `supervisor_service` resource that is provided by the `supervisor` cookbook installed earlier:

```
supervisor_service "webapp" do
  action          [:enable,:restart]
  autostart      true
  user           node[:webapp][:user]
  command        "#{virtual_python} #{src_dir}/server.py"
  #{{node[:webapp][:port]}}
  stdout_logfile "#{{app_root}}/logs/webapp-stdout.log"
  stdout_logfile_backups 5
  stdout_logfile_maxbytes "50MB"
  stdout_capture_maxbytes "0"
  stderr_logfile    "#{{app_root}}/logs/webapp-stderr.log"
  stderr_logfile_backups 5
  stderr_logfile_maxbytes "50MB"
  stderr_capture_maxbytes "0"
  directory       src_dir
  environment     environment_hash
end
```

The previous example will generate a configuration file for `supervisord` with the settings specified in our resource block. Unless you change the location, the configuration files will be located in `/etc/supervisor.d/[service_name].conf`. In our case, the service is named `webapp`, and its configuration file will be `/etc/supervisor.d/webapp.conf`.

Here, we are telling `supervisord` that we want to enable our service and then restart it (which will start it if it's not currently running), where we want to log the process's output, how we want to rotate those log files, where to start our process, what environment variables to use, and most importantly what command to execute.

Now that we've looked at our recipes, let's go ahead and set up our roles, provision some systems, and deploy our application!

Defining roles

Here we will construct our three roles, one each for our base server configuration, database server, and web server. Each role will have a set of recipes to run, with the base server providing the user accounts, SSH keys, and other common components, and then the others providing configuration data for PostgreSQL and nginx, respectively.

Creating the base server role

The key bits that are of interest in our base server role are the run list and the configuration data that specify which group to populate users for. If you look at the file `roles/base_server.json`, you will see that we have defined one group of users to pull from our data bags:

```
"override_attributes": {  
    "shell_users": {  
        "group": "webapp"  
    }  
}
```

And then, the recipe we want to use that will populate the users on the host is in the run list:

```
"run_list": [  
    "users::shell_users"  
],
```

In order to load the role into Chef, you can issue a `knife role from file` command:

```
knife role from file base_server.json  
Updated Role base _server!
```

You can verify that the role was created with a simple `role list` command:

```
[user]% knife role list  
base_server
```

Creating the database server role

Let's take a look at some portions of our database server role, as defined in `roles/postgresql_server.json`. This file contains the description of our PostgreSQL server as modeled earlier in the chapter. What is of interest in this file is our `override_attributes` section; these are settings we want to use in place of the default values provided by our `postgresql` cookbook. As mentioned before, you will want to look at the documentation and the `default_attributes.rb` file to find out what attributes you can set for a given cookbook and its recipes.

The PostgreSQL recipes use a `postgresql` configuration section that contains a `config` section for server-specific configuration and properties, and `pg_hba` for the authentication data. Looking at the `postgresql` section, we can see that we want to install Version 9.3, and we want it to listen on all addresses (`0.0.0.0`) on port 5432:

```
"version": "9.3",
"config": {
  "listen_addresses": "0.0.0.0",
  "port": "5432"
},
```

In addition, the `pg_hba` section contains an array of JSON objects that describes which users have access to the service, by what mechanism they are able to authenticate themselves (MD5, trusted, local ident service, and so on), and from where they can connect. This is coded into our configuration, but the recipes can be extended to use data bags to determine this information as well. It is too long to include all of it here, but if you look at the `pg_hba` data, you will see that there are three entries: one for the user `webuser` to connect from anywhere using an MD5 password, and two for local users to access the default template database, and for the `postgres` user itself to modify the `webapp` database.

In addition to the configuration data, there is a run list – this tells Chef what recipes this role will include. You can see from this example that we will be installing the PostgreSQL server and then provisioning our `webapp`-specific user and database (found in `cookbooks/pythonwebapp/recipes/database.rb`):

```
"run_list": [
  "postgresql::server",
  "pythonwebapp::database"
],
```

In order to use this, we want to load our JSON file into Chef to define our database server role:

```
knife role from file postgresql_server.json
Updated Role postgresql_server!
```

You can check that the role was created with a `role list` command:

```
[user]% knife role list
base_server
postgresql_server
```

Creating the web server role

Our web application role is located in the `roles/web_server.json` file and contains the required information for our web server. If you take a look at the JSON file, you will see that the run list contains four entries:

```
"run_list": [
  "python",
  "python::virtualenv",
  "supervisor",
  "pythonwebapp::webapp"
],
```

Because our application relies on Python, we want to install the required version of Python on our hosts as well as build a Python `virtualenv` tool for our application. In addition, we will be using `supervisord` as the process monitor that is responsible for ensuring that our web service starts and stays running. We also need to install our web application once we have met our prerequisites.

Similar to how we loaded the PostgreSQL role from our JSON file, we can repeat the process for our web server role:

```
knife role from file web_server.json
Updated Role web_server!
```

Again, you can check that the role was created with a simple `role list` command:

```
[user]% knife role list
base_server
postgresql_server
web_server
```

Adding users

We will need a recipe to manage our users; here, we will use the `users` cookbook. We will create one user, `webuser`, which will be the account that is used for deployment and user connectivity. We will define our user in a JSON file similar to what we did in the previous chapter; place the following in a `users/webuser.json` file:

```
{  
  "id" : "webuser",  
  "uid" : "1000",  
  "gid" : 1000,  
  "shell" : "/bin/bash",  
  "comment" : "Webapp deployment user",  
  "groups" : ["webusers"]  
}
```

Then, you can load this user using the `from file` command:

```
knife data bag from file users users
```

Ensure that your hosts have our new users by editing the `base_server` role and adding our `webusers` group so that any users in that group will be provisioned on all our servers that incorporate the `base_server` role:

```
{  
  "shell_users": {  
    "group": "webusers",  
  }  
}
```

Provisioning EC2 instances

Here, we will be provisioning instances in `us-west-1`, but depending on where you have your AWS instances set up, you will need to change your `knife.rb` configuration to specify the region of your choice.

In order for them to communicate securely, we will construct a security group so that all traffic between them is permitted. This is outside the scope of this book, but it would be something to make sure you configure for production systems, as you probably do not want the public on the Internet to have direct access to your database server.

Here, we will assume that you have your AWS credentials and other critical components configured, as we covered in previous chapters.

To provision our database server, we will use the following command:

```
knife ec2 server create -d ubuntu14.04 -I ami-ee4f77ab -f m1.small -Z us-west-1a -S jewartec2 -N db00 --ssh-user ubuntu
```

And to provision the web server, we will use the following command:

```
knife ec2 server create -d ubuntu14.04 -I ami-ee4f77ab -f m1.small -Z us-west-1a -S jewartec2 -N web00 --ssh-user ubuntu
```

Once your instances are up and running, you can now move on to configuring them with the roles and configuration data required!

Configuring the database host

In order to apply the PostgreSQL role to our database host, we need to make sure it's in the run list. We can accomplish this with the following command:

```
knife node run_list add db00 "role[base_server]"  
knife node run_list add db00 "role[postgresql_server]"
```

After ensuring that your node has the `base_server` and `postgresql_server` roles added to the run list, you can run `chef-client` on the newly created host:

```
[jewart]% knife ssh 'name:db00' -x ubuntu 'sudo chef-client'
```

Once this is complete, assuming that everything went well, your new EC2 instance will have:

- PostgreSQL 9.3 server installed and running
- A new database (the name of which is defined from your configuration)
- A database user that is granted permission to connect
- Correct `pg_hba.conf` and `postgresql.conf` files for our service

Now that we have our database server configured, which can be verified by logging onto the server and ensuring that the service is running, let's take a look at setting up our web application.

Configuring the web server

In order for the web server to deploy the web app, we need to add the required roles to the web server, as we did with the database server:

```
knife node run_list add web00 "role[base_server]"  
knife node run_list add web00 "role[web_server]"
```

Now, we can execute `chef-client` on the web host (again making sure to use `sudo` so that it has permission to do its work):

```
[jewart]% knife ssh 'name:web00' -x ubuntu 'sudo chef-client'
```

At this point, our web server will be in the following state:

- The following required packages will be installed:
 - Python 2.7 and development libraries
 - The PostgreSQL client development libraries
 - Git
- The directories our application needs to run are created
- A `virtualenv` tool, which is based on the system Python 2.7 is created
- Our application has been checked out from GitHub
- A configuration file in `/opt/webapp/src/config.py` is created by Chef
- `Supervisord` is configured to run our application and starts the `server.py` daemon

Now, you should be able to visit your newly installed web application at the following URL:

`http://your-new-ec2-instance-hostname:8080`

If you don't see your application, make sure that each of the preceding steps was successful.

Deploying your software

Deploying software should be treated just like your infrastructure; repeated deployments of the same commit and the same configuration should yield a consistent state of your environment. In this example, we will be able to deploy new updates to our web application simply by updating any nodes that use the `web_server` role. The combination of our recipe and our configuration data with the source code hosted in our GitHub repository will ensure that the most up-to-date configuration and source are placed on our host.

Manually deploying updates

Future deployments only require pushing changes to the master branch and then running `chef-client` on any web servers that are in the fleet. This can be accomplished on a single host (`web00`) using the following command:

```
knife ssh 'name:web00' -x ubuntu 'sudo chef-client'
```

This will tell `knife` that we want to SSH to the host whose name is `web00` as the `ubuntu` user (because that's the default EC2 user with `sudo` access) and execute `chef-client` as `root` via `sudo`. This will work well if we only have one host; however, as your capacity increases and you have multiple hosts, you will likely want to execute this on a group of hosts in the future. This can be accomplished using the search capability of Chef that allows you to expand a list of nodes that match a set of criteria. Here, we will want to build a list of all the hosts that have our `web_server` role associated with them. The following command will accomplish this:

```
knife ssh 'role:web_server' -x ubuntu 'sudo chef-client'
```

This will use Chef's search to find all nodes with the `web_server` role and then SSH to them sequentially, the same as before but only across multiple hosts instead of just one.

Automating deployment

The web application recipe is designed so that it syncs the source with the upstream GitHub repository. By doing this, we can execute the recipe multiple times, and any time there are updates, they will be pulled down onto the local host. If we wanted to, the process of deployments could be automated in the following fashion:

- Active development of the application happens in a separate, development branch
- Code is tested thoroughly and then merged into a master (or whatever branch is being deployed onto hosts) when it is stable and ready for production
- Hosts are configured to run `chef-client` on a fixed interval using a tool such as `cron` and will automatically update themselves

The possible issues with this are that bad code gets automatically deployed to endhosts and so on. However, with enough integration testing and a high enough confidence level, our code should be safe to deploy to production if it is in the master branch. Through a combination of tags and proper source management, rollbacks could be as simple as reverting the deploy branch to a known-good tag and they would happen as soon as the next `chef-client` execution or forced using `knife` as outlined previously.

Summary

A major attraction to Chef and infrastructure automation is the ability to deploy software and provision systems quickly and consistently. Using the cookbooks and examples outlined in this chapter, you should be able to model your application and its components, gather cookbooks required to deploy needed software, build cookbooks to configure and deploy custom software, and extend the examples to provide more functionalities or enhance your infrastructure.

Now that you have seen how to take an application from development to deployment, let's take a look at some more advanced examples of cookbook development, including writing custom providers and resources, working with secure data, searching Chef, and other ways of enhancing our recipes and cookbooks.

7

Beyond Basic Recipes and Cookbooks

So far, we have only really looked at how to use cookbooks as a consumer, not as a producer. In order to harness the true power of Chef, it is important to learn how to build our own cookbooks and recipes using the full feature set that Chef provides. This chapter covers:

- Advanced recipe concepts
- Managing your data using data bags
- Searching Chef from recipes
- Advanced scripting in recipes
- Authoring custom providers, resources, and definitions
- Dealing with encrypted data

Managing users

Basic user management in Chef is achieved through the use of the `user` resource. This resource allows you to add, remove, or otherwise manipulate users on your hosts. However, you can't possibly write recipes that contain one resource per user; it simply wouldn't scale. In order to make large-scale user management easier, we can combine some of Chef's capabilities such as data bags, per-role, per-node, and per-environment configuration to enable scalable user management.

Let's take a look at a `user` cookbook that can provide these abilities.

Evolution of a shell user recipe

First, let's take a look at a very naive user management recipe. This cookbook has a hardcoded users list; initially, it contains frodo and samwise and simply iterates through the list, creating users as it goes. Here is what the list may look like:

```
users = [
  {
    'id' => 'frodo',
    'uid' => '100',
    'gid' => 100,
    'shell' => '/bin/hobbitshell',
    'comment' => 'Frodo of the nine fingers'
  },
  {
    'id' => 'samwise',
    'uid' => '101',
    'gid' => 101,
    'shell' => '/bin/gardenshell',
    'comment' => 'Samwise the strong'
  }
]

users.each do |u|
  home_dir = "/home/#{u['id']}"

  user u['id'] do
    uid u['uid']
    gid u['gid']
    shell u['shell']
    comment u['comment']
    supports :manage_home => true
    home home_dir
  end
end
```

This approach will work for a handful of users, but it has the problem of being very limited in scope and difficult to maintain. It also isolates the list of users to this recipe, making it difficult to access data from other recipes and very brittle. The first thing we can improve is make the users accessible to this and any other recipe through the use of data bags. Let's take a look at how we can use data bags to make user data management simpler and more flexible.

Storing data in data bags

Data bags are designed to store arbitrary configuration data that pertains to your entire infrastructure. This may include users, global settings, firewall rules, and so on; if it can be modeled using basic JSON data structures such as arrays and dictionaries, it can be included in a data bag. We haven't touched much on these yet, so now is a good time to take a look at what they can do while modeling users.

Creating a data bag for users

Data bags are collections of data that are related to one another; for example, users, firewall rules, database servers, and so on. Here we will create a data bag that contains our user data. This is not intended to be a replacement for a directory service such as LDAP, though you could potentially use it to store all your user data and then write recipes to populate an LDAP server with user data (in this way, you may be able to keep an Active Directory system and a separate LDAP system in sync by making your Chef data bag the authoritative source for user data). Let's take a look at how to create and manipulate a data bag with user information:

```
[jewart]% knife data bag create users
Created data_bag[users]
```

Now, create a new user, `frodo` (you will need to have the `EDITOR` variable set to a text editor such as `vim` on Linux systems):

```
[jewart]% export EDITOR=vi
[jewart]% knife data bag create users frodo
Data bag users already exists
```

You will be presented with a new entity template that contains only one key, `id`, which is set to the name of the entity you created; in our case, `frodo`:

```
1 {
2   "id": "frodo",
3 }
```

Save this file and you will now have one, mostly empty, entity in your `users` data bag named `frodo`. You can check this with the `show` subcommand:

```
[jewart]% knife data bag show users  
frodo
```

Every item in a data bag has to have a unique identifier, which can be meaningful or just a random identifier; in our case, it will double up as the login name for the user. We can take our previous data from the recipe and convert that to data bag elements by writing them to JSON files and uploading them with `knife`. To take advantage of uploads, we can create a directory, `users`, and create one JSON file per entry:

```
{  
  "id" : "frodo",  
  "uid" : "100",  
  "gid" : 100,  
  "shell" : "/bin/hobbitshell",  
  "comment" : "Frodo of the nine fingers"  
}  
  
{  
  "id" : "samwise",  
  "uid" : "101",  
  "gid" : 101,  
  "shell" : "/bin/gardenshell",  
  "comment" : "Samwise the strong"  
}
```

Once you have created these, you should have two files, `frodo.json` and `samwise.json` inside a `users` directory. In order to bulk upload them, we use a `knife data bag` from the `<dir> <data bag name>` file in the following manner:

```
[jewart]% knife data bag from file users users  
Updated data_bag_item[users::frodo]  
Updated data_bag_item[users::samwise]
```

You can verify whether the entries were created correctly with the `knife data bag show <databag> <entity_id>` command:

```
[jewart]% knife data bag show users frodo  
comment: Frodo of the nine fingers  
gid:    100
```

```
id:      frodo
shell:   /bin/hobbitshell
uid:     100
```

Searching for data

Now that we have our data in a data bag in Chef, we can search for it using the search criteria. For example, if we wanted only all users whose names start with the letter *s*, we can search with the following command:

```
[jewart]% knife search users 'id:s*'
1 items found

chef_type: data_bag_item
comment:   Samwise the strong
data_bag:   users
gid:       101
id:        samwise
shell:    /bin/gardenshell
uid:       101
```

Alternatively, if we wanted all the users in a given data bag, we can perform the following search:

```
[jewart]% knife search users 'id:/*'
2 items found

chef_type: data_bag_item
comment:   Frodo of the nine fingers
data_bag:   users
gid:       100
id:        frodo
shell:    /bin/hobbitshell
uid:       100

chef_type: data_bag_item
comment:   Samwise the strong
```

```
data_bag: users
gid: 101
id: samwise
shell: /bin/gardenshell
uid: 101
```

Searching inside recipes

Now that we have some data bag data created and can perform basic searches, let's see how we can use that to enhance our recipe using the built-in `search` method. This allows us to perform the searches we just ran with `knife` inside our recipes. The search method has a similar format to the `knife` command:

```
search(search_scope, search_criteria)
```

The following are some simple examples:

```
all_users = search(:users, 'id:*')
users_s = search(:users, 'id:s*')
all_nodes = search(:node, '*')
```

With this, we can enhance our shell user recipe to use the entities in the `users` data bag rather than hard code them. Our new recipe would look like the following:

```
# Replace the hard-coded users array with a search:
users = search(:users, 'id:*')

# Same as before, we've just moved our data source
users.each do |u|

  home_dir = "/home/#{u['id']}"

  user u['id'] do
    uid u['uid']
    gid u['gid']
    shell u['shell']
    comment u['comment']
    supports :manage_home => true
    home home_dir
  end

end
```

This is just a simple search; this will work for a small-scale infrastructure with a fixed set of users, where there's no need to restrict certain groups of users to certain hosts. You can easily imagine, however, a situation where some users are provisioned only to certain hosts through groups. Let's look at how we can achieve this with some better user metadata and a more advanced search.

Enhancing your user cookbook

In our previous example, we used the search method to find all of the users in our user's data bag. Here we will go one step further to isolate users based on arbitrary groups and see how we can limit the list of users to be provisioned using a combination of search, user metadata, and node configuration.

First, we need to add a `groups` key to our users. Let's add that to our existing user JSON data files and add a few more users, `legolas` and `gimli`:

```
{
  "id" : "frodo",
  "uid" : 100,
  "gid" : 100,
  "shell" : "/bin/hobbitshell",
  "comment" : "Frodo of the nine fingers",
  "groups" : [ "hobbits", "fellowship" ]
}

{
  "id" : "gimli",
  "uid" : 201,
  "gid" : 201,
  "shell" : "/bin/csh",
  "comment" : "Grumpy old dwarf",
  "groups" : [ "dwarves", "fellowship" ]
}

{
  "id" : "legolas",
  "uid" : 200,
  "gid" : 200,
  "shell" : "/bin/zsh",
  "comment" : "Keen eyed Legolas",
  "groups" : [ "elves", "fellowship" ]
}

{
```

```
"id" : "samwise",
"uid" : "101",
"gid" : 101,
"shell" : "/bin/gardenshell",
"comment" : "Samwise the strong",
"groups" : ["hobbits", "fellowship"]
}
```

Once again, we update the existing records and create our new records using knife data bag from file:

```
[jewart] % knife data bag from file users users
Updated data_bag_item[users::frodo]
Updated data_bag_item[users::gimli]
Updated data_bag_item[users::legolas]
Updated data_bag_item[users::samwise]
```

Now that you have a few additional users in your data bag, and each user has some group metadata attached to it, let's take a look at how we can use this to provision only certain users on specific hosts. First, we need to be able to limit our search scope dynamically; otherwise, we will need to modify our recipe on a per-host basis and that just won't scale. We need to add a dynamic search query to our recipe with something like the following code:

```
search_criteria = "groups:#{node[:shell_users]}[:group]"
```

This creates a search criteria string that will match objects that have the value specified somewhere in their groups key. In order to make this dynamic per host, we will store this value in a shell_users hash under the group key. For example, if you wanted to add all users that are in the hobbits group to a specific node, then your node's configuration would need to contain the following:

```
{
  "shell_users": {
    "group": "hobbits",
  }
}
```

This will build a search criteria of "groups" : "hobbits", which if we pass to the search method will yield all entries in the users data bag that have "hobbits" inside their groups list. Consider the following recipe code:

```
users = search(:users, search_criteria)
```

The node configuration data will expand the search criteria during an execution on this node to be the following:

```
search_criteria = "groups:hobbits"
```

Given the data we have stored in our users data bag, this would match `samwise` and `frodo` as they have the `hobbits` group in their `groups` list. We can verify this by trying the same search on the command line with `knife`:

```
[jewart]% knife search users "groups:hobbits"
2 items found

chef_type: data_bag_item
comment: Frodo of the nine fingers
data_bag: users
gid: 100
groups:
  hobbits
  fellowship
id: frodo
shell: /bin/hobbitshell
uid: 100

chef_type: data_bag_item
comment: Samwise the strong
data_bag: users
gid: 101
groups:
  hobbits
  fellowship
id: samwise
shell: /bin/gardenshell
uid: 101
```

As you can see, this allows us to narrowly define the list of users to be managed through the combination of entity metadata and dynamic search criteria. You can build more advanced applications using this methodology with more advanced search criteria and incorporating more of the entities' metadata.

Distributing SSH keys

In addition to managing user accounts, we can also use Chef to manage SSH keys. Because a given user's accepted SSH keys are stored in a per-user configuration file, it is quite simple to manipulate them. By creating a template for SSH-authorized keys, we can build a recipe that will take the SSH key data from the data bag and populate the authorized keys file on the host. By doing this, users' SSH keys can be stored in Chef and distributed to any number of hosts with just one command. This solves the problems typically associated with distribution and revocation of SSH keys inside an organization.

Templating the authorized keys

Here is a sample template we will use for our user's authorized keys file; this would be defined in an `authorized_keys.erb` file:

```
<% if @ssh_keys.is_a?(Array) %>
<%= @ssh_keys.join("\n") %>
<% else %>
<%= @ssh_keys %>
<% end %>
```

This is a very simple template that has only two cases: if the template variable `ssh_keys` is an array, it will print them out with a new line in between them; otherwise, it will simply print out the contents of the variable.

To use this template, we will simply provide it with a list of SSH-compatible key strings:

```
template "#{home_dir}/.ssh/authorized_keys" do
  source "authorized_keys.erb"
  owner u['id']
  group u['gid'] || u['id']
  mode "0600"
  variables :ssh_keys => u['ssh_keys']
end
```

Now, we can modify one of our previous user JSON entities to add SSH keys:

```
{
  "id" : "frodo",
  "uid" : 100,
  "gid" : 100,
  "shell" : "/bin/hobbitshell",
  "comment" : "Frodo of the nine fingers",
  "groups" : ["hobbits", "fellowship"],
```

```
"ssh_keys": [
    "ssh-dss RG9uJ3Qgd29ycnksIFNhS4gUm9zaWUga25vd3MgYW4gaWRpb3Q
    gd2hlbiBzaGUgc2VlcyBvbmu frodo@shire",
    "ssh-dss TXkgbWFzdGVyLCBTYXVyb24gdGh1IEdyZWF0LCBiWRzIHRoZWUgd
    2VsY29tZS4gSXMcGhlcUgYW55IGluIHRoaXMgcm91dCB3aXRoIGF1dGhvcmloSB0
    byB0cmVhdCB3aXRoIGl1PyA= sauron@mordor"
]
}

% knife data bag from file users ssh_keys/frodo.json
Updated data_bag_item[users::frodo]
```

Once your user has been updated, check whether your newly added metadata has been updated, looking for your new `ssh_keys` key in the entity. In order to do that, you can show the contents of your data bag using the following command:

```
% knife data bag show users frodo
```

The output of this should line up with your newly updated JSON content. With these added, we can write a new recipe that will allow us to deploy `authorized_keys` files for each user on our hosts. Our recipe will use the same search criteria from our previous recipe as we want to apply our SSH keys to all of our shell users.

This recipe is responsible for making sure that the proper directory for SSH is created and has the correct permissions, as well as creating the `authorized_keys` file with the necessary permissions and storing the SSH keys associated with the user in `/home/user/.ssh/authorized_keys`:

```
search_criteria = "groups:#{node[:shell_users] [:group]}"

search(:users, search_criteria) do |u|
    home_dir = "/home/#{u['id']}"

    directory "#{home_dir}/.ssh" do
        owner u['id']
        group u['gid']
        mode "0700"
        recursive true
    end

    template "#{home_dir}/.ssh/authorized_keys" do
        source "authorized_keys.erb"
        owner u['id']
        group u['gid']
```

```
    mode "0600"
    variables :ssh_keys => u['ssh_keys']
end

end
```

Adding deployment keys

If you have ever deployed a Rails application to hosts that need to have access to your source code in a GitHub or BitBucket repository, then you will know how handy it is to manage deployment keys across a fleet of hosts. We can easily generate a recipe that looks at a node's list of deployment users following our previous examples as a starting point. Here, we look for **deploy users** instead of shell users, as these are the ones we want to manage deployment keys for. Note that in this example, these users would also need to be included in the `shell_users` group to ensure that they get created by our previous recipe:

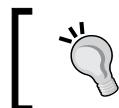
```
search_criteria = "groups:#{node[:deploy_users]}[:group]"

search(:users, search_criteria) do |u|
  home_dir = "/home/#{u['id']}"

  directory "#{home_dir}/.ssh" do
    owner u['id']
    group u['gid'] || u['id']
    mode "0700"
    recursive true
  end

  template "#{home_dir}/.ssh/id_rsa" do
    source "deploy_key.erb"
    owner u['id']
    group u['gid'] || u['id']
    mode "0600"
    variables :key => u['deploy_key']
  end
end
```

To use this new recipe, the deployment users would need to be modified to include a group identifier and their private key. The group would be reserved for users involved in deploying your application and be added to the user's groups key in Chef. Additionally, an unencrypted SSH private key would need to be present in a `deploy_key` field.



Including unencrypted SSH keys can pose a security risk. This can be mitigated using encrypted data bags or an external security material management service.

Writing custom extensions

With Chef, you are given immediate access to a number of resources: files, users, packages, templates, and so on. However, there will always be times when this does not provide you with everything that you need. Fortunately, the built-in Chef resources or LWRPs (light-weight resource providers) are just Ruby code and were built with the intention of providing a framework for end users to build their own components. This means that you can easily build your own custom resources, and these can be shared with others just like any built-in LWRP.

Developing a custom definition

One of the simplest resources that we can build is a **definition** – a definition is like a resource with only one built-in provider. These can be thought of as reusable modules that you can leverage inside of your recipes. If you find yourself writing the same thing *repeatedly* in your recipes, then it is probably a good candidate to write a custom definition. For example, let's look at how we can build two different definitions: one for executing Python's PIP, the Python package installation tool, to install the contents of a `requirements.txt` file for a Python application, and another to install applications that follow the same pattern.

Organizing your code

As discussed before, cookbooks can have a `definitions` directory inside them. The contents of this directory are included in your cookbook runs and should have one per definition. For our PIP resource, we will create a file, `definitions/pip_requirements.rb`, and for our application template, `definitions/python_web_application.rb`. These will each contain the respective definitions.

Writing a definition for using PIP

Definitions look like any Chef component – they are composed of resources, variables, scripts, and anything else you use in a recipe. However, unlike a recipe, they are designed to be reused. Where a recipe is designed with a specific effect in mind such as deploying a specific application, the definition is designed to be consumed by recipes to reduce duplicate code.

Each definition is encapsulated in a `define` block, a no-op version, or our PIP example would look like this:

```
define :pip_requirements do
end
```

This example does absolutely nothing, but it can be used in a recipe as follows:

```
pip_requirements "app_requirements" do
end
```

Just in the same way you would use a file, user, or template block in your recipe, you can use your custom definitions. Now, let's enhance our definition by using the `name` parameter – the string argument passed to the `pip_requirements` block in your recipe; here, it is `app_requirements`:

```
define :pip_requirements , :action => :skip do
  name = params[:name]
end
```

Each invocation of a definition passes the parameters in the block to the definition; these are accessed inside the definition through the `params` hash. There is one special parameter, `:name`, which can come from the first argument before the block, as shown in the previous code, or from the `name` parameter inside the block. This is a convenience parameter designed to make recipes more readable by allowing the developer to write:

```
resource "some handy description" do
  ...
end
```

This code is easier to read than:

```
resource do
  name "some handy description"
end
```

Given this information, let's look at the PIP example from `pip_requirements.rb`:

```
define :pip_requirements , :action => :skip do
  name = params[:name]
  requirements_file = params[:requirements_file]
  pip = params[:pip]
  user = params[:user]
  group = params[:group]

  if params[:action] == :run
```

```
script "pip_install_#{name}" do
  interpreter "bash"
  user "#{user}"
  group "#{group}"
  code <<-EOH
    #{pip} install -r #{requirements_file}
  EOH
  only_if { File.exists?(" #{requirements_file} ") and File.
exists?("#{pip}") }
end
end
```

Here, the definition expects five arguments: the resource name, the path to the requirements.txt file, the pip binary to use, as well as the user and group to execute pip as. The reason that the resource accepts the path to pip is to allow using pip inside a Python virtual environment. By doing this, the definition becomes a little more flexible in situations where you need to install your requirements into a different location on the system.

Also note that we can define default parameters as part of the definition's signature:

```
define :pip_requirements , :action => :skip do
```

In this case, the default action is :skip, but it can be set to anything you want it to be. Here it is set to :skip so that it only gets invoked deliberately rather than by virtue of being used in a recipe.

As this is a simple definition, it only contains one resource—a script block that will effectively execute pip install -r /path/to/requirements.txt as the specified user and group. An example use of this definition can be seen as follows:

```
pip_requirements "requirements" do
  action :run
  pip "/usr/local/bin/pip"
  user node[:app] [:user]
  group node[:app] [:group]
  requirements_file "#{app_root}/src/requirements.txt"
end
```

This can be used in place of the built-in script resource:

```
script "pip_install_#{name}" do
  interpreter "bash"
  user node[:app] [:user]
  group node[:app] [:group]
  code <<-EOH
```

```
/usr/local/bin/pip install -r #{app_root}/src/requirements.txt
EOH
only_if {
  File.exists?( "#{app_root}/src/requirements.txt") and
  File.exists?("/usr/local/bin/pip")
}
end
```

Following Chef's declarative language, building definitions such as this one makes it more obvious as to what is happening, rather than how it is happening. We have abstracted the shell script and guard tests behind a façade, that is the `pip_requirements` definition, which is more clear in its effect when you read a recipe; you don't need to examine the contents of the `script` block to deduce what is happening as the resource name tells you exactly what's going to be done.

Defining a full application template

If you have applications that follow the same structure (think applications that use a common framework such as Rails, Django, Pyramids, Python-tornado, and so on), then you would likely want to define a common definition for what such an application looks like. Consider here a definition to install a Python web application from GitHub using some common idioms:

```
define :tornado_application do
  app_name = params[:name]
  app_root = params[:app_root]
  app_user = params[:user]
  app_group = params[:group]

  python_interpreter = params[:python_interpreter] ||
    "/usr/bin/python3.3"
  github_repo = params[:github_repo]
  deploy_branch = params[:deploy_branch] || "deploy"

  virtualenv = "#{app_root}/python"
  virtual_python = "#{virtualenv}/bin/python"
  app_dir = "#{app_root}/src/#{app_name}"

  # Need to install SSH key for GitHub
  # this comes from the ssh_known_hosts cookbook
  ssh_known_hosts_entry 'github.com'

  # Base package requirements
  package "git"
```

```
package "libpq-dev"
package "libxml2-dev"
package "python3.3"
package "python3.3-dev"

directory "#{app_root}" do
  owner "#{app_user}"
  group "#{app_group}"
  mode "0755"
  action :create
  recursive true
end

# Create directories
["bin", "src", "logs", "conf", "tmp"].each do |child_dir|
  directory "#{app_root}/#{child_dir}" do
    owner "#{app_user}"
    group "#{app_group}"
    mode "0755"
    action :create
    recursive true
  end
end

# Install Python virtualenv
python_virtualenv "#{virtualenv}" do
  owner "#{app_user}"
  group "#{app_group}"
  action :create
  interpreter "#{python_interpreter}"
end

# Application checkout
git "#{app_dir}" do
  repository "#{github_repo}"
  action :sync
  user "#{app_user}"
  branch "#{deploy_branch}"
end

# Python dependencies for app
pip_requirements "tornado_app[#{app_name}]" do
  action :run
```

```
    pip "#{virtualenv}/bin/pip"
    user "#{app_user}"
    group "#{app_group}"
    requirements_file "#{app_dir}/requirements.txt"
  end

end
```

This definition can be used as shown in the following example:

```
tornado_application "image_resizer" do
  app_root "/opt/webapps"
  user "webapp"
  group "webapp"
  deploy_branch "master"
  github_repo "git@github.com:myorg/image_resizer.git"
  python_interpreter "/usr/bin/python3.3"
end
```

According to the previous definition, this would do the following:

- Add a system-wide SSH-known key for `github.com` (required to perform a Git clone and guarantees that future key changes will work)
- Install any required packages if they didn't already exist, including Git, Python, and the `postgresql` client
- Ensure any application-required directories exist for data such as binaries, logs, configuration, and more
- Create a Python virtual environment based on the supplied Python interpreter (3.3) in `<app_root>/python`
- Clone or sync (if it was already cloned) the source code from `<github_repo>` to `<app_root>/src/<app_name>`
- Install the requirements specified in `<app_root>/src/<app_name>/requirements.txt` using the copy of `pip` from the virtual environment in `<app_root>/python`

Assuming you had another similarly structured application, but you wanted to use a different user, group, Python interpreter, and deployment branch, you can easily configure it using the following resource:

```
tornado_application "restful_api" do
  app_root "/opt/webapps"
  user "restapi"
  group "restapi"
  deploy_branch "production"
```

```
github_repo "git@github.com:myorg/restful_api.git"
  python_interpreter "/usr/bin/python3.2"
end
```

As you can see, definitions allow us to define reusable resources in Chef. There are three primary benefits to this approach:

- Simplified recipes are easier to read, have clearer intentions, and less code to audit, which makes them less error prone
- Any changes to the definition are automatically applied to any usage of the definition, which means you don't need to maintain multiple variations
- It's easier to test because it's designed to be parameterized and modular

Now that you see how easy it is to write custom resources for Chef through definitions, let's examine writing a full-blown resource that has a separate provider implementation.

Building a resource

A Chef LWRP is composed of two primary components, a **resource** and a **provider**. The resource is the blueprint for what is being provided; it describes the resource, including what actions can be taken by a resource, the properties that describe the resource, and any other high-level information about it. The provider is responsible for the actual implementation of the resource. In programming terms, the resource is an abstract class or interface where the provider is a concrete class or implementation. For example, one of Chef's built-in resources is the `package` resource; however, this is a very high-level resource. The `package` resource describes what a package is and what a package can do but not how to manage them. That work is left to the providers, including RPM, APT, FreeBSD packages, and other backend systems that are capable of managing on-disk installation of packages.

Defining the resource

As an example, let's take a look at an S3 bucket resource:

```
actions :sync
default_action :sync if defined?(default_action) # Chef > 10.8

# Default action for Chef <= 10.8
def initialize(*args)
  super
  @action = :sync
```

```
end

# Target folder on the host to sync with the S3 bucket
attribute :destination, :kind_of => String,
           :name_attribute => true
# Anything to skip when syncing
attribute :omit, :kind_of => Array
# AWS Access / secret key
attribute :access_key_id, :kind_of => String
attribute :secret_access_key, :kind_of => String
```

Here, our resource is an S3 bucket that declares the actions it can take along with the attributes that it relies on. Here, our resource declares that it has one available action, `sync`, which is the default action and that it has four attributes: the destination, what files to skip, the access key, and the secret key.

Implementing the provider

The provider is where the logic for the resource is placed – it is responsible for acting on the resource being described. For our S3 bucket, it looks like the following:

```
require 'chef/mixin/language'

# Only run as needed
def whyrun_supported?
  true
end

action :sync do
  Chef::Log.debug("Checking #{new_resource} for changes")
  fetch_from_s3(new_resource.source) do |raw_file|
    Chef::Log.debug "copying remote file from origin #{raw_file.path}"
    to destination #{new_resource.destination}"
    FileUtils.cp raw_file.path, new_resource.destination
  end

  new_resource.updated_by_last_action(true)
end

def load_current_resource
  chef_gem 'aws-sdk' do
    action :install
```

```
end

require 'aws/s3'

current_resource = new_resource.destination
current_resource
end

def fetch_from_s3(source)
begin
  protocol, bucket = URI.split(source).compact
  AWS::S3::Base.establish_connection!(
    :access_key_id      => new_resource.access_key_id,
    :secret_access_key  => new_resource.secret_access_key
  )

  bucket.objects.each do |obj|
    name = obj.key

    if !new_resource.skip.contains(name)
      Chef::Log.debug("Downloading #{name} from S3 bucket
#{bucket}")
      obj = AWS::S3::S3Object.find name, bucket

      file = Tempfile.new("chef-s3-file")
      file.write obj.value
      Chef::Log.debug("File #{name} is #{file.size} bytes on disk")
      begin
        yield file
      ensure
        file.close
      end
    else
      Chef::Log.debug("Skipping #{name} because it's in the skip
list")
    end
  end
rescue URI::InvalidURIError
  Chef::Log.warn("Expected an S3 URL but found #{source}")
  nil
end
end
```

Let's take a look at the provider, piece by piece. The first thing the provider does, beyond including any required libraries, is to inform Chef that it supports **why-run**. This is a mechanism that Chef provides so that resources can be more easily tested by effectively not wiring a resource to a provider. This allows developers to test their resources, in what is effectively a **dry-run** mode, before running them live against a system:

```
# Only run as needed
def whyrun_supported?
  true
end
```

Next, there is an action block – this registers the provided block as the logic to be executed for the specified action (in this case, :sync). This has the general form such as:

```
action :<action name> do
  # Real work in here
end
```

In this case, the only supported action is sync, and so there is only one action block:

```
action :sync do
  Chef::Log.debug("Checking #{new_resource} for changes")
  fetch_from_s3(new_resource.source) do |raw_file|
    Chef::Log.debug "copying remote file from origin #{raw_file.path}"
    to destination #{new_resource.destination}"
      FileUtils.cp raw_file.path, new_resource.destination
    end
  new_resource.updated
end
```

Here, the :sync action leverages the `fetch_from_s3` method, which yields a local copy of a file in the remote bucket once it has been downloaded. Then, the file is copied from the temporary location locally into the specified destination.

Modifying resources

Inside of this action, you will notice that there is an actor, `new_resource` (which is actually a built-in method). This describes what the state of the named resource should be when the provider has completed its execution for the specified resource; this may or may not differ from the current state of the resource on the node. In the case of an initial run, `new_resource` will almost certainly be different from `current_resource`, but that may not always be the case on subsequent runs.

As an example, if we have a recipe with the following S3 bucket resource declared:

```
s3_bucket "s3://mychefbucket/.resource" do
  action :sync
  skip ["foo.txt", "bar.txt"]
  destination "/opt/app_data"
  access_key_id node[:app][:aws_access_key]
  secret_access_key node[:app][:aws_secret_key]
  owner node[:app][:user]
  group node[:app][:group]
  mode "0755"
end
```

Then, the `new_resource` actor would have its member variables populated with the parameters passed to the `s3_bucket` resource. Again, this is the expected state of the resource, the way it should be when the execution by the provider is complete. In this case, when the provider code is executed, `new_resource.destination` will be `"/opt/app_data"` and `new_resource.skip` will be a list of `"foo.txt"` and `"bar.txt"` and so on. This allows you to pass data into the instance of the resource in the same way that was possible with the PIP and Tornado application definitions.

Loading an existing resource

One thing that is less obvious about the provider script is the `load_current_resource` method that is not called from within the provider. This method is used by Chef to find a resource on the node based on the attributes that are provided by the recipe. This is useful to determine if anything needs to be done to bring an existing resource on the host such as a file, a user account, or a directory of files, up to date with the data that is provided during execution of the recipe.

It might make sense to extend this provider to precompute the hashes of the files that already exist in the directory on-disk as specified by `destination`. This way, the provider can be updated to only download any remote files in S3 that have a different fingerprint than a similarly named resource on disk. This prevents unnecessary work from being performed, which saves time, bandwidth, and other resources.

Here, however, it is also used to ensure that any dependencies to download files are installed; in this case, the AWS gem is required to use the S3 client. This works because the `load_current_resource` method gets called on early to determine the current state of the resource. If the resources are the same, then the provider has nothing to do. The current implementation just clobbers whatever files are local with the contents of the S3 bucket (more of a one-way download than a sync, really).

Declaring that a resource was updated

Resources have a built-in method, `updated_by_last_action`, which is called inside the `:sync` action all the time in this example. This method notifies the resource that the node was updated successfully. This should only be set to `true` if everything was successfully updated; failures should not make this call unless they set it to `false`. It is useful to know what resources have been updated for reporting or other purposes. For example, you can use this flag to identify what resources have been updated:

```
module SimpleReport
  class UpdatedResources < Chef::Handler
    def report
      Chef::Log.info "Resources updated this run:"
      run_status.updated_resources.each do |r|
        Chef::Log.info "  #{r.to_s}"
      end
    end
  end
end
```

Working with data bags

There are a number of things you can do with data bags.

Securing your data bags

Data bags are just JSON data, but they are stored in the system as plain text, without any security. They are also downloaded onto various hosts throughout the life cycle, which can lead to leaking of potentially sensitive information. Fortunately, Chef has a method that lets you secure this data by using `knife`, along with secret keys to keep data in data bags encrypted.

Secret keys

Encrypting a data bag item requires a secret key; one way of generating a secret key is to generate a random number and use the `Base64` encoding of that number as the secret key. This should have any line endings removed to ensure it works properly on all platforms, regardless of platform-specific line endings. Here is a quick way to generate one using the `openssl` command line tool combined with `tr` to remove any line endings:

```
$ openssl rand -base64 512 | tr -d '\r\n' > ~/.chef/data_bag_secret
```

Encrypting your data

In order to encrypt your data bag item, you must use `knife` and pass the `--secret` or `--secret-file` flags to `knife` when creating the item. For example, to create a data bag called `credentials` and store a new entry, `aws`, inside it, you would use the following command (make sure you set your `EDITOR` environment variable first):

```
$ knife data bag create credentials aws --secret-file ~/.chef/data_bag_secret
```

As mentioned before, you will be presented with the contents of your new data bag item in your editor, unencrypted:

```
1 {
2   "id": "aws",
3 }
```

Here, we can add some properties, such as a secret key:

```
1 {
2   "id": "aws",
3   "secret_key": "A21AbFdeccFB213f"
4 }
```

Once you save this, `knife` will tell you that the new data bag was created, along with the new data bag item in Chef, just as it did with the user data earlier. The only difference will be that this time the data stored in the Chef server has been encrypted using the symmetric key you provided:

```
$ knife data bag create credentials aws --secret-file ~/.chef/data_bag_secret
Created data_bag[credentials]
Created data_bag_item[aws]
```

To check whether your newly created data bag entry was encrypted, use `knife`, as we have before, to show the contents of an item:

```
$ knife data bag show credentials aws
```

```
id:          aws
secret_key:
cipher:      aes-256-cbc
encrypted_data:
```

```
SG4z4jd4VanJ4gG0wPcJWOX7H+ZNSxG5PH+n7EgHFV9e1SciVznjaAbzK61c
EW0/
iv:          rKB0riCr84QhBkw+Wgc/5Q==
version:     1
```

Decrypting your data

In order to decrypt the data in the data bag item, you need to provide the same symmetric key as you provided when you encrypted it, using the `--secret` or `--secret-file` argument, as can be seen here:

```
$ knife data bag show credentials aws --secret-file ~/.chef/data_bag_
secret
id:          aws
secret_key:  A21AbFdeccFB213f
```

If it wasn't already obvious to you, make certain you do not lose this file. Without your secret key or secret file, you will not be able to decrypt the data in your data bag. It may be worth encrypting the secret file with a passphrase if you are going to be transmitting it to nontrusted locations as well.

Storing keys on nodes

An encryption key can also be stored in an alternate file on the nodes that need it, and you can specify the path location to the file inside an attribute; however, `EncryptedDataBagItem.load` expects to see the actual secret as the third argument rather than a path to the secret file. In this case, you can use `EncryptedDataBagItem.load_secret` to slurp the secret file contents and then pass them:

```
# inside your attribute file:
default[:app] [:aws_creds_secret] = "/opt/secret/aws.secret"

# Inside your recipe
aws_secret = Chef::EncryptedDataBagItem.load_secret
  "#{node[:app] [:aws_creds_secret]}"

aws_creds = Chef::EncryptedDataBagItem.load
  "credentials", "aws", aws_secret

aws_creds["secret_key"]
```

Searching your data

As we discussed earlier, you can search through your data bags using Boolean search logic. This permits you to find only the entries in your data bags that you need. The same search query language is used on the command line with `knife` as it is in your recipes, so that you can test your queries on the command line to ensure that they produce the right results before you put them in your recipes. You can also search through other resources as well, not just data bags.

Searching your data bags with `knife`

The `knife` tool uses the `search` command to search through your data bags. The general syntax is:

```
knife search <source> "<search criteria>"
```

Searching your data bags from a recipe

Inside a recipe, the `search` method is used to search through a data bag. The syntax for this is:

```
search(:source, "search criteria")
```

Querying your data

The search query format is reasonably straightforward and looks like most other search engines that support the Boolean logic.

Searches on attributes come in the form of `key:value`; so for example, if you wanted to find all of the users who were dwarves from our earlier data sets, you can use the search query:

```
knife search users "groups:dwarves"
```

Negating a search term can be accomplished by placing `NOT` in front of the search term. For example, all users who are not hobbits will be:

```
knife search users "NOT groups:hobbits"
```

You can also use an `OR` modifier:

```
knife search users "groups:elves OR groups:hobbits"
```

This last search criteria would yield the users `legolas`, `samwise`, and `frodo` as Frodo and Samwise are in the group called `hobbits` and Legolas is in the `elves` group.

While combining search terms, you can logically AND them together as well. For example, all users with a GID starting with 20, who contain the group `elves` can be found using the following query:

```
knife search users "groups:elves AND gid:20*"
```

You can search your nodes with the same query language—in order to find all nodes that are running some form of windows, you can search for the platform being anything that starts with `win`:

```
knife search node "platform:win*"
```

This will yield all Windows hosts (results have been shortened a bit):

```
4 items found
```

```
Node Name: i-13d0bd4f
Roles:
Platform: windows 6.2.9200

Node Name: WIN-CJDQ9DEOJFK
Roles: umbraco_cms
Platform: windows 6.2.9200

Node Name: 00c0ff3300
Roles:
Platform: windows 6.2.9200

Node Name: rs-5889646228538071
Roles:
Platform: windows 6.2.9200
```

Or, you can search for all nodes that are running windows and contain the role `umbraco_cms`:

```
knife search node "platform:win* AND role:umbraco_cms"
```

Or, if you wanted to eliminate those nodes that run the Umbraco CMS, you can easily invert the role condition:

```
knife search node "platform:win* AND NOT role:umbraco_cms"
```

Because Chef uses Apache Solr to search its data, you can refer to the Apache Solr documentation on building more advanced query logic at <http://wiki.apache.org/solr/SolrQuerySyntax>.

Managing multiple machines with search queries

The search criteria can be used for all sorts of places: in recipes, on the command line, through API calls, and more. One very interesting application is being able to use the search query to SSH to multiple machines to perform commands in parallel:

```
knife ssh "fqdn:*.east.mycorp.com AND platform:ubuntu" "chef-client" -x  
app_user
```

This will contact the Chef server and ask for the nodes that match the given query string (machines whose FQDN matches the wildcard expression "`*.east.mycorp.com`" and that are running Ubuntu) and then connect to them via SSH as the user `app_user` and run the `chef-client` command on each of them. Again, you can restrict (or expand) the server list by using a more (or less) specific query.

Once you have mastered this aspect of using `knife`, you can learn more about its support for executing multiple connections concurrently and even interact with terminal multiplexers such as `screen` and `tmux`.

Summary

Chef has lots of mechanisms to build advanced automation, including building your own definitions, resources, and providers, as well as storing and accessing complex configuration data and even securely encrypting it. This chapter has shown you how to manage data in data bags (including encrypted data), use Chef's advanced search engine to find and manipulate data in your system from the command line and in recipes, as well as develop definitions for reusable recipe development, and even build custom resources and providers for use in your cookbooks.

In the next chapter, we will cover some more advanced ways to use Chef, including interacting with the Chef shell, automation and integration with Chef using scripts and APIs, external tools and resources, advanced testing including integration testing, and using Chef-solo and Vagrant to manage your development environments.

8

Extras You Need to Know

This chapter will cover how to use Chef to build custom bootstrap scripts for systems, enhanced command-line tool concepts, leverage Chef for automation, integration, and securely store sensitive data in the system. Some topics that will be covered in this chapter include:

- Using Chef-solo with Vagrant
- Interacting with the Chef shell
- Debugging recipes
- Advanced command-line usage
- Automating and integrating with Chef
- More testing methodologies

Vagrant and Chef-solo

Vagrant is a very useful tool to build development environments, where it provides tools to build virtual machines that contain everything you need to get started with building software. Consider, for a moment, working on a team that builds software and relies on a **service-oriented architecture (SOA)**, and this software is composed of a number of different services. In order for it to work, you may be required to install and configure all of the dependent services to even begin working on a part of the system; this could be a time-consuming and error-prone exercise for even seasoned developers. Now imagine that all you had to do was download a configuration file and execute `vagrant` to do it for you – this is the world of Vagrant.

One of the interesting facets of Vagrant is that it has support to provision new instances using a number of different mechanisms. Currently, this list includes 10 or so different tools, but the most interesting two are Chef-solo and Chef client. By now, you should be comfortable with how you might provision a virtual machine using the Chef client; it's not much different than provisioning an EC2 instance or a dedicated server. However, we haven't discussed using Chef-solo much yet, so this is a good time to learn more about it.

Installing Vagrant

Historically, Vagrant was installed via RubyGems; this is no longer the case, and if you have an older version installed as a gem, it is recommended that you remove it before installing Vagrant. Installers for all supported platforms (OS X, Windows, and Linux) are available at the following URL:

<http://www.vagrantup.com/downloads>

If you are new to Vagrant, then in addition to installing Vagrant, you will want to install VirtualBox for simplicity, as Vagrant has built-in support for VirtualBox. Vagrant does support other providers such as VMWare and AWS, but it requires plugins that are not distributed with the core Vagrant installation in order for them to work.

Once you have installed Vagrant and VirtualBox, then you can continue on with the following examples.

Provisioning a new host with Vagrant

Provisioning a new virtual instance requires that you build a Vagrant configuration file called `vagrantfile`. This file serves two purposes: to denote that the directory is a Vagrant project (similar to how a `Makefile` indicates a project that is built with Make), and to describe the virtual machine that is being run, including how to provision it, what operating system to use, where to find the virtual image, and so on. Because this is just a plain text file, you can include it along with any auxiliary files required to build the image such as cookbooks, recipes, JSON files, installers, and so on, and commit it to the source control for others to use.

In order to begin, you will want to create a directory that will house your new Vagrant project. On Unix-like systems, we would bootstrap our project similarly to the following command:

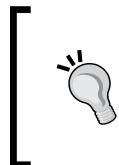
```
mkdir -p ~/vagrant/chef_solo  
cd ~/vagrant/chef_solo
```

Windows hosts will be the same except for different paths and changes in methods of directory creation. Once this step is complete, you will need to create a skeleton configuration located in `~/vagrant/chef_solo/Vagrantfile`. This file can be generated using `vagrant init`, but we would not want to use the contents of the generated file; so, we will skip that step and manually construct our `Vagrantfile` instead (with a simple one-line configuration that uses a base image of Ubuntu 13.10). Your `Vagrantfile` should look like the following code:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
end
```

Here, "2" is the API version, which is currently Version 2.0 as of this writing, and the configured base image (or `box`) is the Ubuntu Trusty (14.04) 64-bit image. Before you use this base image, it needs to be downloaded to your local machine; you can add it to Vagrant using the `box add` command:

```
vagrant box add ubuntu/trusty64
```



This step will take a few minutes on a fast connection, so be prepared to wait while this completes if you are following along interactively. Also note that if you skip this step, the base image will automatically be downloaded while running `vagrant up` to start your virtual machine.



For future references, if you want to find alternative OS images to use for your Vagrant machines, you should look at Vagrant Cloud (<https://vagrantcloud.com/>), where you can find a number of other freely available base images to download for use with Vagrant.

Booting your Vagrant image

Once your base image has completed downloading, you will use the `vagrant up` command to boot up a new virtual machine. By doing this, you will instruct Vagrant to read the `Vagrantfile` and boot a new instance of the base image:

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/trusty64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/trusty64' is up to date...
==> default: Setting the name of the VM: chef_solo_
default_1402875519251_51266
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
```

```
--> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
--> default: Forwarding ports...
    default: 22 => 2222 (adapter 1)
--> default: Booting VM...
--> default: Waiting for machine to boot. This may take a few
minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
--> default: Machine booted and ready!
--> default: Checking for guest additions in VM...
--> default: Mounting shared folders...
    default: /vagrant => /Users/jewart/Temp/vagrant/chef_solo
```

As you can see from the output, Vagrant performed the following things:

- Used the base image `ubuntu/trusty64`
- Configured VirtualBox to use a NAT adapter, mapping port 22 to 2222
- Started the VM in headless mode (such that you don't see the VirtualBox GUI)
- Created a user, `vagrant`, with a private key for authentication
- Mounted a shared folder mapping `/vagrant` on the guest to the Vagrant workspace on the host

Now that you have a running guest, you can control it by running `vagrant` commands from inside of the `vagrant` workspace (`~/vagrant/chef_solo`); for example, you can SSH into it using the following command:

```
vagrant ssh
```

And you can destroy the running instance with the following command:

```
vagrant destroy
```

Go ahead and SSH into your new guest and poke around a little bit—you will notice that it looks just like any other Ubuntu 14.04 host. Once you are done, use `destroy` to destroy it so that you can look at how to provision your Vagrant image using Chef-solo. It's important to know that if you use `destroy` on your guest, changes to your Vagrant image are not persisted; so, any changes you have made inside it will not be saved and will not exist the next time you use `vagrant up` to start the VM.

Combining Vagrant with Chef-solo

In our previous example, our `Vagrantfile` simply declared that our guest relied on the `ubuntu/trusty64` image as the base image via the `config.vm.box` property. Next, we will look at how to extend our configuration file to use the Chef-solo provisioner to install some software on our guest host. Here, we will use Chef-solo to install PostgreSQL, Python, and a web application inside of the guest.

You will probably notice that the configuration sections in the `Vagrantfile` look sort of like resources in Chef—this is because they both leverage Ruby blocks to configure their resources. So with Vagrant, in order to specify the provisioning mechanism being used, the `config.vm.provision` option is set to the desired tool. Here, we will use Chef-solo, which is named "`chef_solo`"; so, we will extend our `Vagrantfile` to indicate this:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "chef_solo" do |chef|
    # ... Chef specific settings block
  end
end
```

Understanding the limitations of Chef-solo

For the most part, Chef-solo operates a lot like the traditional client-server mode of chef-client. The primary differences result from the fact that Chef-solo does not interact with a central Chef server and therefore lacks support for the following:

- Node data storage
- Search indexes
- Centralized distribution of cookbooks
- A centralized API that interacts with and integrates infrastructure components
- Authentication or authorization
- Persistent attributes

As a result, if you are writing recipes to be used with Chef-solo, you will be unable to rely on search for nodes, roles, or other data and may need to modify the way you find data for your recipes. You can still load data from data bags for complex data, but they will not be centrally located; rather, they will be located in a number of JSON files that contain the required data.

Configuring Chef-solo

There are a number of options available for the Chef-solo provisioner in Vagrant. For the most up-to-date documentation of Vagrant, be sure to visit the official Vagrant documentation site at <http://docs.vagrantup.com/v2/>.

Most of the configuration options are ways to provide paths to various Chef resources such as cookbooks, data bags, environments, recipes, and roles. Any paths specified are relative to the Vagrant workspace root (where the `Vagrantfile` is located); this is because these are mounted in the guest under `/vagrant` and are the only way to get data into the host during the bootstrap phase. The ones we will be using are:

- `cookbooks_path`: This consists of a single string or an array of paths to the location where cookbooks are stored. The default location is `cookbooks`.
- `data_bags_path`: This consists of a path to data bags' JSON files. The default path is empty.
- `roles_path`: This consists of an array or a single string of paths where roles are defined in JSON. The default value is empty.

In our case, we will be reusing our example cookbooks from the earlier chapter. You can fetch them from GitHub at http://github.com/johnewart/chef_cookbook_files; either download the ZIP file or clone them using Git locally. Once you have done that, copy `cookbooks`, `roles`, and `data_bags` from the archive to your Vagrant workspace. These will be the resources that you will use for your Vagrant image as well. In order to tell Vagrant's Chef-solo provider how to find these, we will update our `Vagrantfile` again to include the following configuration:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "chef_solo" do |chef|
    chef.cookbooks_path = "cookbooks"
    chef.roles_path = "roles"
    chef.data_bags_path = "data_bags"
  end
end
```

Telling Chef-solo what to run

Inside of the provision block, we have a Chef object that effectively represents a Chef client run. This object has a number of methods (such as the path settings we already saw), one of which is the `add_recipe` method. This allows us to manually build our run list without requiring roles or data bags and can be used, as shown in the following example, to install the PostgreSQL server with no special configuration:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "chef_solo" do |chef|
    chef.cookbooks_path = "cookbooks"
    chef.roles_path = "roles"
    chef.data_bags_path = "data_bags"
    # Build run list
    chef.add_recipe "postgresql::server"
  end
end
```

This will tell Vagrant that we want to use our defined directories to load our resources, and we want to add the `postgresql::server` recipe to our run list. Because cookbooks are by default expected to be in `[vagrant root]/cookbooks`, we can shorten this example as shown in the following code, as we are not yet using roles or data bags:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "chef_solo" do |chef|
    chef.add_recipe "postgresql::server"
  end
end
```

Using roles and data bags with Chef-solo

As you are already aware by now, we may want to perform more complex configuration of our hosts. Let's take a look at how to use both roles and data bags as well as our cookbooks to deploy our Python web application into our Vagrant guest similar to how we deployed it to EC2:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.provision "chef_solo" do |chef|
    chef.cookbooks_path = "cookbooks"
    chef.roles_path = "roles"
    chef.data_bags_path = "data_bags"
    # Build run list
```

```
    chef.add_role("base_server")
    chef.add_role("postgresql_server")
    chef.add_role("web_server")
  end
end
```

Just like the `cookbooks` path, the `roles` path is relative to the project root if a relative path is given.

Injecting custom JSON data

Additional configuration data for Chef attributes can be passed into Chef-solo. This is done by setting the `json` property with a Ruby hash (dictionary-like object), which is converted to JSON and passed into Chef:

```
Vagrant.configure("2") do |config|
  config.vm.provision "chef_solo" do |chef|
    #
    # ...
    chef.json = {
      "apache" => {
        "listen_address" => "0.0.0.0"
      }
    }
  end
end
```

Hashes, arrays, and so on can be used with the JSON configuration object. Basically, anything that can be turned cleanly into JSON works.

Providing a custom node name

You can specify a custom node name by setting the `node_name` property. This is useful for cookbooks that may depend on this being set to some sort of value. For example:

```
Vagrant.configure("2") do |config|
  config.vm.provision "chef_solo" do |chef|
    chef.node_name = "db00"
  end
end
```

Getting to know the Chef shell

The Chef shell, previously called `chef`, provides an interactive tool or **read-eval-print-loop (REPL)** to work with Chef resources. Much in the same way IRB or any other language's REPL shell works, `chef-shell` is a way to interact with `knife`. This is handy for experimenting with resources while writing recipes so that you can see what happens interactively rather than having to upload your cookbook to a server and then executing the chef-client on a target node. Additionally, the Chef shell provides a resource to add breakpoints to recipe execution so that it can be used to debug recipe execution, which is a very handy feature.

Using the Chef shell

As of 11.x, `chef` has been replaced with `chef-shell` and can be used in three different modes: standalone, solo, and client mode. Each of these has a slightly different set of functionalities and expected use cases.

The standalone mode

The standalone mode is used to run Chef in an interactive mode with nothing loaded; this is almost like running an REPL such as `irb` or `python` on the command line. This is also the default behavior of `chef-shell` if nothing is specified.

The solo mode

The solo mode is invoked using the `-s` or `--solo` command-line flag and is a way to use `chef-shell` as a chef-solo client. It will load any cookbooks using the same mechanism that `chef-solo` users would, and it will use any `chef-solo` JSON file provided to it using the `-j` command-line option.

The following are examples of using the solo mode:

```
chef-shell -s
chef-shell -s -j /home/myuser/chef/chef-solo.json
```

The client mode

The client mode is enabled with the `-z` or `--client` command-line flag; this mode causes `chef-shell` to act as though you invoked `chef-client` on the host. The shell will read the local client configuration and perform the normal duties of `chef-client`: connecting to your Chef server and downloading any required run lists, attributes, and cookbooks. However, it will allow for interactive execution so that it is possible to debug or diagnose issues with recipes on the endhost. When using the client mode, you can use an alternate configuration file with the `-c` command-line option, or specify a different Chef server URL via the `-s` command-line option.

The example uses the following:

```
chef-shell --client -c /etc/chef/alternate.conf
chef-shell --client -s http://test.server.url:8080/
```

Interacting with the Chef server using the shell

The Chef shell provides you with the ability to interact with the server quickly in the same way you would use `knife`, but without the overhead of typing `knife search node...` or `knife node list`, and so on. It is a very convenient way to query the data stored in the Chef server interactively. In order to interact with the server from your workstation, you need to make sure that your shell's configuration file, located in `~/.chef/chef_shell.rb`, is configured properly. If you are connecting with `chef-shell` from a node, then the configuration in `/etc/chef/client.rb` (or similar on Windows) will be used instead.

This file, similar to the `knife.rb` or `client.rb` file, contains the required certificate data and configuration data to connect to the Chef server. An example configuration file will resemble the following, with paths, organization, and client names updated accordingly:

```
node_name          'myorg'
client_key        File.expand_path('~/Chef/client.pem')
validation_key    File.expand_path('~/Chef/validator.pem')
validation_client_name "myorg-validator"
chef_server_url   'https://api.opscode.com/organizations/myorg'
```

All of these files are present if your `knife` installation is operational, and the configuration file closely resembles that of `knife.rb`—if you need values for these on your workstation, take a look at the `~/.chef/knife.rb` file. Once you have configured your shell, you can pass the `-z` command-line flag to connect as the chef-client would:

```
[jewart]% chef-shell -z
loading configuration: /Users/jewart/.chef/chef_shell.rb
Session type: client
Loading.....resolving cookbooks for run list: []
Synchronizing Cookbooks:
done.

This is the chef-shell.
Chef Version: 11.12.8
http://www.opscode.com/chef
http://docs.opscode.com/

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

Ohai2u jewart@!
```

Interacting with data

From here, you can interact with the Chef server in a variety of ways, including searching, modifying, and displaying any data elements (roles, nodes, data bags, environments, cookbooks, and clients), performing a client run (including stepping through it, one step at a time), assuming the identity of another node, and printing the attributes of the local node. For example, listing the roles on the Chef server can be performed with the `roles.all` method, shown as follows:

```
chef > roles.all
=> [role[umbraco cms], role[umbraco], role[base_server], role[web_
server], role[postgresql_server]]
```

Searching your data

Searching the data elements is also supported, as each data type has a `find` method attached to it. The `find` method takes a map of the attribute and pattern to look for and returns the results. For example, you can find all roles on the Chef server that begin with "um" with the following command:

```
chef > umbraco_roles = roles.find(:name => "um*")
=> [role[umbraco_cms], role[umbraco]]
```

Editing your data

Any object in your Chef server can be edited directly from the Chef shell using the `edit` command from inside the shell. This will invoke your favorite editor to edit the raw JSON of the object in question, which provides a more direct mechanism over using `knife (node|role|data bag) edit` on the command line, as you can quickly manipulate a number of records a lot more easily. For example, to edit all of the roles that contain the name "apache" and save the results, you can use the following Ruby code:

```
chef > apache_roles = roles.find(:name => "*apache*")
> [... some list... ]
chef> apache_roles.each do |r|
chef>   updated = edit r
chef>   updated.save
chef> end
```

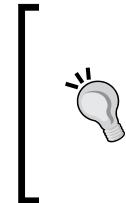
This will find all roles whose name contains "apache". Then for each record, edit the JSON, storing the results in variable named `updated`, and then save that record back to the Chef server.

Transforming data

In this way, you can interact with any of the resources that are available to you, allowing you to quickly find and manipulate any data stored in Chef directly using the Ruby code. For example, to find all clients with a given string in their name and disable their administrative access, you can use the following code:

```
clients.transform("*:*") do |client|
  if client.name =~ /bad_user/i
    client.admin(false)
  true
```

```
else
  nil
end
end
```



Use caution when transforming your data from the Chef shell; it is an incredibly powerful tool, but its effects are destructive. These changes are not reversible (at least not without forethought or backups) and could damage your data if you are not careful. For example, if the previous code was transcribed incorrectly, it could potentially render all users unable to administer the system.

Executing recipes with Chef shell

Two great features of `chef-shell` are the ability to rewind a run (to step backwards) and to be able to step forward in the run one resource at a time. As an example, let's look at how to define a simple recipe in `chef-shell` interactively and then run it, start it over, and step through it.

First, let's fire up `chef-shell` with the following command:

```
[jewart]% chef-shell
loading configuration: none (standalone session)
Session type: standalone
Loading.....done.
```

```
This is the chef-shell.
Chef Version: 11.12.8
http://www.opscode.com/chef
http://docs.opscode.com/

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

```
Ohai2u jewart@!
chef >
```

The `chef-shell` prompt will change based on the state you are in. If you are working with a recipe, the prompt will change to be `chef:recipe >`.

Creating a recipe in the shell

The Chef shell has a number of modes – recipe mode and attribute mode. Recipe mode is activated when working with recipes and will be what we use here. In order to activate it, type `recipe_mode` at the prompt:

```
chef > recipe_mode
chef:recipe >
```

Here, we will create resources to create a file in the current directory interactively using a `file` resource with no associated configuration block, only the name:

```
chef:recipe > file "foo.txt"
=> <file[foo.txt] @name: "foo.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'" @
guard_interpreter: :default @elapsed_time: 0 @resource_name: :file @path:
"foo.txt" @backup: 5 @atomic_update: true @force_unlink: false @manage_
symlink_source: nil @diff: nil @sensitive: false @cookbook_name: nil @
recipe_name: nil>
```



One thing to note here is that the shell will print out the results of the last operation executed in the shell. This is part of an REPL shell's implicit behavior; it is the print part of REPL: input is read and evaluated, then the results are printed out, and the shell loops to wait for more input from the user. This can be controlled by enabling or disabling the echo state; `echo off` will prevent the printed output and `echo on` will turn it back on.

It is critical to note that, at this point, nothing has been executed; we have only described a `file` resource that will be acted upon if the recipe is run. You can verify this by making sure that there is no file named `foo.txt` in the directory you executed `chef-shell` from. The recipe can be run by issuing the `run_chef` command, which will execute all of the steps in the recipe from start to finish. Here is an example of this:

```
chef:recipe > run_chef
INFO: Processing file[foo.txt] action create ((irb#1) line 1)
DEBUG: touching foo.txt to create it
INFO: file[foo.txt] created file foo.txt
DEBUG: found current_mode == nil, so we are creating a new file, updating
mode
```

```

DEBUG: found current_mode == nil, so we are creating a new file, updating mode
DEBUG: found current_uid == nil, so we are creating a new file, updating owner
DEBUG: found current_gid == nil, so we are creating a new file, updating group
DEBUG: found current_uid == nil, so we are creating a new file, updating owner
INFO: file[foo.txt] owner changed to 501
DEBUG: found current_gid == nil, so we are creating a new file, updating group
INFO: file[foo.txt] group changed to 20
DEBUG: found current_mode == nil, so we are creating a new file, updating mode
INFO: file[foo.txt] mode changed to 644
DEBUG: selinux utilities can not be found. Skipping selinux permission fixup.

```

Defining node attributes

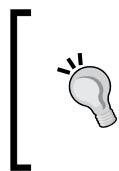
Just as in any recipe, attributes can be used in the recipes defined in the shell. However, in the standalone mode, there will be no attributes defined initially; solo and client modes will likely have attributes defined by their JSON file or the Chef server, respectively. In order to interact with the currently defined attributes, we must switch between the recipe mode and attribute mode. This is achieved using the `attributes_mode` command as shown in the following code:

```

chef:recipe > attributes_mode
chef:attributes >

```

Here we can perform two primary operations: getting and setting node attributes. These are ways of modifying the values that are accessed from inside the node Mash in a recipe.



Remember that the node's attributes are accessed as a Mash, a key-insensitive hash that allows you to interchange string keys with symbol keys. The Mash class is not a built-in structure in Ruby—it is provided by Chef for convenience so that hash keys can be either symbols or strings and have the same effect.

Setting attributes

Setting attributes is achieved using the `set` command, which has the following form:

```
set [:key] = value
```

Here, `:key` can be a single-level key or a multilevel key similar to any entry in the `attributes/default.rb` file. As an example, we can construct an application configuration using the following:

```
set [:webapp] [:path] = "/opt/webapp"
set [:webapp] [:db] [:username] = "dbuser"
set [:webapp] [:db] [:password] = "topsecret"
set [:webapp] [:user] = "webuser"
set [:postgresql] [:config] [:listen] = "0.0.0.0"
```

Any parent keys that are non-existent are implicitly created on the fly, so you do not need to do something like the following:

```
set [:webapp] = {}
set [:webapp] [:path] = "/opt/webapp"
```

Accessing attributes

In order to display an attribute when in the attributes mode, simply type in the name of the key you are interested in. For example, if you had executed the `set` commands listed previously, then asking for the `webapp` hash is as simple as typing `webapp`, as follows:

```
chef:attributes > webapp
=> {"path"=>"/opt/webapp", "db"=>{"username"=>"dbuser",
"password"=>"topsecret"}, "user"=>"webuser"}
```

However, if you wish to access these when in the recipe mode, they are accessed through the `node` hash, as shown here:

```
chef:attributes > recipe_mode
=> :attributes
chef:recipe > node[:webapp]
=> {"path"=>"/opt/webapp", "db"=>{"username"=>"dbuser",
"password"=>"topsecret"}, "user"=>"webuser"}
```

They can be used via the `node` hash in just the same way you would use them in a recipe. If you want to construct a file block that created a `foo.txt` file located in the install path of our `webapp` hash, you can easily use the following example inside your shell:

```
file "#{node[:webapp][:path]}/foo.txt"
```

This makes writing recipes using the interactive shell feel exactly the same as writing recipe files.

Using configuration blocks

A resource in a recipe file can have a Ruby block with attributes, and you can do this in `chef-shell` in exactly the same fashion. Simply insert `do` after the resource name and the shell will behave as a multiline editor, allowing you to complete the block. The following example demonstrates providing a `content` attribute to a `file` resource in this manner:

```
chef:recipe > file "not_empty.txt" do
chef:recipe > content "Not empty!"
chef:recipe ?> end
=> <file[not_empty.txt] @name: "not_empty.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):2:in 'irb_binding'" @guard_interpreter: :default @elapsed_time: 0 @resource_name: :file @path: "not_empty.txt" @backup: 5 @atomic_update: true @force_unlink: false @manage_symlink_source: nil @diff: nil @sensitive: false @cookbook_name: nil @recipe_name: nil @content: "Not empty!">
```

Note that when the shell printed out the previous `file` resource, `@content` was not present. Here, everything but the name remains the same, and there is an additional property inside the object, `@content`, as specified in our attributes block.

Interactively executing recipes

Running a recipe step by step is a good way of slowing down the execution of a recipe so that the state of the system can be inspected before proceeding with the next resource. This can be incredibly useful both for debugging (as will be discussed later) and for developing and exploring resources. It gives you a chance to see what has happened and what side effects your recipe has as the recipe is executed. To achieve this, the Chef shell allows you to *rewind* your recipe to the start and run from the beginning, execute your recipe one *step* at a time, and *resume* execution from the current point to the end.

Restarting our Chef shell, let's take a look at how we can use this:

```
recipe_mode
echo off
file "foo.txt"
file "foo.txt" do
  action :delete
end
file "foo.txt" do
  content "Foo content"
end
```

Here our recipe is quite simple—create an empty file, `foo.txt`, remove it, and then recreate it with "Foo content". If we execute our recipe using `run_chef`, the shell will perform all the operations in one pass without stopping and will not allow us to check whether the delete action occurred. Instead, we can run our recipe and then rewind and use the `chef_run.step` method to interactively walk through our recipe:

```
chef:recipe > run_chef
... execution output ...
chef:recipe > echo on
=> true
chef:recipe > chef_run.rewind
=> 0
chef:recipe > chef_run.step
INFO: Processing file[foo.txt] action create ((irb#1) line 3)
=> 1
chef:recipe > chef_run.step
INFO: Processing file[foo.txt] action delete ((irb#1) line 4)
INFO: file[foo.txt] backed up to /var/chef/backup/foo.txt.chef-
20140615175124.279917
file[foo.txt] deleted file at foo.txt
=> 2
chef:recipe > chef_run.step
INFO: Processing file[foo.txt] action create ((irb#1) line 7)
INFO: file[foo.txt] created file foo.txt
INFO: file[foo.txt] updated file contents foo.txt
```

```
INFO: file[foo.txt] owner changed to 501
INFO: file[foo.txt] group changed to 20
INFO: file[foo.txt] mode changed to 644
=> 3
```

As you can see, here we were able to rewind our recipe back to the first instruction (position 0, as the result of `chef_run.rewind` indicates), and then walk through each resource step by step using `chef_run.step` and see what happened. During this run, you can easily open a terminal after you rewind the recipe, delete the `foo.txt` file from the previous run, and check that initially there is no `foo.txt` file, then step through the next command in the recipe, validate that there is an empty `foo.txt` file, and so on. This is a very good way to learn how resources work and to see what they do without having to formalize your recipe in a cookbook, provision and bootstrap a host, and so on.

Debugging with the Chef shell

Debugging is achieved in two different ways using `chef-shell`: stepping interactively through a recipe or using a special breakpoint resource that is only used by `chef-shell`. Running recipes interactively step by step is good to build recipes locally; experiment with resources to determine the effect of certain attributes, actions and notifications; or to inspect the state of the system after each resource has been acted upon. The breakpoints allow you to inject very specific stopping points into the client run so that the world can be inspected before continuing. Typically, once a breakpoint is encountered, you will want to step through the execution of your script (at least for a while) so that these are not mutually exclusive techniques.

Using the breakpoint resource

The breakpoint resource is structured just like any other Chef resource. The resource's name attribute is the location where you want to insert the breakpoint, and it has only one action, `:break`, which signals `chef-shell` to interrupt execution of the current recipe and provide an interactive shell. Any breakpoint resources in recipes are ignored by the chef-client. That way, if they are forgotten about and left in a recipe, they will not cause havoc in production. That being said, they should only be used when actively debugging an issue and removed before releasing your recipes into your production environment.

The name attribute has the following structure:

```
when resource resource_name
```

Here, `when` has the value of "before" or "after", to indicate whether the breakpoint should stop before or after execution, respectively and `resource` is the type of resource that when combined with `resource_name` is the unique identifier that will trigger the breakpoint. For example:

```
before file '/tmp/foo.txt'
```

This would cause the shell to interrupt execution of the recipes immediately before any file resource that was manipulating `/tmp/foo.txt`. Another example, where we want to stop execution after installing the `git` package, would look like the following:

```
after package 'git'
```

Using this, we will tell `chef-shell` that execution was to be paused once the `git` package was modified. Let's look at how we can form a simple recipe complete with breakpoint resources that would use these examples:

```
breakpoint "before file '/tmp/foo.txt'" do
  action :break
end

breakpoint "after package 'git'" do
  action :break
end

file '/tmp/foo.txt' do
  action :create
end

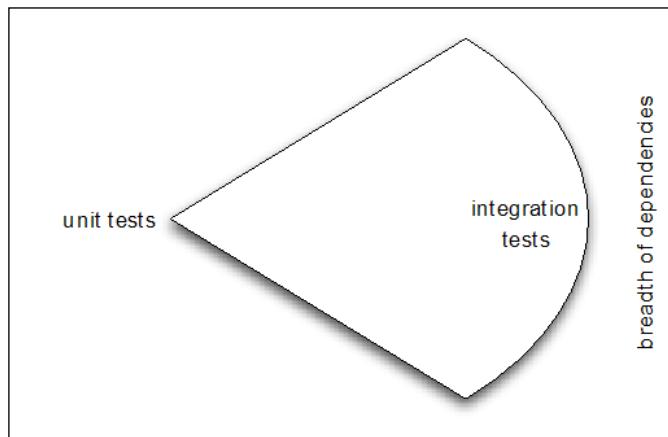
package 'git' do
  action :remove
end
```

For those who have used `gdb` or any other debugger, this will be easy to understand; if you have not used an interactive debugger, then try a few of the interactive examples, and you will get the hang of it in no time at all.

Chef shell provides a comprehensive way to interact with your recipes. Now that you see how to test out and debug your work, let's take a look at how we can go one step further in our testing to perform full end-to-end integration testing of our infrastructure.

Integration testing

With integration testing, your tests move into testing beyond your code. With ChefSpec, we looked at how to verify that a recipe was executing statements as expected. However, unit testing has its limits; integration testing completes the picture by testing cookbooks in conjunction with real hosts that run the desired operating system(s):



This means testing cookbooks from the outside rather than from the inside, where a unit test provides very narrow testing of code inside of cookbooks, integration tests perform deployment of your cookbooks in a test-specific environment and then execute probes to validate that the system is in the desired state. One of the most popular tools for this is **Test Kitchen**, which we will take a brief tour of.

Using Test Kitchen

Test Kitchen is an open source tool that helps to automate integration testing of Chef cookbooks. Where a ChefSpec test would validate that a `file` resource was called during an execution, a similarly tasked Test Kitchen test would spin up a new instance, execute your recipe(s), and then validate that the file was actually created on all platforms that you expect the recipe to work on. One of the great things about Test Kitchen is that it supports a number of different **driver** plugins to manage your target hosts, including Vagrant (the default), EC2, OpenStack, Docker, and Rackspace Cloud among others. This enables you to test your cookbooks not only on local virtual machines using Vagrant, but to verify that they work correctly on a cloud service as well. The ability to perform integration tests on different types of hosts brings your tests that much closer to matching a production environment, thereby increasing your confidence in the changes you are making.

Installing Test Kitchen

Currently, Test Kitchen is distributed as a Ruby gem, so installation is quite straightforward:

```
gem install test-kitchen
```

By installing this gem, you are also installing a command-line tool, `kitchen`, which is used to interact with Test Kitchen. Similar to Vagrant, Bundler, and other tools, Test Kitchen uses a configuration file to store information about what to test, where to test it (on what virtual machines), and how to test it.

Testing with Test Kitchen

As mentioned before, Test Kitchen focuses on integration testing of your Chef components. This means that it needs to be able to execute your recipes on a host (or a set of hosts) and then invoke a set of tests to validate that the expected behavior occurred on the endhost(s). Test Kitchen is responsible for the following:

- Provisioning a clean host for testing
- Installing Chef onto the new host
- Executing our recipes on the host
- Validating the behavior of the recipes on the host

So, let's get started with some actual testing!

Building a simple cookbook

In order to demonstrate how to use Test Kitchen, we will need to write a cookbook that we can run on our test hosts and write tests to validate the behavior. In order to focus on how to use Test Kitchen, we will take a look at a very simple cookbook that just creates a file on the host filesystem so that it is guaranteed that it will work on both Ubuntu and CentOS platforms.

First, create a place to do your work:

```
mkdir -p testfile-cookbook/recipes
```

Then, add a very simple `metadata.rb` in your `testfile-cookbook` directory with the following contents inside:

```
name "testfile"
version "1.0"
```

Once that is complete, add a default recipe, `recipes/default.rb`, with the sample recipe code as follows:

```
file "/tmp/myfile.txt" do
  content "My awesome file!"
end
```

Now that we have a complete (albeit simple) cookbook, let's take a look at how to test it using Test Kitchen.

Preparing your cookbook for the kitchen

In order to start using Test Kitchen, you will need to prepare your test environment. Test Kitchen relies on a configuration file, `.kitchen.yml`, to tell it what to do. You can generate it by hand, or you could use the `init` command as part of the `kitchen` tool:

```
kitchen init
```

This will do a few things for you: first, it will create a `.kitchen.yml` file with some sane defaults in the current directory. Then, it will create a `test/integration/default` directory for your integration tests, and then it will install the Vagrant driver for Test Kitchen so that it can interact with Vagrant virtual machines (if it has not already been installed).

If you look at the `.kitchen.yml` file, you will see that the initial file contains the following YAML code:

```
---
driver:
  name: vagrant

provisioner:
  name: chef_solo

platforms:
  - name: ubuntu-12.04
  - name: centos-6.4

suites:
  - name: default
    run_list:
      - recipe[testfile::default]
    attributes:
```

This configuration file instructs Test Kitchen to use Vagrant to manage the target instances and to use Chef-solo for provisioning, and it should execute the default suite of tests on both Ubuntu 12.04 and CentOS 6.4. Of course, you can always modify or extend this list if you have other systems that you want to run tests on, but this is a reasonable default list for now.

Notice that we don't have any attributes specified as our default recipe and it does not use attributes. If you need to provide attributes to test recipes, this would be the place to do it, which is laid out as a dictionary in YAML. Each test suite has its own run list and defined attributes that allow you to check the behavior of a variety of configuration data and recipe combinations.

Testing your new cookbook

Testing a cookbook with Test Kitchen is outlined in the following three steps:

1. Provisioning a host if needed.
2. Converging the host so that it is up to date.
3. Executing tests.

Let's take a look at how we will perform these with our simple cookbook to check that it works properly.

Provisioning the instance

Before you can test, you will need to prepare the instances for testing – this is done using the `kitchen create <instance name>` command. Only you don't know what instance to bring up just yet. To get the list of instances that can be run, we will use the `list` subcommand:

```
[jewart]$ kitchen list
Instance          Driver  Provisioner  Last Action
default-ubuntu-1204  Vagrant  ChefSolo    <Not Created>
default-centos-64    Vagrant  ChefSolo    <Not Created>
```

You will see that this list is generated by a combination of the platforms and the suites listed in the `.kitchen.yml` file. If you were to define a new suite, named `server`, then your list would include two additional instances, `server-ubuntu-1204` and `server-centos-64`.

Once you have seen this list, you can create an Ubuntu 12.04 instance with the following command:

```
kitchen create default-ubuntu-1204
```

This will use Vagrant and VirtualBox to provision a new headless Ubuntu 12.04 host and boot it up for you to start testing with. If you don't already have an Ubuntu 12.04 Vagrant image downloaded, then it will be downloaded for you automatically (this is a large image so it may take a while to complete this operation, depending on your connection speed, if you are following along).

This will look familiar to those that used Vagrant at the beginning of the chapter:

```
[jewart]% kitchen create default-ubuntu-1204
-----> Starting Kitchen (v1.2.1)
-----> Creating <default-ubuntu-1204>...
Provisioning happens...

      Finished creating <default-ubuntu-1204> (4m2.59s).
-----> Kitchen is finished. (4m2.83s)
```

However, this only builds the virtual machine for our tests; it does not run our recipes or our tests in the newly constructed host. Before we move on to writing a test, let's take a look at how to run our recipe inside our instance.

Converging the newly created instance

The next step is to execute our run list ("converge" in Chef parlance) on the new instance. This is done with Test Kitchen's converge command and can be used for all or one specific instance. In order to converge our Ubuntu 12.04 instance, the following command is used:

```
kitchen converge default-ubuntu-1204
```

What this will do is transfer any required data files, install Chef as needed, and then execute the run list for the specified suite (default in this case) on the instance. Here is a sample run of converge (with some parts removed):

```
[jewart]% kitchen converge default-ubuntu-1204
-----> Starting Kitchen (v1.2.1)
-----> Converging <default-ubuntu-1204>...
      Preparing files for transfer
      Preparing current project directory as a cookbook
      Removing non-cookbook files before transfer
```

```
-----> Installing Chef Omnibus (true)
```

```
Installation of Chef happens...
```

```
Compiling Cookbooks...
```

```
Converging 1 resources
```

```
Recipe: testfile::default
```

And then you will see the output of a normal Chef run after that—at this point, the run list is complete and the instance has been converged to the latest state. Now that it's been converged, we will continue to write a very simple test to verify that our recipe did the right thing.

Writing a simple test

Tests are stored in a directory whose structure is similar to other Chef components—the directory we created previously with `kitchen init`, `tests/integration/default`, allows us to keep our integration tests separate from spec tests or other types of tests. The `integration` directory will contain one directory per suite so that test files are grouped together based on the particular component or aspect of your cookbook that is being tested. Additionally, depending on the type of the test framework being used, your tests will be contained in another child directory for the given suite. In this case, we will take a look at the **BASH Automated Testing System (BATS)**, so our test will be placed in the `tests/integration/default/bats/file_created.bats` file and will look like the following code:

```
#!/usr/bin/env bats

@test "myfile.txt exists in /tmp" {
    [ -f "/tmp/myfile.txt" ]
}
```

This allows us to use the simple `-f` BASH test (which returns `true` if the specified value exists) to guarantee that the file was created on the instance.

Next, we can run this test with `kitchen verify default-ubuntu-1204` and see that the BATS plugin was installed and that our test was executed and passed:

```
[jewart] % kitchen verify default-ubuntu-1204
-----> Starting Kitchen (v1.2.1)
-----> Setting up <default-ubuntu-1204>...
Fetching: thor-0.19.0.gem (100%)
Fetching: busser-0.6.2.gem (100%)
```

```
Successfully installed thor-0.19.0
Successfully installed busser-0.6.2
2 gems installed
----> Setting up Busser
      Creating BUSSER_ROOT in /tmp/busser
      Creating busser binstub
      Plugin bats installed (version 0.2.0)
----> Running postinstall for bats plugin
Installed Bats to /tmp/busser/vendor/bats/bin/bats
      Finished setting up <default-ubuntu-1204> (0m8.94s).
----> Verifying <default-ubuntu-1204>...
      Suite path directory /tmp/busser/suites does not exist, skipping.
Uploading /tmp/busser/suites/bats/file_created.bats (mode=0644)
----> Running bats test suite
✓ myfile.txt exists in /tmp

1 test, 0 failures
      Finished verifying <default-ubuntu-1204> (0m0.67s).
----> Kitchen is finished. (0m9.86s)
```

To demonstrate what happens if the file does not exist, we can clone our test to create a second simple test that validates the existence of /tmp/myotherfile.txt, and run our verify command again without making a corresponding change to our recipe. The output from Test Kitchen will tell us that our test failed and why:

```
----> Running bats test suite
✓ myfile.txt exists in /tmp
✗ myotherfile.txt exists in /tmp
      (in test file /tmp/busser/suites/bats/other_file.bats, line 4)

2 tests, 1 failure
Command [/tmp/busser/vendor/bats/bin/bats /tmp/busser/suites/bats] exit
code was 1
>>>>> Verify failed on instance <default-ubuntu-1204>.
```

Combining all the steps

Fortunately, the fine folks that created Test Kitchen realized that it would be tedious to run all three steps every time you wanted to run some tests. As a result, there is the `kitchen test` command that will provision an instance, execute the run list, verify the results, and then tear down the instance with only one command. In this case, you can replace them with the following single command:

```
kitchen test default-ubuntu-1204
```

This covers the basics of testing your cookbooks with Test Kitchen. There are other things that can be done with Test Kitchen, including using other testing mechanisms, testing cookbook dependencies, validating whether services are running, fully automating Test Kitchen as part of your release process, and plenty more. For more information, visit the project at <http://kitchen.ci/>.

Extending Chef

Chef is developed in the open with flexibility and extensibility in mind. Most of the tools are architected to support loading custom plugins to support development of add-ons for new functionality. As you saw earlier, Knife's cloud service support is provided by plugins, one for each cloud service, including EC2, Azure, and Rackspace cloud. We will look at how that happens so that you can explore writing your own plugins for Knife, Ohai, and other Chef components should the need arise.

In addition to extending Chef's core components directly, it is possible to extend the functionality of your Chef ecosystem by building enhancements to existing tools that can leverage Chef's data APIs to provide data about your infrastructure.

Writing an Ohai plugin

Ohai (a play on the phrase oh, hi!) is a tool that is used to detect attributes on a node, and provide these attributes to the chef-client at the start of every `chef-client` run. Without Ohai, the `chef-client` will not function, and therefore, it must be present on a node in order for Chef to work. The data that is collected is authoritative—it has the highest level of precedence when computing attribute data for client runs on a node.

The types of attributes that Ohai might be used for include:

- Platform details
- Network usage
- Memory usage
- Processor usage

- Kernel data
- Host names
- Information about the network topology
- Cloud-specific information

Ohai implements an extensible architecture through plugins that allows end users to write custom extensions to report information that is collected about a node. For example, there are plugins for EC2 cloud hosts that use EC2-specific mechanisms to determine information about the host, including its internal IP address and other bits of information.

This is incredibly useful for integrating Chef with an existing infrastructure, as you can automatically probe for local configuration data and generate attributes from that data. Once this data is stored in Chef, it can be used in search queries, recipes, and everywhere else that you could otherwise use node attributes.

Every Ohai plugin follows a pattern: it registers itself as providing a certain class of attribute data and contains a combination of general purpose and data collection methods to gather information about the local state of the system and report it back to the Chef server. Ohai already has built-in collectors for a number of platforms, including Linux, Windows, and BSD.

A sample Ohai plugin would look like the following:

```
Ohai.plugin(:Region) do
  provides "region"

  def init
    region Mash.new
  end

  collect_data(:default) do
    # Runs on all hosts whose platform is not specifically handled
    init
    region[:name] = "unknown"
  end

  collect_data(:linux, :freebsd) do
    # Run only on Linux and FreeBSD hosts
    init
    region[:name] = discover_region_unix
  end

  collect_data(:windows) do
```

```
# Run on Windows hosts
init
  region[:public_ip] = discover_region_windows
end

end
```

Here we register our plugin as providing a `region` data, there is an `init` method that creates a `Mash` for our `region` data, and a few data collection callbacks. Each data collection callback is registered as a block that is called for the specified platform(s). In our case, there are three callbacks registered, one for Windows systems, one for Linux and FreeBSD systems, and then a fallback that will be called for any platform not explicitly handled.

A note about writing Ohai plugins

The way we declare the `region` `Mash` in our `init` method is a little bit different than normal variable assignment in Ruby. In our plugin, we define the plugin's `region` property with the following code:

```
region Mash.new
```

On the surface, this might look like someone forgot an equals sign, as in the following Ruby code:

```
region = Mash.new
```

However, if there were an equals sign, the plugin would not work as intended. In this case, there is no equals sign missing and the code is correct. This is because Ohai's `Plugin` class leverages a special Ruby mechanism for intercepting calls to nonexistent methods and dynamically handling them. This mechanism is recognizable by the presence of a special method named `method_missing` in the code. In this case, the `method_missing` handler will call a special Ohai plugin method, `get_attribute`, if no arguments are passed, or it will call the `set_attribute` method if arguments are passed.

To demonstrate why this is used, if you wanted to have the same effect without the `method_missing` mechanism, then the plugin's `init` method could be written as:

```
def init
  set_attribute "region", Mash.new
end
```

Were you to do this, then our subsequent collect data methods would need to be rewritten as well. Here is an example of what they might look like:

```
collect_data(:platform) do
  init
  region = get_attribute "region"
  region[:name] = get_data_mechanism
  set_attribute "region", region
end
```

You can see that the `method_missing` mechanism makes writing plugins more natural, though it takes a little bit of extra work to understand how to write them at first.

Chef with Capistrano

One example of extending Chef through external tools is the `capistrano-chef` gem that extends the popular deployment tool, Capistrano. Written in Ruby, Capistrano was designed to deploy applications and perform light systems administration. If you have existing applications that are being deployed using Capistrano, this is an example of how to leverage your configuration data stored in Chef to make integration as seamless as possible.

If you have an existing application that uses Capistrano, you will have a `deploy.rb` file that defines the various application roles. Each tool has an array of IP addresses to hosts that provide that role and might look something like this:

```
role :web, '10.0.0.2', '10.0.0.3'
role :db, '10.0.0.2', :primary => true
```

By using `capistrano-chef`, you can do this:

```
require 'capistrano/chef'
chef_role :web, 'roles:web'
chef_role :db, 'roles:database_master',
           :primary => true,
           :attribute => :private_ip,
           :limit      => 1
```

Notice that here we have used a simple search query to determine the hosts that should be included in each Capistrano role. In this case, the `:web` role has been replaced with a Chef search query for all nodes that have the `web` Chef role associated with them. This allows you to model your data in Chef, but still use Capistrano to deploy your application stack, increasing the ease of integration.

This works because all of Chef's data is available via an HTTP API making integration as simple as making HTTP calls and parsing some JSON results (which, compared to some other integration mechanisms, is incredibly easy).

Automation and integration

One of the best parts about Chef is that your infrastructure and software model is consistent with what is deployed. What this means for you is that when chef-client runs on an endhost, that host's state is updated to match your modeled environment. For example, consider a scenario where you have 10 EC2 database hosts, and all of them have a special role, `database_server`, applied to them. This role's attributes indicate that PostgreSQL 9.1 is to be installed and its data should be stored in `/opt/postgresql/data`. By executing `chef-client` on all ten nodes, they will have PostgreSQL 9.1 installed and storing data in `/opt/postgresql/data`. Now consider that all of our nodes need to have a new EBS storage device attached to each of them, and PostgreSQL needs to be pointed to our new EBS device. Updating our model to include a recipe that mounts the EBS device gracefully shuts down PostgreSQL, moves the database data, reconfigures PostgreSQL, and starts it up again. We can automate and roll out this configuration to our fleet of ten database hosts. You can easily imagine ten hosts growing to hundreds or even thousands. This is what the power of automation is all about.

Automated updates and deployments

If you have confidence in your model and your cookbooks, then you can take this automation one step further. By automating the execution of `chef-client` on a periodic schedule, you can fully automate updates without needing to SSH into the hosts to run `chef-client`. However, this level of automation requires a high level of confidence in the correctness of your cookbooks. Achieving this requires continuous and in-depth testing of not only the code in the cookbooks developed but also of the dependencies that are needed to make your cookbooks work. To that end, comprehensive integration tests can help to build the confidence needed to move into a fully automated world.

Summary

By now you have been exposed to a lot of what Chef has to offer the DevOps community. You have seen what Chef does, how to install it, and how it works. Throughout this book, you have been introduced to some new ways of thinking about how to model infrastructure and use automated tools to manage it.

At this point, you hopefully understand how to model your infrastructure with Chef as well as install the various components related to Chef, ranging from the server to the client. From here, you can take what you've learned about the various components of Chef and use that information to build more advanced cookbooks to deploy your software and manage your infrastructure, ranging from cloud hosts to physical on-site hardware and even virtual machines using Vagrant. Once you have gotten things working, you can automate your configuration tools and ensure the reliability of your cookbooks through unit and integration tests as well.

Now, it is your turn to take this information and your new skills to automate your systems infrastructure in order to build exciting new things!

Module 2

Chef Infrastructure Automation Cookbook - Second Edition

*Over 80 recipes to automate your cloud and server infrastructure with
Chef and its associated toolset*

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 the 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
- ▶ Downloading and integrating cookbooks as vendor branches into your Git repository
- ▶ Using custom knife plugins
- ▶ Deleting a node from the Chef server
- ▶ Developing recipes with local mode
- ▶ Using roles
- ▶ Using environments
- ▶ Freezing cookbooks
- ▶ Running Chef client as a daemon
- ▶ Using chef-shell

Introduction

This chapter will cover the basics of Chef, including common terminology, workflow practices, and various tools that work in accordance with Chef. We will explore version control using Git, walk through working with community cookbooks, and running those cookbooks on your own servers, so that you can configure them in the way you need them.

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, such as installing MySQL, the most important one being 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 ways 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 part that retrieves the cookbooks from the Chef server and executes 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 taking a look 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 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 with them.

Using version control is a fundamental part of any infrastructure automation. There are multiple solutions (some free, some paid) 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](https://github.com) and create an (free) account 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> while you're still logged in and using `chef-repo` as the repository name.

Make sure you have `wget` installed on your local workstation, in order 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...  
2014-11-30 22:00:43 (1.30 MB/s) - 'master' saved [9309/9309]
```

2. Extract the downloaded tarball:

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

3. Rename the directory:

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

4. Change to your newly created Chef repository:

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

5. Initialize a fresh Git repository:

```
mma@laptop:~/chef-repo $ 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 your repository, added a new cookbook called `other_cookbook`, committed their changes locally, and pushed their changes back to GitHub. Now, it's time for you to get the new cookbook downloaded on 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 case 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/54/articles>
- ▶ You'll use more Git features in the *Downloading and integrating cookbooks as vendor branches into your Git repository* recipe in this chapter

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 'rubocop' succeeded.
Verification of component 'kitchen-vagrant' succeeded.
Verification of component 'chefspec' succeeded.
Verification of component 'berkshelf' succeeded.
Verification of component 'fauxhai' succeeded.
Verification of component 'test-kitchen' succeeded.
Verification of component 'package installation' succeeded.
Verification of component 'chef-dk' succeeded.
Verification of component 'knife-spork' succeeded.
Verification of component 'chef-client' succeeded.
```

3. Add the newly installed Ruby to your path:

```
mma@laptop:~ $ echo 'export PATH="/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 the 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, Linux, and Windows at <https://learn.chef.io>
- ▶ Find ChefDK on GitHub at <https://github.com/opscode/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 the 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 trial or a free account.

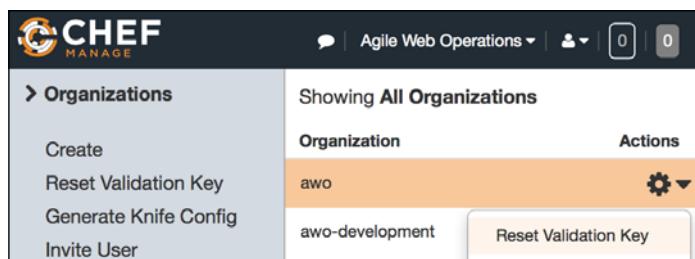
I registered as the user webops with an organization short name of awo.

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

How to do it...

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

1. Create the configuration directory for your Chef client on your local workstation:
`mma@laptop:~/chef-repo $ mkdir .chef`
2. Navigate to <http://manage.chef.io/organizations>. After logging in, you can start downloading your validation keys and configuration file.
3. Select your organization to be able to see its contents using the web UI.



The screenshot shows the Chef Manage web interface. At the top, there is a navigation bar with icons for messaging, Agile Web Operations, users, and notifications. Below the header, the left sidebar has a 'CHEF MANAGE' logo and a 'Organizations' section with options: Create, Reset Validation Key, Generate Knife Config, and Invite User. The main content area is titled 'Showing All Organizations' and lists two organizations: 'awo' and 'awo-development'. The 'awo' row is highlighted with an orange background, and an 'Actions' button is visible next to it. A 'Reset Validation Key' button is also present in the bottom right corner of the organization list.

Regenerate the validation key for your organization and save it as <your-organization-short-name>-validator.pem in the chef directory inside your chef-repo repository.



4. Generate the knife config and put the downloaded knife.rb into the .chef directory inside your chef-repo directory, as well. Make sure you have downloaded your user's private key from <https://www.chef.io/account/password> and replace webops with the username you chose for hosted Chef, and awo with the short name you chose for your organization:

```
current_dir = File.dirname(__FILE__)
log_level           :info
log_location        STDOUT
node_name          "webops"
client_key         "#{current_dir}/webops.pem"
validation_client_name "awo-validator"
validation_key      "#{current_dir}/awo-validator.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"]
```

[Take a look at the following code:

 .chef/*.pem
.chef/encrypted_data_bag_secret

You should add the preceding code to your .gitignore file inside chef-repo to avoid your credentials from ending up in your Git repository.]

5. Use knife to verify that you can connect to your hosted Chef organization. It should only have your validator client, so far. Instead of awo, you'll see your organization's short name:

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

How it works...

Hosted Chef uses two private keys (called validators):

- ▶ one for the organization
- ▶ one for every user.

You need to tell knife where it can find these two keys in your `knife.rb` file.

The following two lines of code in your `knife.rb` file tell the knife about which organization to use and where to find its private key. The `validation_key` is used to allow new clients to authenticate the Chef server before getting their own Client key:

```
validation_client_name      "awo-validator"  
validation_key              "#{current_dir}/awo-validator.pem"
```

The following line of code in your `knife.rb` file tells the knife where to find your users' private key. It is used by your local workstation to authenticate 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:

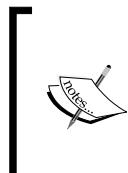
```
chef_server_url            "https://api.chef.io/organizations/awo"
```

Using the `knife.rb` file and your two validators knife, you can now connect to your organization hosted by Chef Software, Inc.

You do not need your own self-hosted Chef server, nor do you need to use Chef client local mode in this setup.

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 of the Chef server on GitHub at <https://github.com/chef/chef>

Managing virtual machines with Vagrant

Developing Chef cookbooks requires you to run your work in progress cookbooks multiple times on your nodes. To make sure they work, you need a clean, initial state of your nodes every time you run them. You can achieve this by using a **virtual machine (VM)**. However, manually setting up and destroying VMs is tedious and breaks your development flow.

Vagrant is a command-line tool, which provides you with a configurable, reproducible, and portable development environment by enabling you to manage 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.

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

Getting ready

Download and install VirtualBox at <https://www.virtualbox.org/wiki/Downloads>.

Download and install Vagrant at <https://www.vagrantup.com/downloads.html>.

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 (1.4.1)'!
```

How to do it...

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

1. Visit <https://github.com/opscode/bento> and choose a Vagrant box to base your VMs on. We'll use `opscode-ubuntu-14.10` in this example.
2. The URL of the `opscode-ubuntu-14.10` box is https://opscode-vm-bento.s3.amazonaws.com/vagrant/virtualbox/opscode_ubuntu-14.10_chef-provisionerless.box.

3. Edit your 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-14.10"
  config.vm.box_url = "https://opscode-vm-bento.s3.amazonaws.com/
vagrant/virtualbox/opscode_ubuntu-14.10_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_ORG>-validator.pem"
    chef.validation_client_name = "<YOUR_ORG>-validator"
    chef.node_name = "server"
  end
end
```

4. Create your virtual node using Vagrant:

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

Bringing machine 'server' up with 'virtualbox' provider...
...TRUNCATED OUTPUT...
==> default: Importing base box 'opscode-ubuntu-14.10'...
...TRUNCATED OUTPUT...
==> default: Installing Chef 11.16.4 Omnibus package...
...TRUNCATED OUTPUT...
==> default: Running provisioner: chef_client...
==> default: Creating folder to hold client key...
==> default: Uploading chef client validation key...
Generating chef JSON and uploading...
==> default: Running chef-client...
==> default: [2014-12-01T22:00:54+00:00] INFO: *** Chef 11.16.4
***
...TRUNCATED OUTPUT...
```

5. Log in to your virtual node using SSH:

```
mma@laptop:~/chef-repo $ vagrant ssh

Welcome to Ubuntu 14.10 (GNU/Linux 3.16.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
```

```
Last login: Mon Oct 27 02:22:37 2014 from 10.0.2.2
```

```
vagrant@server:~$
```

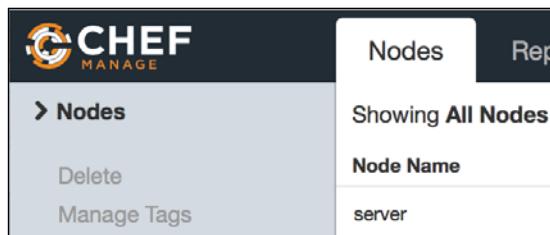
6. Log out of your virtual node:

```
vagrant@server:~$ exit  
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
```

8. 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-14.10"  
config.vm.box_url = "https://opscode-vm-bento.s3.amazonaws.com/  
vagrant/virtualbox/opscode_ubuntu-14.10_chef-provisionerless.box"
```

We want to boot our VM using a so-called Bento Box, provided by Chef. We use Ubuntu Version 14.10 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 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 organization on hosted Chef, you must download your private key. Tell Vagrant where to find this file:

```
chef.validation_key_path = ".chef/<YOUR_ORG>-validator.pem"
```

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

```
chef.validation_client_name = "<YOUR_ORG>-validator"
```

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

- ▶ Find the Vagrant documentation at <http://docs.vagrantup.com/v2/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 the same at <https://github.com/mitchellh/vagrant-aws>

Creating and using cookbooks

Cookbooks are an essential part of Chef. You can easily create them using the Chef executable installed by the Chef DK. In this recipe (and many of the following recipes), I will assume that you're using a Chef server to manage your infrastructure. You can either set up your own cookbook or use the hosted Chef as described previously. You'll use the command-line tool `knife` to interact with the Chef server.

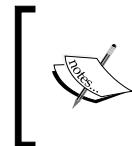
In this recipe, we'll create and apply a simple cookbook using the Chef and `knife` command-line tools.

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 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 default 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
```

```
Compiling Cookbooks...
Recipe: code_generator::cookbook
```

...TRUNCATED OUTPUT...

Before ChefDK was introduced, the only way to generate cookbooks was to use `knife cookbook create my_cookbook`

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` in order 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.

See also

- ▶ 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 that specific cookbook version, which is 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

Make sure that you have the `iptables` cookbook installed locally and uploaded on your Chef server.

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

```
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 given command and code lines:

```
mma@laptop:~/chef-repo $ knife cookbook upload iptables  
Uploading iptables [0.14.0]  
Uploaded 1 cookbook.
```

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 0.14.0
```

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

```
mma@laptop:~/chef-repo $ knife cookbook show iptables 0.14.0  
definitions
```

```
checksum: 45c0b77ff10d7177627694827ce47340  
name: iptables_rule.rb  
path: definitions/iptables_rule.rb  
specificity: default  
url: https://s3.amazonaws.com/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 0.14.0  
definitions iptables_rule.rb
```

```
#  
# Cookbook Name:: iptables
```

```
# Definition:: iptables_rule
#
#
define :iptables_rule, :enable => true, :source => nil, :variables => {} 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...

Since Chef 11, you can pass patterns to the `knife show` command to tell it what exactly you want to see. Showing the contents of the `iptables_rule` definition can be done as follows, in addition to the way we described previously:

```
mma@laptop:~/work/chef_helpster $ knife show cookbooks/iptables/
definitions/*

cookbooks/iptables/definitions/iptables_rule.rb:
#
# Cookbook Name:: iptables
# Definition:: iptables_rule
#
#
define :iptables_rule, :enable => true, :source => nil, :variables => {}
{} do
...TRUNCATED OUTPUT...
end
```

See also

- ▶ To find some more examples on `knife show`, visit http://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. 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` command 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'
depends 'apache2', '>= 1.0.4'
```

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 `berks install` and `berks upload` or `knife cookbook upload . . .`, the Chef server will then send all the required cookbooks to the node.



The first `depends` call tells the Chef server that your cookbook depends on the latest version of the `build-essential` cookbook.

The second `depends` call tells the Chef server that your cookbook depends on a version of the `apache2` cookbook, which is greater or equal to the version `1.0.4`. You may use any of these version constraints with your `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 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 have to 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 `Berksfile`) 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 cookbooks, which really matter.

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 (2.6.0) from https://supermarket.chef.io ([opscode]
https://supermarket.chef.io/api/v1)
...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...
Uploading windows (1.34.8) to: 'https://api.chef.io:443/
organizations/awo'
```

How it works...

Berkshelf comes with the Chef DK.

We create 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 generate creates a Berksfile, besides the metadata.rb file.

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

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

It 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 recipe, and everyone using your recipe 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`. Here, you'll find the exact versions of all the cookbooks that Berkshelf installed:

```
DEPENDENCIES
my_cookbook
  path: .
  metadata: true

GRAPH
apt (2.6.0)
chef-client (3.9.0)
  cron (>= 1.2.0)
  logrotate (>= 1.2.0)
  windows (~> 1.11)
chef_handler (1.1.6)
  cron (1.6.1)
logrotate (1.7.0)
my_cookbook (0.1.0)
  apt (>= 0.0.0)
  chef-client (>= 0.0.0)
  ntp (>= 0.0.0)
  ntp (1.6.8)
  windows (1.34.8)
  chef_handler (>= 0.0.0)
```

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 on 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.

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

```
mma@mma-mbp:~/work/chef-repo (master)$ vagrant plugin install vagrant-berkshelf
```

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

```
Installed the plugin 'vagrant-berkshelf (4.0.1)'!
```

Then, you need to tell Vagrant that you want to use the plugin. You do this by enabling the plugin in `Vagrantfile`:

```
mma@mma-mbp:~/work/chef-repo (master)$ subl Vagrantfile

config.berkshelf.enabled = true
```

Then, you need a `Berksfile` in the root directory of your Chef repository to tell Berkshelf which cookbooks to install on each Vagrant run:

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

cookbook 'my_cookbook', path: 'cookbooks/my_cookbook'
```

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 on 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: Loading Berkshelf datafile...
==> 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.getchef.com...
...TRUNCATED OUTPUT...
```

This way, using Berkshelf together with Vagrant, saves a lot of manual steps and gets faster cycle times for your cookbook development.

See also

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

Downloading and integrating cookbooks as vendor branches into your Git repository

The Chef community offers a wide variety of ready-made cookbooks for many major software packages. They're a great starting point for your own infrastructure. However, usually you need to modify these cookbooks to suit your needs. Modifying your local copy of a community cookbook leaves you with the dilemma of not being able to update to the latest version of the community cookbook without losing your local changes.

Getting ready

You'll need to make sure that your local Git repository is clean and does not have any uncommitted changes:

```
mma@laptop:~/chef-repo $ git status
# On branch master
nothing to commit (working directory clean)
```

How to do it...

Carry out the following given steps:

1. Go to <https://supermarket.chef.io/cookbooks> and search for the cookbook you need. In our example, we will use the mysql cookbook, which you can find by typing mysql in the search box and hitting enter. All we need is to note down the exact name of the cookbook; in this case, it's simply mysql.



2. Use knife to pull down the cookbook and integrate it with your local repository:

```
mma@laptop:~/chef-repo $ knife cookbook site install mysql
```

```
Installing mysql to /Users/mma/work/chef-repo/cookbooks  
...TRUNCATED OUTPUT...
```

Verify the downloaded cookbooks:

```
mma@laptop:~/chef-repo $ cd cookbooks  
mma@laptop:~/chef-repo/cookbooks $ ls -l  
total 8
```

```
-rw-r--r-- 1 mma  staff  3064 27 Sep  2013 README.md

drwxr-xr-x+ 10 mma  staff   340  7 Dez 20:43 mysql
drwxr-xr-x+ 12 mma  staff   408  7 Dez 20:43 yum
drwxr-xr-x+  9 mma  staff   306  7 Dez 20:43 yum-mysql-community
```

Validate the Git status:

```
mma@laptop:~/chef-repo/cookbooks $ git status

# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#
nothing to commit (working directory clean)
```

3. You might have noticed that your local branch received three commits. Let's take a look at those:

```
mma@laptop:~/chef-repo/cookbooks $ git log

commit 271d3de3b95bdc32d68133cdc91cb04e09625f59
Author: Matthias Marschall <mm@agileweboperations.com>
Date:   Sun Dec 7 20:43:50 2014 +0100

    Import yum version 3.5.1

commit cc72319ca7989df26e0ba7c3a76f78f6a7a7a4e2
Author: Matthias Marschall <mm@agileweboperations.com>
Date:   Sun Dec 7 20:43:44 2014 +0100

    Import yum-mysql-community version 0.1.10

commit 30984edb00c12177e25558bdfcd519da508b3ac5
Author: Matthias Marschall <mm@agileweboperations.com>
Date:   Sun Dec 7 20:43:38 2014 +0100

    Import mysql version 5.6.1
```

The `knife` command successfully downloaded and imported the `mysql` cookbook as well as its dependencies.

How it works...

Knife executes a set of commands to download the desired cookbook and to integrate it with your local repository.

Let's take a look at the output of the `knife cookbook site install` command again and go through it step-by-step:

1. First, the command makes sure that you're on the master branch of your repository:
`Checking out the master branch.`

2. The next step is to create a new vendor branch for the `mysql` cookbook, if none exists so far:

`Creating pristine copy branch chef-vendor-mysql.`

3. Then, knife downloads the tarball, removes any older version, uncompresses the new tarball, and removes it after successfully extracting its contents into a new cookbook directory:

```
Downloading mysql from the cookbooks site at version 5.6.1 to /  
Users/mma/work/chef-repo/cookbooks/mysql.tar.gz  
Cookbook saved: /Users/mma/work/chef-repo/cookbooks/mysql.tar.gz  
Removing pre-existing version.  
Uncompressing mysql version 5.6.1.  
Removing downloaded tarball
```

4. Now, it's time to commit the newly extracted files to the vendor branch:

`1 files updated, committing changes`

5. Finally, Git tags the branch with the current version of the cookbook:

`Creating tag cookbook-site-imported-mysql-5.6.1`

The `knife cookbook site install` command executes all the previously mentioned steps for all the cookbooks the desired cookbook depends on, by default.

Eventually, you end up with a separate branch, the so-called vendor branch, for every downloaded cookbook integrated into your master branch and nicely tagged as shown:

```
mma@laptop:~/chef-repo$ git branch -a  
  
  chef-vendor-iptables  
  chef-vendor-mysql  
* master  
  remotes/origin/master
```

This approach enables you to change whatever you like in your master branch and still pull down newer versions of the community cookbook. Git will automatically merge both versions or ask you to remove conflicts manually; these are standard Git procedures.

There's more...

If you want to integrate the desired cookbook into another branch, use the `--branch` `BRANCH_NAME` parameter.

```
mma@laptop:~/chef-repo [experimental] $ knife cookbook site install mysql  
--branch experimental  
  
Installing mysql to /Users/mma/work/chef-repo/cookbooks  
Checking out the experimental branch.  
Pristine copy branch (chef-vendor-mysql) exists, switching to it.  
Downloading mysql from the cookbooks site at version 5.6.1 to /Users/  
mma/work/chef-repo/cookbooks/mysql.tar.gz  
Cookbook saved: /Users/mma/work/chef-repo/cookbooks/mysql.tar.gz  
Removing pre-existing version.  
Uncompressing mysql version 5.6.1.  
removing downloaded tarball  
No changes made to mysql  
Checking out the experimental branch.  
...TRUNCATED OUTPUT...
```

As you can see, instead of checking out the master branch, the `knife cookbook site install` command uses the experimental branch now.

You can use the `-D` switch when running the command in order to avoid downloading all the cookbooks that your desired cookbook depends on.

```
mma@laptop:~/chef-repo $ knife cookbook site install mysql -D  
  
Installing mysql to /Users/mma/work/chef-repo/cookbooks  
Checking out the master branch.  
Pristine copy branch (chef-vendor-mysql) exists, switching to it.  
Downloading mysql from the cookbooks site at version 5.6.1 to /Users/  
mma/work/chef-repo/cookbooks/mysql.tar.gz  
Cookbook saved: /Users/mma/work/chef-repo/cookbooks/mysql.tar.gz  
Removing pre-existing version.  
Uncompressing mysql version 5.6.1.  
removing downloaded tarball  
No changes made to mysql  
Checking out the master branch.
```

You can see that the command stopped after dealing with the `mysql` cookbook. It has not yet gotten the other cookbooks.

See also

- ▶ You can use Berkshelf to manage cookbooks and their dependencies for you, which makes the preceding approach obsolete. See the *Managing cookbook dependencies with Berkshelf* 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 the 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:

```
mma@laptop:~/chef-repo $ chef gem search -r knife-  
*** REMOTE GEMS ***  
...TRUNCATED OUTPUT...  
  
knife-azure (1.3.0)  
...TRUNCATED OUTPUT...  
knife-ec2 (0.10.0)  
...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-ec2-0.10.0.gem (100%)
```

```
Successfully installed knife-ec2-0.10.0
...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 YYYYY:

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

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 at various places.

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

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

Then, it looks into the `.chef` directory, which is located in your home directory, to find the 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, amongst 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

Bootstrapping a node not only installs Chef on that node, but creates a client object on the Chef server as well. Running the Chef client on your node uses the client object to authenticate itself against the Chef server on each run.

Additionally, to registering 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...

It is a bit tedious and error prone when you have to issue two commands. 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 chapter
- ▶ 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 an 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"
end
```

How to do it...

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

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

```
mma@laptop:~/chef-repo $ chef-client -z -o my_cookbook

[2014-12-11T22:54:44+01:00] INFO: Starting chef-zero on host
localhost, port 8889 with repository at repository at /Users/mma/
work/chef-repo
[2014-12-11T22:54:44+01:00] INFO: Forking chef instance to
converge...
Starting Chef Client, version 11.18.0.rc.1
[2014-12-11T22:54:44+01:00] INFO: *** Chef 11.18.0.rc.1 ***
[2014-12-11T22:54:44+01:00] INFO: Chef-client pid: 20179
[2014-12-11T22:54:47+01:00] WARN: Run List override has been
provided.
[2014-12-11T22:54:47+01:00] WARN: Original Run List: []
[2014-12-11T22:54:47+01:00] WARN: Overridden Run List: [recipe[my_
cookbook]]
[2014-12-11T22:54:47+01:00] INFO: Run List is [recipe[my_
cookbook]]
[2014-12-11T22:54:47+01:00] INFO: Run List expands to [my_
cookbook]
[2014-12-11T22:54:47+01:00] INFO: Starting Chef Run for webops
```

2. Validate that the Chef client run creates the desired temporary file:

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

How it works...

The `-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 file system. 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 your 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:

```
mma@laptop:~/chef-repo $ knife upload /
laptop:
  run_list: recipe[my_cookbook]
```

See also

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

Using roles

Roles are there in Chef to group nodes with similar configuration. Typical cases are to have 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. Run the Chef client:

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

```
...TRUNCATED OUTPUT...
[2014-12-25T13:28:24+00:00] INFO: Run List is [role[web_servers]]
[2014-12-25T13:28:24+00:00] INFO: Run List expands to [ntp]
...TRUNCATED OUTPUT...
```

How it works...

You define a role in a Ruby 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, which has 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), which are in the role's run list, will be executed on your nodes.

You need to upload your role on 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 in finding 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 are good ways 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, I 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 book
{
  "name": "book",
  "description": "",
  "cookbook_versions": {
  },
  "json_class": "Chef::Environment",
  "chef_type": "environment",
  "default_attributes": {
  },
  "override_attributes": {
  }
}
Created book
```

2. List the available environments:

```
mma@laptop:~/chef-repo $ knife environment list
_default
book
```

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 book environment yet by listing nodes in the book environment only:

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

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

5. Change the environment of server to book using knife:

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

```
server:
```

```
    chef_environment: book
```

6. List the nodes of the book environment again:

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

```
server
```

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

```
mma@laptop:~/chef-repo $ knife environment edit book
```

```
{
  "name": "book",
  "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 book
```

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 lesser 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 lesser 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 book.rb  
name "book"
```

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

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

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

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



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("book")}'
```

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

```
mma@laptop:~/chef-repo $ knife search node "chef_environment:book"  
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 outage throughout your infrastructure. If you have a cookbook version that is known to work, 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.

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 [1.6.8]  
Uploaded 1 cookbook.
```

2. Try to upload the same cookbook version again:

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

```
Uploading ntp [1.6.8]
ERROR: Version 1.6.8 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 "1.6.9"
```

4. Upload the cookbook again:

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

Uploading ntp [1.6.9]
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. 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://github.com/jonlives/knife-spork>.

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 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 box 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 the daemon mode so that it runs automatically.

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

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

2. Validate that the Chef client runs as 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.

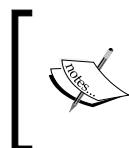
There's more...

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 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
```

Using chef-shell

Writing cookbooks is hard. Being able to try out parts of a recipe interactively and using breakpoints really helps 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 and executeing complete cookbooks on your nodes.

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: 11.18.0
http://www.chef.io/chef
http://docs.chef.io/

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

```
Ohai2u mma@laptop!
chef >
```

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

```
chef > attributes_mode
```

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

```
chef:attributes > set[:title] = "Chef Cookbook"
```

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

```
chef:attributes > quit
```

```
=> :attributes
```

```
chef >
```

4. Switch to the recipe mode:

```
chef > recipe_mode
```

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

```
chef:recipe > 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 >
```

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

```
chef:recipe > run_chef
```

```
[2014-12-12T22:26:42+01:00] 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 file 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 files 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 local box might do, but if you're going to do more invasive stuff, 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 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 the Chef client mode by using the `run_it --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 > nodes.list  
[node [my_server]]
```

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 the chef-shell at https://docs.chef.io/chef_shell.html

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
- ▶ Flagging problems in your Chef cookbooks
- ▶ Test-driven development for cookbooks using ChefSpec
- ▶ 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 why-run mode to find out what a recipe might do
- ▶ Debugging Chef client runs
- ▶ Inspecting the results of your last Chef run
- ▶ Raising and logging exceptions in recipes
- ▶ Diff-ing cookbooks with knife
- ▶ Using community exception and report handlers
- ▶ Creating custom handlers

Introduction

Developing cookbooks and making sure your nodes converge to the desired state is a complex endeavor. You need transparency about what is really 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

You know how annoying this is: you tweak a cookbook, upload it to your Chef server, start a Chef run on your node and, boom! It fails. What's even more annoying is that it fails not because a black hole absorbed your node and the whole data center that node lives in, but 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 real nodes.

Getting ready

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

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

Installing ntp to /Users/mma/work/chef-repo/cookbooks
...TRUNCATED OUTPUT...
Cookbook ntp version 1.7.0 successfully installed
```

How to do it...

Carry out the following steps to test your cookbooks:

1. Run `knife cookbook test` on a working cookbook, for example, the `ntp` cookbook:

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

checking ntp
Running syntax check on ntp
Validating ruby files
Validating templates
```

2. Now, let's break something in the `ntp` cookbook's default recipe by removing the comma at the end of the `node['ntp']['varlibdir']`, line:

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

...
[ node['ntp']['varlibdir'],
  node['ntp']['statsdir'] ].each do |ntpdirectory|
  directory ntpdirectory do
    owner node['ntp']['var_owner']
    group node['ntp']['var_group']
    mode 0755
  end
end
```

3. Run the `test` command again:

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

checking ntp
Running syntax check on ntp
Validating ruby files
FATAL: Cookbook file recipes/default.rb has a ruby syntax error:
FATAL: cookbooks/ntp/recipes/default.rb:25: syntax error,
unexpected TIDENTIFIER, expecting ']'
FATAL:   node['ntp']['statsdir'].each do |ntpdirectory|
FATAL:           ^
FATAL: cookbooks/ntp/recipes/default.rb:25: syntax error,
unexpected ']', expecting $end
FATAL:   node['ntp']['statsdir'].each do |ntpdirectory|
FATAL:           ^
```

How it works...

`knife cookbook test` executes a Ruby syntax check on all Ruby files within the cookbook as well as on all ERB templates. It loops through all Ruby files and runs `ruby -c` against each of them. The `ruby -c` command causes Ruby to check the syntax of the script and quit without running it.

After going through all Ruby files, `knife cookbook test` goes through all ERB templates and pipes the rendered version created by `erubis -x` through `ruby -c`.

There's more...

The `knife cookbook test` command does only a very simple syntax check on the Ruby files and ERB templates. There exists a whole ecosystem of additional tools such as Foodcritic (a lint check for Chef cookbooks), ChefSpec (behavior-driven testing for Chef), and Test-kitchen (an integration testing tool to run cookbooks on virtual servers), and then some. You can go fully test driven, if you want!

See also

- ▶ 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

Writing solid Chef recipes can be quite challenging. There are a couple of pitfalls, which you can easily overlook. Also, writing cookbooks in a consistent style is even harder. 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 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 the excerpts of the places where it found the rules to match.

In the preceding example, it didn't like that the `mysql` cookbook 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 the ones 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 $?
```

3

In this example, we tell Foodcritic to fail if any rule of 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.

See also

- ▶ Learn how to make sure that your cookbooks compile in the *Testing your Chef cookbooks* recipe in this chapter
- ▶ Check out strainer, a tool to test multiple things, such as Foodcritic and knife test as well as other stuff, at once 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 and ensure that your test is for real because it should fail, as long as you haven't written your recipe code.

As soon as you've completed your recipe, your unit tests should pass.

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 `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 and then a recipe, which will pass the test:

1. Create the `spec` directory for your cookbook:

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

2. Create your `spec` file:

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

require 'chefspec'
describe 'my_cookbook::default' do
  let(:chef_run) {
    ChefSpec::ServerRunner.new(
      platform:'ubuntu', version:'12.04'
    ).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
```

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

```
mma@laptop:~/chef-repo $ 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.converge(described_recipe)).to  
create_file_with_content('/tmp/greeting.txt','Hello! ubuntu!')  
File content:  
  does not match expected:  
  Hello! ubuntu!  
# ./cookbooks/my_cookbook/spec/default_spec.rb:11:in `block  
(2 levels) in <top (required)>'
```

Finished in 0.11152 seconds
1 example, 1 failure

Failed examples:

```
rspec ./cookbooks/my_cookbook/spec/default_spec.rb:10 # my_  
cookbook::default creates a greetings file, containing the  
platform name
```

4. 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!'  
end
```

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

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

6. Create the template file:

```
mma@laptop:~/chef-repo $ subl  
cookbooks/my_cookbook/templates/greeting.txt.erb  
  
<%= @greeting %> <%= node['platform'] %>!
```

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

```
mma@laptop:~/chef-repo $ 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 in order to use RSpec with Chef. The 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 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, but 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 12.04. 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 12.04 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:'12.04'
  ).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:

```
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, which was 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 what our recipe should do.

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

On the ChefSpec site, you will find the 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 check whether it fails:

```
$ rspec cookbooks/my_cookbook/spec/default_spec.rb
```

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

```
.F
```

Failures:

```
1) my_cookbook::default uses a node attribute as greeting text
Failure/Error: expect(chef_run.converge(described_recipe)).to
  create_file_with_content('/tmp/greeting.txt','Go! ubuntu!')
  File content:
    Hello! ubuntu! does not match expected:
    Go! ubuntu!
# ./cookbooks/my_cookbook/spec/default_spec.rb:16:in `block
  (2 levels) in <top (required)>'
```

```
Finished in 0.25295 seconds
2 examples, 1 failure
```

Failed examples:

```
rspec ./cookbooks/my_cookbook/spec/default_spec.rb:14 #
  my_cookbook::default uses a node attribute as greeting text
```

Now, we modify our recipe to use the node attribute:

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

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

This makes, our tests pass again:

```
..
Finished in 0.25078 seconds
2 examples, 0 failures
```

See also

- ▶ The ChefSpec repository on GitHub at <https://github.com/sethvargo/chefspec>
- ▶ The source code of Fauxhai at <https://github.com/customink/fauxhai>

- ▶ A talk by Seth Vargo showing an example developing a cookbook test-driven at <http://www.confreaks.com/videos/2364-mwrc2013-tdding-tmux>
- ▶ The RSpec website at <http://rspec.info/>

Integration testing your Chef cookbooks with Test Kitchen

Verifying that your cookbooks really work when converging a node is essential. Only if you can trust your cookbooks, you are 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 tests 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, 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 Serverspec 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. Initialize Test Kitchen for your cookbook:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ kitchen init  
  
create .kitchen.yml  
create test/integration/default  
Successfully installed kitchen-vagrant-0.16.0  
1 gem installed
```

5. Edit the generated Test Kitchen configuration file to only test against Ubuntu 14.04:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ subl .kitchen.yml  
  
...  
platforms:  
- name: ubuntu-14.04  
...
```

6. Create a directory for your Serverspec files inside your cookbook:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ mkdir -p test/  
integration/default/serverspec
```

7. Create your spec, defining what you expect your cookbook to do:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ subl test/  
integration/default/serverspec/greeting_spec.rb  
  
require 'serverspec'  
  
# Required by serverspec  
set :backend, :exec  
  
describe file('/tmp/greeting.txt') do  
  its(:content) { should match 'Ohai, Chefs!' }  
end
```

8. Run Test Kitchen:

```
mma@laptop:~/chef-repo/cookbooks/my_cookbook $ kitchen test  
-----> Starting Kitchen (v1.2.1)  
...TRUNCATED OUTPUT...
```

```
Bringing machine 'default' up with 'virtualbox' provider...
...TRUNCATED OUTPUT...
Finished creating <default-ubuntu-1404> (0m50.31s).
----> Converging <default-ubuntu-1404>...
...TRUNCATED OUTPUT...
----> Installing Chef Omnibus (true)
...TRUNCATED OUTPUT...
Starting Chef Client, version 12.0.3
...TRUNCATED OUTPUT...
Recipe: my_cookbook::default
...TRUNCATED OUTPUT...
      Chef Client finished, 1/1 resources updated in 6.450780111
seconds
...TRUNCATED OUTPUT...
Uploading /tmp/busser/suites/serverspec/greeting_spec.rb
(mode=0644)
----> Running serverspec test suite
----> Installing Serverspec...
...TRUNCATED OUTPUT...
----> serverspec installed (version 2.7.1)
...TRUNCATED OUTPUT...
File "/tmp/greeting.txt"
  content
    should match "Ohai, Chefs!"

Finished in 0.09647 seconds (files took 0.26995 seconds to load)
1 example, 0 failures
      Finished verifying <default-ubuntu-1404> (0m15.46s).
----> Destroying <default-ubuntu-1404>...
...TRUNCATED OUTPUT...
----> Kitchen is finished. (2m13.66s)
```

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 running `kitchen init`, which creates a `.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 solo:

```
provisioner:  
  - name: chef_solo
```

3. Part three defines on which platforms you want to test your cookbook. To keep things simple, we only define Ubuntu 14.04 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`:

```
platforms:  
  - name: ubuntu-14.04
```

4. Part four defines the test suites. We define one called `default`. We include our `my_cookbook::default` recipe here so that we're able to test what it does:

```
suites:  
  - name: default  
    run_list:  
      - recipe [my_cookbook::default]  
    attributes:
```

Now, it's time to create our specs using Serverspec. Test Kitchen uses a naming convention inside the `test/integration` directory to discover the test framework to use for each test suite. In our case, we run the `default` test suite by using `serverspec` as the testing framework. That's why our directory structure looks like this:

```
test/integration/default/serverspec
```

We call our spec `greeting_spec.rb` and put it inside the aforementioned directory, so that Test Kitchen will automatically pick it up and run it.

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
```

Serverspec provides you with custom RSpec matchers to verify the status of your systems. You can find a complete list of the supported resource types here: http://serverspec.org/resource_types.html.

Finally, we can run Test Kitchen. It will first make sure that no old VMs are around and then create a new one. It installs Chef on that brand new VM and starts a Chef run. Test Kitchen uses **Busser** to install the desired test framework (as defined by the directory structure below `test/integration`) and execute our specs 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 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 or CentOS and so on, by adding them to the `platforms` definition in `.kitchen.yml`:

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



You 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-14.04` or `kitchen test 14`.

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	Last Action
default-ubuntu-1404	Vagrant	ChefSolo	Converged

See also

- ▶ Find Test Kitchen at <http://kitchen.ci>
- ▶ Find the source code for Test Kitchen and the associated tools (including drivers for cloud providers and Busser plugins for other test frameworks) on GitHub at <https://github.com/test-kitchen>
- ▶ Find Serverspec at <http://serverspec.org>

Showing affected nodes before uploading cookbooks

You know how it goes. 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 spits its guts at you. 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 at least one role using the `ntp` cookbook in its run list and that you have multiple servers with this role and/or the `ntp` cookbook in their run list directly.

Use Chef to install the `knife-preflight` gem:

```
mma@laptop:~/chef-repo $ chef gem install knife-preflight
Fetching gem metadata from https://rubygems.org/
...TRUNCATED OUTPUT...
Installing knife-preflight (0.1.7)
```

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 obviously see your nodes and roles in the output instead of the exact ones listed here:

```
mma@laptop:~/chef-repo $ knife preflight ntp
Searching for nodes containing ntp OR ntp::default in their
expanded run_list...
2 Nodes found
www-staging.example.com
cms-staging.example.com
Searching for roles containing ntp OR ntp::default in their
expanded run_list...
3 Roles found
your_cms_role
your_www_role
```

```
your_app_role
Found 6 nodes and 3 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 and many more

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.

Eventually, 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 be aware of the fact 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

We all have those snowflake environments that are built using Chef, but we're not comfortable with running the Chef client anymore. We know that some cookbooks have been enhanced but never tested against this specific environment. The risk of bringing it down by a Chef client run is pretty high.

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



Don't overuse this feature! Make sure you fix your environment so that you're comfortable to run Chef client whenever you need to!



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, my 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
...
Run List:      role[base]
Roles:        base
Recipes:      chef-client::delete_validation, runit, chef-client
...
mma@laptop:~/chef-repo $
```

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:~$ chef-client -o 'recipe[users]'

Starting Chef Client, version 12.0.3
[2014-12-23T10:58:10+00:00] WARN: Run List override has been provided.
[2014-12-23T10:58:10+00:00] WARN: Original Run List: []
[2014-12-23T10:58:10+00:00] WARN: Overridden Run List:
[recipe[users]]
resolving cookbooks for run list: ["users"]
...
user@server:~$
```

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 passed 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 it in the *Showing affected nodes before uploading cookbooks* recipe in this chapter

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 about 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 that is 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 1:4.2.6.p5+dfsg-3ubuntu2 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, 10/11 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, 5/11 resources would have been updated
```

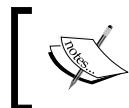
How it works...

The `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 would be 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...

Let's see how we can ask the Chef client to print debug messages:

1. Run the Chef-client with debug output:

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

...TRUNCATED OUTPUT...
[2014-12-27T21:38:34+00:00] DEBUG: Chef::HTTP calling Chef::HTTP::
RemoteRequestID#handle_request
[2014-12-27T21:38:34+00:00] DEBUG: Chef::HTTP calling Chef::HTTP::
ValidateContentLength#handle_request
[2014-12-27T21:38:34+00:00] DEBUG: Initiating GET to https://api.
opscode.com/organizations/awo/nodes/server
[2014-12-27T21:38:34+00:00] DEBUG: ---- HTTP Request Header Data:
-----
[2014-12-27T21:38:34+00:00] DEBUG: Accept: application/json
[2014-12-27T21:38:34+00:00] DEBUG: Accept-Encoding: gzip;q=1.0,def
late;q=0.6,identity;q=0.3
[2014-12-27T21:38:34+00:00] DEBUG: X-OPS-SIGN:
...TRUNCATED OUTPUT...
[2014-12-27T21:38:34+00:00] DEBUG: HOST: api.opscode.com:443
[2014-12-27T21:38:34+00:00] DEBUG: X-REMOTE-REQUEST-ID: d4b07248-
d522-4181-bf02-1405c3ccb110
[2014-12-27T21:38:34+00:00] DEBUG: ---- End HTTP Request Header
Data -----
[2014-12-27T21:38:35+00:00] DEBUG: ---- HTTP Status and Header
Data: -----
[2014-12-27T21:38:35+00:00] DEBUG: HTTP 1.1 200 OK
[2014-12-27T21:38:35+00:00] DEBUG: server: ngx_openresty
[2014-12-27T21:38:35+00:00] DEBUG: date: Sat, 27 Dec 2014 21:38:35
GMT
[2014-12-27T21:38:35+00:00] DEBUG: content-type: application/json
[2014-12-27T21:38:35+00:00] DEBUG: transfer-encoding: chunked
[2014-12-27T21:38:35+00:00] DEBUG: connection: close
[2014-12-27T21:38:35+00:00] DEBUG: x-ops-api-info:
flavor=hec;version=12.0.0;oc_erchef=0.29.2
[2014-12-27T21:38:35+00:00] DEBUG: content-encoding: gzip
[2014-12-27T21:38:35+00:00] DEBUG: ---- End HTTP Status/Header
Data -----
[2014-12-27T21:38:35+00:00] DEBUG: Chef::HTTP calling Chef::HTTP::
ValidateContentLength#handle_response
[2014-12-27T21:38:35+00:00] DEBUG: HTTP server did not include
a Content-Length header in response, cannot identify truncated
downloads.
```

```
[2014-12-27T21:38:35+00:00] DEBUG: Chef::HTTP calling Chef::HTTP::  
RemoteRequestID#handle_response  
[2014-12-27T21:38:35+00:00] DEBUG: Chef::HTTP calling Chef::HTTP::  
Authenticator#handle_response  
[2014-12-27T21:38:35+00:00] DEBUG: Chef::HTTP calling  
Chef::HTTP::Decompressor#handle_response  
...TRUNCATED OUTPUT...
```

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

More often than we like to admit, Chef client runs fail. Especially when developing new cookbooks, we need to know what exactly went wrong.

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.

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) 2014 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 2014-12-27 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.

Raising and logging exceptions in recipes

Running your own cookbooks on your nodes might lead into 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 [1.7.0]  
Uploaded 1 cookbook.
```

3. Run the Chef client on the node:

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

```
...TRUNCATED OUTPUT...  
[2014-12-27T13:53:19+00:00] INFO: Storing updated cookbooks/ntp/  
TESTING.md in the cache.  
[2014-12-27T13:53:19+00:00] INFO: ** Going to install the ntp  
service now...  
[2014-12-27T13:53:19+00:00] INFO: ** ntp service installed and  
started successfully!  
[2014-12-27T13:53:19+00:00] INFO: Processing package[ntp] action  
install (ntp::default line 21)  
...TRUNCATED OUTPUT...
```

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

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

```
Chef::Application.fatal!("Ouch!!! Bailing out!!!!")
```

5. Upload the modified cookbook to the Chef server:

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

```
Uploading ntp [1.7.0]  
Uploaded 1 cookbook.
```

6. Run the Chef client on the node again:

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

```
...TRUNCATED OUTPUT...  
[2014-12-28T11:09:44+00:00] FATAL: 'Ouch!!! Bailing out!!!'  
...TRUNCATED OUTPUT...
```

How it works...

The `fatal! (msg)` method logs the given error message through `Chef::Log.fatal (msg)` and then exits the Chef client process using `Process.exit` with the exit code `-1`.

There's more...

You might want to exit the Chef client run without logging a fatal message. You can do so by using the `exit! (msg)` method in your recipe. It will log the given message as debug and exit the Chef client, the same as it does when calling `fatal! (...)`.

See also

- ▶ Read the documentation for the `fatal!` method here: http://www.rubydoc.info/gems/chef/Chef/Application#fatal%21-class_method
- ▶ 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 like 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 against the version already uploaded to the Chef server.

Getting ready

You need to have at least one cookbook you can modify and which 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      2014-12-29
21:02:50.000000000 +0100
+++ cookbooks/my_cookbook/recipes/default.rb      2014-12-29
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 list files, which have been changed instead of 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 **Handler**, 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 email, to Campfire, 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

In order 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

Compiling Cookbooks...
Recipe: code_generator::cookbook
 * directory[/Users/mma/work/chef-repo/cookbooks/my_handlers]
action create
 - create new directory /Users/mma/work/chef-repo/cookbooks/
my_handlers
...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:

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

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

```
mma@laptop:~/chef-repo $ subl  
cookbooks/my_handlers/recipes/default.rb  
  
include_recipe 'chef_handler'  
  
chef_gem "chef-irc-snitch"  
  
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 $ knife cookbook upload my_handlers  
  
Uploading my_handlers [0.1.0]  
Uploaded 1 cookbook.
```

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

```
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 [2014-12-29T10:19:25+00:00]  
INFO: Enabling chef_handler[IRCSnitch] as a report handler  
  
  - enable chef_handler[IRCSnitch] as a report handler  
[2014-12-29T10:19:25+00:00] INFO: Enabling chef_handler[IRCSnitch]  
as a exception handler  
  
  - enable chef_handler[IRCSnitch] as a exception handler
```

How it works...

The `chef_handler` **Light Weight Resource Provider (LWRP)** provided by the `chef_handler` cookbook helps you enable and configure any custom handler without the need to manually modify `client.rb` on 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 LWRP where it can find the Handler class. The default should be `chef/handlers/...` but more often than not, 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 LWRP 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 install it in `/var/chef/handlers` and use this directory as the source when using the `chef_handler` LWRP.

See also

- ▶ Read more about Exception and Report handlers at <http://docs.chef.io/handlers.html>

Creating custom handlers

Chef handlers can be very helpful to integrate Chef with your tool chain. If there is no handler readily available for the tools you are using, it's pretty simple to write your own.

We'll have a look at how to create an exception handler, reporting Chef client run failures to **Flowdock**, a web-based team inbox and chat tool.

Getting ready...

As we want to publish information to a Flowdock inbox, you need to sign up for an account at <http://www.flowdock.com>. Also, we need to install the API client as a Ruby gem in order to be able to post to our team inbox from Chef.

Install the flowdock gem on your local development box:

```
mma@laptop:~/chef-repo $ chef gem install flowdock
...TRUNCATED OUTPUT...
Fetching: flowdock-0.5.0.gem (100%)
Successfully installed flowdock-0.5.0
2 gems installed
```

How to do it...

Carry out the following steps to create a custom handler to post Chef run failures to Flowdock:

1. Create your handler class:

```
mma@laptop:~/work/chef-handler-flowdock $ mkdir -p
      lib/chef/handler
mma@laptop:~/work/chef-handler-flowdock $ subl
      lib/chef/handler/flowdock_handler.rb

require 'chef/handler'
require 'flowdock'

class Chef
  class Handler
    class FlowdockHandler < Chef::Handler

      def initialize(options = {})
        @from = options[:from] || nil
        @flow = Flowdock::Flow.new(:api_token =>
          options[:api_token],
          :source => options[:source] || "Chef client")
      end

      def report
        if run_status.failed?
          content = "Chef client raised an exception:<br/>"
          content << run_status.formatted_exception
          content << "<br/>"
          content << run_status.backtrace.join("<br/>")

          @from = { :name => "root", :address =>
            "root@#{run_status.node fqdn}" } if @from.nil?

          @flow.push_to_team_inbox(:subject => "Chef client
            run on #{run_status.node} failed!",
```

```
:content => content,
:tags => ["chef",
  run_status.node.chef_environment,
  run_status.node.name], :from => @from)
end
end
end
end
end
```

2. Copy the handler to your node:

```
user@server:~$ sudo mkdir -p /var/chef/handlers
mma@laptop:~/work/chef-handler-flowdock $ scp
  lib/chef/handler/flowdock_handler.rb
  user@server:/var/chef/handlers/flowdock_handler.rb
```

3. Enable the handler in `client.rb` on your node. Replace `FLOWDOCK_API_TOKEN` with your own token:

```
user@server:~$ subl /etc/chef/client.rb

require '/var/chef/handlers/flowdock_handler'
exception_handlers <<
  Chef::Handler::FlowdockHandler.new(:api_token =>
    "FLOWDOCK_API_TOKEN")
```

If you have a failing Chef client run on your node, your handler will report it to your Flowdock flow.

How it works...

To create a Chef handler, your class needs to extend `Chef::Handler`. It should have two methods: `initialize` and `report`. Chef will call the `report` method at the end of every Chef client run.

The handler class can access `run_status` of the Chef client run to retrieve information about the run, for example, the current node object, `success?` or `failed?`, and the exception (if any). You will find a full list of the supported attributes at <http://docs.chef.io/handlers.html#run-status-object>

As we only want to report exceptions, we will execute our logic inside the `report` method only if the Chef run fails

There's more...

Instead of manually installing the handler on all your nodes, you can create a cookbook called `my_handlers` to do it for you (see the *Using community exception and report handlers* recipe in this chapter). The default recipe of your `my_handlers` cookbook should look like this:

```
include_recipe "chef_handler"
chef_gem "flowdock"
remote_file "#{node["chef_handler"]["handler_path"]}/flowdock_handler.rb" do
  source "https://raw.githubusercontent.com/mmarschall/chef-handler-flowdock/master/lib/chef/handler/flowdock_handler.rb"
end
chef_handler "Chef::Handler::FlowdockHandler" do
  source "#{node["chef_handler"]["handler_path"]}/flowdock_handler.rb"
  arguments :api_token => "FLOWDOCK_API_TOKEN"
  action :enable
end
```

In our example, we create the Flowdock API client in the `initialize` method. If you use the LWRP to install the handler, the `initialize` method will receive an options Hash from the `arguments` call inside the `chef_handler` provider.

See also

- ▶ If you just want to use an existing handler, read the *Using community exception and report handlers* recipe in this chapter

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
- ▶ Using definitions
- ▶ Creating your own **Lightweight Resource Providers (LWRP)**
- ▶ 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 take a look at how to use the Chef **Domain Specific Language (DSL)** from 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 really 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 its 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. is used to propose 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 take a 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 styling guideline 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 as the JSON version will be automatically generated from it).

5. Configure your cookbook to the appropriate version by using Semantic Versioning (<http://semver.org>):

```
version "1.1.0"
```

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

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

7. Declare dependencies 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 level 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 did 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 lot of work and headaches.

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 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 (use --force or bump the version number in your metadata.rb file if Berkshelf has frozen your cookbook earlier):

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook --force
Uploading my_cookbook [0.1.0]
```

5. Run Chef-client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2015-01-13T20:48:21+00:00] INFO: ** Saying what I was told to
say: hello world!
...TRUNCATED OUTPUT...
```

How it works...

Chef loads all attributes from the attribute files before it executes the 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 being the lowest, then comes 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 <http://docs.chef.io/attributes.html#attribute-precedence>.

There's more...

You can set and override attributes within roles and environments, as well. 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...
[2015-01-13T20:49:49+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 get 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 <http://docs.chef.io/attributes.html>
- ▶ Learn all about how attributes evolved since Chef 0.7 here: <http://www.opscode.com/blog/2013/02/05/chef-11-in-depth-attributes-changes/>

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 recreate these configuration files with dynamic values that are driven by the attributes we discussed so far in this chapter.

You can retrieve dynamic values from **data bags**, 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 *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...
[2015-01-14T20:41:21+00:00] INFO: Processing template[/tmp/
message] action create (my_cookbook::default line 9)
[2015-01-14T20:41:22+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 least specific(default).

You can place `message.erb` in the `cookbooks/my_cookbook/templates/host-server.vm` ("host-#{node[:fqdn]}") directory it should be specific to that host. It should be specific to a certain platform version. You can place it in `cookbooks/my_cookbook/templates/ubuntu-14.04` ("#{node[:platform]}-#{node[:platform_version]}"), and if it is only be platform-specific, you can place it in `cookbooks/my_cookbook/templates/ubuntu` ("#{node[:platform]}"). Only if your template is the same for any host or platform, you will place it in the default directory.



Be aware of the fact that 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 <http://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 execute parts of your recipe, looping, or even 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:0x0000005010d38 @chef_server_rest=nil, @
name="server",
...TRUNCATED OUTPUT...
```

2. Sort your nodes by name by 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, which 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 earlier.

Another common thing is to use `if`, `else`, or `case` for conditional execution. In the fourth example earlier, we used `if` to only write a warning to the log file, if the `nodes` array would be 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 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, they use Ruby to sort an array of nodes by their name attributes.

Then, they 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: http://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 is an example 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 *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'  
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 Chef client on your node to see whether it works:

```
user@server $ sudo chef-client  
  
...TRUNCATED OUTPUT...  
[2014-12-29T22:29:15+01:00] INFO: Netmask of 192.168.0.1:  
255.255.255.0  
...TRUNCATED OUTPUT...
```

How it works...

A Chef run consists of a *compile* phase, where it instantiates all resources, and an *execute* phase, where Chef runs the resource providers to converge the node.

If you want to use the functionality of a Ruby gem inside your cookbook, you need to install that gem during the *compile* phase. Otherwise, it will not be available during the *execute* phase (only after the Chef run).

The `chef_gem` resource will do exactly that.

The `gem_package` resource, in contrast, installs the gem into the system Ruby. 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. As long as 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 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 *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...
[2014-12-30T21:17:44+01:00] INFO: Netmask of 10.10.0.0/24:
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.



This isn't the cleanest, but it is the simplest way of doing it. The following paragraphs will help you find a cleaner way.

```
class Chef::Recipe
  def netmask(ipaddress)
    ...
  end
end
```

Chef automatically loads your library code in the compile phase that enables you to use the methods that you declare inside your `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, for example, if you define a method inside a library of your own cookbook and someone else defines a method with the same name in the library of another cookbook. Another clash would happen if you accidentally use a method name, which Chef defines in its `Chef::Recipe` class.

It's cleaner to introduce your subclasses inside your libraries and define your methods as class methods. This avoids polluting the `Chef::Recipe` namespace are given:

```
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 the *Using the Chef console (chef-shell)* recipe in *Creating and using cookbooks* recipe of *Chapter 1, Chef Infrastructure*
- ▶ The *Mixing plain Ruby with Chef DSL* recipe in this chapter

Using definitions

Your cookbooks will grow and get pretty long. Silently, some duplication will sneak in, as well. It's time to group resources and give them names in order to regain readability for your cookbook. Also, if you use the same set of resources again and again, it is a good idea to refactor such groups of resources into **definitions**.

In this section, we'll group a set of resources into a definition to make it reusable.

Getting ready

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

How to do it...

Let's see how to create and use a definition:

1. Create a definition in a new file in your cookbook's `definitions` folder:

```
mma@laptop:~/chef-repo $ mkdir -p cookbooks/my_cookbook/
definitions
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/definitions/
capistrano_deploy_dirs.rb
define :capistrano_deploy_dirs, :deploy_to => '' do
  directory "#{params[:deploy_to]}/releases"
  directory "#{params[:deploy_to]}/shared"
  directory "#{params[:deploy_to]}/shared/system"
end
```

2. Use the definition inside your cookbook's default recipe:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/recipes/
default.rb
capistrano_deploy_dirs do
  deploy_to "/srv"
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 to see whether it works:

```
user@server $ sudo chef-client

...TRUNCATED OUTPUT...
* directory[/srv/releases] action create[2014-12-
30T16:23:57+00:00] INFO: directory[/srv/releases] created
directory /srv/releases

- create new directory /srv/releases
* directory[/srv/shared] action create[2014-12-
30T16:23:57+00:00] INFO: directory[/srv/shared] created directory
/srv/shared

- create new directory /srv/shared
* directory[/srv/shared/system] action create[2014-12-
30T16:23:57+00:00] INFO: directory[/srv/shared/system] created
directory /srv/shared/system

- create new directory /srv/shared/system
...TRUNCATED OUTPUT...
```

How it works...

Definitions in Chef are like macros: you group a collection of resources and give this group a name. Chef reads the definition and expands its contents into the recipe during the compile phase.

A definition has a name (here, `capistrano_deploy_dirs`) by which you can call it from your recipe. Also, a definition has a list of parameters (here `deploy_to`):

```
define :capistrano_deploy_dirs, :deploy_to => '' do
  ...
end
```

The code inside the definition has access to a Hash called `params`. It contains all the keys you defined after the definition name. Here, Chef will add the three `directory` resources to the execution list:

```
define ...
  directory "#{params[:deploy_to]}/releases"
  directory "#{params[:deploy_to]}/shared"
  directory "#{params[:deploy_to]}/shared/system"
end
```

In your recipes, you can use the definition name instead of putting all the three directory resources. Inside the block, you use dynamically generated methods to fill each parameter with its value:

```
capistrano_deploy_dirs do
  deploy_to "/srv"
end
```

There's more...

Be aware that definitions are expanded into their containing resources during the compile phase of the Chef run. Definitions are not available during the execution phase. You cannot notify a definition, but only the resources it contains.

You cannot address the definition:

```
notifies :delete, 'capistrano_deploy_dirs', :immediately
```

However, you *can* address the individual resources inside the definition:

```
notifies :delete, 'directory[/srv/releases]', :immediately
```

See also

- ▶ Read more about definitions at <http://docs.chef.io/definitions.html>

Creating your own Lightweight Resource Providers (LWRP)

Chef offers the opportunity to extend the list of available resources by creating a custom **Lightweight Resource Provider (LWRP)**. 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.

Many of the custom resources in Chef's community cookbooks (and elsewhere) are implemented as LWRPs. So, there are many working examples in the real world, such as `iptables_rule`, `apt_repository`, and many more.

In this section, we will create a very simple LWRP 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 *Creating and using cookbooks* recipe of Chapter 1, *Cookbook Infrastructure*.

How to do it...

Let's see how to build a very simple LWRP to create a text file on your node:

1. Create your custom resource in your greeting cookbook:

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

actions :create, :remove
attribute :title, kind_of: String, default: "World"
attribute :path, kind_of: String, default: "/tmp/greeting.txt"
```

2. Create the provider for your resource in your greeting cookbook:

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

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
```

3. Use your new resource by editing your greeting cookbook's default recipe:

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

greeting "Ohai" do
  title "Chef"
  action :create
end
```

4. Run the Chef client on your workstation:

```
mma@laptop:~/chef-repo $ chef-client -z -o 'recipe[greeting]' '
...TRUNCATED OUTPUT...'
```

```
* greeting[Ohai] action create[2014-12-30T18:08:21+01:00] INFO:  
Processing greeting[Ohai] action create (greeting::default line 6)  
[2014-12-30T18:08:21+01:00] INFO: Adding 'Ohai' greeting as /tmp/  
greeting.txt  
(up to date)  
* file[/tmp/greeting.txt] action create[2014-12-  
30T18:08:21+01:00] INFO: Processing file[/tmp/greeting.txt]  
action create (/Users/mma/.chef/local-mode-cache/cache/cookbooks/  
greeting/providers/default.rb line 3)  
[2014-12-30T18:08:21+01:00] INFO: file[/tmp/greeting.txt] created  
file /tmp/greeting.txt  
  
- create new file /tmp/greeting.txt [2014-12-30T18:08:21+01:00]  
WARN: Could not set gid = 0 on /var/folders/fz/dcb5y3qs4m5g1hk8zrx  
d948m000gn/T/.greeting.txt20141230-49531-1rn88me, file modes not  
preserved  
[2014-12-30T18:08:21+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      2014-12-30 18:08:21.000000000 +0100  
+++ /var/folders/fz/dcb5y3qs4m5g1hk8zrx948m000g  
n/T/.greeting.txt20141230-49531-1rn88me      2014-12-30  
18:08:21.000000000 +0100  
@@ -1 +1,2 @@  
+Ohai, Chef!  
. . .TRUNCATED OUTPUT...
```

5. Validate the content of the generated file:

```
mma@laptop:~/chef-repo $ cat /tmp/greeting.txt  
Ohai, Chef!
```

How it works...

LWRPs live in cookbooks. A custom resource, which you will define in a file called `default.rb` in the `resources` directory of your cookbook, will be available under the cookbook name.

We create `greeting/resources/default.rb` and use it in our default recipe, as follows:

```
greeting "...." do  
end
```

Let's see what the resource definition in `greeting/resources/default.rb` looks like.

First, we define the actions, which our resource should support:

```
actions :create, :remove
```

Then, we define attributes you can pass to the resource when using it in your cookbook. In our case, we define two string attributes with their default values:

```
attribute :title, kind_of: String, default: "World"
attribute :path, kind_of: String, default: "/tmp/greeting.txt"
```

Now, we can use those actions and attributes in our recipe:

```
greeting "Ohai" do
  title "Chef"
  action :create
end
```

We defined the resource, now it's time to make it do something. The implementation of a resource lives in one or many providers. You might find multiple providers for the same resource for different operating systems. However, we keep it simple here and create only one provider in `greeting/providers/default.rb`.

The provider has to implement each action defined in the resource. In our case, we need to implement two actions: `create` and `remove`, as shown in the following code:

```
action :create do
  ...
end
action :remove do
  ...
end
```

Now, you can use pure Ruby and the existing Chef resources to make your provider 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 attributes 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`) attribute to fill the text file:

```
file new_resource.path do
  content "#{new_resource.name}, #{new_resource.title}!"
  action :create
end
```

The `remove` action works in a similar way to the `create` action, but calls the `file` resource's `delete` action, instead.

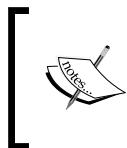
There's more...

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 providers, you need to make sure that your code is idempotent. This means that it only runs if it has to modify something. You should be able to run your code multiple times on the same machine, without executing unnecessary actions on each run.

If you want your resource to support `why-run` mode, you need to add the following to it:

```
def whyrun_supported?
  true
end
```

Then, you can wrap your code with a `converge_by` block. This will produce the message you pass to it during `why-run` mode, instead of executing the code inside, as shown in the following code:

```
converge_by("Doing something with #{@new_resource}") do
  ...
end
```

See also

- ▶ Read more about what LWRPs are at <http://docs.chef.io/lwrp.html>
- ▶ You find a more detailed explanation about how to create LWRPs at http://docs.chef.io/lwrp_custom.html
- ▶ Read more about the `why-run` mode in the *Using why-run mode to find out what a recipe might do* in Chapter 2, Evaluating and Troubleshooting Cookbooks and Chef Runs

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 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 within 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 Berksfile:

```
mma@laptop:~/chef-repo $ subl Berksfile
source 'https://supermarket.getchef.com'
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 my-ntp
Compiling Cookbooks...
Recipe: code_generator::cookbook
```

```
* directory[/Users/mma/work/chef-repo/cookbooks/my-ntp] action
create
  - create new directory /Users/mma/work/chef-repo/cookbooks/my-
ntp
```

2. Add your new my-ntp cookbook to the run list of your node:

```
mma@laptop:~/chef-repo $ knife node run_list add 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'
```

4. Make the default recipe from the ntp cookbook execute another command, which you 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...
[2015-01-02T21:24:54+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, which defines the resource we want to modify the code:

```
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 lines:

```
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, it will execute all the resources it built during the compile phase.

There's more...

If you don't want to modify the existing cookbooks, this is currently the only way to modify parts of recipes, which are not meant to be configured via attributes.

I don't like this approach too much. It is the exact 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 implementation details of another piece of code instead of depending on its public interface (in Chef recipes, the public interface is its attributes).

You should be aware of the fact that what you're doing is dangerous. 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 *Downloading and integrating cookbooks as vendor branches into your Git repository* recipe in *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 plugin 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 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 add server 'chef-client::config'

server:
  run_list:
    recipe[chef-client::config]
```

How to do it...

Let's write a simple Ohai plugin, which lists 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 2.0.1 successfully installed
```

2. Add your plugin to the `ohai` cookbook:

```
mma@laptop:~/chef-repo $ subl cookbooks/ohai/files/default/
plugins/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. Upload the modified ohai cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload ohai
Uploading ohai      [2.0.1]
```

4. 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]
```

5. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...
[2015-01-02T21:21:55+00:00] INFO: ohai[custom_plugins] reloaded
...
- re-run ohai and merge results into node attributes
...
[2015-01-02T21:21:55+00:00] INFO: ohai[custom_plugins] reloaded
...
- re-run ohai and merge results into node attributes
...
[2015-01-02T21:21:55+00:00] INFO: ohai[custom_plugins] reloaded
```

6. 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:

Nodes	Reports	Policy	
Showing All Nodes			
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/plugins` directory. The `ohai` cookbook will then upload your plugins to your node.

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 parameter `:default` says that this `collect_data` method runs on any platform.

You collect the node attributes in a Mash—an extended version of a Hash the code, 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.all_plugins
>> 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 of Ohai here: <https://github.com/opscode/ohai>
- ▶ Find the source code of the Ohai cookbook here: <https://github.com/opscode-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 it. 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, which can use any external libraries, there are no limits for 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 your name!

Getting ready

Make sure you have a Twitter user account and 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 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-5.13.0
5 gems installed
```

How to do it...

Let's create a knife plugin so that we can tweet by using the following knife command:

```
$ knife tweet "having fun building knife plugins"

1. Create a directory for your knife plugin inside your Chef repository:
mma@laptop:~/chef-repo $ mkdir -p .chef/plugins/knife

2. 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>"
```

Chef Language and Style

```
    config.access_token_secret = "<YOUR_ACCESS_TOKEN_SECRET>"  
  end  
  client.update("#{name_args.first} #getchef")  
end  
end  
end
```

3. Send your first tweet:

```
mma@laptop:~/chef-repo $ knife tweet "having fun with building  
knife plugins"
```

4. 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 into our Ruby file in order 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, no matter whether it would 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 the 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
- ▶ Finding the complete list of operating systems you can use in cookbooks
- ▶ Making recipes idempotent by using conditional execution

Introduction

In this chapter, we'll see some of the 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 the 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 there are certain environment variables set in your shell, which are not set during the Chef run. You might have set them manually or in your shell start up scripts—it does not really matter. You'll need to set them again in your recipe.

In this section, you will see how to set environment variables needed 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 --force  
Uploading my_cookbook [0.1.0]
```

3. Run the Chef client to create the tmp file:

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

...TRUNCATED OUTPUT...
[2015-01-02T22:46:30+01:00] 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 child processes run.

The execute resource spawns a child process of the Ruby process by 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 simply 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 you need to pass 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 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...
[2015-01-02T22:55:20+01:00] INFO: execute[echo max memory value
into tmp file] ran successfully

- execute echo 6553.6 > /tmp/max_mem
...TRUNCATED OUTPUT...
```

4. Validate that it worked:

```
user@server:~$ cat /tmp/max_mem
299523.2
```

How it works...

We calculate a value, which 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 multi-line strings, as well. You can define them like this:

```
command <<EOC
echo #{message} > /tmp/message
EOC
```

 EOC is the string delimiter. You're free to use whatever you want here. 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 the 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...  
[2015-01-02T22:59:10+01:00] 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 defined 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 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...  
[2015-01-02T23:04:29+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 a 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.

Your cookbooks cannot rely on hardcoded server names when you need in this situation, 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...  
[2015-02-19T21:32:00+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 criteria. 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, which you neither want to hard code 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

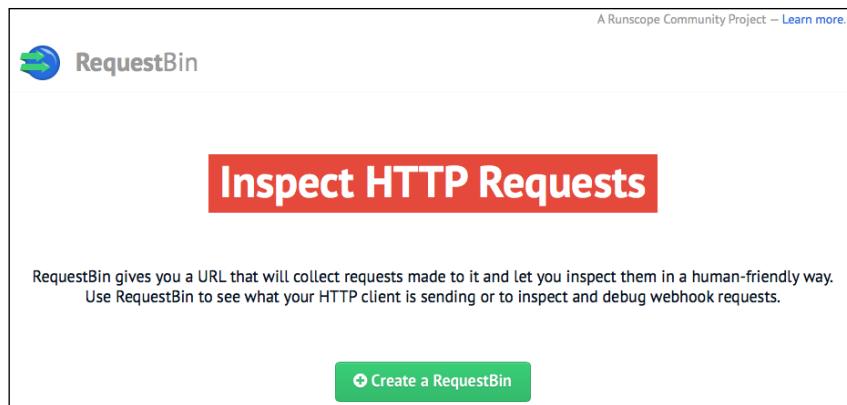
In the following example, we want to illustrate the usage of data bags by sending HTTP requests to a configurable HTTP endpoint. We don't want to hardcode the HTTP endpoint in our recipe. That's why we store it as a data bag item in a data bag.

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 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://requestbin.net> 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 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:

```
mma@laptop:~/chef-repo $ mkdir data_bags/hooks
```

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 $ 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` in order 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:

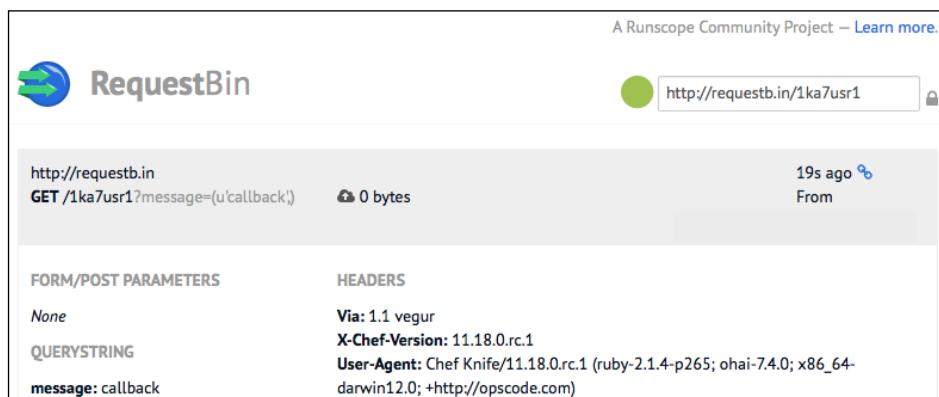
```
user@server:~$ sudo chef-client
```

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

```
[2015-01-02T20:08:20+01:00] 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:



A screenshot of the RequestBin website. The URL in the address bar is `http://requestb.in/1ka7usr1`. The page shows a single log entry from 19 seconds ago. The log entry details a GET request to `http://requestb.in/1ka7usr1?message=(u'callback')` with 0 bytes sent. Below the log entry, there are sections for 'FORM/POST PARAMETERS' and 'HEADERS'. Under 'FORM/POST PARAMETERS', it shows 'None' and 'message: callback'. Under 'HEADERS', it shows 'Via: 1.1 vegur', 'X-Chef-Version: 11.18.0.rc.1', and 'User-Agent: Chef Knife/11.18.0.rc.1 (ruby-2.1.4-p265; ohai-7.4.0; x86_64-darwin12.0; +http://opscode.com)'. The entire screenshot is framed by a light gray border.

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 JSON 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 create 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 end point.

You can use search to find certain data bag items and loop through 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 end points 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 everyone:

1. Edit the default recipe of `my_cookbook` to retrieve all HTTP hooks, which should be called by your recipe from your data bag:

```
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 *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 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 search at http://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
```

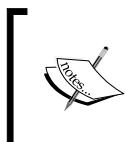
```
{  
    "id": "google",  
    "email": "some.one@gmail.com",  
    "password": "Oh! So secret?"  
}
```

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. Take a look at the `There's more...` section of 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 take a 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.

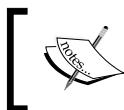
There's more...

Accessing encrypted data bag items from the command line with `knife` is usually not what you want. Let's take a 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 2222 .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 http://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, which 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, 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...
[2015-01-08T21:02:24+01:00] INFO: file[/etc/backup_config.json]
created file /etc/backup_config.json

- create new file /etc/backup_config.json
[2015-01-08T21:02:24+01:00] 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, which 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 take a look at how to retrieve it.

Getting ready

Log in to any of your Chef managed nodes and start the chef-shell:

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

How to do it...

Let's play around with the `node` object and take a 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  
  
=> ["eth0", "lo"]
```

3. List all the existing user accounts:

```
chef > node['etc']['passwd'].keys.sort  
  
=> ["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']

=> "utopic"
```

6. Find out which kernel modules are available:

```
chef > node['kernel']['modules'].keys.sort

=> ["auth_rpcgss", "dm_crypt", "drm", "e1000", "fscache", "i2c_piix4",
"ip_tables", "iptable_filter", "lockd", "mac_hid",
"nfs", "nfs_acl", "nfsd", "parport", "parport_pc", "pata_acpi",
"ppdev", "psmouse", "serio_raw", "sunrpc", "vboxguest", "vboxsf",
"vboxvideo", "x_tables"]
```

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.

Instead of using `node['lsb']['codename']`, you could use `node.lsb.codename` as well.

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. Package names might be different and 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 take a 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...  
[2015-01-08T21:24:39+01:00] INFO: Running on Ubuntu  
...TRUNCATED OUTPUT...
```

4. If you are not interested in a specific platform but you only need to know whether you run 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 will show:

```
[2015-01-08T21:25:20+01:00] INFO: Running on a debian  
derivative
```

How it works...

Ohai discovers the current node's operating system and stores it as a platform attribute with the node object. You can access it like any other attribute by using the Hash syntax, as follows:

```
node['platform']
```

Alternatively, you can use the method style 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 even `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 the 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-runsrvdir" 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 of the 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::Dscl 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/hw-cookbooks/runit>

Finding the complete list of operating systems you can use in cookbooks

You want to write cookbooks that work on different operating systems, such as Ubuntu, RedHat, Debian, or Windows.

Inside your cookbooks, you need to distinguish between these different platforms. And you need to tell your cookbook which platforms it supports. However, you don't know which platform values you can use inside `metadata.rb` or your recipes.

In this section, we'll look at a very simple way to ask Chef which values it defines for a platform.

How to do it...

Let's use plain Ruby to find out all the possible values for `platform` and use a subset of those in `metadata.rb`:

1. Print a list of supported platforms by querying the `Chef::Platform` class:

```
mma@laptop:~/chef-repo/cookbooks $ ruby -rchef -e
  puts Chef::Platform.platforms.keys.sort.join(", ")
aix, amazon, arch, centos, cloudlinux, debian, default, exherbo,
fedora,
... TRUNCATED OUTPUT...
ubuntu, windows, xcp, xenserver
```

2. Tell the users of your cookbook which platforms it supports:

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

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

How it works...

Chef maintains a set of supported operating system platforms it runs on in the `Chef::Platform` class. To query this class for the list of platforms, we use the Ruby command line.

We need to require the `chef` gem by adding an `-r` parameter to the `ruby` call.

The `-e` parameter contains the Ruby code that we want to execute. In our case, we use `puts` to print the result of our query to the console.

The `Chef::Platform` class holds a collection called **platforms**. We get its keys, sort them, and join the contents of the resulting Ruby array to a comma-separated string:

```
Chef::Platform.platforms.keys.sort.join(", ")
```

There's more...

Each platform in the `Chef::Platform` collection not only has the platform name as the key, but a set of default providers as well.

Providers contain the platform-specific implementation details for resources. For example, the package resource has providers to use Apt on Ubuntu, but Yum on RedHat:

```
mma@laptop:~/chef_helpster $ irb

2.1.0 :001 > require 'chef'
=> true
2.1.0 :002 > Chef::Platform.platforms[:ubuntu]
=> { :default=>{:package=>Chef::Provider::Package::Apt,
  :service=>Chef::Provider::Service::Debian,
  :cron=>Chef::Provider::Cron, :mdadm=>Chef::Provider::Mdadm} }
```

Instead of using the Ruby command line, we can use the Chef classes in the **Interactive Ruby shell (IRB)**.

You can change how your recipe works, depending on the platform it runs on (example taken from Opscode's apache cookbook):

```
service "apache2" do
  case node[:platform]
  when "centos", "redhat", "fedora", "suse"
    service_name "httpd"
    ...TRUNCATED OUTPUT...
  when "arch"
    service_name "httpd"
  end
  supports value_for_platform(
    "debian" => { ... },
    "ubuntu" => { ... },
    ...TRUNCATED OUTPUT...
  "default" => { ... }
)
  action :enable
end
```

This version of the apache cookbook sets up the apache service with different names and commands depending on the platform, and tells Chef which actions may be called to manage the apache service.

Chef sets the node attribute `:platform` according to the underlying operating system. You can use this node attribute to tailor your recipe code for each platform you need.

See also

- ▶ To see some examples on how to use the platform values, go to https://docs.chef.io/dsl_recipe.html

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, which 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 take a 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...
[2015-01-08T21:43:39+01:00] 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 simply queried 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 or not 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 shell command as well, 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 undertaken when setting up your nodes. In this chapter, we'll take a look at the various ways in which 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...
```

```
[2015-01-09T10:33:23+01:00] 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      2015-01-08 22:20:17.000000000
+0100
  +++ /var/folders/fz/dcb5y3qs4m5g1hk8zrx948m0000g
n/T/chef-rendered-template20150109-63309-ly6vmk      2015-01-09
10:33:23.000000000 +0100
  @@ -1,2 +1,4 @@
  -dailyrotate 31create
  +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 have to follow the given 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 (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 and set a flag called `enabled` 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/
dcb5y3qs4m5g1hk8zrx948m0000gn/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/dcb5y3qs4m5g1hk8zrx948m000gn
/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
```

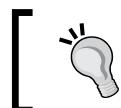
How it works...

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 the 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 `backend` 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 pre-calculated 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 bump into situations in which either the package you need is 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 'apt'
```

2. Install it to your local workstation:

```
mma@laptop:~/chef-repo $ berks install
Resolving cookbook dependencies...
Fetching cookbook index from https://supermarket.getchef.com...
Installing apt (2.6.1)
```

3. Upload it to your Chef server:

```
mma@laptop:~/chef-repo $ berks upload
Uploaded apt (2.6.1) 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 `apt` cookbook installed on your node.

How to do it...

Let's take a look at how you can install the `s3cmd` tool from the repository available at www.s3tools.org on a Ubuntu Lucid 12.04 LTS node:



Ubuntu Utopic 14.04 LTS already comes with `s3cmd` version 1.5.0-rc1-2 installed, and therefore, the following recipe isn't necessary to get the latest version of `s3cmd` installed.

1. Edit your cookbook's default recipe:

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

```
include_recipe "apt"
apt_repository "s3tools" do
  uri "http://s3tools.org/repo/deb-all"
  components ["stable/"]
  key "http://s3tools.org/repo/deb-all/stable/s3tools.key"
  action :add
end
package "s3cmd"
```

2. Edit your cookbook's metadata to add a dependency on the apt cookbook:

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

3. Upload the modified my_cookbook to the Chef server:

```
mma@laptop:~/chef-repo $ knife cookbook upload my_cookbook
Uploading my_cookbook [0.1.0]
```

4. Validate that the s3cmd package is not yet installed:

```
user@server:~$ dpkg -l s3cmd
No packages found matching s3cmd.
```

5. Validate that the default repository will install an older version of s3cmd (1.0.0-1):

```
user@server:~$ apt-cache showpkg s3cmd
Package: s3cmd
Versions:
1.0.0-1 (/var/lib/apt/lists/us.archive.ubuntu.com_ubuntu_dists_
precise_universe_binary-amd64_Packages)
```

6. Run the Chef client on your node:

```
user@server:~$ sudo chef-client
...TRUNCATED OUTPUT...
[2015-01-12T19:59:04+00:00] INFO: execute[apt-get update] ran
successfully

- execute apt-get update -o Dir::Etc::sourcelist='sources.
list.d/s3tools.list' -o Dir::Etc::sourceparts='-' -o
APT::Get::List-Cleanup='0'
[2015-01-12T19:59:04+00:00] INFO: execute[apt-get update] sending
run action to execute[apt-cache gencaches] (immediate)
  * execute[apt-cache gencaches] action runReading package
lists...
```

```
[2015-01-12T19:59:07+00:00] INFO: execute[apt-cache gencaches] ran successfully
```

```
- execute apt-cache gencaches
```

```
* apt_package[s3cmd] action install
  - install version 1.0.0-4 of package s3cmd
...TRUNCATED OUTPUT...
```

7. Ensure that the s3tools repository will install a newer version (1.0.0-4 instead of 1.0.0-1):

```
user@server:~$ apt-cache showpkg s3cmd
```

```
Package: s3cmd
Versions:
1.0.0-4 (/var/lib/apt/lists/s3tools.org_repo_deb-all_stable_
Packages) (/var/lib/dpkg/status)
```

8. Ensure that the s3cmd package is installed:

```
user@server:~$ dpkg -l
```

```
...TRUNCATED OUTPUT...
ii  s3cmd                               1.0.0-4   The ultimate
Amazon S3 and CloudFront command line client
```

How it works...

The apt cookbook provides an easy way to deal with additional APT repositories.



If you don't use Berkshelf, as described in the *Managing cookbook dependencies with Berkshelf* recipe in Chapter 1, *Chef Infrastructure*, you need to use knife cookbook site install to download the apt cookbook to your workstation and knife cookbook upload apt to install it on your Chef server.

We need to tell Chef that we want to use the apt cookbook by adding the depends call to our cookbook's metadata.rb file.

The apt cookbook defines the apt_repository resource. To be able to use it, we need to include the apt recipe in our default recipe:

```
include_recipe "apt"
```

As soon as the `apt` cookbook is available, we can add the third-party repository by using the `apt_repository` resource:

```
apt_repository "s3tools" do
  uri "http://s3tools.org/repo/deb-all"
  components ["stable/"]
  key "http://s3tools.org/repo/deb-all/stable/s3tools.key"
  action :add
end
```

After adding the third-party repository, we can install the desired package from there:

```
package "s3cmd"
```

See also

- ▶ Learn more about the `s3cmd` package at <http://s3tools.org>

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.

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.getchef.com'
cookbook 'apt'
cookbook 'build-essential'
```

2. Install it on your local workstation:

```
mma@laptop:~/chef-repo $ berks install
```

```
Resolving cookbook dependencies...
Fetching cookbook index from https://supermarket.getchef.com...
Installing build-essential (2.1.3)
Installing apt (2.6.1)
```

3. Upload it to your Chef server:

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

```
Uploaded apt (2.6.1) to: 'https://api.opscode.com:443/
organizations/awo'
Uploaded build-essential (2.1.3) 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 the Vagrant provision to get the required cookbooks installed on your node.

How to do it...

Let's take nginx as a well-known example for installing it 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 = "1.7.9"
bash "install_nginx_from_source" do
  cwd Chef::Config['file_cache_path']
  code <<-EOH
    wget http://nginx.org/download/nginx-#{version}.tar.gz
    tar zxf 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 a dependency on the `apt` cookbook:

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

```
...
depends "apt"
depends "build-essential"
```

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-1.7.9'
[2015-01-12T20:45:20+00:00] INFO: bash[install_nginx_from_source]
ran successfully
```

```
- execute "bash" "/tmp/chef-script20150112-13681-1m0wt4u"
...TRUNCATED OUTPUT...
```

5. Validate that nginx is installed:

```
user@server:~$ /usr/local/nginx/sbin/nginx -v
```

```
nginx version: nginx/1.7.9
```



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.



How it works...

The `bash` resource executes only if the `nginx` executable is not yet there. 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 reboot. In order 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`.



<<-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 as long as 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—it will only download the file if it needs to.

Change your default recipe in the following way to use the `remote_file` resource:

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

version = "1.7.9"

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 zxf 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/opscode-cookbooks/nginx/blob/master/recipes/source.rb>
- ▶ 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[2015-01-12T20:52:19+00:00] INFO: template[/tmp/trigger] created file /tmp/trigger
```

```
- create new file /tmp/trigger[2015-01-12T20:52:19+00:00] INFO: template[/tmp/trigger] updated file contents /tmp/trigger
```

```
- update content in file /tmp/trigger from none to e3b0c4  
(no diff)
```

```
[2015-01-12T20:52:19+00:00] INFO: template[/tmp/trigger] sending run action to bash[run_on_trigger] (immediate)
```

```
* bash[run_on_trigger] action runTriggered
```

```
[2015-01-12T20:52:19+00:00] INFO: bash[run_on_trigger] ran successfully
```

```
- execute "bash" "/tmp/chef-script20150112-16221-1q4r38y"
```

```
...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, which 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 case. 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 too big so that it doesn't take 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...
```

```
[2015-01-15T08:48:25+01:00] 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
[2015-01-15T08:48:25+01:00] INFO: Processing directory[/tmp/chef.github.com/images] action create (dynamically defined)
[2015-01-15T08:48:25+01:00] INFO: directory[/tmp/chef.github.com/images] created directory /tmp/chef.github.com/images
```

```
  - create new directory /tmp/chef.github.com/images
[2015-01-15T08:48:25+01:00] INFO: directory[/tmp/chef.github.com/images] owner changed to 0
[2015-01-15T08:48:25+01:00] INFO: directory[/tmp/chef.github.com/images] group changed to 0
```

```
[2015-01-15T08:48:25+01:00] 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 the same. 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...
```

```
[2015-01-15T08:58:24+01:00] 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 14.04 nodes:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
ubuntu-14.04/message.erb
```

```
Hello from Ubuntu 14.04!
```

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...
[2015-01-16T18:19:16+01:00] INFO: template[/tmp/message] created
file /tmp/message

- create new file /tmp/message
[2015-01-16T18:19:16+01:00] WARN: Could not set gid = 0 on /
var/folders/fz/dcb5y3qs4m5g1hk8zrxd948m0000gn/T/chef-rendered-
template20150115-74876-coftwo, file modes not preserved
```

```
[2015-01-16T18:19:16+01:00] 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 14.04!
```

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 14.04:

```
my_cookbook/templates/my_node.example.com/message.erb  
my_cookbook/templates/ubuntu-14.04/message.erb  
my_cookbook/templates/ubuntu-14/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-14.04` or `ubuntu-14`, and so on.

The only directory, which is mandatory, is the `default` directory.

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 (SSHD)
- ▶ Enabling passwordless sudo
- ▶ Managing NTP
- ▶ Managing nginx
- ▶ Creating nginx virtual hosts
- ▶ Creating MySQL databases and users
- ▶ Managing WordPress sites
- ▶ Managing Ruby on Rails applications
- ▶ Managing Varnish
- ▶ Managing your local workstation

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 take a look at how to install and manage more advanced applications. Our examples mainly cover a web application stack using nginx as a web server, MySQL as a database, and WordPress or Ruby on Rails for the web application.

We'll close the chapter with showing you 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 they) have access. You definitely 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 let a recipe create and remove the users, accordingly.

Let's take a look at how you can do that.

Getting ready

Make sure the 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 `Berksfile` in your Chef repository which 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 our 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"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install  
...TRUNCATED OUTPUT...  
Installing users (1.7.0)  
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
...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` **Light Weight Resource Provider (LWRP)**, provided by the `users` cookbook.

The `manage_users` LWRP takes its name attribute "staff" as the group name it should manage. It searches for data bag items, which have this group in their `groups` entry, and uses every entry found to create those users and groups.

[ The `manage_users` LWRP 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 LWRP, we make sure that it searches for both: users to remove and users to add.

There's more...

Let's take a 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
...
- remove user user[mma]
...

```

4. Validate that the user `mma` does not exist anymore:

```
user@server:~$ fgrep mma /etc/passwd
...

```



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/opscode-cookbooks/users>
- ▶ The *Using data Bags* recipe in *Chapter 4, Writing Better Cookbooks*

Securing the Secure Shell Daemon (SSHD)

Depending on your Linux flavor, the `ssh` daemon might listen on all network interfaces on the default port, and allow root and password logins.

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.

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 `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile  
  
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```



Note that 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"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install
```

```
Resolving cookbook dependencies...
...
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['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
```

```
...
Uploading my_cookbook (0.1.0) to: 'https://api.opscode.com:443/
organizations/awo'
...

```

5. Run the Chef client on your node:

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

...
[2015-02-09T20:15:22+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]
...

```

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 price of moving from a privileged port to a non-privileged port on your node. This presents the risk that someone on your box might highjack 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/opscode-cookbooks/openssh>
- ▶ Find a detailed list of all attributes the openssh cookbook offers to configure sshd at <https://github.com/opscode-cookbooks/openssh/blob/master/attributes/default.rb>

Enabling passwordless sudo

You have secured your sshd so that your 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 `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 make 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"
```

2. Install your cookbook's dependencies:

```
mma@laptop:~/chef-repo $ berks install

Installing sudo (2.7.1)
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 (2.7.1) 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
[2015-02-09T20:22:20+00:00] INFO: template[/etc/sudoers] backed up
to /var/chef/backup/etc/sudoers.chef-20150209202220.755848
[2015-02-09T20:22:20+00:00] INFO: template[/etc/sudoers] updated
file contents /etc/sudoers

  - update content in file /etc/sudoers from d49b83 to cd7842
  --- /etc/sudoers2014-10-27 02:18:28.623220955 +0000
  +++ /tmp/chef-rendered-template20150209-5333-cviwbs2015-02-09
20:22:20.751179595 +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 LWRP from the `sudo` cookbook, you can manage each group or user individually. The LWRP will place configuration fragments inside `/etc/sudoers.d`. You can use 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/opscode-cookbooks/sudo>

Managing NTP

Your nodes should always have synchronized clocks, if nothing else, because 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. Better avoid them in the first place by using NTP on every node.

Getting ready

Create `Berksfile` in your Chef repository including the `ntp` cookbook:

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

Install the `ntp` cookbook:

```
mma@laptop:~/chef-repo $ berks install
```

```
Resolving cookbook dependencies...
Using ntp (1.7.0)
```

Upload the `ntp` cookbook to the Chef server:

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

...TRUNCATED OUTPUT...
Uploading ntp (1.7.0) to: 'https://api.opscode.com:443/organizations/
awo'
...TRUNCATED OUTPUT...
```

How to do it...

Let's create a role called "`base`", which 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[2015-02-10T07:56:03+00:00] INFO: template[/etc/ntp.conf] backed up to /var/chef/backup/etc/ntp.conf.chef-20150210075603.587108
[2015-02-10T07:56:03+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[2015-02-10T07:56:05+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 server is running
```

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 default attributes in the `ntp` namespace. In the preceding example, we configured the upstream NTP servers for our node to query.



If you're on Debian or Ubuntu, the `ntp` cookbook installs `ntpdate` as well. `ntpdate` is there to quickly synchronize and set a node's date and time.



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/gmiranda23/ntp>
- ▶ The *Overriding attributes* recipe in Chapter 4, *Writing Better Cookbooks*

Managing nginx

Say you need to set up a website, which handles a lot of traffic simultaneously. nginx is a web server that is designed to handle many connections 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` community cookbook to do just that.

Getting ready

Let's get ready for set up of nginx:

1. Create `Berksfile` in your Chef repository including the `nginx` cookbook:

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

2. Install the `nginx` cookbook:

```
mma@laptop:~/chef-repo $ berks install
...TRUNCATED OUTPUT...
Installing nginx (2.7.4)
...TRUNCATED OUTPUT...
```

3. Upload the `nginx` cookbook to your Chef server:

```
mma@laptop:~/chef-repo $ berks upload
Using nginx (1.7.0)
...TRUNCATED OUTPUT...
Uploading nginx (1.7.0) 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" => "1.7.9",  
  "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" =>  
    "1aafc7f0f530ddaa86d4ba24c14941ce0a584cc896f8261d6218f99625318f8a",  
    "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-1.7.9.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...
[2015-02-10T19:47:37+00:00] INFO: bash[compile_nginx_source]
sending restart action to service[nginx] (delayed)
  * service[nginx] action restart[2015-02-10T19:47:39+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-1.7.9/sbin/nginx -v

nginx version: nginx/1.7.9
built by gcc 4.9.1 (Ubuntu 4.9.1-16ubuntu6)
TLS SNI support enabled
configure arguments: --prefix=/opt/nginx-1.7.9 --conf-path=/etc/
nginx/nginx.conf --sbin-path=/opt/nginx-1.7.9/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 would have been to use `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 `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-1.7.9/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 preceding examples.



The `nginx` cookbook sets up handling of sites and its configuration in a similar way to Debian's way of configuring Apache2. You can use `nxdissite` 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/master/recipes/source.rb>
- ▶ Find the `application_nginx` cookbook on GitHub at https://github.com/poise/application_nginx
- ▶ Find the HTTP Upload Progress Module at <http://wiki.nginx.org/HttpUploadProgressModule>
- ▶ Learn how to calculate checksums for the `remote_file` resource at [https://coderwall.com/p/bbfjrw/calculate-checksum-for-chef-s-remote_file-resource](https://coderwall.com/p/bbfjrw/calculate-checksum-for-chef-s-remote-file-resource)
- ▶ The *Overriding attributes* recipe *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 `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",
  "enable_default_site" => 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 "nginx"
```

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 "nginx"  
  
nginx_site "default" do  
  enable false  
end  
  
app_name = "my_app"  
app_home = "/srv/#{app_name}"  
  
template "#{node[:nginx][:dir]}/sites-available/#{app_name}" do
```

```
source "nginx-site-#{app_name}.erb"
owner  "root"
group  "root"
mode    "0644"
variables :app_home => app_home
notifies :restart, resources(:service => "nginx")
end

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}"
```

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...
Uploading 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...
[2015-02-22T22:59:34+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
--2015-02-22 23:27:36-- 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'

100% [=====] 21      -.- K/s   in 0s

2015-02-22 23:27:36 (2.00 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 installs a template for the `nginx` configuration file. The template ends up as `/etc/nginx/sites-available/my_app` and gets symlinked to `/etc/nginx/sites-enabled/my_app`.

Next, we create 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, consider the following line:

```
nginx_site "#{app_name}"
```

Replace the preceding line with the following:

```
nginx_site "#{app_name}" do
  enable false
end
```

After uploading the modified cookbook and running the Chef client again, you should not be able to retrieve `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` in the *Managing nginx* recipe in this chapter
- ▶ Read more about the `nginx_site` resource at https://github.com/opscode-cookbooks/nginx/blob/master/definitions/nginx_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 `mysql` cookbook.

The database cookbook provides resources for managing databases and database users for MySQL, PostgreSQL, and Microsoft SQL Server. The `mysql` cookbook installs MySQL client and server.

Let's see how we can install 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 `Berksfile` in your Chef repository includes `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
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 the dependencies on the database and mysql cookbooks:

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

...
depends "mysql2_chef_gem"
depends "database"
depends "mysql"
```

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 '5.5'
  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 we can log in to our MySQL server with the user that we have just created and 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 5.5 listening to port 3306 and start it:

```
mysql_service 'default' do
  port '3306'
  version '5.5'
  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/opscode-cookbooks/database>.
- ▶ Find the mysql cookbook on GitHub at <https://github.com/chef-cookbooks/mysql>.

Managing WordPress sites

You need to enable your business users to manage their own website. Fast. WordPress has come a long way. You might have seen it as a simple blogging tool, however, in recent years, it has grown into a fully featured content management system. Fortunately, managing WordPress with Chef is pretty straightforward.

Let's take a look at how to do it.

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 `Berksfile` in your Chef repository including `my_cookbook`:

```
mma@laptop:~/chef-repo $ subl Berksfile
cookbook 'my_cookbook', path: './cookbooks/my_cookbook'
```

If you're using Vagrant for your node, you need to configure a bridged network to be able to browse to your WordPress installation from your local workstation:

```
Vagrant::Config.run do |config|
  config.vm.network :bridged
end
```

How to do it...

We'll install WordPress by using the community cookbook:

1. Edit your cookbook's metadata to make sure it depends on the `wordpress` cookbook:

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

...
depends "wordpress"
```

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 to set some attributes and include the `wordpress` cookbook:

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

node.default['apache']['mpm'] = 'prefork'
node.default['wordpress']['db']['database'] = "my_wordpress"
node.default['wordpress']['db']['user'] = "me"
node.default['wordpress']['db']['password'] = "my_password_11"

include_recipe 'wordpress'
```

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...
[2015-02-25T21:55:03+00:00] INFO: execute[a2ensite wordpress.conf]
ran successfully

- execute /usr/sbin/a2ensite wordpress.conf

...TRUNCATED OUTPUT...
```

6. Validate whether WordPress is installed on your host by firing up your browser and navigating to the WordPress installation page:



How it works...

The `wordpress` cookbook installs a full Apache-MySQL-PHP-stack. That's why, when you're installing it in your repository, it will install quite a few supporting cookbooks as well.

To use the `wordpress` cookbook, you simply include it in your own cookbook's recipe:

```
include_recipe 'wordpress'
```

Because we do not like the default values for the database name, database user, and the password for the database user, we override those attributes at the beginning of our default recipe:

```
node.default['wordpress']['db']['database'] = 'my_wordpress'  
node.default['wordpress']['db']['user'] = 'me'  
node.default['wordpress']['db']['password'] = 'my_password_11'
```

You can lookup the default values in `cookbooks/wordpress/attributes/default.rb`.

There's more...

The `wordpress` cookbook installs the complete stack but does not set up your first blog. It asks you to call the WordPress installation page with your browser to set up your first blog.

If you already have a tarball of your blog available, you could deliver it to your node, as described in the *Distributing directory trees* recipe in *Chapter 5, Working with Files and Packages*.

See also

- ▶ You can find the `wordpress` cookbook on GitHub at <https://github.com/brint/wordpress-cookbook>.

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 unicorn 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 `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"
```

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

application "rails-app" do
  packages %w[build-essential ruby2.1-dev runit git libsqlite3-dev
libssl-dev]

  path "/usr/local/www/rails-app"
  owner "www-data"
  group "www-data"

  environment_name "development"

  repository "https://github.com/mmarschall/rails-app.git"

  rails do
    gems %w[bundler]

    database_template "sqlite3_database.yml.erb"

    database do
      adapter "sqlite3"
      database "db/rails-app.sqlite3"
    end
  end

  unicorn do
  end
end
```

4. Add your own template file to your database.yml file:

```
mma@laptop:~/chef-repo $ subl cookbooks/my_cookbook/templates/
default/sqlite3_database.yml.erb

<%= @rails_env %>
  adapter: <%= @database['adapter'] %>
  host: <%= @host %>
  database: <%= @database['database'] %>
  pool: 5
  timeout: 5000
```

5. 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...
```

6. Run the Chef client on your node:

```
user@server:~$ sudo chef-client  
...TRUNCATED OUTPUT...  
- update release history data[2015-02-27T20:45:06+00:00] INFO:  
deploy_revision[rails-app] deployed to /usr/local/www/rails-app  
  
* execute[/etc/init.d/rails-app hup] action run  
[2015-02-27T20:45:06+00:00] INFO: execute[/etc/init.d/rails-app  
hup] ran successfully  
  
- execute /etc/init.d/rails-app hup  
...TRUNCATED OUTPUT...
```

7. Validate whether your Rails application is up and running by hitting your node at port 8080:

```
user@server:~$ wget localhost:8080  
2015-02-27 21:35:37 (346 MB/s) - 'index.html' saved [14935]
```

8. Then, you can have a look at the downloaded file to verify whether the **Welcome page** of your Rails app shows up:

```
user@server:~$ cat index.html  
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Ruby on Rails: Welcome aboard</title>  
  ...
```

How it works...

Chef provides an abstract application cookbook to deploy web applications. We call our application `rails-app`:

```
application "rails-app" do  
  ...  
end
```

Inside the application block, we define the details of our web app. First, we need to install a few operating system packages as follows:

```
packages %w[build-essential ruby2.1-dev runit git libsqlite3-dev  
libssl-dev]
```

The `ruby2.1-dev` package will make sure that we have 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.

Since we're going to use unicorn to run our Rails application, we need to install `runit` because that's the way that unicorn is installed at the time of writing this book.

Git is required to be able to check out our repository from www.github.com.

Finally, we're using SQLite for our Rails application and need to install it first.

The next step is to configure the deployment details—where our app should go (`path`), which user and group should own the application (`owner`, `group`), and where our app should find the source code of our app (`repository`):

```
path "/usr/local/www/rails-app"  
  owner "www-data"  
  group "www-data"  
  ...  
  repository "https://github.com/mmarschall/rails-app.git"
```

Make sure you enabled the `unicorn` gem in your Rails application's `Gemfile`:

```
gem 'unicorn'
```

If you don't want to run your application in the production environment, you can specify the desired `environment_name` in your cookbook, just as we did:

```
environment_name "development"
```

Your application will be fetched from www.github.com and the cookbook will install it in a directory structure similar to a directory structure that is maintained by **Capistrano**. It will put the current revision of your app into the `releases` directory and create a **symlink** to it as `current`.

Now, it's time to define the Rails-specific things. First of all, we want to install the `bundler` gem because our Rails application uses `Gemfile` for its dependencies:

```
rails do  
  gems %w[bundler]  
  ...  
end
```

The %w[] syntax creates an array of strings. You could write ["bundler"] instead. It doesn't make any difference for one element, but when placing multiple elements into your array, you save a lot of double quotes and commas with the %w syntax.

Since our Rails application uses SQLite as its persistence store, we need to use our own template for the database.yml file:

```
database_template "sqlite3_database.yml.erb"
```

Then, we can use a database block to populate it with the values we need:

```
database do
  adapter "sqlite3"
  database "db/rails-app.sqlite3"
end
```

We need to tell our Rails application that we're using a SQLite database and want it to store its data in a file called db/rails-app.sqlite3.

Finally, we need to tell our cookbook that we want to run our Rails application by using unicorn. An empty block will suffice, as long as we don't want to change any default attributes such as port number or number of workers:

```
unicorn do
end
```

There's more...

Usually, the application cookbook's deploy resource will only deploy new revisions of your Rails app. If you want to ensure that it grabs the same revision again and again, you need to call the force_deploy action on your application resource:

```
application "rails-app" do
  ...
  action :force_deploy
end
```

If you want to use a new or existing MySQL server, you can assign it a role, for example, rails_database_master, and pass that role name to the application resource. It will then search for the node and use its IP address in database.yml:

```
application "rails-app" do
  ...
end
```

```
database_master_role " rails_database_master"
end
```

In this case, you don't need to use your own `database.yml` template.

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
- ▶ The *Using search to find nodes* recipe in Chapter 4, *Writing Better Cookbooks*

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 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 8080. We'll set up Varnish to use `localhost:8080` 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 `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"
```



If you don't have the `apt` cookbook in your node's run list (which you should), you need `include_recipe "apt"` in your cookbook's default recipe.

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

include_recipe 'apt'

varnish_install "webapp" do
  vendor_repo true
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...
* apt_package [varnish] action install
```

```
- install version 4.0.1-1 of package varnish
* service[varnish] action enable (up to date)
* service[varnish] action restart[2015-02-27T23:38:54+00:00]
INFO: service[varnish] restarted

- restart service service[varnish]
...TRUNCATED OUTPUT...
```

6. Validate whether your Varnish cache is up and running by hitting your node at port 6081:

```
user@server:~$ wget localhost:6081
2015-02-27 23:39:41 (16.4 MB/s) - `index.html' saved [14900]
```

How it works...

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 "webapp" do
  vendor_repo true
end
```

This call will install, configure, and start the Varnish server listening to its default port 6081.

There's more...

If you want to build your own configuration file instead of the default one provided by the apt package, you can use `varnish_default_config` in your recipe.

You can change the backend host Varnish uses, as shown in the following code:

```
varnish_default_vcl do
  backend_host 'webapp.example.com'
  backend_port 8000
end
```

Use `varnish_log` 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

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 take a look at how to install applications and tweak settings on your local development box with Chef.

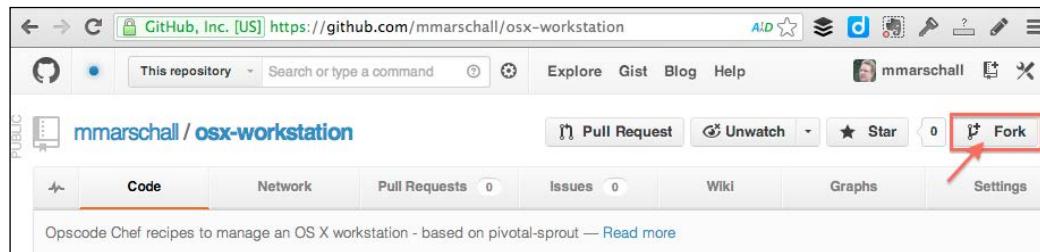


This example is based on recipes for OS X only, but you can tweak it to run on Windows or Linux, too.

Getting ready

First, we need to prepare our own repository for our individual setup:

1. Fork the `github.com/mmarschall/kitchenplan-config` repository:



2. Clone it to your local development box, replacing <YOUR GITHUB USER> with the name of your GitHub account:

```
mma@laptop:~/ $ git clone https://github.com/<YOUR GITHUB USER>/kitchenplan-config.git
```

3. Go into your clone of the `kitchenplan-config` repository:
`mma@laptop:~/ $ cd kitchenplan-config`
4. Make sure you have the `kitchenplan` gem installed by running the following code:

```
mma@laptop:~/kitchenplan-config $ gem install kitchenplan  
...TRUNCATED OUTPUT...  
Installing kitchenplan (2.1.18)  
...TRUNCATED OUTPUT...
```

How to do it...

Let's set up Kitchenplan to use a few readymade recipes:

1. Make sure you have a configuration for your user that tells Kitchenplan what to install on your box:

```
mma@laptop:~/kitchenplan-config $ subl config/people/<YOUR OSX  
USER>.yml  
  
recipes:  
  global:  
    mac_os_x:  
      - osxdefaults::dock_position_the_dock_on_the_left_side  
  attributes:  
    homebrewalt:  
      cask_apps:  
        - free-ruler
```

2. Commit and push your changes to your `kitchenplan-config` repository:

```
mma@laptop:~/kitchenplan-config $ git add config/people/<YOUR OSX  
USER>.yml  
mma@laptop:~/kitchenplan-config $ git commit -a -m "initial  
config"  
mma@laptop:~/kitchenplan-config $ git push origin master
```

3. Set up Kitchenplan on your development box. Choose `y` when asked whether you have a config repository and provide the GitHub URL of your `kitchenplan-config` repository:

```
mma@laptop:~/kitchenplan-config $ kitchenplan setup
```

```
Do you have a config repository? [y,n] y  
Please enter the clone URL of your git config repository: git@  
github.com:mmarschall/kitchenplan-config.git
```

```
-> Making sure /opt exists and I can write to it
    run sudo mkdir -p /opt from "."
Password:
    run sudo chown -R mma /opt from "."
sh: line 0: cd: /opt/kitchenplan: No such file or directory
-> Fetching git@github.com:mmarschall/kitchenplan-config.git to /
opt/kitchenplan.
    run git clone -q git@github.com:mmarschall/kitchenplan-
config.git kitchenplan from "/opt"
    run git submodule init from "/opt/kitchenplan"
    run git submodule update from "/opt/kitchenplan"
...TRUNCATED OUTPUT...
```

4. Let Kitchenplan provision your box:

```
mma@laptop:~/kitchenplan-config $ kitchenplan provision
...TRUNCATED OUTPUT...
* homebrewalt_cask[free-ruler] action cask

* execute[installing cask free-ruler] action run
  - execute sudo -u mma /usr/local/bin/brew cask install
--appdir=/Applications free-ruler
...TRUNCATED OUTPUT...
Recipe: osxdefaults::dock_position_the_dock_on_the_left_side
  * osxdefaults_defaults[Move the Dock to the left side of the
screen] action write

  * 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...
=> Installation complete!
```

5. Now, your dock should be located at the left-hand side of the screen and the application **Free Ruler** should show up in your Applications folder.

How it works...

Kitchenplan is a quick and easy way to configure your box. It uses **Librarian** and Chef solo to manage cookbooks. That's why Kitchenplan still uses Chef 11 at the time of writing this book.

Kitchenplan setup puts everything it needs into the `/opt/kitchenplan` directory. It will check out your `kitchenplan-config` repository and run Chef solo from there.

You can configure what to install in YAML files that are placed in the config directory of your kitchenplan-config repository. You can configure OS X using the mac_os_x recipe and you can install applications using homebrew by listing them in the attributes section of your config file:

```
recipes:  
  global:  
    mac_os_x:  
      - osxdefaults::dock_position_the_dock_on_the_left_side  
  attributes:  
    homebrewalt:  
      cask_apps:  
        - free-ruler
```

Kitchenplan will use Librarian to install all cookbooks defined in Cheffile which is located in your kitchenplan-config repository and then converge all recipes listed in your YAML files. Before converging, it will set all the given node attributes that will be used by the recipes.

There's more...

You can create various config files for your users and groups, and a default.yml. Kitchenplan will merge all YAML files before provisioning your box using the currently logged in username to find the correct file in the config/people directory.

See also

- ▶ Find Kitchenplan at <https://github.com/kitchenplan/kitchenplan>.
- ▶ You can find more configuration examples here: <https://github.com/kitchenplan/kitchenplan-config-test> and here <https://github.com/roderik/kitchenplan-config>.

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 your infrastructure using Chef Provisioning
- ▶ 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
- ▶ Building high-availability services using heartbeat
- ▶ 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
- ▶ Loading your Chef infrastructure from a file with spiceweasel and knife

Introduction

In the preceding chapters, we mostly looked at individual nodes. Now, it's time to consider your infrastructure as a whole. 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 your infrastructure using Chef Provisioning

You know how to use Chef to manage the software on individual machines and you know how to use knife to bootstrap individual nodes. Chef Provisioning helps you to use the power of Chef to create your whole infrastructure for you.

No matter whether you want to create a cluster of Vagrant boxes, Docker instances, or Cloud servers, Chef Provisioning lets you define your infrastructure in a simple recipe and run it idempotently.

Let's see how to create a Vagrant machine using a Chef recipe.

Getting ready

Make sure that you have your `Berksfile`, `my_cookbook` and `web_server` roles ready to create an nginx site, as described in *Creating nginx virtual hosts* section in chapter 6, *Users and Applications*.

How to do it...

Let's see how to create a Vagrant machine and install nginx on it:

1. Describe your Vagrant machine in a recipe called `mycluster.rb`:

```
mma@laptop:~/chef-repo $ subl mycluster.rb

require 'chef/provisioning'

with_driver 'vagrant'
with_machine_options :vagrant_options => { 'vm.box' => 'opscode-ubuntu-14.04' }

machine 'web01' do
  role 'web_server'
end
```

2. Install all required cookbooks in your local chef-repo:

```
mma@laptop:~/chef-repo $ berks install
mma@laptop:~/chef-repo $ berks vendor cookbooks

Resolving cookbook dependencies...
Using apt (2.6.1)
...TRUNCATED OUTPUT...
Vendorizing yum-epel (0.6.0) to cookbooks/yum-epel
```

3. Run the Chef client in local mode to bring up the Vagrant machine and execute a Chef run on it:

```
mma@laptop:~/chef-repo $ chef-client -z mycluster.rb

[2015-03-08T21:09:39+01:00] INFO: Starting chef-zero on host
localhost, port 8889 with repository at repository at /Users/mma/
work/chef-repo
...TRUNCATED OUTPUT...
Recipe: @recipe_files:::/Users/mma/work/chef-repo/mycluster.rb
  * machine[webserver] action converge[2015-03-08T21:09:43+01:00]
INFO: Processing machine[web01] action converge (@recipe_files::/
Users/mma/work/chef-repo/mycluster.rb line 6)
...TRUNCATED OUTPUT...
[2015-03-08T21:09:47+01:00] INFO: Executing sudo chef-client -l
info on vagrant@127.0.0.1

[web01] [2015-03-08T20:09:21+00:00] INFO: Forking chef
instance to converge...
      Starting Chef Client, version 12.1.0
      ...TRUNCATED OUTPUT...
      Chef Client finished, 18/25 resources updated in
73.839065458 seconds
...TRUNCATED OUTPUT...
[2015-03-08T21:11:05+01:00] INFO: Completed chef-client -l info on
vagrant@127.0.0.1: exit status 0
      - run 'chef-client -l info' on web01
[2015-03-08T21:11:05+01:00] INFO: Chef Run complete in 82.948293
seconds
...TRUNCATED OUTPUT...
Chef Client finished, 1/1 resources updated in 85.914979 seconds
```

4. Change into the directory where Chef put the Vagrant configuration:

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

5. Validate that there is a Vagrant machine named web01 running:

```
mma@laptop:~/chef/vms $ vagrant status
```

Current machine states:

```
web01           running (virtualbox)
```

6. Validate that nginx is installed and running on the Vagrant machine:

```
mma@laptop:~/chef/vms $ vagrant ssh
```

```
vagrant@web01:~$ wget localhost:80
```

...TRUNCATED OUTPUT...

```
2015-03-08 22:14:45 (2.80 MB/s) - 'index.html' saved [21/21]
```

How it works...

Chef Provisioning comes with a selection of drivers for all kinds of infrastructures, including Fog (supporting Amazon EC2, OpenStack, and others), VMware VSphere, Vagrant (supporting Virtualbox and VMware Fusion), various Containers, such as LXC Docker and Secure Shell (SSH).

In this recipe, we make sure that we can use the directives provided by Chef Provisioning by requiring `chef/provisioning` library.

Then, we configure the driver that we want to use. We use Vagrant and tell Chef to use the `opscode-ubuntu-14.04` Vagrant box to spin up our machine.

Using the `machine` resource, we ask Chef to spin up a Vagrant machine and configure it using Chef by applying the role `web_server`.

The `web_server` role uses the cookbook `my_cookbook` to configure the newly created Vagrant machine. To make sure that all the required cookbooks are available to Chef, we use `berks install` and `berks vendor cookbooks`. The `berks vendor cookbooks` installs all the required cookbooks in the local `cookbooks` directory. The Chef client can access the cookbooks here, without the need for a Chef server.

Finally, we use the Chef client to execute our Chef Provisioning recipe. It will spin up the defined Vagrant machine and execute a Chef client run on it.

Chef Provisioning will put the Vagrant Virtual Machine (VM) definition into the directory `~/chef/vms`. To manage the Vagrant VM, you need to change to this directory.

There's more...

Instead of using the `with_driver` directive, you can use the `CHEF_DRIVER` environment variable:

```
mma@laptop:~/chef-repo $ CHEF_DRIVER=vagrant chef-client -z  
mycluster.rb
```

You can create multiple instances of a machine by using the `machine_image` directive in your recipe:

```
machine_image 'web_server' do
  role 'web_server'
end
1.upto(2) do |i|
  machine "web0#{i}" do
    from_image 'web_server'
  end
end
```

See also

- ▶ Find the source code of the Chef Provisioning library at GitHub: <https://github.com/chef/chef-provisioning>
- ▶ Find the Chef Provisioning documentation at <https://docs.chef.io/provisioning.html>
- ▶ Learn how to set up a Chef server using Chef Provisioning: <https://www.chef.io/blog/2014/12/15/sysadvent-day-14-using-chef-provisioning-to-build-chef-server/>

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 2013.02.13-1ubuntu1
apt zlib1g-dev 1:1.2.8.dfsg-1ubuntu1
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, which 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 have to specify a query to find your nodes. It is usually a good idea to test your queries by running a command such as `uptime` (instead of dangerous commands such as `sudo restart now`). Your query will obviously use the node index and the complete `knife search` query syntax is available.

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 the following location: 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"
```

2. Install the dependent cookbooks:

```
mma@laptop:~/chef-repo $ berks install  
...TRUNCATED OUTPUT...  
Installing snmp (3.0.1)  
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 (3.0.1) 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"
```

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 spread monitoring packages 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'
cookbook 'nagios'
```

2. Install the `nagios` cookbook:

```
mma@laptop:~/chef-repo $ berks install

Using nagios (7.0.2)
...TRUNCATED OUTPUT...
```

3. Upload the `nagios` cookbook to the Chef server:

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

...TRUNCATED OUTPUT...
Uploading nagios (7.0.2) 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 email 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...
[2015-06-12T22:50:09+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, there is a navigation menu with sections like General, Home, Documentation, Current Status, and Problems. Under Current Status, there are links for Tactical Overview, Map, Hosts, Services, Host Groups (Summary, Grid), Service Groups (Summary, Grid), and Problems (Services (Unhandled), Hosts (Unhandled), Network Outages). A Quick Search bar is also present. On the right, a large box displays the "Current Network Status" with the last update being Fri Mar 13 20:49:24 UTC 2015, updated every 90 seconds. It shows Nagios® Core™ 3.5.1 - www.nagios.org, and the user is logged in as mma. Notifications are disabled. Below this, there are three blue links: View History For all hosts, View Notifications For All Hosts, and View Host Status Detail For All Hosts. Further down, it shows "Display Filters:" with Host Status Types: All, Host Properties: Any, Service Status Types: All, and Service Properties: Active Checks. A dropdown for "Limit Results" is set to 100. Below that, a table shows a single service: vagrant ssh OK 2015-03-13 20:48:35. At the bottom, it says "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 email 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

- ▶ Find the `nagios` cookbook on GitHub at: <https://github.com/tas50/nagios>

Building high-availability services using heartbeat

If you want to offer any IP-based service with automatic failover to provide **high availability (HA)**, you can use heartbeat to create an HA cluster.

Heartbeat will run on two or more nodes and ensure that the IP address you chose to make highly available will switch to a working node, if one of them goes down. This way, you have a failover IP address, which is guaranteed to reach a running host, as long as there is one left.

Let's take a look at how to install heartbeat on your nodes and configure it with a failover IP address.

Getting ready

Make sure that you have two machines, named `ha1` and `ha2`, up and running. You can use a `Vagrantfile` like this or use Chef Provisioning to set them up. You need to replace `awo` with a short name of your own Chef organization:

```
def define_node(config, node_name, ip_address=nil, the_recipe=nil)
  config.vm.define node_name do |node|
    node.vm.hostname = node_name.to_s
    node.vm.network :private_network, ip: ip_address if ip_address
```

```
node.vm.provision :chef_client do |chef|
  chef.provisioning_path = "/etc/chef"
  chef.chef_server_url = "https://api.opscode.com/organizations/
awo"
  chef.validation_key_path = ".chef/awo-validator.pem"
  chef.validation_client_name = "awo-validator"
  chef.node_name = node_name.to_s
  chef.add_recipe the_recipe if the_recipe
end
end
end

Vagrant.configure("2") do |config|
  config.vm.box = "opscode-ubuntu-14.04"
  config.vm.box_url = "https://opscode-vm-bento.s3.amazonaws.com/
vagrant/virtualbox/opscode_ubuntu-14.04_chef-provisionerless.box"
  config.omnibus.chef_version = :latest
  config.berkshelf.enabled = true

  config.vm.provider :virtualbox do |vb|
    vb.customize ['modifyvm', :id, '--natdnshostresolver1', 'on']
  end

  define_node(config, :hal, "192.168.0.101", 'my_cookbook')
  define_node(config, :ha2, "192.168.0.102", 'my_cookbook')
end
```

Start both machines by using `vagrant up`.



You need to have the Vagrant Berkshelf plugin installed:

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

Make sure that you have a cookbook called `my_cookbook` and `run_list` of all the nodes you want to add to your HA cluster, including `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 use the community-provided heartbeat cookbook and configure it to work with our nodes:

1. Edit your cookbook's `metadata.rb` to add the dependency to the heartbeat cookbook:

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

depends 'apt'
depends "heartbeat"
Install your cookbook's dependencies:
mma@laptop:~/chef-repo $ berks install
Using my_cookbook (0.1.0) at './cookbooks/my_cookbook'
...TRUNCATED OUTPUT...
```

2. Edit your cookbook's default recipe, replacing `192.168.0.100` with the IP address, which should be highly available (your failover IP address) and `eth1` with the network interface that you want to use:

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

include_recipe "apt"
include_recipe "heartbeat"

heartbeat "heartbeat" do
  authkeys "MySecretAuthPassword"
  autojoin "none"
  warntime 5
  deadtime 15
  initdead 60
  keepalive 2
  logfacility "syslog"
  interface "eth1"
  mode "bcast"
  udpport 694
  auto_fallback true

  resources "192.168.0.100"

  search "name:ha*"
end
```

3. 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...
```

4. Run the Chef client on both nodes:

```
user@server:~$ sudo chef-client  
...TRUNCATED OUTPUT...  
* service[heartbeat] action restart [2015-03-14T22:02:26+00:00]  
INFO: service[heartbeat] restarted  
  
- restart service service[heartbeat]
```

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

5. Validate that your first node holds the failover IP address:

```
user@ha1:~$ ifconfig -a  
eth1      Link encap:Ethernet  HWaddr 08:00:27:12:7b:fc  
...TRUNCATED OUTPUT...  
eth1:0    Link encap:Ethernet  HWaddr 08:00:27:12:7b:fc  
          inet addr:192.168.0.100  Bcast:192.168.0.255  
          Mask:255.255.255.0  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
...TRUNCATED OUTPUT...  
lo        Link encap:Local Loopback
```

6. Validate that your second node does not hold the failover IP address:

```
user@ha2:~$ ifconfig -a  
eth1      Link encap:Ethernet  HWaddr 08:00:27:62:11:58  
  
lo        Link encap:Local Loopback
```

7. Stop the heartbeat service on your first node and validate that the failover IP address moves to your second node:

```
user@ha1:~$ sudo service heartbeat stop  
user@ha2:~$ ifconfig -a
```

```
eth1      Link encap:Ethernet HWaddr 08:00:27:62:11:58
          ...TRUNCATED OUTPUT...

eth1:0    Link encap:Ethernet HWaddr 08:00:27:62:11:58
          inet addr:192.168.0.100 Bcast:192.168.0.255
          Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500
          Metric:1

lo       Link encap:Local Loopback
...TRUNCATED OUTPUT...
```

How it works...

The heartbeat cookbook installs the heartbeat service on all your nodes. In this example, we assume that your hostnames are ha1, ha2, and so on.

Then, we need to configure our HA cluster. In the preceding example, we do this within our recipe.

First, you need to define a password. The nodes will use this password to authenticate themselves to each other.

Setting `autojoin` to `none` will make it impossible that new nodes get added outside of your Chef client runs.

Next, we set the timeouts to tell heartbeat when to act, if something seems wrong. The timeouts are given in seconds.

In the preceding example, we ask heartbeat to use the broadcast method on the network interface `eth1`.

The `resources` is your failover IP address. This IP address will be highly available in your setup.

The `search` call contains the query to find all the nodes to be included in the heartbeat setup. In our example, we search for nodes with names starting with `ha`.

After uploading all cookbooks and running the Chef client, we can verify our setup by querying the heartbeat status on both the nodes.

By stopping the heartbeat service on the node that currently has the failover IP address assigned to it, the second node will take over automatically.

There's more...

You can configure the heartbeat by setting attributes in a role as well. In this case, it would make sense to set the search attribute to find all the nodes that have the role.

See also

- ▶ Find the heartbeat cookbook on GitHub at <https://github.com/opscode-cookbooks/heartbeat>
- ▶ Read more about how to configure heartbeat at http://www.linux-ha.org/doc/users-guide/_creating_an_initial_heartbeat_configuration.html
- ▶ Find the complete reference of the heartbeat configuration file at <http://linux-ha.org/wiki/Ha.cf>

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. If you configure it on an HA cluster by using heartbeat (see the *Building high-availability services using heartbeat* recipe in this chapter), you have a complete high-availability solution available.

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.

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 see how to set up a simple HAProxy balancing to all nodes that have the `web_server` role:

1. Edit your cookbook's `metadata.rb`:

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

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

node.default['haproxy']['httpchk'] = true
node.default['haproxy']['x_forwarded_for'] = true
node.default['haproxy']['app_server_role'] = "web_server"

include_recipe "haproxy::app_lb"
include_recipe "haproxy::default"
```

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...
[2015-03-15T21:08:42+00:00] INFO: service[haproxy] restarted
...TRUNCATED OUTPUT...
```

6. Validate that the HAProxy admin interface runs on your node on port 22002:

HAProxy version 1.4.24, released 2013/06/17																
Statistics Report for pid 3537																
> General process information																
pid = 3537 (process #1, nbproc = 1) uptime = 0d 0h00m41s system limits: memmax = unlimited; ulimit-n = 8206 maxsock = 8206; maxconn = 4096; maxpipes = 0 current conns = 1; current pipes = 0/0 Running tasks: 1/3																
 Note: UP with load-balancing disabled is reported as "NOLB"																
admin																
	Queue			Session rate			Sessions				Bytes		Denied		Errors	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req
Frontend				1..	1..	-	1	1	2 000	1..		0	0	0	0	0
Backend	0	0		0	0		0	0	2 000	0..		0	0	0	0	0
http																
	Queue			Session rate			Sessions				Bytes		Denied		Errors	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req
Frontend				0..	0..	-	0	0	2 000	0..		0	0	0	0	0
servers-http																

How it works...

First, we change some of the default values: setting `httpchk` to `true` makes sure that HAProxy takes backend servers out of the cluster, if they don't respond anymore.

The `x_forwarded_for` attribute tells HAProxy to set the X-Forwarded-For HTTP header. It will contain the client IP address. If you don't set that header, your web servers will only see the IP address of your HAProxy server in their access logs, instead of your client's IP addresses. This will make it very difficult to debug problems with your web applications.

The third attribute that we need to change is `app_server_role`. You can set whatever role your backend application servers have. The `haproxy` cookbook will include every node (using its `ipaddress` node attribute, as returned by Ohai) having this role within its cluster.

After overriding those attributes, we run the `app_lb` recipe from the `haproxy` cookbook. The `app_lb` recipe will install HAProxy from a package and run a search for all nodes having the configured role.

After uploading all cookbooks and running the Chef client, you'll find the HAProxy admin interface on your node at port 22002. 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
- ▶ Find HAProxy at <http://www.haproxy.org>
- ▶ Find the `haproxy` cookbook on GitHub at <https://github.com/hw-cookbooks/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 take a 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                                         Dload  Upload  Total  Spent
Left  Speed
100  2352  100  2352      0       0  23106      0  --::--  --::--
--::--  23287
```

2. Edit your custom bootstrap script:

```
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
--template-file bootstrap/my-chef-full.erb --sudo
```

```
192.168.0.100 [2015-03-21T15:30:44+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 as 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 whenever 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 like Vagrant to test your bootstrap script, you might need to use localhost as the node's IP address, and add `-p 2222` to your command line to tell knife to connect through the forwarded SSH port of your VM.

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  
--template-file bootstrap/my-chef-full.erb --sudo -r 'role[web_server]'
```

Here, we added the `web_server` role to the run 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
- ▶ Find the `chef-full` bootstrap script here: <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 only those network connections that hit the services you decide to allow are accepted.

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"
```

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"

execute "ensure iptables is activated" do
  command "/usr/sbin/rebuild-iptables"
  creates "/etc/iptables/general"
  action :run
end
```

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. 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...
```

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 FWR (1 references)  
target     prot opt source          destination  
ACCEPT    all   --  anywhere        anywhere  
ACCEPT    all   --  anywhere        anywhere  
state RELATED,ESTABLISHED  
ACCEPT    icmp  --  anywhere       anywhere  
ACCEPT    tcp   --  anywhere       anywhere           tcp  
dpt:http  
ACCEPT    tcp   --  anywhere       anywhere           tcp  
dpt:ssh  
REJECT    tcp   --  anywhere       anywhere           tcp  
flags:SYN,RST,ACK/SYN reject-with icmp-port-unreachable  
REJECT    udp   --  anywhere       anywhere           tcp  
reject-with icmp-port-unreachable
```

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 it 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 up 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.

The `iptables` cookbook will drop off those templates in `/etc/iptables.d` and configure `iptables` so that it loads all those files on startup. It installs the script `rebuild-iptables` to do this.

Finally, we make sure that `iptables` has been activated. We add this step because I saw that the `iptables` cookbook ran, but did not load all the rules. This is fatal because you deem your box 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 see the `ACCEPT` lines for ports `http` and `ssh`. That's a good sign. The last two lines shut down all other services.

See also

- ▶ Find the `iptables` cookbook on GitHub at <https://github.com/opscode-cookbooks/iptables>

Managing fail2ban to ban malicious IP addresses

Brute-force attacks against any of your password protected services, such as SSH, and break-in attempts against your web server happen frequently for every public-facing system.

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 take a look at how to set up a basic protection for by 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"
depends "fail2ban"
```

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. 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...
* 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
- ▶ Find the fail2ban manual at the following location: http://www.fail2ban.org/wiki/index.php/MANUAL_0_8
- ▶ Find the fail2ban cookbook on GitHub at <https://github.com/opscode-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/Using_SettingUpUser.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 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...

Let's use the `knife-ec2` plugin to instantiate and bootstrap an EC2 node with Ubuntu 14.04 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-ec2-0.10.0.gem (100%)
Successfully installed knife-ec2-0.10.0
1 gem installed
```

2. Create your EC2 instance:

```
mma@laptop:~/chef-repo $ knife ec2 server create -d 'chef-full'  
-r 'recipe[apt]' -S 'aws_knife_key' -x ubuntu -i ~/.ssh/aws_knife_  
key.pem -I 'ami-fc99c294' -f 'm1.small' -A 'Your AWS Access Key  
ID' -K 'Your AWS Secret Access Key'  
  
Instance ID: i-01e227fd  
Flavor: m1.small  
Image: ami-fc99c294  
Region: us-east-1  
Availability Zone: us-east-1c  
Security Groups: default  
Tags: Name: i-01e227fd  
SSH Key: aws_knife_key  
  
Waiting for EC2 to create the instance.....  
Public DNS Name: ec2-54-161-86-119.compute-1.amazonaws.com  
Public IP Address: 54.161.86.119  
Private DNS Name: ip-10-145-212-110.ec2.internal  
Private IP Address: 10.145.212.110  
  
Waiting for sshd access to become available.....done  
Connecting to ec2-54-161-86-119.compute-1.amazonaws.com  
ec2-54-161-86-119.compute-1.amazonaws.com Installing Chef  
Client...  
...TRUNCATED OUTPUT...  
ec2-54-161-86-119.compute-1.amazonaws.com Chef Client finished,  
3/4 resources updated in 26.807762127 seconds  
...TRUNCATED OUTPUT...
```



You need to look up the most current AMI ID for your node at <http://cloud-images.ubuntu.com/locator/ec2/> and use it in your knife call, instead of ami-fc99c294. See the following 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@  
ec2-54-161-86-119.compute-1.amazonaws.com  
  
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-46-generic x86_64)  
...TRUNCATED OUTPUT...  
ubuntu@ip-10-145-212-110:~$
```

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 14.04 LTS code named `trusty`, using `instance-store`. At the time of writing this, the most current version was `ami-fc99c294`.

As soon as you know what you want to achieve, it's time to construct the launch command. It consists of the following parts:

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:

- ▶ `-d '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-fc99c294'`: This names the AMI ID. You need to take the latest one, as described in the preceding paragraph.
- ▶ `-f 'm1.small'`: This is the instance type, as described in the preceding point
- ▶ `-A 'Your AWS Access Key ID'`: This is the ID of your IAM user's AWS Access Key
- ▶ `-K '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 is allowed to 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 hard coding 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 instance data (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
- ▶ Find the `knife-ec2` plugin on GitHub at <https://github.com/chef/knife-ec2>

Loading your Chef infrastructure from a file with spiceweasel and knife

Having all your cookbooks, roles, and data bags as code under version control is great, but having your repository alone is not enough to be able to spin up your complete environment from scratch again. Starting from the repository alone, you will need to spin up nodes, upload cookbooks to your Chef server, and recreate data bags there.

Especially when you use a Cloud provider to spin up your nodes, it would be great if you could spin up your nodes automatically and hook them up to your freshly created and filled Chef server.

The spiceweasel tool lets you define all your cookbooks, data bags, and nodes, and generates all the necessary knife commands to recreate your complete environment, including spinning up nodes and populating your empty Chef server or organization on the hosted Chef.

Let's see how to dump our current repository to spiceweasel and how to recreate our infrastructure with it.

Getting ready

Make sure that you are able to spin up Amazon EC2 instances using knife, as described in the *Managing Amazon EC2 instances* section in this chapter.

How to do it...

Let's use spiceweasel to dump our current configuration, add some EC2 nodes and recreate our complete environment:

1. Install the spiceweasel Ruby gem:

```
mma@laptop:~/chef-repo $ chef gem install spiceweasel
Fetching: spiceweasel-2.8.0.gem (100%)
Successfully installed spiceweasel-2.8.0
1 gem installed
```

2. Let spiceweasel dump your current repository into a `infrastructure.yml` file:

```
mma@laptop:~/chef-repo $ spiceweasel --extractyaml >
infrastructure.yml
```

3. Look into your new `infrastructure.yml` file (the contents depend on the current state of your Chef repository):

```
mma@laptop:~/chef-repo $ cat infrastructure.yml
---
berksfile:
cookbooks:
- apt:
    version: 2.6.1
- heartbeat:
    version: 1.0.0
roles:
- monitoring:
- web_server:
...TRUNCATED OUTPUT...
```

4. Print all the knife commands and spiceweasel will run:

```
mma@laptop:~/chef-repo $ spiceweasel infrastructure.yml
berks upload -b ./Berksfile
knife cookbook upload apt heartbeat
knife role from file monitoring.rb web_server.json
knife data bag create accounts
knife data bag from file accounts google.json
knife data bag create hooks
knife data bag from file hooks request_bin.json
knife data bag create nagios_services
```

```
knife data bag from file nagios_services ssh.json
knife data bag create users
knife data bag from file users mma.json
```

5. Let spiceweasel run the knife commands, as follows:

```
mma@laptop:~/chef-repo $ spiceweasel -e infrastructure.yml
...TRUNCATED OUTPUT...
Data bag users already exists
Updated data_bag_item[users::mma]
```

How it works...

The spiceweasel tool scans your local Chef repository and notes down everything as a `YAML` file.

When reading a `YAML` file, it generates knife commands to make the contents of the Chef repository available on the Chef server.

There's more...

You can define nodes in your `infrastructure.yml` file either as local nodes, which spiceweasel will then bootstrap, or nodes for cloud providers. spiceweasel will then create `knife <provider> server create` commands for each specified node.

Using nodes in your `infrastructure.yml` file enables you to recreate a complete environment, including all the necessary VMs using spiceweasel.

You can use spiceweasel to delete your setup from your Chef server by using the `--delete` flag when running spiceweasel:

```
mma@laptop:~/chef-repo $ spiceweasel --delete infrastructure.yml
```

See also

- ▶ You can find the source code of spiceweasel on GitHub at <https://github.com/mattray/spiceweasel>

Module 3

Mastering Chef Provisioning

Learn Chef Provisioning like a boss and finally own your infrastructure

1

Setting Up a Development Environment on Your Workstation

In today's fast-paced IT world, it is a requirement to deliver infrastructure at warp speed. There's really only one way to achieve this requirement, and that is through automation.

One of the best paths to infrastructure automation is via Chef. Using Chef, you can turn your infrastructure into code. With infrastructure as code, you can automate the way you build, deploy, and manage all of it. Using Chef, your infrastructure becomes very consistent, very duplicable, and version-controlled. Using Chef, you can easily test your infrastructure setup and configuration. With Chef, you can become an IT superhero!

In this chapter, we're going to detail the setup and configuration of a complete development system or workstation. The main focus will be centered on deploying the **Chef Development Kit (ChefDK)** and preparing the various components of a development environment. The chapter also includes suggestions for additional tools to round out the DevOps toolbox. However, before we dive into ChefDK, we will fly through a high-level review of the what-and-how of Chef so that we are all on the same page.

"If you wish to make an apple pie truly from scratch, you must first invent the universe." – Carl Sagan

Here is what you will find in this chapter:

- Filling in the gaps in your Chef knowledge base
- Exploring ChefDK
- Installing ChefDK
- Setting up a standard Chef repo
- What else do you need?
- Using chef-apply

Re-introducing Chef

If you are reading this book, then you've probably been working with Chef for some time now, and you know the many benefits it brings. You may also know that, in the past, it was a somewhat daunting task to set up a new Chef workstation. You had to download and install Chef, then download and install a variety of community tools, and make sure that all the versions were compatible and configured correctly.

Today, a lot of the work is done for you via the ChefDK. Once you install it, you have a basic workstation setup and are ready to create, modify, and test Chef code.

What version of the Chef tools do I use?

To make it easy to follow along with the contents, let's go over the versions of the tools that will be used throughout this book.

In March 2016, [chef.io](#) announced the release of ChefDK 0.12.0, which includes Chef client 12.8.1. As this is the latest version at the time of writing, I will be using it as the version for this book. All of the examples shown will be based on this version of the ChefDK and Chef client. This is very exciting because a lot of really exciting changes are in this release, including Policies and the transition from Resource Providers to Custom Resources.

Currently, there are ChefDK installers available for Mac OS, Windows OS, and Linux OS (RHEL, Debian, and Ubuntu). In the examples within this book, the workstation used will be Mac OS X, so the ChefDK version will be the Mac OS version.

With the many choices available to use for your Chef Server mode, including Hosted Chef Server, Private Chef Server, Open Source Chef Server, and Chef-Solo, it would be difficult to show examples for each mode. Therefore, the majority of examples you'll see in this book will be based on using the Hosted Chef Server mode. Later in this chapter, I will briefly review the installation and setup of a Private Chef Server onto an Ubuntu server, for readers who want to use Chef Server on-premise.

Which OS do I use for my workstation? Everyone has their own, nearly religious, choice for the best workstation platform. However, it would make this book way too long to provide examples for the major OSes alone. Therefore, to keep the focus on Chef content and not on the differences between workstation implementations, I will be using a Mac OS X (Yosemite version 10.10) workstation for the examples. I may at times show additional examples on a Windows or Ubuntu workstation where the differences are significant and worth the extra detail. And for the nodes used in the examples, a variety of OSes will be represented in the hope of having some overlap with the reader's real environment.

References

- IT infrastructure automation begins at <https://www.chef.io/>
- Additional opportunities to learn Chef can be found at <https://learn.chef.io/>

Filling in the gaps in your Chef knowledge base

First things first. Let's have a quick, high-level review of Chef. A Chef 101 class, if you will. The plan here is to make sure that all readers are on the same page with me so that the rest of the book will be easier to follow and benefit from. If you are already a Chef ninja, you can probably skip right on past this section and start exploring the ChefDK in the next section. Otherwise, let's audit our 101 class now.

Major pieces of a Chef puzzle

Chef is a ruby framework, and its main purpose is to facilitate the automation, reuse, and documentation of server configurations. Chef allows you to treat server configuration just like any other type of software code.

There are three major components of this framework for almost all corporate-level Chef deployments—the workstation, the Chef Server, and the nodes. Let's talk a little about each of these components:

- **Workstations:** The Chef workstation is, as the name would suggest, where the real "work" is done. This is where developers will create their infrastructure code. It's also where they will test their creations. The workstation is where developers will interact with their source code control systems. It is where genius is born. It is where the "desired state" of nodes is defined via code.
- **Chef servers:** The Chef server is a database-backed web API server with a browsable user interface. It is like the matrix housing the hive mind of the Chef universe. The "work" created in the workstation is uploaded into the Chef server so that it can be used to automate the configuration of the many nodes. It stands ready to provide the desired state information to the Chef client on a node so that it can bring that node into alignment or convergence with the desired state of configuration.
- **Nodes:** Nodes are the ultimate targets of the "work" that's been created on the workstation and uploaded to the Chef server. A node is where the automation occurs via the Chef client. Nodes are transformed into the desired state configuration. Commonly, nodes are servers, either virtual or physical, but a node can be anything that needs to be configured, such as a Docker container, or a network device such as a switch or router. Every node has a unique name, such as the FQDN of a server.

The Chef client

The Chef client is the tool that is deployed to all nodes and used to configure or reconfigure the node to its desired state. The Chef client is what executes the automation. It is the magician that, through the magic of Chef, transforms an ordinary server into the desired state node that the developer defined back on his workstation.

The Ohai Chef tool

Ohai is the tool that gathers information about a node. Information such as platform details, operating system data, and processor information is made available to the Chef client so that the latter can have the know-how to bring the node to the desired state. Ohai is executed at the beginning of a Chef client run to gather the state of the node. At the end of a Chef client run, all of the data gathered by Ohai (usually) is shared as node data with the Chef server, and the shared data is available for searches done against the Chef server. Ohai can be extended via plugins, and we'll take a look at that in a later chapter.

Recipes and cookbooks

Recipes are the building blocks used to define the desired states. Recipes are files of ruby code that define the commands to be run on nodes. They are like blueprints used to "build" a node. Recipes are collections of Chef resources. We will explain more about Chef resources shortly.

Cookbooks are collections of related recipes, templates, files, and custom resources. They provide organization and versioning for recipes. Each unique version of a cookbook represents unique sets of functionality, such as bug fixes or added features.

Cookbooks define a scenario, such as everything needed to install and configure apt-docker or Sublime text, and they contain all the elements needed to support the defined scenario.

Recipes and cookbooks provide modularity and let you easily reuse code.

Chef run lists

A "run list" is, as the name would suggest, a list of, and the sequence for, the recipes, cookbooks, and policies (spoiler alert!) to be applied to a node. A run list contains all of the information required to configure a node to a desired state. That is, a Chef run list describes the desired final state of the node. It is important to note that, if the node's state already matches what the resources in the run list describe, then no action will be taken to change the node's state.

Chef roles

Roles are functional groupings of recipes and cookbooks used to describe the full blueprint needed for a node to become everything it is intended to be. Roles are reusable configurations, and they can be applied to multiple nodes to make functionally identical servers, such as a farm of web servers.

Chef resources

Chef resources are statements of configuration policy. They are defined in recipes and take actions through the Chef client to put the node into the desired state. Chef resources have some types: Package, Template, Service, and so on. They have a name and parameters. Also, Chef resources can send notifications to other resources.

Chef resources define what we want to happen to a node. They don't say how to do it. The how to do it is left to the providers which are platform-specific. That is to say, the way you install a package will be different depending on the OS, and the provider determines the correct way to do it – the "how". The Chef resource simply defines the "what," such as "install ntp".

- **Package:** This contains software or applications, such as apache, ntp, and cron, and the action to be performed on that software or application, such as "install".
- **Template:** These are files with place-holders for attributes that are transformed into configuration files for package installation and execution.
- **Service:** This is the installed executable of the package and the actions that the executable can perform, such as start, stop, or restart. Service also defines whether the software or application is launched at node startup.

More things to consider

Apart from the three types of Chef resources we saw earlier, let's look at some other important aspects of recipes and cookbooks.

- **Attributes:** These are variables used in recipes and templates. Generally speaking, a recipe will represent the pattern used in configuration. The attributes provide the specific details for the recipes. For example, the recipe will say "configure the port" and the attribute will say "which port?". Attributes can be provided in a large variety of places, such as cookbooks, roles, and environments. As such, there is a necessity for an order of precedence. There is a complete description of attribute precedence on the chef.io site. You can find it at <https://docs.chef.io/attributes.html>.

- **Order matters:** When creating recipes, the resources need to be listed in order. First, the package, then the template, and finally the service; when creating run lists, the order of policies, roles, cookbooks, and recipes is the order in which they are evaluated. Any recipe that appears in a run list (or as a dependency of a policy, role, or cookbook) more than once will only be evaluated the first time.
- **Convergence and idempotence:** A Chef client run converges the node into the desired state. What this means is that only things that do not equal the desired state on the node are modified during the Chef client run. If, for example, the recipe says that the desired state of the node is to have the ntp demon running, the ntp package is already installed on the node, and the ntp demon is already running, then Chef client will take no action to install or start ntp.

Chef resources are idempotent. That is, applying them more than once results in the same outcome every time. If no inputs related to the resource have changed, then applying that resource won't change anything after the first application. In fact, if none of the inputs have changed, the corresponding commands don't even get run.

This is actually one of the most important concepts and features of Chef.

Data bags

Data bags are containers for information that is not tied to a specific node. Data bags are the global variables of a Chef server. They can be used in recipes and can be searched like node data via the Chef server. One common use case for data bags is for user and group information. Items in a data bag can be encrypted. This allows secret information to be stored in them, for example, passwords.

Environments

Environments allow you to define specific cookbook versions that are applied to a given set of nodes. They permit you to model the stages of your infrastructure workflow, that is, Development, Test, Stage, and Production. By identifying specific nodes as Development, and other nodes as Production, for example, you can apply different versions of your cookbooks to the nodes based on their environment membership.

The Chef supermarket

The supermarket is a site that provides shared cookbooks. There is a public supermarket that contains community-created and -maintained cookbooks. This site is hosted by Chef and is available at <https://supermarket.chef.io>. In addition to the public supermarket, anyone can create and manage a private supermarket that can host cookbooks intended for the private consumption of you and your organization.



It is always best practice to do a thorough code review of any cookbooks obtained from the public community supermarket before using them in your production environments.



Chef Development Kit

The ChefDK contains everything you need to start working with Chef on a workstation. It provides all the tools that a developer needs to create and modify cookbooks and upload them to a Chef server. We are going to go into a lot of detail regarding the ChefDK later in this chapter.

You can jump ahead to the ChefDK sections now, or you can read on and learn how to set up your very own private Chef server.

Setting up an on-premise (private) Chef Server

Many companies will have strong reasons to keep their infrastructure configuration data within the firewalls of their datacenters, and Chef server has a mode for that. It's called on-premise or private Chef server.

There are some real benefits to using an in-house solution, including control. With a private installation, you have full control of your system, and for some that reason is enough. But there are additional considerations such as performance. Since a private Chef server will likely be physically closer to the node's network, Chef client runs will be faster. For example, if Chef is used in a load-based on-demand server deployment solution, then the extra speed the private Chef server can provide when converging your new Nodes can make all the difference in meeting the load demand in time.

With all the reasons to use an on-premise Chef server, there is a downside: you are responsible for the deployment, configuration, and maintenance of your Chef servers.

To get you started, let's take a look at the deployment and initial configuration of a standalone private Chef server now. We'll go through the steps to get a new Chef server set up on an Ubuntu 14.04 server.

The first step is downloading the required Chef server installer version. Visit the Chef server page at this link <https://downloads.chef.io/chef-server/> to find and download the version you will use for your installation.

The screenshot shows the Chef Server download page. At the top, it says "Chef Server" and "The Chef 12 server features new high availability, replication, and analytics capabilities." Below this, there are two sections: "Red Hat Enterprise Linux" and "Ubuntu Linux". Under "Ubuntu Linux", there are three entries:

- Ubuntu Linux 14.04**: Works on 64 bit (x86_64) versions of Ubuntu 14.04. Includes SHA1, URL, and a "Download" button.
- Ubuntu Linux 12.04**: Works on 64 bit (x86_64) versions of Ubuntu 12.04. Includes SHA1, URL, and a "Download" button.
- Ubuntu Linux 10.04**: Works on 64 bit (x86_64) versions of Ubuntu 10.04. Includes SHA1, URL, and a "Download" button.

Next, you will want to transfer the downloaded installer package to your soon to be Chef server.

Once the installer is available on your server, you will want to install the package. The command to do this is:

```
sudo dpkg -i chef-server-core_12.2.0-1_amd64.deb
```

Once the Chef server package has been installed onto your server, the first thing to do is start the Chef server services. The command to do this is:

```
sudo chef-server-ctl reconfigure
```

The Chef server startup does a lot of work and as such you will see a lot of activity displayed as the setup progresses. When the startup command finally finishes successfully (a few minutes later; you might want to get a cup of coffee), the result will be a running Chef server.



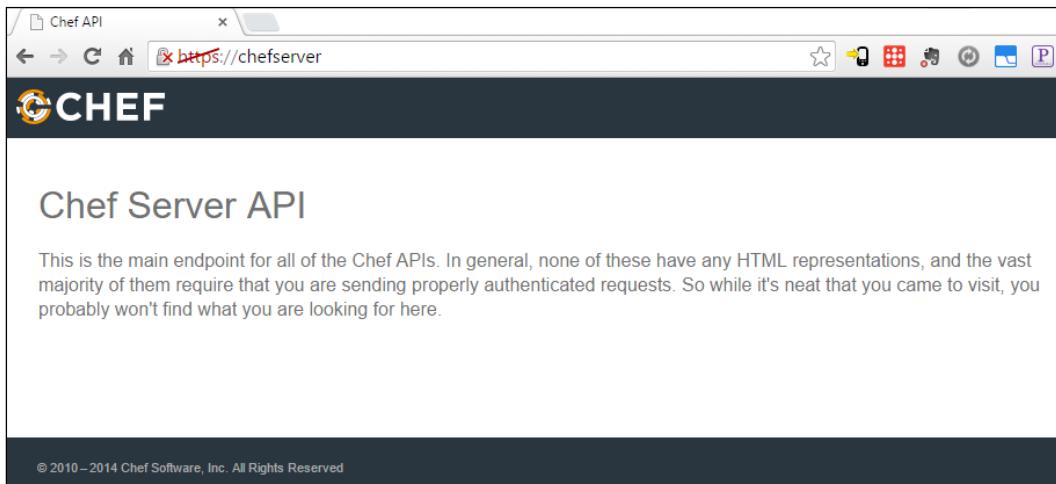
Do you get an error in the nginx.rb file during the reconfigure?

The private Chef server has a dependency on the nginx cookbook, and in some cases you may experience an error when this dependency is resolved. The error may look something like this:

"common_name nil currently does not overwrite the value of common_name."

What this error message is probably indicating is that there is no valid host entry in the /etc/hosts file. You should check that file, and if there isn't a valid entry, create one.

Are we there yet? Well, we do have a Chef server up-and-running, but it is not much good to us yet. If you browse to your new Chef server, you will see something like this:



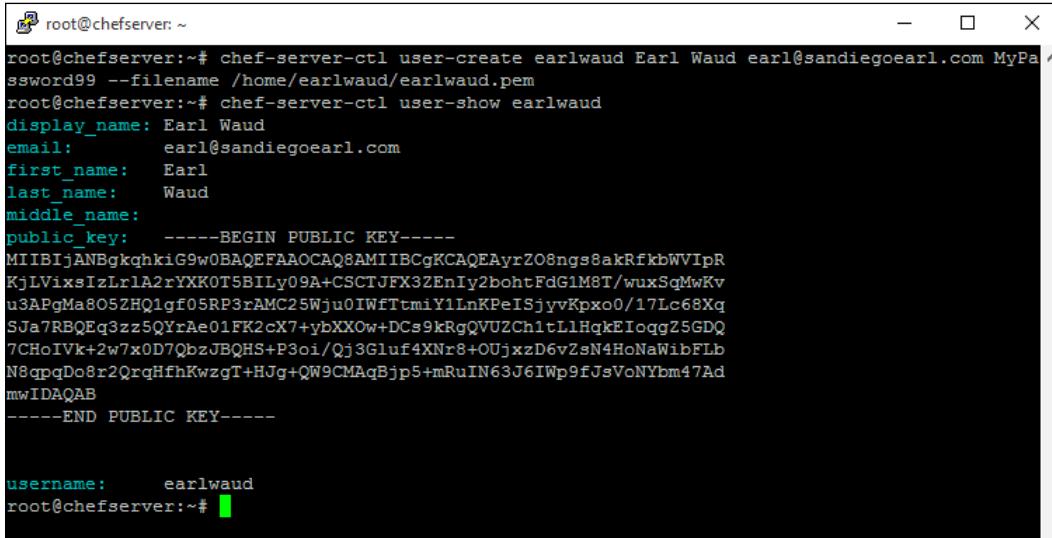
That is a good start, but there is still work to be done. Next up is creating your first Chef user on the Chef server with the `user-create` command. That user will be the initial Chef admin for the server. The format of the `user-create` command is:

```
sudo chef-server-ctl user-create user_name first_name last_name email  
password --filename FILE_NAME
```

Here is an example of what that command might look like:

```
sudo chef-server-ctl user-create earlwaud Earl Waud earl@sandiegoearl.com  
MyPassword99 --filename /home/earlwaud/earlwaud.pem
```

Here is what it looks like when you issue the command on your Chef server (along with the `user-show` command):



```

root@chefserver:~# chef-server-ctl user-create earlwaud Earl Waud earl@sandiegoearl.com MyPa
ssword99 --filename /home/earlwaud/earlwaud.pem
root@chefserver:~# chef-server-ctl user-show earlwaud
display_name: Earl Waud
email: earl@sandiegoearl.com
first_name: Earl
last_name: Waud
middle_name:
public_key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBgKCAQEAYrZ08ngs8akRfkWVIpR
KjLVixsIzLrlA2rYXKOTSBILy09A+CSCTJFX3ZEnIy2bohtFdG1M8T/wuxSqMwKv
u3APgMa805ZHQ1gf05RP3rAMC25WjuIWfItmiY1LnKPeISjyvKpxo0/17Lc68Xq
SJa7RBQEg3zz5QYrAe01FK2cX7+ybXXOw+DCs9kRgQVUZCh1tLlHqkEIoqgZ5GDQ
7CHoIVk+2w7x0D7QbzJBQHS+P3oi/Qj3Gluf4XNr8+OUjxzD6vzsN4HoNaWibFLb
N8qpqDo8r2QrqHfhKwzgT+HJg+QW9CMaQBjp5+mRuIN63J6IWp9fJsVoNYbm47Ad
mwIDAQAB
-----END PUBLIC KEY-----

username: earlwaud
root@chefserver:~#

```

The filename parameter used in the `user-create` command provides the location at which to store the user's private key. The contents of this key file should be securely stored so that they can be provided to the user whose account has been created.

The next step is to create the first organization in your Chef server. The format of that command is as follows:

```
sudo chef-server-ctl org-create short_name "full_organization_name"
--association_user user_name --filename ORGANIZATION-validator.pem
```

Here is the command I used to create the organization on my example Chef server:

```
sudo chef-server-ctl org-create sdearl "San Diego Earl" --association_
user earlwaud --filename /home/earlwaud/sdearl-validator.pem
```

You can see that, in the example, the key file was saved at `/home/earlwaud/sdearl-validator.pem`.

If the filename parameter is not provided in the `org-create` command, the validator key will be displayed to `stdout`.

Either way, it is vital that the key information be captured and securely stored since it is not saved in the Chef server database and cannot be recovered if lost. The only option available when an organization's private key is completely lost is to reset the validation key.

Setting Up a Development Environment on Your Workstation

Next, we need to install a package that will allow management of the Chef server via a web interface. The package is named `opscode-manage`. The command to install the package is:

```
sudo chef-server-ctl install opscode-manage
```

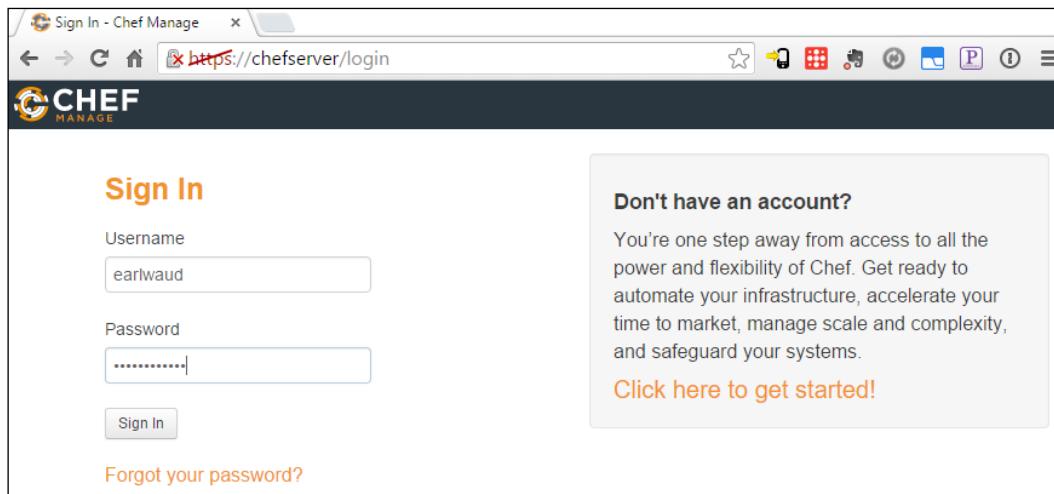
Now we need to restart the Chef server services by issuing the `reconfigure` command again:

```
sudo chef-server-ctl reconfigure
```

And finally, we need to start the `opscode-manage` services. This is done by issuing the command:

```
sudo opscode-manage-ctl reconfigure
```

If everything went as planned, you should now be able to browse to the new Chef server, and log in with your newly created admin user. Of course, there is still more work to do, such as integrating your new Chef server with your corporate domain to allow domain user access control. Still, this is a good start, right?



References

- Explore the Chef community Supermarket at this link:
<https://supermarket.chef.io>
- Download Chef server from <https://downloads.chef.io/chef-server/>

- Learn more about installing Chef server at https://docs.chef.io/release/server_12-2/install_server.html
- Learn more about attributes and attribute precedence at <https://docs.chef.io/attributes.html>
- Read more about the "common name" error at <https://github.com/chef-cookbooks/chef-server/issues/108>

Exploring the Chef Development Kit

The Chef Development Kit, or ChefDK, is the omnibus installer used to set up a Chef development environment on a workstation. It includes the Chef client itself, an early version of the new Chef tool, an embedded version of Ruby, RubyGems, OpenSSL, and command-line utilities such as Knife, Ohai, and Chef Zero. It also includes community tools, such as Test Kitchen, Foodcritic, Berkshelf, Chef Vault, Rubocop, and ChefSpec.

Included in the ChefDK installer is the new Chef command-line tool. It is the new go-to way to generate cookbooks, recipes, templates, custom resources, and more. It also downloads RubyGems for the chef-client environment, and it will verify the ChefDK installation and configuration.

The Chef client is a key component of everything Chef. The Chef client is not only used on nodes for converging configuration data, but it is also used on workstations and the Chef server.

The Chef Knife

Knife is a command-line tool that supplies the interface between your chef-repo and your Chef server. It allows developers to upload their work on the workstation to the Chef server, from where it can be distributed to the nodes during their Chef client runs. Knife also allows you to obtain cookbooks from the supermarket, and it provides a mechanism to bootstrap nodes.

Community tools

Some of the most commonly used community tools are installed as part of the ChefDK. Test Kitchen is installed, so you can test the results of your recipes and cookbook development on deployed test nodes. Berkshelf is installed to manage your cookbooks and their dependencies. Foodcritic is included in the ChefDK installation. Foodcritic will help you check your cookbooks for common errors and omissions.

Another valuable community tool installed with the ChefDK is Rubocop. You can use it to give you cookbooks some "style" and make sure that you're following the same conventions used by other developers in your organization. There is also ChefSpec, which is based on the ruby tool RSpec. ChefSpec will allow you to unit-test your cookbooks and recipes. We will have to exercise a lot of these community tools in later chapters, so please stay tuned.

References

- Learn more about the Chef Development Kit at https://docs.chef.io/chef_dk.html
- Learn more about Berkshelf in this video-recorded interview at <https://www.chef.io/blog/chefconf-talks/the-berkshelf-way-jamie-winsor/>
- Learn more about Foodcritic at this link <http://www.foodcritic.io/>
- Learn about the Knife command line tool at this link: <https://docs.chef.io/knife.html>
- And download an excellent printable Knife reference guide at this link https://github.com/chef/quick-reference/blob/master/qr_knife_web.png

Installing Chef Development Kit

In this section, you will learn how easy it is to get your workstation up and running using the Chef Development Kit. First, you will download ChefDK. Next, you will install ChefDK; and finally, you will validate the installation with some quick version checks.

Downloading ChefDK

You are going to download Chef Development Kit by opening your favorite browser and visiting the Chef.io ChefDK downloads page found at this URL: <https://downloads.chef.io/chef-dk/>.

1. Select the platform you are going to install the Chef Development Kit on and then click on the download button that corresponds to the OS Version you are using. In my case, I am selecting the Mac OS X 10.10 version of the ChefDK installer.
2. As expected, clicking on the download button will transfer the ChefDK installer to your local workstation.

3. Expand the Downloads folder and click on the ChefDK disk image file to mount the image on your desktop.

Installing ChefDK

To initiate the installer for the Chef Development Kit, double-click on the mounted ChefDK image icon.

1. The ChefDK image will open, exposing the chefdk package. Double-click on the package icon to begin the installation process.
2. You will see the Chef Development Kit installer introduction screen. Click on the **Continue** button to advance the installation.
3. You will be presented with license information for the Chef Development Kit, which is the commonly used Apache License. I would suggest that you read through the entire license document and perhaps even consult with your lawyer to make sure that you are able to accept this license agreement. Then, click on the **Continue** button to advance the installation.
4. Once you've carefully read and understood the license agreement, click on the **Agree** button to advance the installation.
5. At this point, you can begin to customize the installation. The options available here are to install for all users or only the currently logged in user. In the example, I am installing for all users of this computer (even though I am the only user).
6. Next, you can choose the install location. For consistency with regard to the examples in this book, I have left the install location at the default. Click on the **Install** button when you are ready to begin the actual installation.
7. As always, you will have to provide credentials to allow the system to install new software. Enter your username and password and click on the **Install Software** button to continue.

The installer will now go about its business of installing the super fantastic Chef Development Kit, including the corresponding versions of community tools such as Test Kitchen, Food Critic, and Rubocup.

Finally, you will get the installation summary screen congratulating you on your successful installation and keen DevOps Kung-Fu.

The captain has turned off the seat-belt sign, so it is now OK to unmount the ChefDK image, and move about the installation.

Installing ChefDK on Ubuntu

 It is even easier to install the ChefDK on an Ubuntu system. Once you have downloaded the installer, you simply issue a `dpkg` command as follows:

```
sudo dpkg -i ~/Downloads/chefdk_0.12.0-1_amd64.deb
```

That's it. You should see the "Thank you for installing Chef Development Kit" message.

Verifying ChefDK

The ChefDK installer is a superhero for DevOps aficionados, saving them time and energy by installing the major necessities for a Chef workstation. Still, let's double-check the work done by the installer to make sure that we are really ready to go.

First, let's just try a simple command to see if things look right. Open a terminal window and issue the command `chef --help`. If the install was successful, you should see information about using Chef.

Next, let's use the new Chef tool to do a deeper validation of the install. Issue the following command in your terminal session:

```
chef verify
```

You should see the Chef tool validate each of the major packages installed during the ChefDK installation. Note that, depending on the state of your (OS X) workstation at the time you run this command, you may receive an error message regarding the need to install the command-line tools for gcc. It may look something like this:

The screenshot shows a terminal window titled "earlwaud — ruby —". The terminal output indicates an error executing a "run" action on a "execute[initialize-git]" resource. The error message is:

```
Error executing action `run` on resource `execute[initialize-git]`
=====
Mixlib::ShellOut::ShellCommandFailed
-----
Expected process to exit with [0], but received '1'
---- Begin output of git init . ----
STDOUT:
STDERR: xcode-select: note: no developer tools were found at '/Applications/Xcode.app'. Please select developer tools from the App Store.
```

Below the terminal, a modal dialog box from ChefDK is displayed. It features a circular icon with arrows and the text:

The "gcc" command requires the command line developer tools. Would you like to install the tools now?

It includes three buttons: "Get Xcode", "Not Now", and a highlighted "Install" button. The "Not Now" button is greyed out.

At the bottom of the terminal window, there is more command-line output related to the ChefDK skeleton.

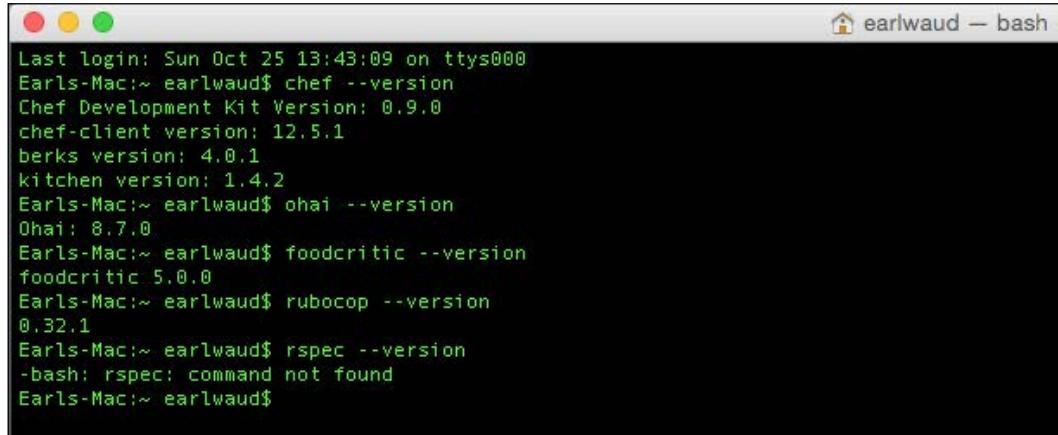
If you get a message such as the preceding one, go ahead and install the command-line tools and then run the `chef verify` command again. This time, you should get a clean run of the command, and the output should be very similar to that shown in the preceding screenshot.

Now, at this point, things are looking pretty good, and you should be feeling fine. So, let's do one last validation check just to be 100% certain. Let's check the versions of some of the expected tools installed via the ChefDK installer.

Issue the following commands:

```
chef --version
ohai --version
foodcritic --version
rubocop --version
rspec --version
```

Your results should look something like this:



```
Last login: Sun Oct 25 13:43:09 on ttys000
Earls-Mac:~ earlwaud$ chef --version
Chef Development Kit Version: 0.9.0
chef-client version: 12.5.1
berks version: 4.0.1
kitchen version: 1.4.2
Earls-Mac:~ earlwaud$ ohai --version
Ohai: 8.7.0
Earls-Mac:~ earlwaud$ foodcritic --version
foodcritic 5.0.0
Earls-Mac:~ earlwaud$ rubocop --version
0.32.1
Earls-Mac:~ earlwaud$ rspec --version
-bash: rspec: command not found
Earls-Mac:~ earlwaud$
```

Hey wait... what happened to rspec? The Version 0.12 ChefDK installer does not seem to have successfully installed rspec, which is a requirement for the Chef-specific tool ChefSpec. That's not good. What went wrong? Everything seemed to have been installed based on the verify command we just used. What gives?

Well, rspec is installed, there is just one more step that we need to do. By the way, I think this is something that should be done by the ChefDK installer, but for whatever reason it's not, and it falls on our shoulders to handle it.

So what do we have to do? We need to set up the environment used in our terminal window for our Chef workstation development. How do we do that?

Fortunately, Chef has a command for that. It is the `chef shell-init` command. You can run this command interactively every time you open a new terminal window, or you can update your profile so that it is run automatically. I would recommend updating your profile, but let me show you both ways so you can make the call.

To make the interactive change, you would issue a command in the following format:

```
eval "$(chef shell-init SHELL_NAME)"
```

The actual call will be the following for the bash shell:

```
eval "$(chef shell-init bash)"
```

If you want to make the change more permanent, then you need to issue a command in the following format:

```
echo 'eval "$(chef shell-init SHELL_NAME)"' >> ~/.YOUR_SHELL_RC_FILE
```

If you are using Mac, the commands will be:

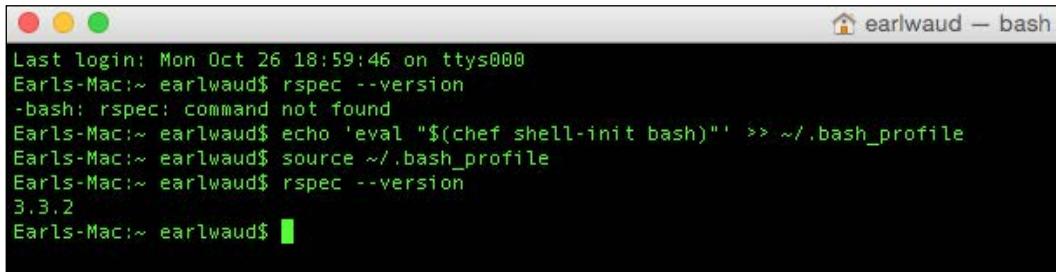
```
echo 'eval "$(chef shell-init bash)"' >> ~/.bash_profile
source ~/.bash_profile
```

If you are using Ubuntu, the commands will be:

```
echo 'eval "$(chef shell-init bash)"' >> ~/.bashrc
source ~/.bashrc
```

This echo command will add the eval command to your bash startup so that each run of your bash session will already have the desired environment ready to go for your Chef development work.

It should look something like this:



```
Last login: Mon Oct 26 18:59:46 on ttys000
Earls-Mac:~ earlwaud$ rspec --version
-bash: rspec: command not found
Earls-Mac:~ earlwaud$ echo 'eval "$(chef shell-init bash)"' >> ~/.bash_profile
Earls-Mac:~ earlwaud$ source ~/.bash_profile
Earls-Mac:~ earlwaud$ rspec --version
3.3.2
Earls-Mac:~ earlwaud$
```

By the way, when you install the ChefDK onto a Windows system, the installation creates a Chef Development Kit icon on your desktop. When you execute from icon, a Chef-specific PowerShell session is launched. It will automatically run the chef shell-init command and set up the full environment needed for Chef. So there is no need to do this last step on a Windows workstation.



Chef Development Kit on a Windows workstation

When you launch Chef Development Kit from your Windows workstation, remember to right-click the icon and choose "Run as administrator." This allows PowerShell to use the correct permissions to set up the Chef environment.

The screenshot shows a Windows PowerShell window titled "Administrator: ChefDK (EarlWaud)". The window displays the following text:
PowerShell 4.0 (Microsoft Windows NT 6.3.9600.0)
Ohai, welcome to ChefDK!
PS C:\Windows\system32> rspec --version
3.3.2
PS C:\Windows\system32>

Well, there you have it: the successful installation of the ChefDK. Your workstation is ready to do some real Chef work now. You should feel pretty awesome!

References

- Visit this link to download the installer for the ChefDK or Chef server at <https://www.chef.io/chef/get-chef/>.
- Here is Chef.io's information on installing the Chef Development Kit: https://docs.chef.io/install_dk.html. The Chef DK change log can be found at <https://github.com/chef/chef-dk/blob/master/CHANGELOG.md>.

Setting up a standard Chef repo

Ever Chef workstation has at least one Chef repository. The Chef repo is where all working copies of the cookbooks, recipes, templates, and so on live. It is where you do your development work. It is from where you check your work into your source code repository, and it is from where you upload your work to the Chef server.

Let's take a look at how to set up a new empty Chef repo on your new ChefDK-installed workstation.

Using the Chef tool for Chef repo

Another benefit of the new Chef command-line tool is that it will create a standard chef repo for you. You no longer need to download or clone a starter repo from GitHub. You can use the `chef generate` command to create your new repo. Let's take a look at the `chef generate repo` command's help:

```
chef generate repo -help
```

Based on the description of the default values for the `chef generate repo` command, we can simply issue the following command on our workstation:

```
chef generate repo ~/chef-repo
```

That was easy! So what do we get? Well, let's have a look:

```
tree -a chef-repo
chef-repo
├── .chef-repo.txt
├── .git
│   ├── HEAD
│   ├── branches
│   ├── config
│   ├── description
│   ├── hooks
│   │   ├── applypatch-msg.sample
│   │   ├── commit-msg.sample
│   │   ├── post-update.sample
│   │   ├── pre-applypatch.sample
│   │   ├── pre-commit.sample
│   │   ├── pre-push.sample
│   │   ├── pre-rebase.sample
│   │   └── prepare-commit-msg.sample
│   └── update.sample
└── .gitignore
```

```
|── LICENSE
|── README.md
|── .chefignore
|── cookbooks
|   ├── README.md
|   └── example
|       ├── README.md
|       ├── attributes
|       |   └── default.rb
|       ├── metadata.rb
|       └── recipes
|           └── default.rb
|── data_bags
|   ├── README.md
|   └── example
|       └── example_item.json
|── environments
|   ├── README.md
|   └── example.json
└── roles
    ├── README.md
    └── example.json
```

This list of files looks pretty good but, as you may have noted, an essential folder is missing: the `.chef` folder, which needs to contain the private key files and the `knife.rb` file, used to communicate with the Chef server. Let's take a look at creating or obtaining these files. One way is to use the starter kit. It is an OK way if this is a new Chef server. But, if it's not, be sure to read the following Starter Kit info; it could save your job!

The repo starter kit

You still need the `.chef` files for your new workstation to securely communicate with your Chef server. These files are your user private key file, the organization's validator private key file, and the `knife.rb` file. These files need to be placed in a `.chef` folder in your `chef-repo`.

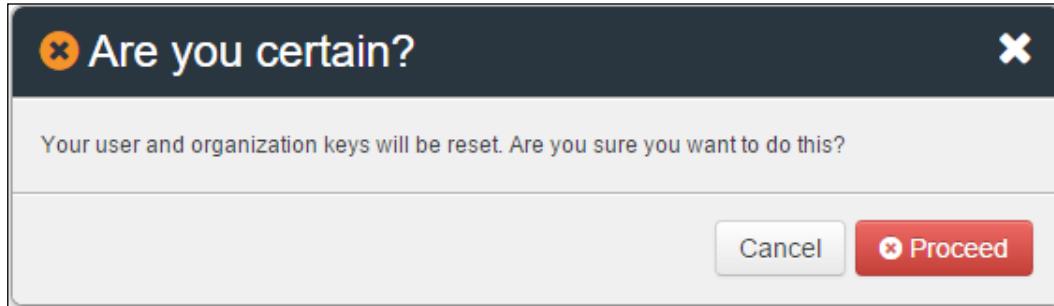
When setting up a new organization in your Chef server, there is an easy way to get the files: with the repo starter kit.

Warning!!! Only download the starter kit if you are the Chef server admin and are the first (or only) person setting up your repo for a brand new organization. Currently, the act of downloading the starter kit will reset both your user private key and the organization validator key.



Let me say this again. Downloading the Starter Kit will reset the private key values for both your user and your organization.

Downloading the starter kit on a mature organization will essentially break the Internet.



With this warning, if you are still going to use the starter kit, proceed at your own risk.

To get the starter kit, you will want to browse to your Chef server and log in. Once there, click on **Administration** and then select your Organization. Now click on **Starter Kit** and you will see the **Download Starter Kit** button. When you click the button, you will get the warning shown earlier. Click on the **Proceed** button to transfer a zip archive to your workstation, then unzip it into your chef-repo folder, and you're done.

Files for your .chef folder – a safer way

Really, the only files you need from the starter kit are the .chef folder files:

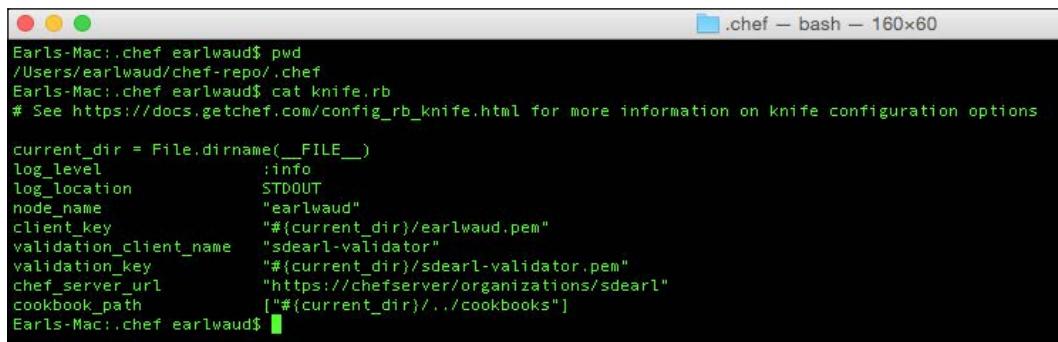
- The organization's validator private key file
- Your user private key file
- A knife.rb file

The organization's key file can be provided by an administrator of the Chef server. The organization's validator key is generated when the organization is created. When the command-line tool is used to create the org, the location of the validator key file may be provided in the command. Otherwise, it will be displayed to stdout. The Chef server admin will have captured the validator key info and will be able to provide it to you now.

The user private key is much like the organization's private key in that it is created when the user is created and must be captured and securely stored at that time as it is not saved to the Chef server database. A user can reset their private key through the use of the Reset Key feature available when viewing the user profile on a Chef server. A user resetting their private key will have no negative impact on other users or the organization, so it is relatively safe to do.

Finally, you need the `knife.rb` file. This file is the easy one. It can be downloaded from the Chef server any time by logging in and browsing to the organization. From there, you can click on any of the Generate Knife Config links in the UI. This will create the `knife.rb` file and download it to your workstation. Another option is to use the `knife configure` command. This will prompt you through the process of creating your `knife.rb` file.

Put all three of these files into the `.chef` folder of your new chef-repo. Make sure that you save the two key files using the names in the `knife.rb` file (or conversely, update the contents of the `knife.rb` file to match the file names used to save your key files).



```
Earls-Mac:.chef earlwaud$ pwd
/Users/earlwaud/chef-repo/.chef
Earls-Mac:.chef earlwaud$ cat knife.rb
# See https://docs.getchef.com/config_rb_knife.html for more information on knife configuration options

current_dir = File.dirname(__FILE__)
log_level         :info
log_location      STDOUT
node_name         "earlwaud"
client_key        "#{current_dir}/earlwaud.pem"
validation_client_name "sdearl-validator"
validation_key    "#{current_dir}/sdearl-validator.pem"
chef_server_url   "https://chefserver/organizations/sdearl"
cookbook_path     ["#{current_dir}/../cookbooks"]
Earls-Mac:.chef earlwaud$
```

Growing your own tree

OS X doesn't come with a version of the `tree` command installed so, if you like that tool and use a Mac, I've found a clever way to have your cake and eat it too by creating your own `tree` script. This script was used to create the "tree" output shown in the preceding screenshot. This idea was shared by users Ahmed Masud and JakeGould on <http://superuser.com>. It goes like this:

There isn't a formal `tree` command per se however you can do this:

Save the following script to `/usr/local/bin/tree`

```
#!/bin/bash

SEDMAGIC='s;[^/]*;/____;g;s;____|; |;g'

if [ "$#" -gt 0 ] ; then
    dirlist="$@"
else
    dirlist=."
fi

for x in $dirlist; do
    find "$x" -print | sed -e "$SEDMAGIC"
done
```

Change the permissions so you can run it:

```
chmod 755 /usr/local/bin/tree
```

Of course you may have to create `/usr/local/bin`:

```
sudo mkdir -p /usr/local/bin/tree
```

share improve this answer

edited Jan 21 at 22:17



JakeGould

17.1k ● 5 ▪ 49 ▬ 61

answered Nov 21 '11 at 10:53



Ahmed Masud

356 ● 1 ▬ 6

Self-signed certificates

When you are using an On Premise Chef server, you need to do one more thing to finish setting up your chef-repo. If the Chef server you are connecting to has a self-signed certificate, then you will need to add the server's certificate to your chef-repo to allow your workstation to trust your Chef server. An easy way to check if this applies to you is to run the `ssl check` command. It goes like this:

```
knife ssl check
```

If the Chef Server you are communicating with has a self-signed certificate, you will see a message like this:

```
Earls-Mac:chef-repo earlwaud$ knife ssl check
Connecting to host chefserver:443
ERROR: The SSL certificate of chefserver could not be verified
Certificate issuer data: /C=US/O=YouCorp/OU=Operations/CN=chefserver

Configuration Info:

OpenSSL Configuration:
* Version: OpenSSL 1.0.1p 9 Jul 2015
* Certificate file: /opt/chefdk/embedded/ssl/cert.pem
* Certificate directory: /opt/chefdk/embedded/ssl/certs
Chef SSL Configuration:
* ssl_ca_path: nil
* ssl_ca_file: nil
* trusted_certs_dir: "/Users/earlwaud/chef-repo/.chef/trusted_certs"

TO FIX THIS ERROR:

If the server you are connecting to uses a self-signed certificate, you must
configure chef to trust that server's certificate.

By default, the certificate is stored in the following location on the host
where your chef-server runs:

    /var/opt/opscode/nginx/ca/SERVER_HOSTNAME.crt

Copy that file to your trusted_certs_dir (currently: /Users/earlwaud/chef-repo/.chef/trusted_certs)
using SSH/SCP or some other secure method, then re-run this command to confirm
that the server's certificate is now trusted.

Earls-Mac:chef-repo earlwaud$
```

As you can see in the error message generated by the `ssl check` command, the Chef server's certificate needs to be obtained and placed into a `trusted_certs` folder in your `chef-repo/.chef` folder. Again, you will need to contact your Chef server administrator to get a copy of the certificate file. Once you have it, just place it in the `trusted_certs` folder; when you run your `ssl check` command again, you should get a message back that says something like:

```
knife ssl check
Connecting to host chefserver:443
Successfully verified certificates from 'chefserver'
```

With that successful knife command completed, you have verified that you have successfully set up your chef-repo and are ready to start creating your infrastructure as code.

References

- Learn about setting up your Chef repo at the link https://docs.chef.io/chef_repo.html.
- Learn more about setting up a chef-repo at the link <https://learn.chef.io/manage-a-web-app/ubuntu/get-set-up/>.
- OS X doesn't come with a version of tree installed so, if you like that tool and use Mac, there is a clever way to have your cake and eat it too by creating your own a tree script. You can find an example shared by Ahmed Masud and JakeGould at the link <http://superuser.com/questions/359723/mac-os-x-equivalent-of-the-ubuntu-tree-command>.
- Keep up-to-date on the Starter Kit resetting the organization's validator key at the link <https://feedback.chef.io/forums/301644-chef-product-feedback/suggestions/9618459-starter-kit-should-not-reset-validator-and-user-ke>.

What else do you need?

You will need just a few more things to round out your workstation toolbox. First, you will want to make sure that you have your favorite text editor and have it configured to integrate with Chef. Next, since you are creating infrastructure code, you really must use a source code control system. I don't want to give anything away on that topic yet but ... Git Git Git! Then, you will want to add a hypervisor to your workstation to deploy test nodes. Finally, you'll want to add Vagrant to manage those test nodes. So, let's take a look at these finishing touches for your workstation.

Using your favorite editor

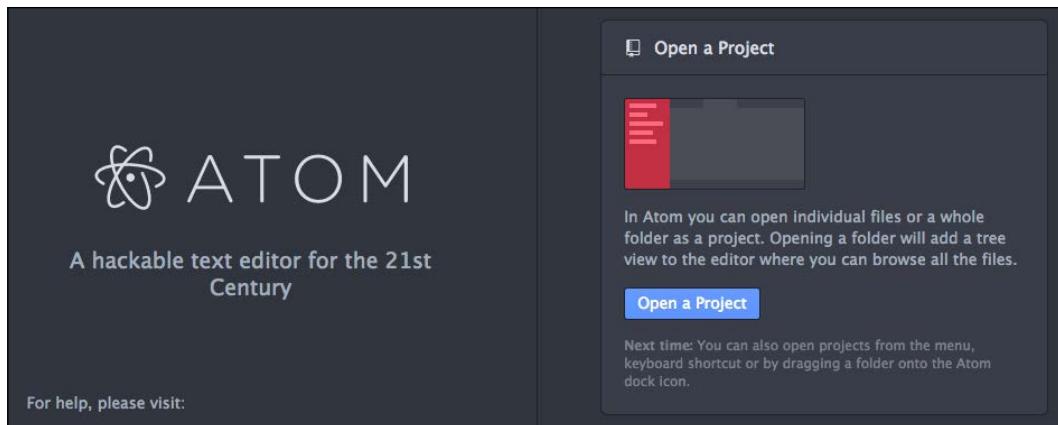
Everyone has their favorite text editor. For a lot of people that editor is Sublime Text, but there are many other choices available. A lot of old schoolers still prefer to use a flavor of the vi editor. Many OS X users swear by Mac-only TextMate. And now there's a new kid on the block from the makers of GitHub: Atom 1.0. Sublime Text and Atom are both available for multiple platforms, so whatever OS is your poison, they have a version for you.



I would recommend that, if you don't already use an editor that has a "project space" feature, you switch to one that does. This feature allows you to quickly and easily switch to and edit different files within a project without the delay of the "save current then open new" workflow needed in an editor that does not have the project space feature.

The project space feature is one of the many reasons I usually recommend Sublime Text, but this feature is also present in other editors such as the new Atom 1.0.

To keep things current, I will use the Atom 1.0 editor in the remainder of this book's examples.



Once you have decided on the editor you plan to use for your Chef workstation and have that editor installed and working, there is one more step. You will need to set the configuration to allow you to integrate your editor with the Chef command-line tools. The best way to do that is to edit your `.chef/knife.rb` file and add a line to set the editor you want to use with your knife commands. (You could also add an `export EDITOR=` command to your shell's `.bashrc` or `.bash_profile` file.)

Here are some sample `knife.rb` entries:

Sublime Text:

```
knife[:editor] = '/Applications/Sublime\ Text.app/Contents/  
SharedSupport/bin/subl -w'
```

Atom:

```
knife[:editor] = '/Applications/Atom.app/Contents/MacOS/Atom -w'
```

On Mac, the Atom editor has a built-in feature to create symbolic links in the `/usr/local/bin` folder. With the Atom editor running, open the Atom menu and select **Install Shell Commands** from the submenu. This will create two new links for you. If you do this before updating your `knife.rb` file, the `knife[:editor]` entry simply becomes this:

```
knife[:editor] = 'atom -w'
```

When you're using a Windows OS for your workstation, you will have to take greater care in formatting your `knife[:editor]` line to escape the slash characters in the path. A convenient way to do that is to use single quotes around the full value and use double quotes for the path. For Sublime Text, it might look something like this:

```
knife[:editor] = '"C:\Program Files\Sublime Text 2\sublime_text.exe"  
--wait'
```

You can test your editor setting by issuing a `knife` command that requires using the editor. For example, try the command:

```
knife client create editor-tester
```

If your editor integration is configured correctly, you will see your editor open up and show you some code. Save the code and exit the editor, and you should see the private key value print out. (This key is not important at this time because we are just testing the editor, and will immediately delete the client.)

If your editor opens, you are done. If you get an error that says **Please set EDITOR environment variable**, then your `knife[:editor]` value is incorrect. There is probably a typo in the path or some other error. Double-check the value and try again until you successfully launch the editor with your `knife client create` command.

Once you have validated the integration with your editor, you can delete the test client with the following command:

```
knife client delete editor-tester
```

Now you are ready to edit your infrastructure code! When you are creating amazing infrastructure code, you are going to want to make sure that it gets preserved and to ensure that happens you going to want to use a source control tool. Say hello to my little friend, Git.

Version control systems

Git, initially designed and developed by Linus Torvalds for Linux kernel development back in 2005, is now the most widely used version control system for developers everywhere. It's the number-one choice due to its speed, data integrity, and its distributed workflow model. Git is certainly the primary choice as the version control system to be used when developing infrastructure code with Chef.

I will be emphasizing the use of Git throughout the examples in the rest of the book. Git is included with Mac OS X, but it will need to be installed on Windows and Ubuntu workstations. I recommend you to use the GitHub desktop installer where available (Windows and OS X) and allow it to deploy the command-line tools as part of the installation. This installation of Git is easy and will be left to the reader to execute.

Virtualization hypervisor

There are several virtualization choices you can use for your Chef testing. The overall goal is to have a system that you can use to deploy a temporary Chef node to test your infrastructure code. One of the best ways to accomplish this is to install a local virtualization tool on your workstation. Among the choices for a local hypervisor are:

- VirtualBox
- Docker
- VMware Workstation or Fusion

One of the best choices for this kind of Chef testing is VirtualBox. The reasons include its functionality, its ease of use, and its free price tag. Either of the VMware solutions will cost you not only for the hypervisor layer, but also for the integration plugins we'll use with Vagrant to manage our test nodes.

Visit the VirtualBox downloads page at <https://www.virtualbox.org/wiki/Downloads> to download the installer. Once downloaded, it is an easy matter to install it, so I'll leave that to the reader to take care of.

Vagrant

Last, but certainly not least, is Vagrant. Vagrant is another item I consider to be one of the key pieces of the Chef testing puzzle. It is the "glue" between Chef and the hypervisor that deploys the test nodes during testing. You will want to download and install Vagrant on your workstation. You can find the downloaders at <https://www.vagrantup.com/downloads.html>. Vagrant is also an easy install, so I'll leave that to the reader to handle.

I'll go into using Vagrant with VirtualBox (and Docker) in some detail in a later chapter.

References

- You can download Sublime Text 2 from <http://www.sublimetext.com/>
- If you want to try the long-running beta of Sublime Text 3, it can be found at <http://www.sublimetext.com/3>
- If you are looking to try Atom 1.0, you can get it at <https://atom.io/>
- The OS X editor, TextMate, can be had at <https://macromates.com>
- Learn more about GitHub at <https://github.com/>
- Learn about and download VirtualBox from <https://www.virtualbox.org/>
- Learn more about Docker at <https://www.docker.com/>
- To get VMware Workstation go to <https://www.vmware.com/products/workstation>
- Or for Mac OS X, get VMware Fusion at <https://www.vmware.com/products/fusion>

What is chef-apply?

One often overlooked tool that is installed with the ChefDK is `chef-apply`. This tool basically allows you to execute chef to converge a single recipe on your local machine. There is no Chef server involved. Everything is local.

Note that `chef-apply` is not a tool for the deployment of production nodes. It's just a quick and easy way to use Chef to configure your local system.

Why is that useful?

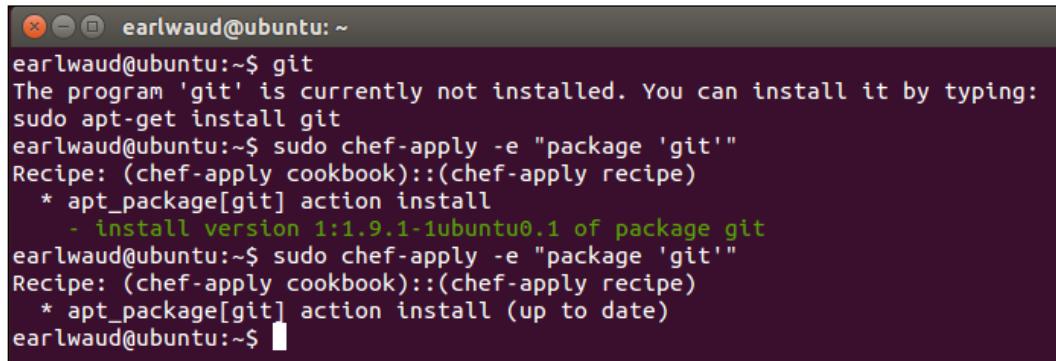
One use case for `chef-apply` is to learn what a recipe does. You can use `chef-apply` to run a recipe in a "test only" mode by using the `-w` or `--why-run` parameter. This will converge the recipe, showing you everything that would happen with the recipe without making actual changes to your system.

You can also execute some one-liner scripts. You can easily use `chef-apply` in a scripting-like manner to install software. For example, if you need to use Git on your system and you find it's not installed. Just issue the command:

```
sudo chef-apply -e "package 'git'"
```

This will check to see if Git is currently installed. If it is, then the command will just inform you that that is the case and exit. If Git is not installed, then it will go about using the correct installer (aka Provider) to obtain and install the package.

Testing this on my Ubuntu workstation, you can see that Git was not installed initially. Then, by using the `chef-apply` command, the `git` package is installed. Then to confirm that nothing happens if you try to use `chef-apply` to install a package that is already installed, I ran the same `chef-apply` again to show what happens:



```
earlwaud@ubuntu:~$ git
The program 'git' is currently not installed. You can install it by typing:
sudo apt-get install git
earlwaud@ubuntu:~$ sudo chef-apply -e "package 'git'"
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * apt_package[git] action install
    - install version 1:1.9.1-1ubuntu0.1 of package git
earlwaud@ubuntu:~$ sudo chef-apply -e "package 'git'"
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * apt_package[git] action install (up to date)
earlwaud@ubuntu:~$ █
```

How about creating a "workstation" recipe to set up all the "missing" items from a new ChefDK workstations setup? You create the recipe that defines the state of having your editor installed and integrated with Chef, defines the state of having `git` installed, and defines the state of having your custom OS X "tree" script created in `/usr/local/bin`. You create this recipe once, and you (and anyone in your organization) can quickly complete their workstation setup by using `chef-apply` with your workstation recipe.

References

- The link to get the details of an interesting use case for `chef-apply` posted by Chris Doherty on `chef.io` is <https://www.chef.io/blog/2015/01/16/convert-your-bash-scripts-with-chef-apply/>.
- The link to the `chef.io` documentation on `chef-apply` is https://docs.chef.io/ctl_chef_apply.html.

Summary

In this chapter, we've taken a quick stroll down memory lane and reviewed some Chef basics so that those concepts are fresh and easy to mentally access during the rest of the journey through this book. Next up, we took a deeper dive into the Chef Development Kit and learned how to download, install, and configure it for our Chef workstation. We've exposed the small gap in the ChefDK installer when used on Mac OS X and Linux systems, which is the setup of the shell environment and learned the simple way (provided by the new Chef tool) to fill that gap. We also explored the setup and configuration of your very own on-premise Chef server, including creating your admin user and the first organization. After this, we took a look at setting up our Chef repo, giving a BIG WARNING about the starter kit. And then we considered what was still missing from a well-rounded Chef workstation. That will be our favorite editor, a version control system, a virtualization layer to simplify our testing flow, and Vagrant to glue things together. Finally, as an added bonus, we talked about the ChefDK tool `chef-apply`.

Wow! That was a lot of ground for a single chapter, but I hope that you were able to stick with me till the end. This chapter is a great foundation for the Chef knowledge transfer coming your way in the chapters ahead.

So without any further delay, let's charge ahead into *Chapter 2, Knife Tooling and Plugins*, and carve things up with some Swiss Army knife skills.

2

Knife Tooling and Plugins

In this chapter, we're going to cover the primary command-line tool in the Chef repertoire. Basic Knife functionality is detailed, as well as the use of community plugins and instructions on how to create your own new plugins.

"Knife skills are the gateway drug to good cooking" – Chef Todd Mohr, Certified Culinary Educator (CCE)

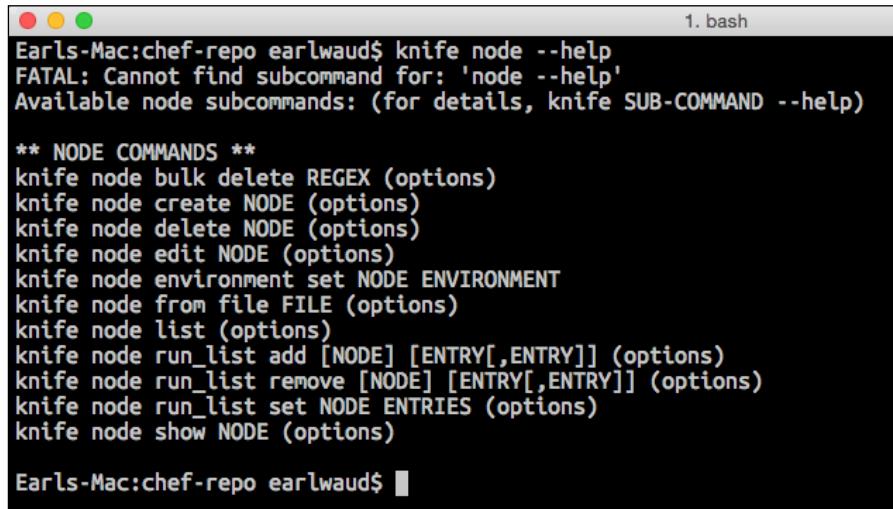
In this chapter, the reader will learn to wield Knife and extend its capabilities as needed. Here is the list of topics that will be covered in this chapter:

- Knife help
- Knife subcommands
- Knife to editor integration
- Knife Windows plugin
- Knife EC2 plugin
- Creating new Knife plugins

Knife help

Let's begin with the basics. As with all command-line tools in the Chef toolbox, knife has a rich help system built in. Issuing the command `knife --help` will display the basic knife syntax help contents. When you issue the `knife --help` command, you get a list of all the optional parameters, plus a list of all the knife subcommands. In addition, each of the subcommands has its own help content. For example, ask for help with the subcommand `node`, and you will get the syntax for the full set of node-specific subcommands and options.

Here is what it will look like when you issue the `knife node --help` command:



A screenshot of a terminal window titled "1. bash". The window shows the output of the command `knife node --help`. The output includes error messages about not finding a subcommand, a list of available node subcommands, and a detailed list of node commands with their descriptions and options.

```
Earls-Mac:chef-repo earlwaud$ knife node --help
FATAL: Cannot find subcommand for: 'node --help'
Available node subcommands: (for details, knife SUB-COMMAND --help)

** NODE COMMANDS **
knife node bulk delete REGEX (options)
knife node create NODE (options)
knife node delete NODE (options)
knife node edit NODE (options)
knife node environment set NODE ENVIRONMENT
knife node from file FILE (options)
knife node list (options)
knife node run_list add [NODE] [ENTRY[,ENTRY]] (options)
knife node run_list remove [NODE] [ENTRY[,ENTRY]] (options)
knife node run_list set NODE ENTRIES (options)
knife node show NODE (options)

Earls-Mac:chef-repo earlwaud$
```

You can see from this help text that, using the `knife node` command, you can create nodes, delete nodes, list nodes, show the details of nodes, and much more. Each subcommand available to the `knife` command has a similar set of rich help details available on demand.

Most of the actions associated with the subcommands have additional syntax help details available, and they can be accessed in the same fashion as earlier. For example, issuing the command `knife node show --help` will give you all of the options available for the action "show" of the subcommand `node`.

The frequency of use of the `knife` subcommands vary greatly, and having this level of syntax help at your fingertips is very valuable. When you are trying to use a subcommand that you don't regularly execute, remembering to check the syntax and options using this help system will help you avoid frustration and mistakes.

However, if you need more than syntax help on a specific subcommand, you can invoke the full help text for that command by using the full `knife help` command. For example, entering the command `knife help node` will present you with the full `knife-node` help man page. This content will detail all of the options available to the subcommand, in this case, the `node` subcommand. It includes a description of the function of the subcommand, the syntax, the options, and example uses of the subcommand – in short, a rich man page for the subcommand.

References

There is an excellent help reference available on GitHub that takes the built-in syntax help information to the next level. You can download and print out your own reference guide from <https://github.com/chef/quick-reference>.

Knife subcommands

Now that we have some help skills under our belt, let's take a look at some of the other knife subcommands in greater detail. Using the various knife subcommands, we will manage nodes, cookbooks, recipes, roles, and environments. We will also use knife to do searching of our Chef server data. All of this and more is available using Knife.

We will have a look at some of the most used commands, but also dig deeper into some of the less frequently used commands to better understand what they can do for us.

Knife Bootstrap

You use the knife bootstrap subcommand to install the chef-client on systems so that it can be used on those systems to communicate with the Chef server. The bootstrap process also registers the system with the Chef server to facilitate communication between the two. In addition to installing the chef-client and registering it with the Chef server, bootstrapping also creates a Node object on the Chef server that is used to manage the configuration of the bootstrapped system.

The primary syntax of the bootstrap command is `knife bootstrap <IP | FQDN>`. For example, to bootstrap a server by IP address, you would use a command like `knife bootstrap 192.168.10.44`. This instructs knife to install the chef-client on the system with the IP address 192.168.10.44, register that system with the Chef server, and create a node representing the system on the Chef server.



To obtain the Chef server's SSL certificate, you can use the knife subcommand `ssl_fetch` to download the on-premise Chef server's SSL certificate and install it in your `trusted certs` folder. This certificate is used in the knife bootstrap subcommands.

Knife Tooling and Plugins

Here is an example knife bootstrap command that I used to bootstrap an opscode-centos-7.1 vagrant node running on VirtualBox:

```
knife bootstrap 127.0.0.1 --node-name vagrant-node01 --ssh-port 2222  
--ssh-user vagrant --identity-file ~/chef-repo/.vagrant/machines/default/  
virtualbox/private_key --sudo
```

Here is the result of running that bootstrap command:

```
1. bash  
Earls-Mac:chef-repo earlwaud$ knife bootstrap 127.0.0.1 --node-name vagrant-node01 --ssh-port 2222 --ssh-user  
vagrant --identity-file ~/chef-repo/.vagrant/machines/default/virtualbox/private_key --sudo  
Doing old-style registration with the validation key at /Users/earlwaud/chef-repo/.chef/hosted/sdearl-validation.pem...  
Delete your validation key in order to use your user credentials instead  
  
Connecting to 127.0.0.1  
127.0.0.1 -----> Installing Chef Omnibus (-v 12)  
127.0.0.1 downloading https://omnitruck-direct.chef.io/chef/install.sh  
127.0.0.1 to file /tmp/install.sh.12437/install.sh  
127.0.0.1 trying wget...  
127.0.0.1 el 7 x86_64  
127.0.0.1 Getting information for chef stable 12 for el...  
127.0.0.1 downloading https://www.chef.io/stable/chef/metadata?v=12&p=el&pv=7&m=x86_64  
127.0.0.1 to file /tmp/install.sh.12442/metadata.txt  
127.0.0.1 trying wget...  
127.0.0.1 sha1 5907edce1a3b0f7bd42359fe64960da8833385e9  
127.0.0.1 sha256 1c3e680f106ab6829c3713307e447116f7bb3f2d9c30fd3943638b01d6246fe2  
127.0.0.1 url https://packages.chef.io/stable/el/7/chef-12.9.38-1.el7.x86_64.rpm  
127.0.0.1 version 12.9.38  
127.0.0.1 downloaded metadata file looks valid...  
127.0.0.1 downloading https://packages.chef.io/stable/el/7/chef-12.9.38-1.el7.x86_64.rpm  
127.0.0.1 to file /tmp/install.sh.12442/chef-12.9.38-1.el7.x86_64.rpm  
127.0.0.1 trying wget...  
127.0.0.1 Comparing checksum with sha256sum...  
127.0.0.1 Installing chef 12  
127.0.0.1 installing with rpm...  
127.0.0.1 warning: /tmp/install.sh.12442/chef-12.9.38-1.el7.x86_64.rpm: Header V4 DSA/SHA1 Signature, key ID  
83ef826a: NOKEY  
127.0.0.1 Preparing... ##### [100%]  
127.0.0.1 Updating / installing... ##### [100%]  
127.0.0.1 1:chef-12.9.38-1.el7 ##### [100%]  
127.0.0.1 Thank you for installing Chef!  
127.0.0.1 Starting the first Chef Client run...  
127.0.0.1 Starting Chef Client, version 12.9.38  
127.0.0.1 Creating a new client identity for vagrant-node01 using the validator key.  
127.0.0.1 resolving cookbooks for run list: []  
127.0.0.1 Synchronizing Cookbooks:  
127.0.0.1 Installing Cookbook Gems:  
127.0.0.1 Compiling Cookbooks...  
127.0.0.1 [2016-04-17T22:59:55+00:00] WARN: Node vagrant-node01 has an empty run list.  
127.0.0.1 Converging 0 resources  
127.0.0.1  
127.0.0.1 Running handlers:  
127.0.0.1 Running handlers complete  
127.0.0.1 Chef Client finished, 0/0 resources updated in 24 seconds  
Earls-Mac:chef-repo earlwaud$ █
```

The optional parameters that I used in this command are:

- `--node-name vagrant`: Create a Node object on the Chef server with the name "vagrant".
- `--ssh-port 2222`: Use port "2222" when connecting to the system being bootstrapped.

- `--ssh-user vagrant`: When ssh'ing to the system, log in with username "vagrant".
- `--identity-file ~/chef-repo/`: Use key-based authentication and use the listed key file "`~/chef-repo/...`".
- `--sudo`: Execute the commands on the bootstrapped system using sudo.

There is one significant option that I did not include in the example shown earlier. Did you notice its absence? That optional parameter is `--run-list <RUN_LIST>`. The run list option allows you to specify a comma-separated list of policies, roles, and recipes, that are used in the initial chef-client run.

Another common optional parameter used is `--environment`. This parameter can be used to associate an environment with the system/node, so that only policies, roles, and recipes that are part of the associated environment are used when configuring the server.

Knife Node

The knife node subcommand is used to query and set the various configuration values for a node. One of the most common uses of this subcommand is to get a list of nodes known to your Chef server. The command to do this is `knife node list`. This will return the complete list of nodes on your Chef server. The node subcommand can also be used to create, delete, or edit node objects on your Chef server.

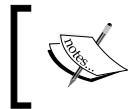


Make sure that you have set up your editor to work with Knife, as shown in the *Knife to editor integration* section of this chapter.

Other uses for the node subcommand include adding or removing a node's run list. And of course, there is the `knife node show <node name>` command to let you see the current configuration settings stored in the Node object on the Chef server.

Knife Diff

Another useful knife subcommand is `knife diff`. Knife diff is used to compare the files in your current Workstation repo with what is stored on the Chef server. You can compare files before you upload them to the Chef server. You can also use `knife diff` to compare the repo files with versions used for different environments. You will find this subcommand to be very valuable when modifying resources, such as cookbooks and recipes, to compare your changes to the state of the resource on the Chef server. It provides an excellent sanity check on your work.



Remember that you can use knife diff to compare individual files, like a single recipe file, or you can use it to compare folders, like an entire cookbook.



Knife Exec

Exec is a knife subcommand that lets you run Ruby scripts against the Chef server in the context of a fully configured chef-client. What that means is that you can do things to the Chef server via Ruby scripts that would be much more difficult or impossible to accomplish using other subcommands. For example, if you want to move nodes from one environment to another, you can issue a knife exec subcommand. Here is what a knife exec command to move all nodes in the "_default" environment into the "development" environment:

```
knife exec -E 'nodes.transform("chef_environment:_default") { |n| n.chef_environment("development") }'
```

This is something that you would not need to do very often, but having the ability to do it with a single knife exec command would still save you a lot of time. Another example use case would be adding a role or recipe to a subset of your nodes, such as all the nodes in production. If, for example, you want to add the recipe "base" to all of your nodes in production, you could issue a knife exec command like this:

```
knife exec -E 'nodes.transform("chef_environment:production") { |n| puts n.run_list << "recipe[base]"; n.save }'
```

Executing this command will update the `run_list` for all of the nodes in the production environment adding the recipe "base".



Issuing the command without the `n.save` as a test run

If you issue the preceding command without including the `n.save` first, then you can see the effects of the `knife exec` command without applying the changes to the server. If everything looks right, run it again with the `n.save` to apply the changes. This lets you try test your commands before they make changes to your environment.

Issue the following command first to test the results:

```
knife exec -E 'nodes.transform("chef_environment:production") { |n| puts n.run_list << "recipe[base]" }'
```

Then, issue the command again with `n.save` to apply the changes:

```
knife exec -E 'nodes.transform("chef_environment:production") { |n| puts n.run_list << "recipe[base]"; n.save }'
```

Knife SSH

The knife ssh subcommand lets you run ssh commands on all or a subset of the nodes known to your Chef Server. The nodes that will have the ssh command invoked on them are determined by a search or a query against the Chef Server's nodes. For example, if you want to run the uptime command on all of the nodes in the production environment, you can issue a command like this one:

```
knife ssh "chef_environment:production" "uptime" -p 2222 -x earl -P MyPW99
```

The search used is `chef_environment:production`. The command being issued on the matching nodes is "uptime". In this case, the port being used for ssh is 2222, and the credentials are self-explanatory.

Another good use of the knife ssh subcommand is to combine it with the knife exec subcommand. This combination lets you do some very powerful things. For example, you can check the status of the ntpd service on all your production nodes with a command something like this:

```
knife ssh -p 2222 -x earl -P MyPW99 -m "`knife exec -E "search(:node, 'chef_environment:production').each { |host| puts host[:hostname] }" | xargs ^ '/etc/init.d/ntpd status'
```

Here, we are searching for all of the nodes in the production environment, and for each node, we are sshing into the system and issuing the command `/etc/init.d/ntpd status`.

Using a combination of the knife ssh subcommand and the knife exec command is an excellent and easy way to create ad hoc reports.

Knife Search

The knife search subcommand is one of the most used commands in the knife set. knife search lets you do queries against all of the data stored in the Chef server. Since the chef-client run sends system data collected via the Ohai command to the Chef server, there is a wealth of search criteria that is available to search against. The knife search subcommand uses two main parts: an index and a query. A knife search is executed against one of the indexes on the Chef server. The possible indexes are "client," "environment," "node," or "role."

Note that if you do not specify an index, knife search uses the default of "node". The other half of the knife search subcommand is the query. The query is the filter applied to the index to limit the results of the search. In addition, there are several optional parameters that modify the data returned from the search. For example, you can use the optional parameter of "-i" to instruct knife search to return only the ID of the objects in the index that match the filter used. Here, a sample command will better explain. If you want to see a list of all the nodes in your Chef server, you can issue the following command:

```
knife search node '*' -i
```

Here, the knife search command is using the index of "node" and the query or filter is `'*':*`, and the `-i` says to just give me the index IDs for the matching objects. The index IDs for the node index are the node names.

Remember that the default index is "node", so the same results will be delivered if you issue the command like this:

```
knife search '*' -i
```

Without the optional `-i`, the knife search command will return the Node object details, and there are varying levels of detail you can have the search command return. Here are the options:

- The `-i` parameter: just the node name, no object details
- No parameter: short list of the object's details
- The `-m` parameter: medium list of the object's details
- The `-l` parameter: full list of all of the object's details

You can also specify specific attributes of the node object, that is, the IP Address, with the `-a` parameter. The following command will return the IP address of all the nodes in the Chef server:

```
knife search node '*' -a server_ip
```

This is only the tip of the knife search iceberg. In these basic examples, we are essentially ignoring the query part of the search command by using a `"*":*`. Say that you need to see a list of all of the Nodes that are in the 192.168.112.x subnet. You can issue the following knife search command to create the list:

```
knife search node 'server_ip*:192.168.112.*' -i
```

Truly, the only limit to what you can search for is your imagination. The knife search command is one of the most powerful commands in the knife set.

References

- You can read more about the knife command and all of its subcommands on the Chef site at <https://docs.chef.io/knife.html>.
- The link to the Chef documentation on knife search is at https://docs.chef.io/knife_search.html.
- The link to the Chef bootstrapping data is at https://docs.chef.io/knife_bootstrap.html.

Knife to editor integration

Many of the interactive knife subcommands use an editor to create or modify the infrastructure code. The proper integration of your favorite editor is important to a fluid knife experience.

Using the "wait" option

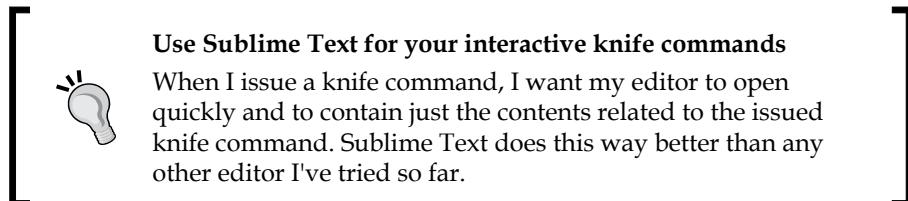
The key to successfully using your editor with knife is to configure it to use the editor's "wait" option. The wait option tells your editor to launch, open the specified file or content, but not return or exit from the launch command until the editor is exited.

In your standard modern OS, launching most applications from the command line will invoke the application and then immediately return control to the command-line shell. The command, in this case, invokes the editor; actually, it executes completely by launching the editor. This is the behavior that needs to be avoided when using the knife commands. Depending on the editor of choice and OS, we may need to configure the "invoke the editor" command so that it does not return as soon as the editor is invoked, but instead waits until the editor is closed before it returns control to the command-line shell. Each editor has its own way of indicating a desire to wait for exit before returning control. The best way to configure your editor to use its wait option is to add the parameter to the editor line in your `~/.chef/knife.rb` file. When needed, the setting will take the following format:

```
knife[:editor] = '<path to your editor> <wait option>'
```

Does my editor require the wait option?

As you might expect, every editor has its own unique features and behaviors. Two common editors, Sublime Text and Atom, both require a wait parameter to allow them to be used successfully with knife. Both use a "-w" for the wait parameter. If you use VI or TextMate, then no wait option is necessary, as both editors, when invoked, hold control until the editor is exited.



On my OS X system, I use Sublime Text as my editor of choice for the interactive knife commands. The `knife.rb` file entry that I use looks as follows:

```
knife[:editor] = '/Applications/Sublime\ Text.app/Contents/  
SharedSupport/bin/subl -w'
```

Moreover, even though I am using Atom as my current code editor of choice, I still prefer to use Sublime Text with my interactive knife commands. This is because I have not found a way to have Atom launch the knife command into a new window that does not also open tabs for the files that were open the last time Atom was launched. In addition to this being an undesirable behavior for me, it also slows down the launch of the editor. I have tried all the various combinations of the command-line parameters available for Atom, including --wait, --foreground, and --new-window, without finding a combination that works for me. Your mileage may vary.

References

- You can get the Sublime Text editor from <http://www.sublimetext.com/2>.
- You can get the Atom editor from <https://atom.io/>.
- You can get TextMate (Mac OS only) from <https://macromates.com/download>.

Knife Windows plugin

Now, let's take a look at a couple of community knife plugins. We'll start with the Windows plugin. We use the knife Windows plugin when we are working with nodes running the Windows operating system. First, you will need to install the knife Windows plugin and then we will review how to use it to bootstrap a Windows node. Then, we will use some of the search features to see some of the Windows-specific node data.

Installing the Knife Windows plugin

Like many things related to the new ChefDK, installing Chef-specific gems is pretty straightforward and easy. In this case, we want to install the knife-windows gem. The command to do this is:

```
chef gem install knife-windows
```

That is all it takes. You are now ready to use the knife Windows plugin.

Bootstrapping a Windows node

Now that we have the knife Windows plugin installed, we can bootstrap our Windows nodes. There are two methods to interface with a Windows system when bootstrapping a node using the knife Windows plugin. One uses ssh just as though the node is a Linux system. The other is using the WinRM interface. The benefit of using the WinRM protocol is that it allows the use of standard Windows objects to be used, such as batch scripts and PowerShell scripts. Here is an example of using the WinRM method to bootstrap a node named "node03":

```
knife bootstrap windows winrm node03 -p 55985 -x Administrator -P MyPW99
-r 'recipe[winbase]'
```

In the preceding command, you can see that we are using the Windows plugin, and the WinRM protocol to bootstrap a node named "node03." We are specifying port 55985 and using the Administrator credentials.



Remember to use a local admin account

Although a node can be connected to an Active Directory domain, it is still necessary to bootstrap the node using a local administrator account.

The last part of the bootstrap command was to tell the node to use the recipe "winbase" in the first chef-client run.

Searching for Windows node data

Now that we know how to bootstrap some Windows nodes, let's take a look at some knife searches that use some Windows-specific attributes. Starting off with something easy, say we need to create a list of all the windows nodes in production:

```
knife search node 'chef_environment:production && os:windows' -i
```

But now we've been asked to provide a list of all the production Windows nodes that have not been properly joined to the domain (that is, they still have a domain of WORKGROUP). This one is a bit tricky because the knife search command does not handle nested attributes all that well, and the attribute we need is very nested (kernel.cs_info.domain). We can see the values with a command like this:

```
knife search node 'chef_environment:production && os:windows' -a kernel.cs_info.domain
```

This will provide a list of all production windows nodes, but we will still have to manually filter that list down to those nodes that aren't joined to the domain.

If we try to use a command that includes a nested attribute, the search will find zero matches. That is, a command like this will not work for us:

```
knife search node 'chef_environment:production && os:windows && kernel.cs_info.domain:WORKGROUP' -i
```

So, what can we do? It's the knife exec subcommand that comes to the rescue. We can use nested attributes in the find of a knife exec command, something like this:

```
knife exec -E 'nodes.find("chef_environment:production && os:windows && kernel.cs_info.domain:WORKGROUP"){|n| puts "#{n.name}"}'
```

Here, we are using all three conditions in nodes.find to provide a list of nodes that are in the production environment, have windows as the OS, and have the (nested) attribute kernel.cs_info.domain equal to WORKGROUP. Using this type of "search" will net us the list of nodes we desire. Hurray!

References

- The knife Windows plugin on the Chef site is discussed in detail at https://docs.chef.io/plugin_knife_windows.html.
- The GitHub repo for the knife Windows plugin is discussed in detail at <https://github.com/chef/knife-windows>.

Knife EC2 plugin

Let's continue our community plugin exploration with the knife EC2 plugin. This plugin is used to integrate knife with Amazon cloud instances that allow you to create and manage instances from the command line.

Installing the Knife EC2 plugin

Much like installing the knife Windows plugin, the installation process for the knife EC2 plugin is super easy. The Chef command-line tool comes to our aid once again. Simply issue the following command:

```
chef gem install knife-ec2
```

Your installation will be done in a few seconds. You might note that there is a dependent gem installed; it is the fog-aws gem, which the EC2 plugin uses.

Unlike the knife Windows plugin, after the plugin is installed, there is still more to configure to be able to use it. The knife EC2 plugin requires some specific data related to your Amazon account. There are several ways to make the Amazon data available to the EC2 plugin. The data can be used via environment variables, or as knife.rb configuration settings, or by creating an AWS credential file and referring to it in your knife.rb file. This is the method I would recommend because sometimes, the knife.rb file is uploaded to your git repo and that would put your AWS access key ID and secret access key at risk. However, if you still want to add the Amazon data to your knife.rb file, it should be added in this format:

```
knife[:aws_access_key_id] = "Your AWS Access Key ID"
knife[:aws_secret_access_key] = "Your AWS Secret Access Key"
```

If, instead, you want to use the safer AWS credential file method, decide on or create a location to store the credential file, and create a new file entering in the credential data in the format shown later. Note that you do not enclose your ID or Key in any kind of quotes like you would if you are adding them directly to the knife.rb file:

```
[default]
aws_access_key_id = Your AWS Access Key ID
aws_secret_access_key = Your AWS Secret Access Key
```

Now update your knife.rb file to point to the credential file, by adding a line as follows:

```
knife[:aws_credential_file] = "/path/to/credentials/file"
```

Okay, now you are ready to use the knife EC2 plugin. If you already have some EC2 instances running, go ahead and test your new plugin by listing them out. You can issue a command like this:

```
knife ec2 server list -r us-west-2
```

The `-r` option tells the EC2 plugin what region you are accessing.

Create a "default" region by adding it to your `knife.rb` file.

If you want to save yourself the extra typing needed to add the `-r <region>` to all of your EC2 plugin commands, you can add an entry into your `knife.rb` file to specify a "default" region. The entry will look as follows:



```
knife[:region] = "us-west-2"
```

With a default region defined in your `knife.rb` file, you can skip the `-r` parameter and your knife ec2 plugin will use the region specified in the `knife.rb` file.

If you need to temporarily access a different region, you can still override the default region by using the `-r <some other region>` parameter again.

Creating an EC2 instance

Using the Knife EC2 command to create a new EC2 instance is somewhat complicated. A "simple" version of the command might look something like this:

```
knife ec2 server create --node-name ec2-node08 \
--groups=launch-wizard-1 \
--region=us-west-2 \
--availability-zone=us-west-2a \
--image=ami-5189a661 \
--server-connect-attribute=public_ip_address \
--flavor=t2.micro \
--ssh-user=ubuntu \
--ssh-key=my_chef_service \
--identity-file=/Users/earlwaud/.aws/my_chef_service.pem \
--run-list=recipe['base']
```

There is a lot going on there, so let's go over each part of the command:

- `knife ec2 server create`: This is the syntax of the create command
- `--node-name ec2-node08`: Name to give the new instance and the Chef node
- `--groups=launch-wizard-1`: Use the pre-existing security group named launch-wizard-1
- `--region=us-west-2`: Selects the region to create instance into
- `--availability-zone=us-west-2a`: The AZ to use
- `--image=ami-5189a661`: The specific AMI image to deploy
- `--server-connect-attribute=public_ip_address`: Use the public IP for ssh
- `--flavor=t2.micro`: The size of the image to deploy
- `--ssh-user=ubuntu`: The user to use in ssh commands
- `--ssh-key=my_chef_service`: The user access key name
- `--identity-file=/Users/earlwaud/.aws/my_chef_service.pem`: This is a key file
- `--run-list=recipe['base']`: The initial run list for the new node

When you run the command, you will see that the knife ec2 server create command details the progression of the instance creation plus the node bootstrap process. That's right, the create command also bootstraps the node and converges the specified run list. Pretty handy, right?

In addition to listing the actions done to create the new instance, the server create command also provides the data needed to access the new instance, including DNS name and both the public and private IP addresses.

So what about existing EC2 instances? You can bootstrap an existing instance pretty much the same way you would bootstrap any other server. Let's take a look.

Bootstrapping an existing EC2 instance

To bootstrap an existing EC2 instance, you will issue a command much like the one used to bootstrap any other server. It might look something like this:

```
knife bootstrap 54.213.235.40 \
  --node-name my_old_ec2_node \
  --ssh-user ubuntu \
  --identity-file /Users/earlwaud/.aws/my_chef_service.pem \
  --sudo
```

There is a significant consideration here. If your Chef server is an on-premise server, you may not have access to it from your EC2 instance. This same consideration applies to the knife ec2 server create command.

Deleting EC2 instances

Eventually, you are going to want to delete some instances. You can use the knife EC2 plugin in order to handle this task. The command you use is `knife ec2 server delete --node-name <node-name> --purge`. The `--purge` option tells knife that you also want to remove the corresponding node and client from the Chef server. Here is what deleting the instance we created earlier looks like:

```
Earls-Mac:chef-repo earlwaud$ knife ec2 server delete --node-name ec2-node08 --purge
no instance id is specific, trying to retrieve it from node name
  Instance ID: i-2c9781eb
  Instance Name: ec2-node08
    Flavor: t2.micro
      Image: ami-5189a661
    Region: us-west-2
  Availability Zone: us-west-2a
  Security Groups: launch-wizard-1
    SSH Key: my_chef_service
  Root Device Type: ebs
  Public DNS Name: ec2-54-213-66-101.us-west-2.compute.amazonaws.com
  Public IP Address: 54.213.66.101
  Private DNS Name: ip-172-31-24-37.us-west-2.compute.internal
  Private IP Address: 172.31.24.37
\n
Do you really want to delete this server? (Y/N) Y
WARNING: Deleted server i-2c9781eb
WARNING: Deleted node ec2-node08
WARNING: Deleted client ec2-node08
Earls-Mac:chef-repo earlwaud$
```

References

- The link to the Chef knife EC2 plugin page is https://docs.chef.io/plugin_knife_ec2.html.
- Another excellent reference can be found on the GitHub page for the knife EC2 plugin at <https://github.com/chef/knife-ec2>.
- The link to the knife Cloud plugins at https://docs.chef.io/plugin_knife.html.
- The GitHub knife EC2 plugin link is <https://github.com/chef/knife-ec2>.
- There is an interesting blog post by Mark Birbeck on using the knife EC2 plugin to create and manage EC2 instances without a Chef server at <http://markbirbeck.com/2012/03/16/using-knife-to-launch-ec2-instances-without-a-chef-server/>.

Creating new Knife plugins

Sometimes, you will be dealing with unique situations and need to create your own knife plugin. In this section, we will review how that is accomplished.

The basic plugin file format

A custom knife plugin begins by creating a new plugin ruby file. The plugin file has to have a specific format. That format, in its simplest form, looks like this:

```
require 'chef/knife'

module ModuleName
  class SubClassName < Chef::Knife

    deps do
      require 'chef/dependency'
    end

    banner 'knife subcommand argument VALUE (options)'

    option :name_of_option,
      :short => "-l VALUE",
      :long => "--long-option-name VALUE",
      :description => "The description for the option.",
      :proc => Proc.new { code_to_run }
      :boolean => true | false
      :default => default_value

    def run
    end
  end
end
```

The essential sections of this file are:

- The required `chef/knife` directive provides the calls all knife plugins inherit from
- `ModuleName` is the name space for our plugin
- `SubClassName` is important because it defines how the knife plugin will be called
- The `deps` do block provides the list of module-specific dependencies
- The `banner` method call provides the header that is shown when the user needs help

- The `option` method call defines the optional parameters for your plugin
- The `run` definition is the method `knife` calls to execute our plugin

Once you have created the plugin's ruby file, you have the choice of saving the file to one of two places. You can save it within your chef-repo, in a location like `~/chef-repo/.chef/plugins/knife/shinnynewplugin.rb`. Otherwise, you can save the file in a location outside of any chef-repo, which would be like `~/.chef/plugins/knife/shinnynewplugin.rb`. This second option, outside of any chef-repo, is where I would recommend placing the file. Remember that the file name must have the `.rb` extension.

Example – A simpler EC2 Create Server plugin

If you are like me and don't want to type in all those parameter names and values every time you create new EC2 instances, then this example is for you. We're going to create a shortcut plugin that allows us to create EC2 instances using a predefined set of parameters, so that we don't have to do all that typing every time we want to spin up another instance. Of course, this is an exaggerated example, and if you were going to create such a shortcut plugin, you would use the `knife.rb` file to hold the predefined parameters so that it would not require a code change to update a parameter value. Hmmmm.... Anyway, here is the example code:

```
require 'chef/knife'

module EasyCreate
  class EasyCreate < Chef::Knife
    deps do
    end

    banner 'knife easy create NODENAME'

    def run
      puts "node-name #{name_args.first}"

      cmd = "knife ec2 server create \
        --node-name #{name_args.first} \
        --groups=launch-wizard-1 \
        --region=us-west-2 \
        --availability-zone=us-west-2a \
        --image=ami-5189a661 \
        --server-connect-attribute=public_ip_address \
        --flavor=t2.micro \
        --ssh-user=ubuntu \
        --ssh-key=my_chef_service \
```

```
--identity-file=/Users/earlwaud/.aws/my_chef_service.pem\  
--run-list=recipe['base']"  
  
    system( cmd )  
  end  
end  
end
```

Now when you want to create a new EC2 instance, you can issue a command such as the following:

```
knife easy create ec2-node10
```

References

For the complete custom knife plugins documentation, go to https://docs.chef.io/plugin_knife_custom.html.

Summary

In this chapter, we've seen how to quickly verify the syntax of a knife subcommand or to review at length the full wealth of the help system contents. We have looked at some of the most often used knife subcommands as well as a few that may be generally overlooked. We also took a deep dive into the knife search subcommand. Then, we shifted gears to explore some existing community knife plugins—one for Windows and one for EC2. Finally, we cut our way through the exciting topic of creating our own knife plugins. Now, with our newly sharpened knife skills, let's begin to explore the management of your infrastructure code with the use of roles, environments, and the new and exciting feature that is policies, in the next chapter.

3

Leveraging Roles, Environments, and Policies

In this chapter, the reader will learn the power of using roles and to follow the best practices to manage large-scale environments by leveraging roles, environments, organizations, and policies.

"My doctor told me I had to stop throwing intimate dinners for four unless there are three other people." – Orson Welles

Here is a list of the topics covered in this chapter:

- Using Chef roles
- Using Chef environments
- Using Chef organizations
- Understanding nested roles
- Learning precedence hierarchy
- Exploring policies

Using Chef roles

Roles provide a way of looking at our infrastructure code as a representation of the functions performed by the servers we associate with them. Said another way, a "web server" role is created and assigned to a server that performs the function of a web server.

The benefit here is functional consistency. Whether your goal is to be able to reproduce a server in the event of failure or upgrade, or to deploy multiple identical servers to load balance a function, using a role will allow you to achieve that functional consistency.

A role's definition can be anywhere between very general, such as "Base" role, to very specific, such as "App-A Inventory Backup PostgreSQL DB Server". It is a common practice to use a role to group functionality that goes together, like all the recipes that should be run on every node grouped in a "Base" role.

Sometimes, servers will be assigned a single role, such as "UI Server". Other times, a server will be assigned more than one role. For example, when you want to deploy a server as a 3-tier all in one, that server might have the roles of "UI Server", "DB Server", and "App Server" assigned to it. This distinction can be one of environment. For example, in a development environment, you may deploy a server as an all in one, with all three roles assigned to it, but in production, you deploy each role to a separate server or multiple servers, as load necessitates.

How to create a role

There are two ways to create a new Chef role. The first way to create a role is to use a Knife create command that will allow you to edit a role template and then transfer that edited template to the Chef server directly, creating the role (without creating a saved copy of the role on your workstation). The format of the create command is:

```
knife role create <role_name> [options]
```

For example to create a "base" role using default options, you would issue a command like this:

```
knife role create base
```

That command will launch your editor with a template that contains the various parts of a role, in JSON data format. Here is an example of using the Knife command to create the "base" role:

```
roles — ubuntu@ip-172-31-17-154: ~ — subl — 103x64
Earls-Mac:chef-repo earlwaud$ cd roles
Earls-Mac:roles earlwaud$ knife role create base
[]

knife-edit-20151206-2606-jeeee6.json
knife-edit-20151206-2606-jeeee6.json ×
1  {
2    "name": "base",
3    "description": "",
4    "json_class": "Chef::Role",
5    "default_attributes": {
6      },
7      "override_attributes": {
8        },
9        "chef_type": "role",
10       "run_list": [
11         ],
12         "env_run_lists": {
13           }
14       }
15     }
16   }
17 }
18 }
```

Of course, you will need to populate the various parts of the role, such as the description and the run_list. Once you have added the content desired for your role, you will save and then exit the editor. This will allow Knife to upload your new role directly to the Chef server. Similarly, you can edit an existing role by using the Knife command in this format:

```
knife role edit <role_name> [options]
```

For example, to edit the "base" role using default options, you can issue the following command:

```
knife role edit base
```

When creating or editing roles this way, you will get a confirmation message that indicates success, such as "Created role[base]" or "Saved role[base]".

The second way, which is the way I would recommend, is to first create a role file on your workstation and then use Knife to upload the role from that file to the Chef server. A role file can be created in one of two formats. One is a JSON data file that is in the same form as the template used in the `knife role create` command. The other is to create the role file as a ruby file that contains domain-specific language. Either way, the file contains the definition of the role it represents. The standard practice is to have the name of the role file equal to the name of the role and to save the role file in the `repo/roles` folder. For example, a ruby-formatted role file for the "Web Server" role would be saved as `~/chef-repo/roles/webserver.rb`.



This is the best practice way to create roles because you can save the role file in your git repo, like the other aspects of your infrastructure code, and capture the history of the changes made to the role over time.

Once the role file is created, it is then uploaded to the Chef server with an upload command that takes the following form:

```
knife role from file <path to the role file> [options]
```

For example, to update the webserver role, you would have a command like this:

```
knife role from file roles/webserver.rb
```

When all goes well, you will get a message back that says "Updated Role webserver!".

Using per-environment run-lists in roles

You may have noted that the section for the `env_run_lists` in the JSON data role template is shown in the previous section. That section allows you to create different run lists that get used when the role is applied to the various environments in your organization, such as development and production. Per environment run lists can provide a powerful way to deploy server functionality that differs between the stages of the development process.

Per environment run lists in roles can be defined for any number of environments, but there can only be one run list per environment. When a role that has env_run_lists defined is applied to an environment not defined in the role, the default run list, defined in the run_list section of the role, is applied to the server. Similarly, when an environment defined in the env_run_list section has an empty run list, the default run list defined in run_list will be used. Here is a sample role that uses env_run_lists:

```
name 'big_app'
description 'Big App server role'
run_list 'role[base]'
env_run_lists(
  'production' => ['role[base] \
                      recipe[app_insall] \
                      recipe[prod_monitoring]'],
  'development' => ['role[base] \
                      recipe:mysql \
                      recipe[app_install] \
                      recipe[load_test_data]'],
  'stage' => [],
  '_default' => []
)
override_attributes(
  'base_name' => 'primary',
  'user_name' => 'prod_svc_acct'
)
```

You can see that if the sample role is applied to a server in either the production or the development environments, there are specifically defined run lists that will be used. It is also anticipated that if this role is applied to a server in the stage environment or the _default environment, the default run_list will be used. Also, note that if the role is applied to a server in an environment not listed in the role file, say the "test" environment, then the default run_list will be used. This is exactly what happens when there are no values in the role file for the env_run_lists, that is, all servers that have this role get the default run_list regardless of what environment they are in.

Using nested roles

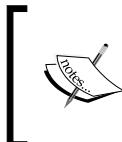
Having recipes grouped together into a role is very useful, and extending this idea by adding or nesting roles into the `run_list` of other roles makes for a powerful way to organize and define the functionality of a role or node. For example, if you have created a base role that includes all the recipes that should be applied to every node, it is much easier to add all of that functionality to your web server role by including the base role in its `run_list` before the other recipes that define a webserver. The resulting `run_list` is also much easier to understand. For example, a simple web server role might look like this:

```
name 'webserver'
description 'Simple Web Server'
run_list 'role[base] recipe[apache]'
default_attributes(
  'apache' => {
    'sites' => {
      'customers' => {
        'port' => 8080
      },
      'admins' => {
        'port' => 8081
      }
    }
  }
)
```

It is very clear that the web server role will always have the recipes that are part of the base role applied to the server before the apache recipe is applied.

The way this works is that during a chef-client run, the `run_list` is expanded. In the case of the preceding web server role, the recipes (and possibly roles) found in the "base" role will be evaluated first. You can think of it like this: if the base role has two recipes, say, `recipe[motd]` and `recipe[ntp]`, then the resulting expanded `run_list` would look like this:

```
run_list 'recipe[motd] recipe[ntp] recipe[apache]'
```



It is important to remember that everything is expanded in the order it is listed. If the base role had another role in between the `motd` and `ntp` recipes, the resulting expanded `run_list` would have the recipes from that role in between the `motd` and `ntp` recipes.

There are some important considerations here. The first is that during a chef-client run, a recipe will only be applied to a server once, the first time it appears in the expanded run_list. That is why the order in which roles are expanded is an important consideration. Next, this order or sequence is important to the order of the attributes. It contributes to the determination of attribute precedence. We will talk more about the precedence hierarchy later in this chapter. Finally, you will want to review the run_lists containing roles when we discuss policies, because currently, policies don't support roles in the run_list.

There is a downside to roles

Roles provide a lot of benefits to our infrastructure code, and until recently, they were the best way to encapsulate the cookbooks and related data into a single construct to be applied to our servers to define their functional purpose. But they have a major shortcoming. That shortcoming is the absence of any versioning capability.

What this means to you is that any changes to a role used in servers deployed to your environment will always be applied to those servers during the next chef-client run. There is no way to use a single role and define something like "use DB Server role version 1.0 in production and DB Server role version 1.1 in development". When the dev team updates the DB Server role for their ongoing development, that updated role is now the de facto standard for DB servers across all servers that the role is assigned to.

The Chef team knows that this is a shortcoming and has recently provided a solution. That solution is policies, and we will explore policies later in this chapter.

References

- The Chef.io documentation for roles can be found at <https://docs.chef.io/roles.html>.
- There is a nice tutorial on using roles and environments by Justin Ellingwood that can be found at <https://www.digitalocean.com/community/tutorials/how-to-use-roles-and-environments-in-chef-to-control-server-configurations>.

Using Chef environments

If your code development cycle goes through phases, such as development, test, stage, preproduction, and production, then you should also have servers that represent those same phases of development. Environments allow us to reproduce the workflow of the development life cycle by assigning servers to the different phases of that life cycle. Environments offer a way to designate a server as being within a specific part of the development process, such as test or production. Each server can be in one and only one environment. Put another way, a server cannot be in both development and production at the same time. The obvious benefit of using environments is that changes made to one environment will not impact other environments.

The `_default` environment

Every Chef server will have an environment created automatically when the server is deployed. That environment is named "`_default`". It is a permanent environment that cannot be renamed, modified, or deleted. Any node that is created on the Chef server will automatically be assigned to the `_default` environment unless another environment is specifically used.

Creating environments

The creation of an environment is done in much the same way as the creation of a role. There are two methods when using Knife—creating the environment using a `knife environment create` command—or creating an environment file in your chef repo and then uploading that file to create the environment using the `knife environment from file` command. Just like with roles, I recommend that you always create environment files in your chef repo and then upload those files to the Chef server to create the environment. This allows the environment to be checked into your source code control system (that is, Git) and have a history of all the changes made to the environment.

Again, like role files, the environment file can be created in one of two formats: either the JSON data format, or the Ruby DSL format. Here is a sample environment file for a development environment created using the Ruby format:

```
name 'test'
description 'The test environment for the bigapp project'
cookbook_versions(
  'bigapp' => '>= 2.1',
  'testdata' => '= 2.1.1'
)
override_attributes(
```

```
'bigapp' => {
  'listen' => ['8080', '4443']
},
'testdata' => {
  'testset' => 'customer'
}
)
```

You create environment files in a folder named environments off your `chef-repo` folder. For example, the test environment file would be saved to `~/chef-repo/environments/test.rb`. The best practice is to name the file using the name of the environment. Once you've created the environment file, you would upload that file to the Chef Server with a command in the following format:

```
knife environment from file <path to environment file> [options]
```

For example, to upload the test environment, you would use the following command:

```
knife environment from file environments/test.rb
```

A successful upload will result in a response such as `Updated Environment test.` Note that changes that you make to the environment file are uploaded using the same command used to create the environment, and the resulting success message will also be the same.

Use the `-a` or `--all` parameter to upload all environments or roles at once.



Both the Knife environment from a file and the knife role from a file can be used to upload all of the environments or roles at once if you supply a folder instead of a file name and use the `-a` or `--all` parameter. For example, `knife environment from file environments/ -a` would upload all environment files in the environments folder.

Specifying cookbook versions in environments

Chef provides a method of specifying cookbook versions to be used within an environment via the "cookbook" or "cookbook_versions" attributes. In the sample shown in the previous section, you saw the use of the "cookbook_versions" attribute. This attribute allows you to specify the version or versions of multiple cookbooks that can be applied to nodes within the environment, in this case, the "bigapp" and "testdata" cookbooks. You can specify the version of a single cookbook by using the "cookbook" attribute instead of "cookbook_versions" (although a single cookbook could also be specified using the "cookbook_versions" attribute).

The limits or constraints that you can specify for the cookbooks used in an environment are the standard operators that you are probably already familiar with, such as "`=`" (exactly this version), "`>=`" (this version or any greater version), "`~>`" (this version or any greater version that does not exceed the specified version number element), and others. If a version constraint is not specified for a cookbook in an environment, it is assumed that any version of that cookbook is acceptable for nodes within that environment.

References

- The Chef.io documentation for roles can be found at <https://docs.chef.io/environments.html>.
- A complete description of the cookbook version constraints can be found at https://docs.chef.io/cookbook_versions.html.

When you want to move nodes from one environment to another, you can use the Knife Exec command. Read more about that in the Knife subcommands section found in *Chapter 2, Knife Tooling and Plugins*.

Using Chef organizations

A single Chef Server can be configured to manage the infrastructure of multiple organizations. For example, if your company has different business units that each have their own application and infrastructure servers, they can all use the same Chef server or set of Chef servers by defining unique organizations for each business unit on those Chef servers. Organizations allow the siloing of your infrastructure code, including the cookbooks, nodes, roles, environments, and so on. Said another way, cookbooks, nodes, roles, and so on, are not shared across organizations.

A cookbook with the same name can exist in multiple organizations and have completely different content in each organization. This is the mechanism that allows for multitenancy on a Chef server. Each organization will have its own set of users. It is possible for a user to have access to multiple organizations, but such a user would have to be invited into each organization.

Creating organizations

An organization can be created on the Chef server via the `chef-server-ctl` command, as shown in the *Setting Up an On-premise Chef Server* section of *Chapter 1, Setting Up a Development Environment on Your Workstation*. For reference, the format of that command goes like this:

```
chef-server-ctl org-create short_name "full_organization_name"  
--association_user user_name --filename ORGANIZATION-validator.pem
```

The initial (default) organization will always be created with this command. Additional organizations can be created using this command or through the Chef UI when the Chef Management Console has been installed on the Chef server (also shown in *Chapter 1, Setting Up a Development Environment on Your Workstation*).

Access control

Chef provides access control through a role-based system that uses organizations, groups, and users. The organization is the top-level entity for controlling access. The organization allows for access to the Chef server itself. Users need to be invited into an organization, and they have to accept that invitation to access the Chef server content belonging to that organization.

Each organization will contain a set of groups, some default, for example, "admins" and "users," and optionally some custom groups. Groups control access to object types and objects within an organization. Depending on the need, custom groups can be created within an organization to further fine tune the access control within that organization.

A default user is created during the initial setup of the Chef server, which is automatically added to the admins group. In addition, the Chef server can have any number of other users, all of which will belong to one or more organizations.

References

- The link to the Chef.io reference on organization and groups is https://docs.chef.io/server_orgs.html.
- There is a Knife plugin that can be useful when it is necessary to work with multiple organizations or multiple Chef servers. It is named Knife Block, and you can find it at <https://github.com/knife-block/knife-block>.
- The information on the Chef Management Console is at https://docs.chef.io/ctl_manage.html.

Learning the attribute precedence hierarchy

With roles, environments, cookbooks, recipes, and so on, there are many places that attributes can be defined. In addition, there are many types of attributes that can be used. If, as a slight exaggeration, the same attribute is defined in all possible resources, each with a different value, how do you know which value would be applied when chef-client converges the node? Consider a more real-world example if you use an application cookbook that has a default attribute defined for the install directory, and you need that app to install into a different directory, what type of attribute can you use, and where should you set it so that your value would be used instead of the default? The answer to these questions is found in the precedence hierarchy. The factors that determine precedence include the type of attribute, the source of the attribute, and of course, the order evaluated. Let's take a closer look at these factors now.

Understanding attribute precedence factors

There are six types of attributes. Each one represents a different level of precedence. The six types, in order of precedence, are: "default", "force_default", "normal", "override", "force_override", and "automatic."

When developing cookbooks, it is the best practice to always provide a default value for the attributes defined in your cookbook. And it is the best practice to use the type "default" when assigning those values. The reason you should use the "default" attribute type is that "default" attributes are at the lowest level of precedence in the hierarchy, and they are always reset at the start of a chef-client run. By using the "default" type of attribute, you allow users of your cookbooks to supply role- and environment-specific values for the attributes using the same "default" type and still supersede your defaults with theirs. And when a developer decides to create a wrapper cookbook, having your cookbook attributes defined with the "default" type makes it easy for the developer to assign new attribute values that will be used instead of yours.



Remember that the "default" attribute type is the best choice for your cookbook's default attribute values.

Again, the "default" attribute type is the type with the lowest level of precedence. The attribute type that is the next level up in the precedence hierarchy is the "force_default". When you use the attribute type force_default in your cookbooks, you are stating that the attribute values set should take precedence over the "default" attribute values set in any roles or environments.

Next up in the precedence hierarchy is the "normal" type of attribute, sometimes referred to as the "set" type. Normal attribute values have a higher precedence than both default and force_default attributes. In addition, the normal type attribute values are preserved on the node between chef-client runs. This is unlike the "default" type attributes, which are reset on every chef-client run.

The next higher level of attribute is the "override" attribute. While override attributes can be set in roles or environments, typically if necessary, you will use override attributes directly in a recipe. There is also a "force_override" attribute type that has a higher level of precedence than an override attribute. A force_override attribute would be used to make sure that the attribute value set in a cookbook was used instead of any defined in a role or environment.

Finally, at the highest precedence level, we have the "automatic" type attributes. The automatic type of attribute is set by Ohai on the node and always has the highest level of precedence. Examples of automatic attributes would be things specific to the node, like CPU speed or amount of RAM. Automatic attributes cannot be modified during a chef-client run.



With all of these attribute types available, it can be challenging to know which type to use. I have found that it is usually best to limit your use of attribute types to the default type. There will be situations when you will need to use the other types, but if most of the attribute values you create are type default, you will be successful.

Attribute values can come from a variety of sources. These include nodes, cookbooks (in attribute files), recipes, environments, and roles. These sources, combined with the attribute types listed earlier, generate 15 possible levels of precedence.

Resulting precedence hierarchy

So what does all this mean? When chef-client converges the attributes in a run, the attributes collected from all the various sources are evaluated to determine which value is actually used when applying the recipes. Since there are 15 levels of precedence, there is a mapping of those levels. Chef.io has provided a very nice table to be able to understand the order of precedence. The higher the number in this table, the higher the level of precedence. Said another way, if an attribute is assigned a value in more than one place, the place with the highest number in this table is the value that will be used:

	Attribute Files	Node/Recipe	Environment	Role
default	1	2	3	4
force_default	5	6		
normal	7	8		
override	9	10	12	11
force_override	13	14		
automatic			15	

Let's consider a specific example. You have a `default.rb` file in your apache cookbook/attributes/ folder that contains the following default attribute values:

```
default['apache']['dir'] = '/etc/apache2'  
default['apache']['listen_ports'] = [ '80', '443' ]
```

Your organization requires that all web servers get installed into /var/www folder and that they use the ports 8080 and 4443 instead of the standard defaults.

Armed with the knowledge provided in the precedence table earlier, you can add "default" type attributes to your web server role, as follows:

```
name 'webserver'  
description 'webserver'  
run_list 'role[base] recipe[apache]'  
default_attributes(  
  'apache' => {  
    'dir' => '/var/www',  
    'listen_ports' => %w(8080 4443)  
  }  
)
```

And know that your values will be used instead of the cookbook's defaults because a "default" attribute type in a role has a higher level of precedence than the same default type attributes defined in the cookbook's attributes file.

Exploring policies

One of the best things about cookbooks is that they can be versioned. And as you have learned, the best application of cookbooks is to combine them into roles.

Unfortunately, there is no concept of version for roles. The proper application of roles to your infrastructure obscures the desirable standard of versions.

Enter policies. Policies allow you to group all of the various resources into a high-level object that defines the specific versions that are used to create the object. Policies combine the best parts of roles, environments, and tools such as Berkshelf, into a single-versioned resource. You can apply a policy in a development environment, test it thoroughly, and promote that policy into the stage and then production environments, progressing your infrastructure code through its lifecycle in a safe, reliable way.

What exactly is a policy?

A policy is built by creating a policyfile. A policyfile is kind of like a Chef role mixed with a Berkshelf berksfile. Policyfiles are used to create artifacts that are used by the chef-client to configure a node. These artifacts are a lock file and a cookbook bundle. They are versioned and cannot be modified without changing the version, and because of this, your chef-client runs become 100 percent reproducible.

A policyfile contains the policy description for a node. They are written using Ruby DSL and contain the `run_list` of recipes, where to get the recipe's cookbooks (including the version of the cookbooks that are valid), and the attributes needed for the recipes. Using ChefDK's `chef generate` command with the policyfile, you create a policy lock file and cache the required cookbooks. The policy and cookbooks can then be uploaded to the Chef server or export the policy and cookbooks into a directory to be used with chef-client or chef solo.

Caching the cookbooks provides several benefits. For example, each cookbook becomes a unique artifact, including each version of the cookbook used. Each of these artifacts can be tracked in a source control. This provides a guarantee that policies will be consistently applied to nodes. The `chef generate` command creates a policy lock file that includes a checksum. Any change in any of the cookbooks used causes a change in the checksum value, which prevents any unexpected cookbook changes being applied to your nodes. This allows a level of certainty that you can deploy a policy to a node across the full development life cycle with 100 percent consistency.

Why would I use a policy?

Although the use of roles can be very effective, there are still significant challenges that are not addressed very well by them, the biggest of the challenges being that roles don't have versions. Another big challenge is that the effort of dependency resolution for cookbooks requires the use of an external tool to work effectively. For most Chef users, that tool is Berkshelf. Chef Environments fills in some of the version deficiency, but the combination of roles, environments, and dependency resolvers like Berkshelf quickly become a source of confusion and potential mistakes. To be clear, I am not saying that there is anything wrong with using Berkshelf. Frankly, until now, it was a must-use tool. But with the introduction of Chef Policy, Berkshelf is no longer needed. The Policyfile includes all the necessary details to resolve all cookbook dependencies.

Another reason that you would want to use Chef Policy is the attribute precedence hierarchy that was covered in a previous section. With the use of Chef Policy, the 15 levels of precedence are effectively reduced by about seven. This removes nearly half of the attribute precedence complexity. This alone can be a compelling reason to switch from roles to policy.

How do I use a policy?

Aha! The part you have been waiting for: how to create and use Chef Policy. First, you will create a Policyfile that describes the desired node state. Let's begin by showing a very simplified example that uses the "example" cookbook that is generated with a `chef generate repo` command (which we explored in *Chapter 1, Setting Up a Development Environment on Your Workstation*). Here is our policyfile:

```
name 'cookbook-dir'  
default_source :chef_repo, File.dirname(__FILE__)  
  
run_list 'example'
```

To create the policy lock file from a Policyfile, execute a command in the following format:

```
chef install <path to policy file> [options]
```

For example, here is what it looked like when I created our policy lock file for "cookbook-dir":

```
chef install cookbook-dir.rb  
Building policy cookbook-dir  
Expanded run list: recipe[example]  
Caching Cookbooks...
```

Installing example 1.0.0

```
Lockfile written to /Users/earlwaud/chef-repo/cookbook-dir.lock.json
```

Let's take a look at the generated policy lock file. Inside the `cookbook-dir.lock.json` file, you will find quite a bit of specific information about our policy. One bit of data to notice is a key "revision_id". Look at the value that was created for it. Here is the one from my lock file:

```
"revision_id":  
"66d367a388f645c251b15db7586830a36be2aadaf9f2aaa96065b1691d0d5995"
```

This is the unique identifier for this lock file and serves as its version number. You can see the name of the policy in the "name" field. You see the run_list that was specified in the policyfile.



Note that it expanded the recipe to identify the specific cookbook recipe that is used, in this case "default."



Next, you see the specifics about the cookbook captured in this lock file. There is the friendly version number and then a unique identifier. Even if a developer makes changes to this cookbook without updating the friendly version number, the identifier version generated will change, thus providing a new "version number" for the lock file. This makes sure that you get the exact version of the cookbook you intended when you apply the policy to your nodes. There's more data in the lock file, mostly related to attributes. Note that if the values for the attributes change, so will the lock file identifier.

When changes are made to the underlying cookbooks, and a new version of the policy is desired, you issue the command to update the policy in the following format:

```
chef update <path to policy file> [options]
```

For example, after changes were made to the "examples" recipe, I updated the policy lock file with the chef update command. Here is the command I used:

```
chef update cookbook-dir.rb
```

The policy lock file is updated, and it contains a new `revision_id` and a new cookbook identifier (even though I didn't update the cookbook-friendly version number from 1.0.0). Here is the new "revision_id" from the updated lock file for comparison with the higher version:

```
"revision_id":  
"f80ae69ae021e0b577a8c0b9dd23f4bb5ea4b91c66ce921cf868721b9a01cafaf"
```



You still need to update the friendly version number

Even though the policy lock file has its own unique version number, it is still going to be the best practice to update the friendly version number in the metadata of your cookbooks. The version number will always play an important role for cookbook versions when collaborating with others or publishing cookbooks to the supermarket.

Another example

So now, let's walk through the workflow of creating and using a new policy. Begin by changing into your home or preferred working directory. Then, execute the following commands:

```
chef generate repo policyfile-book
cd policyfile-book/
git init
git add .
git commit -m 'Initial repo commit'
chef generate policyfile
```

So far, you have created a new repo, initialized git, and then added and committed all the files in the new repo. Finally, you generated a new policyfile template. Since you didn't specify a name for the policy file, you created a template file named `Policyfile.rb`. Here is what that policy file template looks like:

```
cat Policyfile.rb
# Policyfile.rb - Describe how you want Chef to build your system.
#
# For more information on the Policyfile feature, visit
# https://github.com/opscode/chef-dk/blob/master/POLICYFILE_README.md

# A name that describes what the system you're building with Chef does.
name "example-application-service"
```

```
# Where to find external cookbooks:
default_source :supermarket

# run_list: chef-client will run these recipes in the order specified.
run_list "example_cookbook::default"

# Specify a custom source for a single cookbook:
# cookbook "example_cookbook", path: "../cookbooks/example_cookbook"
```

Now you will want to make changes to the policy file template to turn it into the desired policy file. I've edited my `Policyfile.rb` file, making the simple changes resulting in the contents shown here:

```
cat Policyfile.rb
name "policyfile-book"
default_source :supermarket
run_list "motd", 'ntp'
  le_cookbook", path: "../cookbooks/example_cookbook"
```

Specifically, I changed the name of the policy to `policyfile-book`. I set the `run_list` to include two recipes, `motd` and `ntp`.



It is important to note that the `run_list` of a policy can only use recipes. You cannot specify a role, or another policy. These options may be available in future releases, but for now, it is only recipes in the `run_list` of your policies.

Next we need to generate our policy lock file and cache our desired cookbooks. We do that by issuing the `chef install` command. It will look something like this:

```
chef install
Building policy policyfile-book
Expanded run list: recipe[motd], recipe[ntp]
Caching Cookbooks...
Installing motd          0.6.3
Installing ntp            1.8.6
Installing chef_handler   1.2.0

Lockfile written to /Users/earlwaud/policyfile-book/Policyfile.lock.json
```

Note that in my example, there is a dependency on the `chef_handler` cookbook, and it was automatically dealt with, and that cookbook was also cached. So, if the cookbooks are cached, where do they go? Well, if you don't specify otherwise, they go into a cache folder off your home directory. For me, that is `~/.chefdk/cache/cookbooks/`. Here is what you will find if you look in that folder, at this point in the example:

```
ls -la ~/.chefdk/cache/cookbooks/
total 0
drwxr-xr-x  7 earlwaud  staff  238 Dec 13 23:07 .
drwxr-xr-x  5 earlwaud  staff  170 Dec 13 22:42 ..
drwxr-xr-x 11 earlwaud  staff  374 Dec 13 23:07 chef_handler-1.2.0-
supermarket.chef.io
drwxr-xr-x 22 earlwaud  staff  748 Dec 13 23:07 motd-0.6.3-supermarket.
chef.io
drwxr-xr-x 10 earlwaud  staff  340 Dec 13 23:07 ntp-1.8.6-supermarket.
chef.io
```



Note that the specific versions have been identified in the name of the cookbooks.



Next up, we are going to upload our policy and the related files to our Chef server. Remember that we just created a new repo, and it will be necessary to configure that repo to communicate with our Chef server before we can upload the policy. I took care of that by using a shortcut. I copied the contents of the `.chef` folder from another repo into this new one. Here is what that looks like:

```
mkdir .chef
cp -r ../chef-repo/.chef/* ./chef/
```

Okay, so now we are ready to upload our policy. We upload the policy and its related files with the `chef push` command. You will provide a name for this policy so that it can be identified for use later. In my case, I am using the name "book". Here is what that looks like:

```
chef push book
Uploading policy to policy group book
Uploaded vim 2.0.0 (8f81a5b1)
Uploaded zsh 1.0.3 (cb687099)
```

Alternatively, you can export the policy and related files to a local folder or to a tar file using the `chef export` command. Here is what using those commands looks like, first the export to a folder:

```
chef export chef-repo
Exported policy 'policyfile-book' to chef-repo

ls -la chef-repo/
total 16
drwxr-xr-x  6 earlwaud  staff   204 Dec 13 22:54 .
drwxr-xr-x 15 earlwaud  staff   510 Dec 13 22:54 ..
-rw-r--r--  1 earlwaud  staff  1566 Dec 13 22:54 Policyfile.lock.json
-rw-r--r--  1 earlwaud  staff   396 Dec 13 22:54 client.rb
drwxr-xr-x  4 earlwaud  staff   136 Dec 13 22:54 cookbooks
drwxr-xr-x  3 earlwaud  staff   102 Dec 13 22:54 data_bags
```

Or exporting to create a tar file:

```
chef export -a chef-repo
Exported policy 'policyfile-book' to /Users/earlwaud/policyfile-book/
chef-repo/policyfile-book-c465f6433cc209d0fd8ba9196a9e90beaccb3ab88161556
8f2041aa433cb54c5.tgz
```

Now you have two artifacts you can use to configure your "Book" nodes. Using these exact same artifacts, first in dev, then in test, and finally in production, assures that the configuration of the servers in each stage of your lifecycle is 100 percent consistent.

How to specify the source options for cookbooks

Since you can specify the source you want to obtain cookbooks from in your Policyfile, you need to know the options you have for those sources. Here is a list of sources, with examples:

```
# cookbooks with version constraint
cookbook 'yum', '= 3.6.1'
cookbook 'yum', '~> 3.6.0'

# cookbooks from a local path
cookbook 'yum', '/Users/earlwaud/chef-repo/cookbooks/yum'
```

```
# cookbooks from a git repository
cookbook 'yum', git: 'https://github.com/chef-cookbooks/yum'

# cookbooks from a github repository
cookbook 'yum', github: 'chef-cookbooks/yum'

# cookbooks from a github repo with a release tag
cookbook 'yum', github: 'chef-cookbooks/yum', tag: '3.7.0'
```

Also, there are options for what you want to use as a default source for when you don't specify a cookbook source. The default is defined in the `default_source` value, and here are the options you have, with examples:

```
# use the chef supermarket
default_source :supermarket
default_source :community

# use a local or private supermarket
default_source :supermarket, 'https://mysupermarket.domain.com'

# use the cookbooks in the current working directory
default_source :chef_repo, FILE.dirname(__FILE__)

# use the cookbooks in another filesystem location
default_source :chef_repo, '/home/src/other-chef-repo'
```

:community = :supermarket

In early versions of the policy generator, the `default_source` for cookbooks was set to "`:community`". Later, this was updated to "`:supermarket`". If you see `:community` used in policy files, know that it is the same as using `:supermarket`.

What are the "Gotchas"?

The number one consideration before using Chef Policy is that at the time of this writing, it is just now "nearing" 1.0 status. With the latest Chef release, all of the major features are available, but there is still significant development in flight that will make policies better in the near future.

If you want to know the specific limitations in the current release, here is my list:

- Must be using the latest released versions of Chef, ChefDK, and Chef Server
- There is no support for Chef roles

- You cannot "nest" or "include" other policies like you can with Chef roles
- Using Chef zero, which is a local in-memory Chef Server, requires compatibility mode
- Not yet officially branded as version 1.0 by Chef.io
- And my #1 limitation... Not yet implemented for use with Knife bootstrap

References

- The (limited) documentation about policies on the Chef.io site can be found at <https://docs.chef.io/policy.html>. Hopefully, it will be expanded by the time you visit it.
- You will probably want to check out the policyfile documentation on GitHub found at https://github.com/chef/chef-dk/blob/master/POLICYFILE_README.md.
- You will find the chef-zero repo on GitHub at <https://github.com/chef/chef-zero>.
- The link to the Chef.io blog post titled "Policyfiles: A Guided Tour" at <https://www.chef.io/blog/2015/08/18/policyfiles-a-guided-tour>.
- There is an excellent Chef.io webinar recording on the why, what, and how of Chef policies, that can be watched at https://www.chef.io/webinars/?commid=171697&utm_campaign=add-to-calendar&utm_medium=calendar&utm_source=brighttalk-transact.

Summary

In this chapter, you learned about Chef roles and how valuable they are to deploying fully functional servers. We saw that even though they are a global resource, they can be leveraged across environments to deliver different content, based on the environment of the node. We also discussed the big shortcoming of roles, that being their lack of a version component. Then, we looked at using Chef Environments to mimic the development life cycle of your infrastructure code. That was followed by talking about how to provide multitenancy with your Chef servers using organizations. Next, we took a deep dive into the attribute precedence hierarchy, and then we finished up with a visit to the exciting new policy feature, which delivers an object that combines the best of roles and environments (and dependency tools such as Berkshelf too).

In our next chapter, we will investigate custom resources, explaining how to create and use them effectively. Are you ready? Then read on...

4

Custom Resources

In this chapter, we're going to discuss the exciting changes released in Chef 12.5 that transform the old LWRPs and HWRPs into the much easier-to-create and use Custom Resources. We'll start with a look at the differences between the old and new, but by the end of the chapter, the reader will know how to build and use Custom Resources.

"To me, my recipes are priceless." – Colonel Sanders

Here is a list of the topics covered in this chapter:

- Out with the old (LWRPs), in with the new (Custom Resources)
- Creating an example use case
- Creating and using a Custom Resource
- Making a Custom Resource more reusable
- Subclassing a Custom Resource

Out with the old (LWRPs), in with the new (Custom Resources)

When Chef was created, the Chef developers delivered a tool that had a lot of capabilities right out of the box. The variety of resources that it supported natively was very useful and quite extensive. But, of course, being a finite tool, it could not handle every possible type of resource that existed or could exist in the future. Knowing this, the Chef developers built in a feature set to allow users to augment the capabilities and support an ever-growing set of resources.

This feature set is known as **Lightweight Resource Providers (LWRP)** and **Heavyweight Resource Providers (HWRP)**. These features allow users to define new resources and create the providers to support those resources, making Chef infinitely customizable.

The two types, LWRP and HWRP, are differentiated by the level of complexity used to define the resources and the providers being created. LWRPs are created using the **Chef DSL (Domain-specific Language)** and are capable of handling most customization needs. HWRP are created using pure Ruby. HWRPs are the go-to feature when the requirements are too complex to be handled by straight Chef DSL.

Unfortunately, creating and using LWRPs and HWRPs is more challenging than most of the other features of Chef. This is because the Chef developers created Custom Resources. Custom Resources provide the same feature set as LWRP and HWRP, but do it in a way that is much easier to create and use. None of the capabilities are lost in translation either. A Custom Resource can easily handle the task of providing new resources using the Chef DSL, but it can also handle the times when you need to support more complex scenarios that require using pure Ruby.

What is the difference between the old way and the new way of defining resources?

As of version 12.5, LWRPs and HWRPs are no longer used. All existing code that creates and uses them will still work because version 12.5 is backward compatible with cookbooks from previous versions, but new resources should now be created only as Custom Resources.

LWRP and HWRP required you to create both the Resource and the Provider. With Custom Resources, it is only necessary to create the Resource. There is no need to create a folder named providers in the cookbook when creating Custom Resources.

Creating and using Custom Resources is much easier than it was to create and use LWRPs and HWRPs.

Custom Resources use "properties". LWRPs and HWRPs used attributes, which was easily confused with Node attributes.

It is much easier to create OS-specific resources by subclassing a base class, and we will take a look at doing that later in the chapter.



Although Custom Resources are not deprecating "Definitions", it is recommended that going forward, you should create Custom Resources instead.

There are many other differences between the old Resource Provider model and the Custom Resource model introduced in version 12.5, but what is listed earlier should provide reason enough to embrace the change.

What is a Custom Resource?

A Custom Resource is an extension to Chef. It is a way to create and use new resources that were not provided by Chef. A Custom Resource is created as part of a cookbook. A Custom Resource can be created with Chef DSL and can use built-in resources, or you can create a Custom Resource using Ruby if the need is complex enough. A Custom Resource provides the mechanism to create and use resources that are not built in to Chef. For example, Chef has built-in resources for things, such as directories, files, and services. However, there is no built-in resource for websites. So, as a way to demonstrate Custom Resources, we will create a website Custom Resource later in this chapter.

References

- The link to Chef's new Custom Resources information is
https://docs.chef.io/custom_resources.html.
- The link to the Chef page for Custom Resources prior to version 12.5 is
https://docs.chef.io/release/12-4/custom_resources.html. It describes LWRP and HWRP.
- The Chef documentation for (almost deprecated) definitions is at
<https://docs.chef.io/definitions.html>.
- There is an excellent book that really details the creation and use of LWRPs and HWRPs. The book is titled *Customizing Chef* and is by *Jon Cowie*. You can check out a sample at https://books.google.com/books/about/Customizing_Chef.html?id=XOBdBAAQBAJ&printsec=frontcover&source=kp_read_button&hl=en#v=onepage&q&f=false.

Creating an example use case

In order to understand the creation and use of Custom Resources, let's create a somewhat contrived example use case. In order to avoid getting lost in the details, we'll keep the example simple, using just enough detail so that you can see the power of Custom Resources.

What is the goal?

We have been asked to create a new cookbook to support a brand new web application. The application is going to use several different websites such as a "Users" site, a "Suppliers" site, and a "Customers" site. The developers are saying that there will be several other sites needed in the future, but they want to get started now with these three.

Based on this short user story, we are going to create a new cookbook named "mywebapp", and it is going to create the beginnings of three websites: "Users", "Suppliers", and "Customers". We should keep in mind that these three sites are only the beginning, and our cookbook should be written to make adding more websites easy. We are going to begin with the non-custom resource model, so let's get started.

Creating the cookbook

As described in *Chapter 1, Setting Up a Development Environment on Your Workstation*, we will use the chef command-line tool to create our cookbook template. We do that by issuing the following commands:

```
cd ~/chef-repo/cookbooks  
chef generate cookbook mywebapp
```

Let's take a look at what we are starting with. Go into the new cookbook folder and issue the tree command like this:

```
cd mywebapp  
tree
```

This is what you should see:

```

mywebapp — ubuntu@ip-172-31-17-154: ~ — bash — 103x43
Earls-Mac:~ earlwaud$ pwd
/Users/earlwaud
Earls-Mac:~ earlwaud$ cd ~/chef-repo/cookbooks
Earls-Mac:cookbooks earlwaud$ chef generate cookbook mywebapp >/dev/null
Earls-Mac:cookbooks earlwaud$ cd mywebapp
Earls-Mac:mywebapp earlwaud$ tree
.
├── .gitignore
├── .kitchen.yml
├── chefignore
├── metadata.rb
├── Policyfile.rb
├── README.md
├── recipes
│   └── default.rb
└── spec
    ├── .gitignore
    ├── .kitchen.yml
    ├── chefignore
    ├── metadata.rb
    ├── Policyfile.rb
    └── README.md

1 # 
2 # Cookbook Name:: mywebapp
3 # Recipe:: default
4 #
5 # Copyright (c) 2016 The Authors, All
6 |

```

We have a new cookbook template named `mywebapp` ready to customize to our needs.

Creating the default recipe

Now that we have our cookbook template ready to go, let's create our default recipe. Adding the following code to the `default.rb` file will give us the resources needed to create three (very simplistic) websites. The code is as follows:

```

#
# Cookbook Name:: mywebapp
# Recipe:: default
#
# Copyright (c) 2016 The Authors, All Rights Reserved.

directory '/var/www/users' do
  recursive true
  mode 0755

```

Custom Resources

```
user 'earlwaud'
group 'wheel'
action :create
end

file '/var/www/users/index.html' do
  content '<html><title>Users</title><body><h1>Hello to our Users</h1></body></html>'
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  action :nothing
end

directory '/var/www/suppliers' do
  recursive true
  mode 0755
  user 'earlwaud'
  group 'wheel'
  action :create
end

file '/var/www/suppliers/index.html' do
  content '<html><title>Suppliers</title><body><h1>Hello to our Suppliers</h1></body></html>'
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  action :nothing
end

directory '/var/www/customers' do
  recursive true
  mode 0755
  user 'earlwaud'
  group 'wheel'
```

```
action :create
end

file '/var/www/customers/index.html' do
  content '<html><title>Customers</title><body><h1>Hello to our
Customers</h1></body></html>'
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  action :nothing
end
```

Looking at the preceding code, we can see that we have repeated a "set" of resources three times, one for each of the three websites we wish to create. First, we have the directory resource that defines the location where we want to put our website files. The description includes instructions to recursively create the segments of the folder path and apply the permissions, owner, and group values to each one. Next, we have a file resource that creates an `index.html` file in the target folder, again applying the permissions, owner, and group values. Also, we are providing a very basic description of the content for the file. In this case, it is a site-specific title, and a site-specific "Hello" message for the body of the web page.

If we want to validate that what we have created so far is working, we can execute a local override chef-client run with the `-z` and `-o` parameters, as follows:

```
cd ~/chef-repo
sudo chef-client -z -o mywebapp
```

This chef-client run will generate a lot of output. Here is a subset of what you will see:

```
(truncated) ...
Synchronizing Cookbooks:
[2016-01-18T12:20:02-08:00] INFO: Storing updated cookbooks/mywebapp/
recipes/default.rb in the cache.
- mywebapp (0.1.0)
Compiling Cookbooks...
Converging 9 resources
Recipe: mywebapp::default
 * directory[/var/www/users] action create[2016-01-18T12:20:02-08:00]
INFO: Processing directory[/var/www/users] action create
(mywebapp::default line 7)
```

Custom Resources

```
[2016-01-18T12:20:02-08:00] INFO: directory[/var/www/users] created
directory /var/www/users

  - create new directory /var/www/users[2016-01-18T12:20:02-08:00]
INFO: directory[/var/www/users] owner changed to 501
[2016-01-18T12:20:02-08:00] INFO: directory[/var/www/users] group
changed to 0
[2016-01-18T12:20:02-08:00] INFO: directory[/var/www/users] mode
changed to 755

  - change mode from '' to '0755'
  - change owner from '' to 'earlwaud'
  - change group from '' to 'wheel'
* file[/var/www/users/index.html] action create[2016-01-
18T12:20:02-08:00] INFO: Processing file[/var/www/users/index.html]
action create (mywebapp::default line 15)
[2016-01-18T12:20:02-08:00] INFO: file[/var/www/users/index.html]
created file /var/www/users/index.html

  - create new file /var/www/users/index.html[2016-01-
18T12:20:02-08:00] INFO: file[/var/www/users/index.html] updated file
contents /var/www/users/index.html

  - update content in file /var/www/users/index.html from none to
23195b
    --- /var/www/users/index.html      2016-01-18 12:20:02.000000000
-0800
    +++ /var/www/users/.index.html20160118-4946-1nzzlcz 2016-01-18
12:20:02.000000000 -0800
    @@ -1 +1,2 @@
    +<html><title>Users</title><body><h1>Hello to our Users</h1></
body></html>[2016-01-18T12:20:02-08:00] INFO: file[/var/www/users/
index.html] owner changed to 501
[2016-01-18T12:20:02-08:00] INFO: file[/var/www/users/index.html]
group changed to 0
[2016-01-18T12:20:02-08:00] INFO: file[/var/www/users/index.html] mode
changed to 755
...(truncated)
```

The preceding subset of the chef-client run output is related to the set of resources creating the first website, "Users". You will note that it creates the new `/var/www/users` directory and then applies the owner, group, and permissions to it. Next, it creates the new `/var/www/users/index.html` file. Then, it sets the `index.html` file contents to the desired site content described in the file resource. Finally, it sets the owner, group, and permissions for the newly created file. All of this is as expected.

Our simple, but repetitive, recipe did the job. It created three "websites" as requested. Although that recipe worked, it does not meet all of the requirements of the developers. So what is missing?

The recipe worked, but what is missing?

Although the recipe worked and created our three websites, there is a significant request made by the developers that was not addressed. They wanted the recipe to be written so that it could be easily expanded to create many more additional websites.

Of course, we could copy the resources of one of the websites and paste them in at the bottom, editing the definitions to reflect a new website. This would work, but it is not very efficient, and it will make an already repetitive recipe even more repetitive. Also, this plan is a potential source of errors. For example, after pasting the copied resources, the person editing the recipe might forget to update one or more of the "custom" values, forgetting to change the file location, for example. What's more is that if there needs to be a change made, say changing the permissions from 755 to 644, it would be necessary to update the recipe in six places, even with just our three websites. Again, this method would be inefficient and error-prone.

So what can we do about meeting the developers' request, and at the same time, address all this inefficient repetition and potential for errors? Say hello to my little friend, "Custom Resources".

References

The link to chef-client documentation that includes the details for using local mode is https://docs.chef.io/ctl_chef_client.html#run-in-local-mode.

Creating and using a Custom Resource

Now that we have our use case defined and a working solution created, we want to introduce updates to the cookbook to turn what we have into a better solution that uses Custom Resources. Specifically, we are going to create a "website" Custom Resource.

What makes up a Custom Resource?

Custom Resources are created by making a Ruby file in the cookbook's resources folder. The resource name will be the name of the file created. For example, if we want to create a "website" Custom Resource, we would create a new Ruby file named `website.rb`, as follows:

```
mkdir -p ~/chef-repo/cookbooks/mywebapp/resources  
touch ~/chef-repo/cookbooks/mywebapp/resources/website.rb
```

This will create a new folder named `resources` and create an empty file named `website.rb` in that folder.



Creating a default.rb resource file: If you create a resource file named `default.rb`, the resource will have the same name as the cookbook.



A cookbook can have multiple resources. Each one would be defined in a separate Ruby file in the `resources` folder.

The contents of the resource file will have three basic parts:

- Declarations of the properties used in the Custom Resource
- Loading of the current value of the properties, if they already exist
- Definitions of the actions that the Custom Resource can take

A Custom Resource is much simpler to create when compared with the LWRP and HWRP.

Creating the "website" Custom Resource

We are going to "re-use" some of the code in the `default.rb` recipe to create our new "website" Custom Resource. First, let's create a new file to hold our Custom Resource:

```
mkdir ~/chef-repo/cookbooks/mywebapp/resources  
touch ~/chef-repo/cookbooks/mywebapp/resources/website.rb
```

Next, let's copy one set of the website resources from our `recipe/default.rb` file, the Users site resources, and paste them into our new `website.rb` file. Here is the section we are copying and/or pasting:

```
directory '/var/www/users' do  
  recursive true  
  mode 0755  
  user 'earlwaud'
```

```
group 'wheel'
  action :create
end

file '/var/www/users/index.html' do
  content '<html><title>Users</title><body><h1>Hello to our Users</h1></body></html>'
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  action :nothing
end
```

Next, we need to "wrap" our copied code in an action block with the format
action :start do ... end. Now our website Custom Resource looks as follows:

```
action :start do
  directory '/var/www/users' do
    recursive true
    mode 0755
    user 'earlwaud'
    group 'wheel'
    action :create
  end

  file '/var/www/users/index.html' do
    content '<html><title>Users</title><body><h1>Hello to our Users</h1></body></html>'
    mode 0755
    user 'earlwaud'
    group 'wheel'
    notifies :restart, 'service[httpd]'
  end

  service 'httpd' do
    action :nothing
  end
end
```

This code block is our action definition for our "website" Custom Resource.



When you have more than one action defined in a Custom Resource file, the first action listed will be the default action.



In our example, you can plainly see that we have some "constants" defined in our action block, and because we want this to be a highly reusable Custom Resource, this simply will not do. So, let's create some properties for our Custom Resource, which we'll use to replace these constants.

At the top of our `website.rb` file, let's add two new properties. One will be the "site name" and the other will be the "title". Add the following two lines to the `website.rb` file:

```
property :site_name, String, name_property: true
property :title, String
```

The first line is for the site name. This property is defined to be a string and includes the `name_property` setting, which says to use the value of the resource name for the value of this property.

All resources defined in Chef have a `name_property`. For example, our `directory` resource for the `Users` site has the `name_property` of `/var/www/users`.

When a recipe uses our new website resource, the name it provides will be the value that gets set for our `site_name` property. This will make more sense when you see how we use our Custom Resource in the section ahead.

Second, we are adding a `title` property, also of type string.

Once we have our properties declared, we need to do some modifications to our Custom Resource's action to use the new properties instead of using the constant values currently defined.

Let's update our `website.rb` file to use the new properties now. Once updated, our file should contain the following:

```
property :site_name, String, name_property: true
property :title, String

action :start do
  directory "/var/www/#{site_name}" do
    recursive true
    mode 0755
    user 'earlwaud'
```

```
group 'wheel'
  action :create
end

file "/var/www/#{$site_name}/index.html" do
  content "<html><title>#{$title}</title><body><h1>Hello to our
#{$title}</h1></body></html>"
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  action :nothing
end
end
```

That is all we need for this very simplified example of a Custom Resource. Let's move on to how to use our new website Custom Resource.

Using our Custom Resource

Now that we have created our "website" Custom Resource, let's put it to use. We are going to update our cookbook's recipe file. Edit the `default.rb` file we created earlier, and cut out all of the code. Then, add the following three blocks of code to the now empty file:

```
mywebapp_website 'users' do
  title 'Users'
end

mywebapp_website 'suppliers' do
  title 'Suppliers'
end

mywebapp_website 'customers' do
  title 'Customers'
end
```

These three blocks of code are using our new Custom Resource. You will note from this example that the way you use a Custom Resource is to use the name of the cookbook, an underscore, and the name of the resource. In this case, our cookbook is named "mywebapp", and our Custom Resource is named (from the file that defines it) "website". Based on this, we can use our newly defined resource as "mywebapp_website".

Our two properties are given values for each instance of our Custom Resource. First, `site_name` is given the values from the resource names, such as "users", "suppliers", and "customers". Second, the title is given its values from the `title` properties.

Using our new Custom Resource has delivered on the request made by the developers. We have a new recipe that will be easy to expand to create future websites. In addition, we have removed a lot of repetition from the first version of our cookbook, and the result is a more user-friendly cookbook that is much less error-prone and easier to understand and use. Not bad, right?

So, let's take it for a spin. Use the local override chef-client run again to see how this has changed the way our resources are converged. Issue the same commands used earlier:

```
cd ~/chef-repo  
sudo chef-client -z -o mywebapp
```

Here is what you should see in the chef-client output this time:

```
(truncated)...  
Synchronizing Cookbooks:  
[2016-01-18T14:03:42-08:00] INFO: Storing updated cookbooks/mywebapp/  
recipes/default.rb in the cache.  
[2016-01-18T14:03:42-08:00] INFO: Storing updated cookbooks/mywebapp/  
resources/website.rb in the cache.  
  - mywebapp (0.1.0)  
Compiling Cookbooks...  
Converging 3 resources  
Recipe: mywebapp::default  
  * mywebapp_website[users] action start[2016-01-18T14:03:42-08:00] INFO:  
    Processing mywebapp_website[users] action start (mywebapp::default line  
    7)  
  
    * directory[/var/www/users] action create[2016-01-18T14:03:42-08:00]  
INFO: Processing directory[/var/www/users] action create (/Users/  
earlwaud/chef-repo/.chef/local-mode-cache/cache/cookbooks/mywebapp/  
resources/website.rb line 5)  
  (up to date)  
  * file[/var/www/users/index.html] action create[2016-01-  
18T14:03:42-08:00] INFO: Processing file[/var/www/users/index.html]  
action create (/Users/earlwaud/chef-repo/.chef/local-mode-cache/cache/  
cookbooks/mywebapp/resources/website.rb line 13)  
  (up to date)
```

```
* macosx_service[httpd] action nothing[2016-01-18T14:03:42-08:00]
INFO: Processing macosx_service[httpd] action nothing (/Users/earlwaud/
chef-repo/.chef/local-mode-cache/cache/cookbooks/mywebapp/resources/
website.rb line 21)
  (skipped due to action :nothing)
    (up to date)
... (truncated)
```

As in the output sample shown earlier, I have included only the subset of the output that relates to the creation of the Users website.

 A couple of differences you should notice here are:
Only three resources are converged this time, compared to nine before. These three are our new Custom Resources. Also note that pretty much everything resulted in no action taken (up to date). This is because the new Custom Resource we created does the same work that was done (using nine resources) in our model that didn't use Custom Resources.

Both versions converge to the same state, but by using Custom Resources, we have achieved that desired state much more efficiently and in a way that is much more extensible.

References

The link to a Chef.io slide deck on creating Custom Resources is
https://docs.chef.io/decks/custom_resources.html.

Making a Custom Resource more reusable

The Custom Resource we just created, "mywebapp_website", is easily reusable within our mywebapp cookbook and could be used with other cookbooks, but what if we want to expand on its reusability? What if we want to make a cookbook that only has a Custom Resource that we want to use in several cookbooks? The answer is pretty simple. Let's walk through the steps to convert our "website" Custom Resource into a more reusable Custom Resource-only cookbook.

Making a Custom Resource-only cookbook

The Custom Resource `website.rb` file in the cookbook `mywebapp`, which we created in the previous section, is almost all we need to create our more reusable Custom Resource-only cookbook. Let's create a new cookbook named `appsite`, which will be our new reusable cookbook:

```
cd ~/chef-repo/cookbooks
chef generate cookbook appsite
```

Now delete the entire recipe folder. The best practice for reusable Custom Resource cookbooks is that they do not have a recipe folder at all. They should only include the resources folder (and maybe a libraries folder if it uses helper functions, which we will talk about in the next section). Our reusable Custom Resource cookbook will not have its own recipes, but it will only contain the `website` Custom Resource found in the resources folder from our previous example.

Reuse the code from the "mywebapp" example

Now that we have a new cookbook (with no recipe folder), let's move the resources folder from our previous example into the "appsite" cookbook. This will completely remove the resources folder and the `website.rb` file within from the "mywebapp" cookbook and place them in our new "appsite" cookbook.

You're done. This completes the work on our reusable Custom Resource-only cookbook. Wow, that was really easy, right? So how do we use it?

Using the reusable Custom Resource-only cookbook

We now have a brand new shiny cookbook that supplies a Custom Resource to create websites. Using the cookbook is pretty easy. We need to do two things to our "mywebapp" cookbook example to leverage our "appsite" cookbook.

First, we need to add a "depends" directive to our cookbook `metadata.rb` file. Making this one line change will result in a `metadata.rb` file that looks like this:

```
name 'mywebapp'
maintainer 'The Authors'
maintainer_email 'you@example.com'
license 'all_rights'
description 'Installs/Configures mywebapp'
```

```
long_description 'Installs/Configures mywebapp'  
version '0.1.0'  
depends 'appsite'
```

Adding the depends directive to our cookbook ensures that the specified cookbook will be loaded and available when our cookbook converges. If the specified cookbook, in this case "appsite", is not available for some reason, the chef-client run would fail to converge.

Secondly, we need to change the name we use when calling our Custom Resource. In the default.rb file of our "mywebapp" cookbook, change the three resource calls so that they use the new name for our Custom Resource. The resulting default.rb file will look like this:

```
appsite_website 'customers' do  
  title 'Customers'  
end  
  
appsite_website 'suppliers' do  
  title 'Suppliers'  
end  
  
appsite_website 'users' do  
  title 'Users'  
end
```

That was pretty easy too, right? When Chef.io created the Custom Resource feature, a driving goal was to make it much easier to create and use compared with LWRP and HWRP, and I think they have succeeded very well in that goal.

However, with that said, what if you need to do something a little more complex? For example, if you want to handle websites installed to different operating systems? We can do that too, and we'll take a look at doing it later in the chapter, with an example that uses subclassing in a Custom Resource. But now, let's have a quick look at handling idempotence.

Handling idempotence

In our simple example, the resources handle the check for idempotence, making sure that actions are only executed when the resulting state would be different than the existing state. But if we were doing something more complex, such as using Ruby code in our actions or making calls to APIs to affect change, then we need our code to handle the idempotence tasks. Fortunately, Chef has the helper functions "load_current_value" and "converge_if_changed" to help us with this need.

Without actually changing our action so that it truly needs to handle the idempotence, let's look at what the Custom Resource code might look like using the previously-mentioned helper functions.



I'm simplifying the contents of `index.html` to keep the example simple.



If we needed to check the state of the contents of the `index.html` file and only change it if the resulting state (or contents) would be different than it currently is, then we can update the code to look like this:

```
property :site_name, String, name_property: true
property :title, String

load_current_value do
  title IO.read("/var/www/#{site_name}/index.html")
end

action :start do
  converge_if_changed do
    directory "/var/www/#{site_name}" do
      recursive true
      mode 0755
      user 'earlwaud'
      group 'wheel'
      action :create
    end

    file "/var/www/#{site_name}/index.html" do
      content title
      mode 0755
      user 'earlwaud'
      group 'wheel'
      notifies :restart, 'service[httpd]'
    end

    service 'httpd' do
      action :nothing
    end
  end
end
```

You can see in the preceding example that we are checking whether there is a change to "title" by using the call to `load_current_value`. Then, with `converge_if_changed`, we are executing the action only if there was a change to the value of title. This is a somewhat forced example, but it should serve as a template to handling idempotence in your Custom Resources.



Remember that this type of custom test and set code is not necessary when leveraging the built-in resources provided by Chef within your Custom Resource. You will only need to add this type of functionality if your Custom Resources are using Ruby for your actions, or coding for some other, more complex scenario.

References

The link to the Chef Documentation for Custom Resource DSL with details about `converge_if_changed` and `load_current_value` is https://docs.chef.io/dsl_custom_resource.html.

Subclassing a Custom Resource

Sometimes, you need to handle more complex situations, such as supporting distinctly different operating systems, such as Linux and Windows. This can easily be handled by subclassing your Custom Resource. Let's consider a new requirement for our use case, that is, we need to extend our new `appsite` Custom Resource cookbook so that it will allow us to create websites on OS X and Linux using Apache, and on Windows using IIS.

In order to provide for subclassing of our Custom Resource, we will need to convert them into a library module, so let's start there.

Convert our resource into a library module

We begin by creating a new folder named "libraries" in our cookbook. Next, we create a new file in our libraries folder named `website.rb` and add the following contents to the file:

```
module AppsiteCookbook
  class Website < Chef::Resource
    resource_name :appsite_website

  end
end
```

This is the template used to define our library module. We are creating a new library module that will be used by calling the name "appsite_website". It is a new class that is of type `Chef::Resource`.

Next, we can copy all of the code from our `resource/website.rb` file and paste it into our new library module definition. The resulting file will then look like this:

```
module AppsiteCookbook
  class Website < Chef::Resource
    resource_name :appsite_website

    property :site_name, String, name_property: true
    property :title, String

    action :start do
      directory "/var/www/#{site_name}" do
        recursive true
        mode 0755
        user 'earlwaud'
        group 'wheel'
        action :create
      end

      file "/var/www/#{site_name}/index.html" do
        content "<html><title>#{title}</title><body><h1>Hello to our
#{$title}</h1></body></html>"
        mode 0755
        user 'earlwaud'
        group 'wheel'
        notifies :restart, 'service[httpd]'
      end

      service 'httpd' do
        action :nothing
      end
    end
  end
end
```

So far, we have only converted our Custom Resource into a library module Custom Resource. At this point, you can delete the folder named resources and then run the `override local chef-client` command again. The results will be the same state as before with no actions taken because everything is "(up to date)".

So, let's move on to the subclassing of the Custom Resource now.

Subclassing the library module Custom Resource

Now we are going to convert our library module Custom Resource into a subclassed Custom Resource. In this example, we are going to handle the scenario of deploying our websites for either Apache on Linux and OS X systems, or for IIS on Windows systems. Let's change the base class first. We are going to simplify our `libraries/website.rb` file changing the contents to the following:

```
module AppsiteCookbook
  class Website < Chef::Resource
    resource_name :appsite_website

    property :site_name, String, name_property: true
    property :title, String

    action :start do
      puts "Executing Base Class"
    end

  end
end
```

This simple base class is just going to provide the class name, provide for the properties, and execute any required base class actions – in this case, just writing out a message.

Next, let's create our first base class for the Linux-based operating systems. Create a new file in the `libraries` folder named `website_linux.rb`. Put the following code into the file:

```
module AppsiteCookbook
  class WebsiteLinux < Website
    resource_name :appsite_website_linux
    provides :appsite_website, os: [ 'linux', 'darwin' ]

    action :start do
      puts 'Installing on Linux OS'
      super()
      puts 'Executing Linux Sub-Class'
```

```
directory "/var/www/#{$site_name}" do
  recursive true
  mode 0755
  user 'earlwaud'
  group 'wheel'
  action :create
end

file "/var/www/#{$site_name}/index.html" do
  content "<html><title>#{title}</title><body><h1>Hello to our
#{title}</h1></body></html>"
  mode 0755
  user 'earlwaud'
  group 'wheel'
  notifies :restart, 'service[httpd]'
end

service 'httpd' do
  supports :restart => true
  action :nothing
end

end
end
end
```

This is the code that will get executed for the OS types "Linux" or "Darwin" (OS X).

 The key things to notice are that the class is named WebsiteLinux and is derived from our base class Website. Next, note the "provides" directive. This is what instructs chef-client to use this subclass when the OS type matches either "linux" or "darwin". The rest of the code is pretty much identical to our earlier example.

Finally, let's create the Windows subclass file for completeness. Again in the libraries folder, create a new file named website_windows.rb. Put the following code into the new file:

```
module AppsiteCookbook
  class WebsiteWindows < Website
    resource_name :appsite_website_windows
    provides :appsite_website, os: 'windows'
```

```
action :start do
  puts 'Installing on Windows OS'
  super()
  puts 'Executing Windows Sub-Class'

  directory "#{$ENV['SYSTEMDRIVE']}\\inetpub\\wwwroot\\#{site_name}" do
    recursive true
    mode 0755
    user 'earlwaud'
    group 'wheel'
    action :create
  end

  file "#{$ENV['SYSTEMDRIVE']}\\inetpub\\wwwroot\\#{site_name}\\index.html" do
    content "<html><title>#{title}</title><body><h1>Hello to our
#{title}</h1></body></html>"
    mode 0755
    user 'earlwaud'
    group 'wheel'
    notifies :restart, 'service[w3svc]'
  end

  service 'w3svc' do
    action :nothing
  end
end
end
```

Again, notice the class line—it defines the class name as `WebsiteWindows` and says that the class is derived from our `Website` base class. And like in the Linux subclass, we have a "providers" directive that instructs the chef-client to use this base class only when running on a "windows" OS.

Because all of this code is functionally equivalent to our earlier examples, executing our override local chef-client run will result in no state modifying actions taken, as everything is (up to date).

References

The link to the Chef documentation on Libraries is <https://docs.chef.io/libraries.html>.

Summary

In this chapter, we saw how the new 12.5 version of Chef has brought us the Custom Resource, and we explored how the Custom Resource has improved upon the old concept of LWRPs and HWRPs. We realized that Chef has built-in resources for a lot of things, but not for creating websites. Therefore, you learned how to create a new "website" Custom Resource. We expanded on the website resource to make it easy to use in different cookbooks, and expanded it even further so that we can use it for cookbooks that will be converged in multiple operating systems. By now, you should really appreciate the humble Custom Resource. It is a great improvement over the old Resource Provider model and can greatly simplify your recipes.

Next, let's dive into the use of Chef to provision into the traditional datacenter. There is a lot to cover, so let's get started.

5

Provisioning in the Traditional Data Center

In this chapter, we're going to pull together the concepts of the earlier chapters to deliver the software-defined data center in the traditional on-premise model. The reader will take control of their traditional data center, integrating Chef into industry-wide compute deployment and configuration environments.

"The secret of success in life is to eat what you like and let the food fight it out inside." – *Mark Twain*

The following topics will be covered in this chapter:

- VMware provisioning
- Open Stack provisioning
- Network automation

VMware provisioning

VMware is the blue 600-pound gorilla in the data center. If you have any exposure to virtualization in a corporate setting, then the odds are that you have worked extensively with VMware's product line.

Since early 2000s, VMware has had an ever-increasing presence in the data center, which started initially as a way to deliver more computing power in the same space through server consolidation. Then, VMware continued to grow as a way to provide faster server provisioning to meet the ever-increasing demands for rapid deployment.

And now, VMware is the fast, fault-tolerant, and highly available on-site data center cloud provider. Regardless of the reasons, VMware virtualization environments are here to stay in our data centers (at least in the short term).

Because of this, you will need to be able to leverage Chef to manage the configuration of all VMware virtual machines.

Since VMware deployments are such a common part of the traditional data center, I am not going to take you through vSphere environment deployment. If for some reason, you do not already have access to a vSphere environment, you can deploy a "lab in a box" using VMware workstation or fusion. There are many guides available on the Internet that can detail the requirements and steps needed to set up your lab. For example, there is a fairly complete guide on the EnterpriseDaddy website, and the link is <http://www.enterprisedaddy.com/building-a-home-lab-for-vmware-vsphere-6-0/>. You can download a PDF of their guide or review the steps in the pages of their blog.

Let's proceed now with the expectation that you have access to a vSphere environment or that you've created your own vSphere lab to work with. There are many ways to leverage Chef in the VMware-centric data center environment. Starting with just post-provisioning customization, the knife command will very effectively integrate with a VM previously provisioned in any number of ways, allowing the DevOps person to manage the configuration and application deployment on their VMs with relative ease. You can bootstrap a VM with knife, installing the chef-client and applying any desired run lists or perhaps, with the new feature, a policy.



But if you want to provision your VMs with Chef, there are many ways to do that. You can use Chef to work directly with ESX hosts using the knife-esx plugin. With this plugin, you can clone, bootstrap, and manage virtual machines on your ESX or ESXi hosts. Although this is very useful, most large VMware deployments don't stop with the utilization of this free version of VMware's hypervisor. Instead, they will opt for the more complete solution, paying for the feature-rich vSphere product line.

So, another Chef to VMware provisioning option is to use the knife-vsphere plugin and interact directly with the vSphere server. This plugin provides a lot of functionality beyond just creating virtual machines. You are also able to handle some management tasks, such as powering on or off your VMs and adding additional drives by adding VMDKs. We will talk more about some of these features shortly.

Then, there is a knife plugin for VMware's **vCloud Director (vCD)** product. It is appropriately named `knife-vcloud`. If you have an active deployment of vCloud Director, like my company does, then you might be very interested in the knife plugin that lets you work with vCD. However, as the vCloud Director product line is nearing the end of its life, I'll not be talking about using the `knife-vcloud` plugin here. However, I will provide a link in the references section if you want to investigate it on your own.

There's also integration between Chef and **VMware vCloud Automation Center (vCAC)**. This is a great area to familiarize yourself with because it is the direction that VMware is moving its product line. There's a plugin named `knife-vrealize`. This is in the same vein because VMware vRealize is the new product branding for vCAC. There's also another, relatively new, VMware knife plugin named `knife-vcair`. This plugin allows you to create, bootstrap, and manage instances in VMware's Hybrid Service known as vCloud Air (`vcair`). I will provide references to these plugins later, but due to space limitations, I will leave the discovery of these to the reader's discretion.

Because this chapter focuses on the traditional data center where vSphere is certain to be found, we will continue to look at the `knife-vsphere` plugin, starting with the installation of the plugin itself.

Installing the `knife-vsphere` plugin

Like many other plugins we have installed so far, we will use Chef to install our new knife gem. Here is the command:

```
chef gem install knife-vsphere
```

Once you've executed the `gem install`, you need to configure your `knife.rb` file. So, edit your file with the following command:

```
vi ~/chef-repo/.chef/knife.rb
```

Add the following lines to your knife file, replacing the values with values appropriate to your vSphere environment:

```
knife[:vsphere_host] = 'vcenter-hostname'  
knife[:vsphere_user] = 'privileged username'  
knife[:vsphere_pass] = 'your password'  
knife[:vsphere_dc] = 'your-datacenter'
```

Here are the values I've used in my `knife.rb` file:

```
knife[:vsphere_host] = 'vcenter6.nat.local.net'  
knife[:vsphere_user] = 'earlwaud'  
knife[:vsphere_pass] = '1Password'  
knife[:vsphere_dc] = 'SanDiego'
```

One big concern here is the public nature of your password. This is very undesirable in a real production environment. So, the `knife-vsphere` plugin provides the ability to use a base64 encoded version. To do this, you would need to prepend the tag `base64:` to your password value string. That would look something like this:

```
knife[:vsphere_pass] = 'base64:Yk2nZuJwHp=='
```

An alternative to saving the password in the `knife.rb` file is to pass the credentials on the command line when you execute your `knife` commands.

Also, depending on your environment, you may experience SSL connection issues when you run your `knife` commands. This is usually due to certificate errors, and you can configure your `knife.rb` file to just ignore these errors with the additional setting as follows:

```
knife[:vsphere_insecure] = true
```

With these commands saved in our `knife.rb` file, we are ready to start provisioning to our vSphere environment. Of course, you are going to need a template in your vSphere environment, so let's check our `knife.rb` configuration with a quick query to see the templates that are available to provision (or clone) from. The command you will use for this check is as follows:

```
knife vsphere template list
```

Another query you will want to try is to list out the available datastores. Use the following command:

```
knife vsphere datastore list
```

Similarly, you can get a list of several object types, such as `hosts`, `vm`, `cluster`, `folder`, `pool`, and `vlan` by issuing a command in the same format:

```
knife vsphere hosts list  
knife vsphere vm list  
knife vsphere cluster list  
knife vsphere vlan list
```

Here is what running these commands looks like in my shiny new vSphere 6 lab:

```

1. bash
Earls-Mac:chef-repo earlwaud$ knife vsphere template list
Template Name: dsl-template
Earls-Mac:chef-repo earlwaud$ knife vsphere datastore list
Datastore: esxi6-01-datastore1 (31.31 GB / 32.50 GB)
Datastore: esxi6-02-datastore1 (31.55 GB / 32.50 GB)
Earls-Mac:chef-repo earlwaud$ knife vsphere hosts list
Pool: Prod
  Host: esxi6-01.nat.local.net
  Host: esxi6-02.nat.local.net
Earls-Mac:chef-repo earlwaud$ knife vsphere vm list
Folder:
  VM Name: DSL-4.4.10
    IP:
    RAM: 64
    State: off
Earls-Mac:chef-repo earlwaud$ knife vsphere cluster list
Cluster: Prod
Earls-Mac:chef-repo earlwaud$ knife vsphere vlan list
VLAN: VM Network
Earls-Mac:chef-repo earlwaud$ █

```

Next, let's take a look at how we provision using the knife-vsphere plugin.

Provisioning VMs using the knife-vsphere plugin

Using the knife-vsphere plugin to provision VMs into VMware Vsphere can range from a fairly simple command to a very complex command.

Starting out with baby steps, here is an example of a simple knife command to clone a VM template to create a new VM:

```
knife vsphere vm clone api_vm01 \
  --template centos6-template \
  --cspec centos6 \
  -v
```

Examining the preceding command, we start off, with the typical format for a knife plugin command, using knife vsphere for the vsphere plugin. Then, we provide the name of the VM we wish to create. In this case, name is `api_vm01`.

Next, we provide the parameter `--template` with a value of the name of the VM template that we want to clone. This is typically one of your company's baseline templates. In my case, it is a simple centos6 template I created for my lab.

Next, using the `--cspec` parameter, you tell knife the name of the customization spec to use to customize the VM to your environment once it has been provisioned. In my case, I have a Linux customization spec named "centos6".

Finally, we are using the `-v` parameter to tell knife to provide verbose output.

Remember that we can use a second V, as in `-vv`, to get very verbose, debug-level output from the command. You might consider using the double V when you are trying these commands out for the first few times.

Now, let's take our provisioning command up a notch. Let's do some additional VM customization with our knife command. Here is another example of using the `knife-vsphere` plugin to provision a VM:

```
knife vsphere vm clone api_vm02 \
  --template centos6-template \
  --cspec centos6 \
  --ccpu 1 \
  --cram 2 \
  --dest-folder Public \
  --datastore esxi6-01-datastore1 \
  --resource-pool Production \
  --annotation "This VM was provisioned using the knife-vsphere plugin.

The VM is owned by Earl Waud" \
  --start \
  -v
```

So, what is new here. We have added the parameter `--ccpu`. The value used for this parameter tells knife to provision the VM with that number of vCPUs. Similarly, the `--cram` parameter and value tells knife to provision the VM with that many GB of RAM. The next new parameter is the `--dest-folder`. The value of this parameter tells knife the name of the folder you want your VM placed in (visible in the **Inventory | VMs and Templates vCenter view**). Remember that the folder has to exist in vCenter for the knife command to execute successfully. If the folder does not exist, your knife command will fail with a "no such folder" error. The next parameter in our knife command is the `--datastore`. The value provided to this parameter is the exact name of the datastore that you want to house the VMs files when it is provisioned. If the datastore does not exist, the knife command will fail with a "no such datastore" error.

Next, we added the `--resource-pool` parameter and value. This parameter is somewhat overloaded. At its most basic use, the value is the name of the vSphere cluster that you want to provision into. But as the name would imply, you are actually targeting the automatic default resource pool that exists in every cluster. You can, as desired, specify the name of a resource pool that exists, and the VM will be added to that resource pool.

 Note that to target a resource pool, you need to provide the value in the form of "Cluster/ResourcePool". For example, if you have a cluster named Production and you have resource pools within that cluster named Gold, Silver, and Bronze, and you want your new VM to go into the Silver resource pool, the parameter value should be provided as "Production/Silver".

Finally, we added the `--annotation` parameter and value. The value provided for this parameter will be inserted into the **Annotations** section shown on the VM's **Summary** view. Oh yeah, we also added the parameter `--start`, which as you might expect, tells knife to have the VM started after the successful provision completes.

So, you may have noted that while we did provision a VM, and we did some pretty awesome customizations, we didn't do anything Chef to our VM. So, let's Chef this thing up. We are going to do another provision, and this time, we're going to add two Chef-specific parameters, one parameter to tell Knife to bootstrap our VM and the other parameter to provide the node name to our Chef Server to register the VM under. We also need to add some authentication parameters. For this example, I am using ssh user credentials, so the new knife command looks like this:

```
knife vsphere vm clone api_vm03 \
  --template "centos6-template" \
  --cspec centos6 \
  --ccpu 1 \
  --cram 2 \
  --dest-folder Public \
  --datastore esxi6-01-datastore1 \
  --resource-pool "Development" \
  --annotation "This VM is owned by Earl Waud." \
  --start \
  --node-name api_vm03 \
  --bootstrap \
  --bootstrap-protocol ssh \
```

```
--ssh-user centos \
--ssh-password password \
--ssh-port 22 \
--run-list "recipe[base]" \
-v
```

This is not the best way to authenticate when cloning a VM with knife for several reasons, not the least of which is that your password is right there for all to see. But in addition, when Knife gets to the step where it uses sudo to gain root-level privileges on your new VM, it may ask you for the sudo password (unless the user is configured for NOPASSWD access in the sudoers file). This does not lend itself to automation very well. This of course implies that the ssh user you provide on the command line must be added to the sudoers file in the template for the chef-client install to work.

Now we have a Knife command that will provision VM, customize that VM, bootstrap the VM, and set the desired state to converge with the run list specified. The VM will be registered with the Chef Server and can be managed from there. Not too shabby.

Cleaning up our vSphere and Chef environments

Perhaps we have been doing a lot of test provisions, or we have system that have been around for a while that we need to retire to reclaim our compute resources. For whatever the reason maybe, we need to get rid of some of our VMs. The knife-vsphere plugin has been covered. We can control the power state of our VMs using the state subcommand. For example, if we want to power off our VM, we can issue a command in this format:

```
knife vsphere vm state <VM NAME> --state off
```

For example, if we want to power off the VM we provisioned previously, we can use the command:

```
knife vsphere vm state api_vm03 --state off
```

This command will do a hard power off of the VM. If we want to be more respectful of the guest OS running within VM, we can add the --shutdown parameter to the command, and the knife plugin will execute a guest shutdown on the VM instead of a hard power off. Once we are completely finished with our VM, we can remove them from vCenter by issuing a delete command. The command takes the following form:

```
knife vsphere vm delete <VM NAME>
```

 Big note! This command is destructive in that it not only removes the VM from vCenter, but it does it with the "Delete from Disk" command, resulting in the VM being removed from vCenter and all of the VM's files get deleted from the datastore that contained them. They are gone, and unless you have some external backup (such as NAS snapshots and many more), they are gone for good. So, be careful with using this feature.

As an example of using this command, if we want to delete the VM we recently provisioned, we can issue the following command:

```
knife vsphere vm delete api_vm03
```

There is a very useful parameter that we can apply with the `delete` subcommand. It is `--purge`. When we add this parameter to the `delete` subcommand, the node representing the VM in the Chef Server will also be deleted.

Using Chef with VMware and Vagrant

Even though this chapter focuses on using Chef in the traditional data center, I want to take a few minutes to cover using Chef with either VMware Workstation or VMware Fusion and a tool named Vagrant to provision development environments. This is an important technology because it creates development environments that are running the same Chef code that is used in production vSphere environments. "Sounds good," you say; "How do I do it," you say, well let's take a look now.

Disclaimer

 The Vagrant plugins for either VMware Workstation or VMware Fusion are commercial (not open source) products and must be purchased from HashiCorp. At the time of this writing, they are \$79 per license. The licenses are unique between Workstation and Fusion, so if you work using both desktop hypervisors, you would need to purchase a separate license for each. If you are in a corporate setting and developing Chef cookbooks and your company is providing Workstation or Fusion software, then they should willingly spring for the Vagrant plugin license. If you are not able or not willing to purchase the license for the VMware desktop hypervisor, then there is a popular alternative that does not require the purchase of a commercial product. That alternative is using VirtualBox from Oracle as the desktop hypervisor. The general functionality is the same, although the templates used for provisioning will differ.

Once you have purchased your Vagrant license, you will get an e-mail from HashiCorp that details instructions on downloading your plugin and license file. Once you have them downloaded and have Vagrant installed (of course), you can install and license your new plugin. The command will be slightly different depending on if you are using Workstation or Fusion.

Here are the commands to install and license the Fusion plugin:

```
vagrant plugin install vagrant-vmware-fusion  
vagrant plugin license vagrant-vmware-fusion license.lic
```

And here are the commands to install and license the Workstation plugin (it should be run from a command prompt with Administrator privileges):

```
vagrant plugin install vagrant-vmware-workstation  
vagrant plugin license vagrant-vmware-workstation license.lic
```

The `license.lic` parameter in these commands is the filename (and path) to the downloaded plugin license key file.

You can confirm that the plugin and license were successfully deployed by issuing the following command:

```
vagrant plugin list
```

We want to download some VM boxes to use in our development environment. You can find many options in the Vagrant box share found at <http://www.vagrantbox.es>. For my testing, I have downloaded a CentOS5.10 box. Here is the command:

```
vagrant box add centos-5.10 https://dl.dropboxusercontent.com/s/  
r5okkx8330h3tzh/vagrant-centos-5.10-x86_64.box
```

This will download the VM box and deploy it in your VMware Workstation environment. The box is placed into a common Vagrant cache area so that you won't need to download the box file each time you deploy a new environment. You can see a list of the boxes that have been downloaded and are stored in the cache with the following command:

```
vagrant box list
```

Now we're ready to provision some VMs with our desktop hypervisor. For brevity, in the remaining examples in this section, I will be using Workstation. The Fusion commands are nearly identical.

Now you need to initialize your new project. Issue the following command:

```
cd ~/chef-repo  
vagrant init centos-5.10
```

Now we want to bring up our new Vagrant box with the following command:

```
vagrant up --provider vmware_workstation
```

You can see a list of the templates that have been downloaded and are stored in the cache with the following command:

```
vagrant box list
```

We now have a new VM running in our VMware Workstation system. However, here is another case where we haven't done anything with Chef yet. So, let's go ahead and get rid of this environment with the following command:

```
vagrant destroy
```

This will power down the VM and then delete it from the workstation (and from the hard drive).

OK, with our first test system deployed and cleaned up, we are ready to add some Chef to this party. We need to edit the Vagrantfile that was created for us when we ran the `vagrant init` command earlier. Initially, this default Vagrantfile will comprise mostly comments. For the sake of space in this chapter, we will remove the comments and only share the active content of the Vagrantfile. Here is what a simple Vagrant file looks like:

```
# -*- mode: ruby -*-  
# vi: set ft=ruby :  
  
VAGRANTFILE_API_VERSION = '2'  
  
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|  
  config.vm.box = 'centos-5.10'  
  
  config.vm.network 'public_network'  
  
  config.vm.hostname = 'vagrant-vm01'  
  
  config.vm.provision :chef_client do |chef|  
    chef.chef_server_url = \  
      'https://chefserver/organizations/sdearl'  
    chef.validation_client_name = 'sdearl-validator'  
    chef.validation_key_path = \  
      '
```

```
'/Users/earlwaud/chef-repo/.chef/sdearl-validator.pem'  
chef.run_list = 'recipe[base] '  
chef.validation_client_name = 'sdearl-validator'  
chef.environment = 'production'  
end  
end
```

Now, when we issue the command `vagrant up`, the VM being provisioned will also have the chef-client installed and will be registered with the Chef server, and will converge with the run list of "recipe[base]".

Now that we are deploying VMs using Vagrant and registering them with our Chef server, when we clean up those VMs, we also want to clean up the Chef-client and node in our Chef Server. Of course, you can do this manually every time, but that is not what we are all about. We want automation. Well, there's a Vagrant plugin for that. It is called **Butcher**. What the plugin does for us is when we issue our `vagrant destroy` commands, it will not only destroy the VM but will also reach out to the Chef Server and do the necessary cleanup of the Node there as well. Installing the plugin is easy. Just enter the following command:

```
vagrant plugin install vagrant-butcher
```

Once it is installed, there are a few additional configuration lines you will want to add to your Vagrantfile. Here are the lines to add:

```
config.butcher.client_key = \  
'/Users/earlwaud/chef-repo/.vagrant/butcher/default-client.pem'  
config.butcher.enabled = true  
config.butcher.verify_ssl = false
```

Now when you execute your `vagrant destroy` commands, everything will be cleaned up automatically. Great.

References

- You can read about the knife-vsphere plugin at <https://github.com/chef-partners/knife-vsphere>.
- How to guide on setting up a vSphere 6 lab in VMware Workstation is at <http://www.enterprisedaddy.com/building-a-home-lab-for-vmware-vsphere-6-0/>.
- Learn more about the many Chef + VMware integrations at [https://www.chef.io/solutions/vmware/](http://www.chef.io/solutions/vmware/).
- You can read about the knife-vcair plugin at <https://github.com/chef-partners/knife-vcair>.

- If you are working directly with ESX instead of vSphere, you might like the info for the knife-esx plugin, which can be found at <https://github.com/maintux/knife-esx>.
- Finally, if you want to work with VMware Fusion or VMware Workstation, you can integrate with Vagrant and the corresponding plugin. Learn more at <https://www.vagrantup.com> and <https://www.vagrantup.com/docs/vmware/installation.html>.
- You can find a variety of Vagrant base boxes at <http://www.vagrantbox.es>.
- If you are interested in making your own templates or boxes for Vagrant, there is an excellent tutorial found at <https://blog.engineyard.com/2014/building-a-vagrant-box>.
- You can read about the Vagrant Plugin Butcher at <https://github.com/cassianoleal/vagrant-butcher>.

OpenStack provisioning

Using Chef to provision to OpenStack instances requires an OpenStack environment to work with. This seems obvious and simple, but with OpenStack, nothing is really obvious and simple. To emphasize this point, let me tell you a story.

My first exposure to OpenStack was in early 2014, when the company I was working for hired a well-respected training company to come onsite and train our team in the setup and use of OpenStack. It was a very expensive engagement, but we were considering the possibility of reducing our reliance on VMware and were looking at OpenStack as a path toward that end. The training was scheduled for full 5 days for about a dozen people. The instructor arrived on Monday morning, and after introductions, we got down to it. After 3 full days of "work," only two people in the class had a working OpenStack environment. Even the instructor's example environment was not functional. On that day, the instructor apologized and bowed out, ending our training early. The costs were refunded by the training company, and our team decided that, at least for the short term, OpenStack was not our path to VMware independence.

OpenStack has come a long way since then, and it is much less troublesome to get an environment up and running today. Still, it is a complex deployment and can present a lot of challenges. So, before we dive into our use of Chef to provision to OpenStack instances, let's take a look at the way we can set up a development OpenStack environment.

Know that the information in the section ahead was not easily obtained. Bits and pieces had to be learned from all over the place, refined, and reworked to be able to detail a way to deploy a fully functional DevStack OpenStack environment using Neutron networking in a VMWare Workstation VM. Even though this is a book on Chef Provisioning, the following section alone is worth the price of the book!

To paraphrase a quote by Matt Damon's character in the move The Martian -- "I had to science the crap out of this".

So, let's get to work...

Setting up a DevStack OpenStack all-in-one environment

There is a community production of OpenStack named DevStack. This "tool" is not something meant for production use and does not support all configuration and platforms. However, it is a perfect tool for education and dev and test scenarios. You can quickly set up a working OpenStack environment on a workstation or a beefy laptop. So, if you want to use OpenStack for development, the DevStack production is a good option for you. Let's walk through the setup so that if you don't already have access to OpenStack, you can test out the Chef provisioning examples coming up later in the chapter. Let's begin with the setup of the "host" of our DevStack.

Deploying the DevStack Host VM

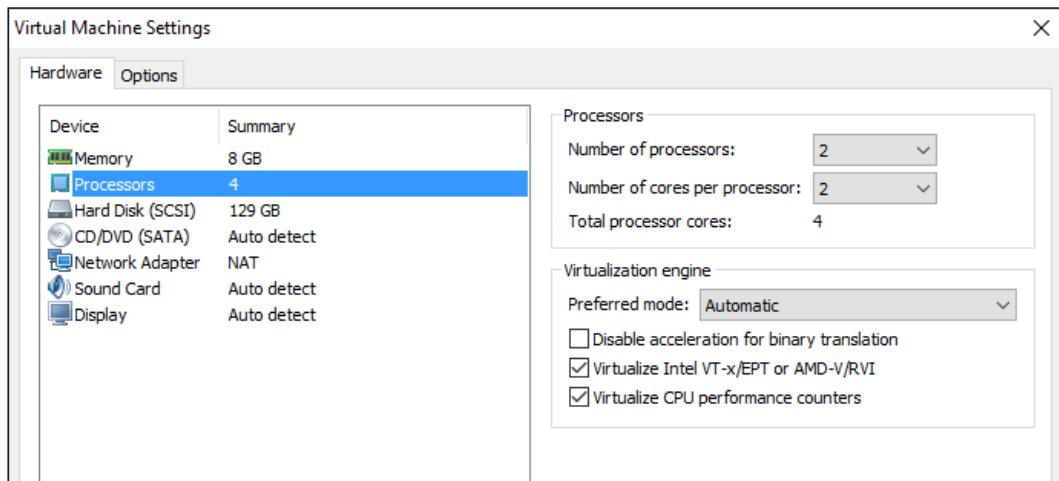
You can set up DevStack directly on physical hardware, but I would highly recommend that you use a desktop virtualization tool such as VMWare Workstation or Fusion. You probably already have it installed in your system. We are going to set up a new VM for our DevStack and set up specific networking for that VM to assure that everything works as desired.

I suggest you to download the Ubuntu 14.04 desktop 64-bit installer ISO for your host VM. At the time of this writing, the current master repo for DevStack works well with this version of Ubuntu. Plus, it is painless to install in the latest version of VMWare's desktop hypervisors, using the "Easy Install" feature, which answers nearly all of the deployment questions automatically.

Because we are in fact going to install all of our OpenStack environment into this host VM, you will want to set the configuration a little larger than you might be used to for normal desktop OS VMs. I would recommend that you give the VM at least 6 GB of memory. With only 4 GB, you will experience a small amount of memory swapping just to stand up the OpenStack environment. It will work, but if you have a system that has the RAM, go ahead and set the VM to 6 GB of memory or more. Since my workstation has 32 GB of RAM, I am using 8 GB for my VM. Next, you will want to give the VM at least 2 CPUs. This is the bare minimum. Since I have a system that can handle it, I am setting my VM's CPU count to 4. Be sure to enable the Virtualization options on the CPUs in **Virtual Machine Settings**, as shown in the following image. Next, you should allow the VM to have some disk breathing room by setting a large hard disk size. I went with a 180 GB drive.

 Remember that the disk will not use the full allocated capacity set here, but it will only use what is actually used by the VM.

I set my disk files to use a single file instead of breaking them into small chunks.



For now, in the networking option, select NAT for **Network Adapter**. We will talk more about the networking shortly.

Next, I removed the USB controller and the printer as they are not necessary for the DevStack host. I also turned off the **accelerate 3D Graphics** option on the Display.

Once you've set your Virtual Machine Settings, save them and power on the VM to let it run the OS installer. Enter your name, username, and password and remember them so you can log in once the VM is running. When the installation is complete, log in and make the VM your own by setting your normal desktop things, such as wallpaper, terminal settings, and running software updates (restart as necessary).

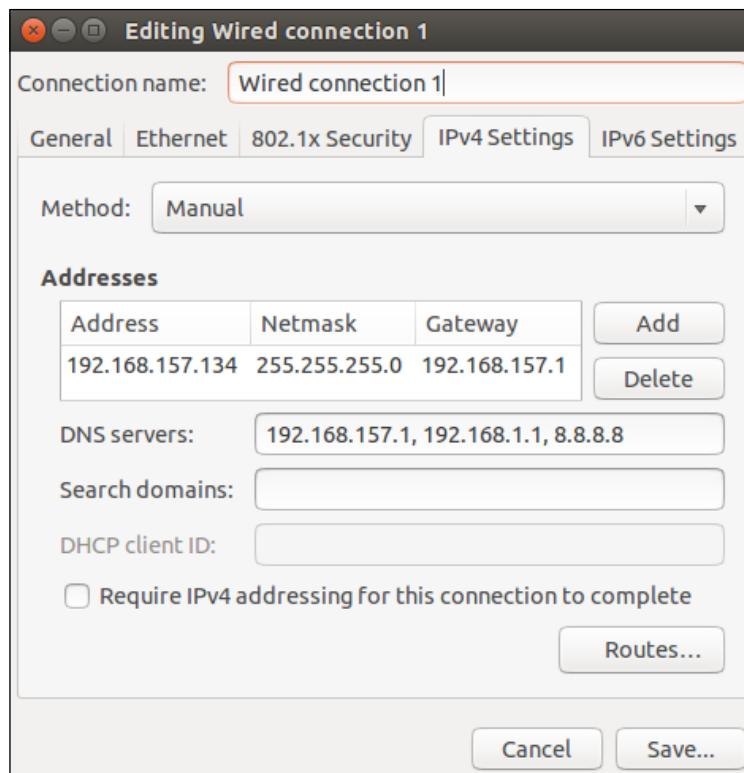
In addition, I would recommend installing the ntp, landscape-common (for the landscape-sysinfo command), and htop (it's just a better top). Here is the command:

```
sudo apt-get install ntp landscape-common htop
```

You might want to consider taking a VM Snapshot at this point and labeling something like Clean Ubuntu 14.04 Install.

Change host interface from DHCP to static.

We want to control the IP address assigned to our OpenStack host, so we need to change the interface configuration from a DHCP autoassigned IP configuration to one that is static. In my setup, the NAT networking interfaces are on the 192.168.157.0/24 subnet. So, here is what my updated networking will look like:



Save your new settings and then, to make sure the settings take effect, open a terminal window and restart the network manager with the following command:

```
sudo service network-manager restart
```

Now check your settings by with the ifconfig command and pinging some well-known servers such as www.google.com.

Downloading and installing DevStack

Now that we have our host VM up and running, it is time to download and install OpenStack DevStack. First things first, we need to install git. Open up a Terminal session on your host VM and issue the following command:

```
sudo apt-get install git -y
```

This will install git so that we can use it to download the DevStack repo. I would recommend doing an update next with the following command:

```
sudo apt-get update
```

Then, issue the following git command to clone the DevStack repo:

```
cd ~/  
git clone https://git.openstack.org/openstack-dev/devstack -b stable/kilo
```

Although you can deploy the master branch for DevStack, or even the Liberty branch, at the time of writing, the knife-openstack plugin has not been blessed for anything later than the OpenStack stable kilo release. Therefore, I am working with stable/kilo branch so that you will have a working example. The earlier stable branches are probably ok, but I would recommend going with kilo since I know that it works with the config file I am about to share.

Now that we have the DevStack installer bits, we can do the installation as soon as we are ready.

Next, we want to create a configuration file to help DevStack configure OpenStack correctly for our Host VM.

Create and edit a file named local.conf:

```
cd ~/devstack  
touch devstack/local.conf  
vi devstack/local.conf
```

Enter the following data into the new file:

```
[[local|localrc]]  
  
# Passwords (modify to match your VM)  
ADMIN_PASSWORD=password  
  
DATABASE_PASSWORD=$ADMIN_PASSWORD  
RABBIT_PASSWORD=$ADMIN_PASSWORD  
SERVICE_PASSWORD=$ADMIN_PASSWORD  
SERVICE_TOKEN=$ADMIN_PASSWORD  
  
# Branches  
KEYSTONE_BRANCH=stable/kilo  
NOVA_BRANCH=stable/kilo  
NEUTRON_BRANCH=stable/kilo  
SWIFT_BRANCH=stable/kilo  
GLANCE_BRANCH=stable/kilo  
CINDER_BRANCH=stable/kilo  
HEAT_BRANCH=stable/kilo  
TROVE_BRANCH=stable/kilo  
HORIZON_BRANCH=stable/kilo  
SAHARA_BRANCH=stable/kilo  
  
# Host VM IP address (replace with the IP of your VM)  
HOST_IP=192.168.157.134  
SERVICE_HOST=192.168.157.134  
MYSQL_HOST=192.168.157.134  
RABBIT_HOST=192.168.157.134  
GLANCE_HOSTPORT=192.168.157.134:9292  
  
# domain (should be the same as chef server domain)  
# dhcp_domain=nat.local.net  
  
# Enable the console auth service  
enable_service n-cauth  
  
# Disable tempest service to save stack time  
disable_service tempest  
  
# Do not use Nova-Network  
disable_service n-net  
  
# Enable Neutron
```

```
ENABLED_SERVICES+=,q-svc,q-dhcp,q-meta,q-agt,q-l3

# Neutron options
Q_USE_SECGROUP=True
Q_L3_ENABLED=True
Q_USE_PROVIDERNET_FOR_PUBLIC=True
Q_AGENT=linuxbridge

# IP range for fixed and floating IP assignments
FIXED_RANGE="10.0.0.0/24"
PUBLIC_INTERFACE=eth0
FLOATING_RANGE="192.168.157.0/24"
Q_FLOATING_ALLOCATION_POOL=start=192.168.157.250,end=192.168.157.254
PUBLIC_NETWORK_GATEWAY="192.168.157.1"

# Linuxbridge Settings
LB_PHYSICAL_INTERFACE=eth0
LB_INTERFACE_MAPPINGS=default:eth0
PUBLIC_PHYSICAL_NETWORK=default

# Enable Logging
LOGFILE=/opt/stack/logs/stack.sh.log
VERBOSE=True
LOG_COLOR=True
SCREEN_LOGDIR=/opt/stack/logs
```

You will need to make a few changes to reflect your installation.

First, change the `ADMIN_PASSWORD` value to the password you use for your login account on the Host VM.

Then, replace all of the 192.168.157.134 values with the IP address you are using for the Host VM.

Then, set the `dhcp_domain` value to the same domain that your Chef Server belongs to. This value will be to domain that provisioned instances belong to. If they do not match the domain of your Chef Server, they will not be able to register with the Chef Server.

Then, replace the `FLOATING_RANGE` and `Q_FLOATING_ALLOCATION_POOL` values with values in the same subnet as your Host VM.

Finally, replace the `PUBLIC_NETWORK_GATEWAY` value with the gateway address of your NAT network.

Now save the file with all the changes, and you are ready to stack your DevStack.



Don't use root to run DevStack

An important note here is that you do not want to run DevStack as root. The web page documentation suggests you to use a script provided in the repo to create a "stack" user. I have found that there are permission issues that can get introduced if you do that. So, I recommend that you use your own user account to run DevStack, and don't bother with the creation of a specialized "stack" user.

Keep your network active during the stacking of DevStack. When you are "stacking," there is a lot of dependency on the eth0 interface of the Host VM to stay active. I've found that depending on your system's connection, the eth0 interface can temporarily lose connectivity with the NAT network. I've found that if you open another terminal window and start up a continuous ping of the host computer's IP address that the stack is much more likely to run to completion. This might just be a quirk of my systems, but I've found that it really is does the trick for me. Once the stack is up and running, you can stop the ping.

Ok, we have our second terminal window open and pinging our VMs host IP address, so we are ready to install and start our OpenStack environment. Change into the devstack folder and launch DevStack with the following commands:

```
cd devstack  
./stack.sh
```

This will take a while, depending on your system's horsepower and your network bandwidth. It may be a good time to go get a coffee, grab some lunch, or take a nap. Naps are nice. Expect about 30 minutes, give or take, for the stack command to complete.

You will see a lot of messages across the screen, but if everything goes according to plan, you will get a "DevStack Components Timed" message that will look something like this:

```
This is your host ip: 192.168.157.134  
Horizon is now available at http://192.168.157.134/  
Keystone is serving at http://192.168.157.134:5000/  
The default users are: admin and demo  
The password: password
```

You can now open up your browser and enter the Horizon URL shown in your stack. sh output and log into your OpenStack environment. However, before you get too far, I would suggest you to take another snapshot of your VM as a safety net.

Downloading the demo project rc file and applying it

We now have a working OpenStack environment, but there is one more thing we want to do. Since this is a test tool and we are going to want to make testing easy, let's configure the system to permit all ping and ssh traffic; so when we use commands to access the instances we are going to provision, we will have no surprises.

The first step will be to log in to our environment, so open up a browser on your DevStack VM and log in as admin. For me, the URL is <http://192.168.157.134/>. Use the user name `admin` and the password that you used in the `local.conf` file described in the preceding section. For me, that was `password`.

Now make sure that you are in the demo project space by selecting `demo` in the projects drop-down menu near the top-left part of the web page. Next, using the menu accordion on the left, select the option **Project**, then **Compute**, and **Access & Security**. Now in the **Access & Security** view, select the tab for **API Access**. Press the **Download OpenStack RC File** button to download the rc file. Save the file to your DevStack Host VM.

Adding the security rules to allow ping and ssh traffic

Now we have the rc file to configure the environment in order to add our security rules. Open up a terminal window and, assuming that you saved the rc file in your Downloads folder, issue the following command:

```
source ~/Downloads/demo-openrc.sh
```

You will be prompted for your password. Again, enter the password you used in the `local.conf` file (the same one shown at the end of your successful run of `./stack.sh`).

Shortly, you will be entering two commands, one for the ICMP protocol for ping, and the other for tcp protocol on port 22 for the ssh commands.

The two commands will be in this format:

```
neutron security-group-rule-create \
<SECURITY GROUP ID> \
--direction ingress --ethertype IPv4 --protocol icmp \
--remote-ip-prefix 0.0.0.0/0

neutron security-group-rule-create \
<SECURITY GROUP ID> \
--direction ingress --ethertype IPv4 --protocol tcp \
--port-range-min 22 --port-range-max 22 \
--remote-ip-prefix 0.0.0.0/0
```

However, you'll first need to get the <SECURITY GROUP ID> value for your DevStack's default security group to use in the commands. To get the security group ID, you will issue the nova secgroup-list command as follows:

```
nova secgroup-list
```

Here is the results from my execution of these commands as an example:

```
earlwaud@ubuntu:~$ source ~/Downloads/demo-openrc.sh
Please enter your OpenStack Password:
earlwaud@ubuntu:~$
earlwaud@ubuntu:~$ nova secgroup-list
+-----+-----+
| Id | Name | Description |
+-----+-----+
| a209661a-cb15-4cc7-b42f-45a5c507aba0 | default | Default security group |
+-----+-----+
earlwaud@ubuntu:~$ neutron security-group-rule-create \
> a209661a-cb15-4cc7-b42f-45a5c507aba0 \
> --direction ingress --ethertype IPv4 --protocol icmp \
> --remote-ip-prefix 0.0.0.0/0
Created a new security_group_rule:
```

```
+-----+-----+
| Field | Value |
+-----+-----+
| direction | ingress |
| ethertype | IPv4 |
| id | fabb4c57-b2c6-4661-a1a7-f7083b1d1743 |
| port_range_max | |
| port_range_min | |
| protocol | icmp |
| remote_group_id | |
| remote_ip_prefix | 0.0.0.0/0 |
| security_group_id | a209661a-cb15-4cc7-b42f-45a5c507aba0 |
| tenant_id | e2af8dbd4ae44ecf84b1f9b30ff1aad8 |
+-----+
earlwaud@ubuntu:~$  

earlwaud@ubuntu:~$ neutron security-group-rule-create \  

> a209661a-cb15-4cc7-b42f-45a5c507aba0 \  

> --direction ingress --ethertype IPv4 --protocol tcp \  

> --port-range-min 22 --port-range-max 22 \  

> --remote-ip-prefix 0.0.0.0/0  

Created a new security_group_rule:  

+-----+-----+
| Field | Value |
+-----+-----+
| direction | ingress |
| ethertype | IPv4 |
| id | 5db1c6e-e57d-4739-82b2-5153270f033f |
| port_range_max | 22 |
| port_range_min | 22 |
| protocol | tcp |
| remote_group_id | |
| remote_ip_prefix | 0.0.0.0/0 |
| security_group_id | a209661a-cb15-4cc7-b42f-45a5c507aba0 |
| tenant_id | e2af8dbd4ae44ecf84b1f9b30ff1aad8 |
+-----+
earlwaud@ubuntu:~$
```

With these rules in place, all of the ICMP and TCP traffic on port 22 will pass into your provisioned instances. Your new DevStack OpenStack all-in-one system is now ready to use for development and testing.

You will also want to provide a *public* DNS address for your *private* network so that your provisioning VMs are able to resolve external host names. I am using a local DNS address of 192.168.157.101 and a public DNS address of 8.8.8.8. So, the command to do that is as follows:

```
neutron subnet-update private-subnet --dns-nameservers list=true  
192.168.157.101 8.8.8.8
```

Since I was rebuilding my DevStack a lot during testing (and writing this book), I created a small script to add rules. The script still needs to source your `demo_openrc.sh` file, so it will ask for your password, but once that's done, the script to create the security group rules and inject the public DNS server:

```
#!/bin/bash  
#new_rules.sh  
  
# load the demo project access settings  
source ~/Downloads/demo-openrc.sh  
  
# capture the ID for the "default" security group  
ID=$(nova secgroup-list | grep default | awk -F"|" '{ print $2 }')"  
  
# add a rule for ping  
neutron security-group-rule-create $ID \  
    --direction ingress --ethertype IPv4 \  
    --protocol icmp --remote-ip-prefix 0.0.0.0/0  
  
# add a rule for ssh  
neutron security-group-rule-create $ID \  
    --direction ingress --ethertype IPv4 \  
    --protocol tcp --port-range-min 22 --port-range-max 22 \  
    --remote-ip-prefix 0.0.0.0/0  
  
# add a rule for dns  
neutron security-group-rule-create $ID \  
    --direction ingress --ethertype IPv4 \  
    --protocol tcp --port-range-min 53 --port-range-max 53 \  
    --remote-ip-prefix 0.0.0.0/0  
  
# add a public DNS to the private instances  
neutron subnet-update private-subnet --dns-nameservers list=true  
192.167.157.101 8.8.8.8
```

With that large task behind us, let's now return to our Chef workstation and do some OpenStack provisioning.

Installing the knife OpenStack plugin

Now that we have a working OpenStack environment, there are some steps we need to take on our Chef workstation to set up for provisioning into that environment. The first of those steps is to install the knife-openstack gem. To accomplish this, back on your Chef development workstation, change directories to your `chef-repo` folder and issue the `chef gem install` command, like this:

```
cd ~/chef-repo  
sudo chef gem install knife-openstack
```

This command will download and install the gem and make it ready and available for our knife commands to use. There is still more setup work that we need to do; therefore, we will set up some knife OpenStack-specific configuration so that knife can communicate with our OpenStack environment.

We need some data from the OpenStack environment, and the best way to get it is to log in and download it. So, on your Chef workstation, browse to the OpenStack Horizon dashboard and log in. If you are using the DevStack environment that we set up earlier, you will use the URL that you got from running `stack.sh`. In my case, it is `http://192.168.157.134/`. You will log in with the admin credentials also provided from running the `stack.sh` command.

Setting up the DevStack public key

We are going to create and download a security key pair to allow our knife bootstrap commands to ssh into our instances without using username/password credentials.

So, once you log in to OpenStack, set your project to be "admin" by selecting admin from the projects drop-down at the top of the web page. You should see the green "switch to project "admin" successful" message, and your project label should now display as admin.

Next, using the accordion menu on the left, select the **Project** top menu, and from its submenu options, select **Access & Security**. Now in the **Access & Security** area, select the tab for **Key Pairs**.

If you just deployed your DevStack, you probably don't have any existing key pairs to use, so press the **+ Create Key Pair** button to create you first one. Give it a significant **Key Pair Name**, then press the **Create Key Pair** button. I named my key pair `admin` since I will be working with the admin project and login credentials. Creating the key pair will initiate a download of the public key file. If it does not, be sure to download the key before moving away from the new key pair page. It is important to remember the name you use to create the key pair as it will be vital to the successful use of our provisioning with Chef, (and it is the name given to the public key file).

Now copy the public key file to a location you can access easily from your chef repo. For simplicity of my examples, I am copying the public key file to my `chef-repo` folder.

```
cp ~/Downloads/admin.pem ~/chef-repo/
```



One important tip to remember is that public key files need to be secure, and most applications won't use them unless the permissions are tight.



So, the next step will be to make sure to tighten the permissions on the public key file. Issuing the following command will take care of this step:

```
chmod 400 ~/chef-repo/admin.pem
```

The public key is now ready to use in our chef OpenStack provisioning. But before we can start using it, we need to poke some holes in our OpenStack firewall.

Allocating some floating IP addresses to the project

We worked very hard to create an OpenStack environment that allows us to use the knife-openstack plugin to allocate floating IP addresses, so we might as well add some to our project so that we can allocate them when the time comes.

Browse to your horizon dashboard and log in. Make sure that your project is set to demo again (it should default to that because it was used previously, but check to be sure). Select the **Project** menu from the menu accordion on the left and then select **Access & Security** from the **Project** submenu options. Now, select the **Floating IP** tab.

Click on the **Allocate Floating IP Address** button and add an IP to the public group. Repeat this process a few times to give us a few floating IPs to work with.

These public IP addresses will be assigned to the OpenStack instances to allow a route to access the private subnet addresses that are assigned to the instances during provisioning. The floating IP will be used by knife to install the chef-client and to execute the chef-client run that will register the node with the Chef Server. Apply the configuration settings based on the run list used.

Configuring the knife-openstack plugin knife settings

Do you remember the `knife.rb` file we configured in *Chapter 2, Knife Tooling and Plugins*? Well, we are going to update it now to include the settings for the knife-openstack plugin. Once again, we need information from OpenStack, and the best way to get it is to log in and collect it.

Once more, browse to your horizon dashboard and log in. Again, make sure that your project is set to admin. Again, select the **Project** menu from the menu accordion on the left, and select **Access & Security** from the **Project** submenu options. This time, in the **Access & Security** view, select the **API Access** tab.

Once there, you are going to want to view the credentials needed for the API access. As you probably have already figured out, you can view them by clicking on the button marked **+ View Credentials**. Here is what my API access credentials look like:

The screenshot shows a modal dialog titled "User Credentials". It contains four input fields: "User Name" (admin), "Project Name" (admin), "Project ID" (ae3bbe1e367f44189424511bd8c123a0), and "Authentication URL" (http://192.168.157.134:5000/v2.0). Each field has a placeholder text and a corresponding input box.

User Credentials	
User Name	admin
Project Name	admin
Project ID	ae3bbe1e367f44189424511bd8c123a0
Authentication URL	http://192.168.157.134:5000/v2.0

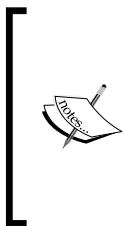
You will want to capture the values shown in the **User Credentials** dialog as we are going to need them for our `knife.rb` file. Once you have copied the values, you will want to edit your `knife.rb` file:

```
cd ~/chef-repo  
vi .chef/knife.rb
```

Go to the end of the file and add the following values, replacing the **Project ID** and **Authentication URL** values with the data from your **User Credentials** dialog. Here are my entries:

```
knife[:openstack_auth_url] = 'http://192.168.157.134:5000/v2.0/tokens'  
knife[:openstack_username] = 'admin'  
knife[:openstack_password] = 'password'  
knife[:openstack_tenant] = 'demo'  
knife[:openstack_tenant_id] = 'ae3bbe1e367f44189424511bd8c123a0'  
knife[:openstack_region] = 'RegionOne'
```

Notice that the `openstack_auth_url` value has an extra `/tokens` string at the end that is not shown in the **User Credentials** dialog. Make sure that you add it to your `knife.rb` file.



Note that each OpenStack entry should be on its own, single line in the `knife.rb` file. The code shown above may appear to have some lines split into two. The only other comment on these lines is the one for the OpenStack region. This is the default value used for my DevStack environment. It will be the same default value for your environment too. Now, save your changes, so we can test our `knife-openstack` plugin integration.

Here, for reference, is the full `knife.rb` file that contains settings for knife, the editor integration, the `ec2` plugin, and the OpenStack plugin:

```
# See https://docs.getchef.com/config_rb_knife.html for more  
# information on knife configuration options  
  
current_dir = File.dirname(__FILE__)  
log_level :info  
log_location STDOUT  
node_name 'earlwaud'  
client_key "#{current_dir}/earlwaud.pem"  
validation_client_name 'sdearl-validator'  
validation_key "#{current_dir}/sdearl-validator.pem"  
chef_server_url 'https://chefserver/organizations/sdearl'  
cookbook_path ["#{current_dir}/../cookbooks"]
```

```
#ssl_verify_mode      :verify_none
knife[:editor] = '/Applications/Sublime\ Text.app/Contents/
SharedSupport/bin/subl -w'
knife[:aws_credential_file] = '/Users/earlwaud/.aws/credential_file'

#knife-vsphere
knife[:vsphere_host] = 'vcenter6.nat.local.net'
knife[:vsphere_user] = 'earlwaud'
knife[:vsphere_pass] = '1Password'
knife[:vsphere_dc] = 'SanDiego'
knife[:vsphere_insecure] = true

#knife-openstack
knife[:openstack_auth_url] = 'http://192.168.157.134:5000/v2.0/tokens'
knife[:openstack_username] = 'admin'
knife[:openstack_password] = 'password'
knife[:openstack_tenant] = 'demo'
knife[:openstack_tenant_id] = '181735be7b8a4a0c80c68200dffabe77'
knife[:openstack_region] = 'RegionOne'
knife[:openstack_ssh_key_id] = 'admin'
```



Add knife openstack server create parameters to your `knife.rb` file. You may note the `openstack_ssh_key_id` value shown in this listing. I've added it to my `knife.rb` file so that I can avoid adding it to every `server create` command. I'll include it in the upcoming example, but know that you can add other OpenStack parameters to your `knife.rb` file to shorten your `knife` commands.

With our updated `knife.rb` file, we should be able to successfully communicate with our OpenStack environment using `knife`. We can test this integration easily with the following commands:

```
cd ~/chef-repo
knife openstack server list
```

If your configuration is correct, issuing the `server list` command shown earlier will return a single header line of information. It is only showing a header line because we haven't deployed any instances yet. Another command you might want to try at this point is the `image list` command. We are going to need the image information when we try to provision an instance. Give this a try now:

```
knife openstack image list
```

You should get back a list of the default images in your OpenStack environment. In my case, issuing that command looks like this:

```
Earls-Mac:chef-repo earlwaud$ sudo knife openstack image list
Name           ID
Snapshot
cirros-0.3.4-x86_64-uec      84dff654-7f0d-45c8-a20b-b08c0ef39fd1  no
cirros-0.3.4-x86_64-uec-ramdisk  d03ec228-a5c9-4200-8058-3bde074c67a1  no
```

Let's get a better option for our knife-openstack plugin testing.

Adding a CentOS instance image to provision

The default deployment of DevStack detailed earlier comes with some sample cirros images to test and validate the OpenStack deployment. Those images are listed in the output example from the previous section. For our use, we want to deploy a more production-like image. For the examples in the rest of this chapter, I am going to use a CentOS image for provisioning OpenStack instances.

You can obtain the same image by visiting the official Get Images OpenStack page at this URL: <http://docs.openstack.org/image-guide/obtain-images.html>. Near the top of the page, you will find the Official CentOS 6 Images link. Clicking that link will take you to a list of CentOS 6 images you can download. I'll use the CentOS image for the rest of the testing in this chapter. Click on the link for that image and download it to your Chef workstation. The exact name of the image I downloaded for the examples is as follows:

```
CentOS-6-x86_64-GenericCloud-1508.qcow2
```

Once it has been downloaded, you can log in to your OpenStack environment and add the image to the list of available images. Make sure that you are in the demo project, then using the menu accordion on the left, select **Project | Compute | Images**. Now press the **+ Create Image** button to create our new `centos6` image. A dialog window will open up, and you can populate the inputs as follows:

Create An Image

Name *	centos6	Description:
Description	CentOS 6	Specify an image to upload to the Image Service. Currently only images available via an HTTP URL are supported. The image location must be accessible to the Image Service. Compressed image binaries are supported (.zip and .tar.gz.)
Image Source	Image File	Please note: The Image Location field MUST be a valid and direct URL to the image binary. URLs that redirect or serve error pages will result in unusable images.
Image File ⓘ	Choose File CentOS-6-x8...1508.qcow2	
Format *	QCOW2 - QEMU Emulator	
Architecture		

Leave all of the other inputs blank. Also, since this is an image in the demo project, and we will provision into the demo project, you don't need to check the **Public** check-box, but if you want this image to be available to any other projects, check the public check-box. Now click on the **Create Image** button, and we are ready to provision an OpenStack instance with Chef. Let's give it a try.

Provisioning a new instance

Now that we finally have everything configured for OpenStack provisioning, let's see what our first knife OpenStack provisioning command will look like. Here is a command that I used to provision a new OpenStack instance (and deploy chef-client onto to it and so on):

```
knife openstack server create \
--availability-zone "nova" \
--image "centos6" \
--flavor "m1.small" \
--sudo \
```

```
--bootstrap-protocol ssh \
--ssh-user "centos" \
--identity-file "/Users/earlwaud/chef-repo/admin.pem" \
--openstack-ssh-key-id admin \
--openstack-floating-ip \
--run-list "recipe[base]" \
--node-name api_vm01 \
-v
```

There are a lot of parameters in that command, so let's talk about each one. By the way, if you want to see the full list of parameters for this create command, ask for help with this command:

```
knife openstack server create --help
```

Ok, so here is what's going on:

- `knife openstack server create`: This is telling knife to use the newly installed openstack plugin and to issue a create server command. The `server create` command will talk to OpenStack, and using the API information we added to the `.chef` file, will execute a "Launch Instance" command. Several of the parameters in our knife command are used to provide the inputs for the launch instance command.
- `availability-zone "nova"`: This parameter tells OpenStack that we want our instance to go into the "nova" zone. In our dev environment, we only have the single zone, but if this were a production deployment, we would have multiple zones, which would provide our physical fault isolation. The objective would be to deploy redundant instances in different zones to provide protection against physical failure of the hosting resources.
- `image "centos6"`: This is the name of the image from which we want to build our instance. This is the image we installed in the earlier section.
- `flavor "m1.small"`: This tells OpenStack what resources we want our instance to have. This sets the number of vCPUs, the amount of memory, the disk size, and so on. The configured "Flavors" can be found in your OpenStack dashboard in the **Admin** accordion menu on the left-hand side.
- `Sudo`: This tells knife to execute the node bootstrap using sudo.
- `bootstrap-protocol ssh`: This instructs the knife command to use ssh to connect to the new instance when it bootstraps the node.

- `ssh-user "centos"`: This is the username to use when executing the ssh command for bootstrapping the node. All of the images provided on the OpenStack image pages use an OS distro-related user name. For CentOS images, the user name is `centos`, for Ubuntu images, it is `ubuntu`, and for the cirros images, the username is `cirros`.
- `identity-file "/Users/earlwaud/chef-repo/admin.pem"`: This parameter is providing the location on your Chef workstation to the key file that will be used for authentication during the connection of the ssh session.
- `openstack-ssh-key-id admin`: This parameter tells OpenStack the name of the key pair to use when deploying the instance thus providing the access via key authentication in the ssh command. (There is no password access provided for the OpenStack instances deployed.)
- `openstack-floating-ip`: This parameter is telling OpenStack to associate one of the allocated floating IP addresses to the provisioned instance. This IP address will provide the public access to the node. It is an address on the public network interface. In our environment, the instance is provisioned with an IP address in the 10.0.0.x subnet on a private network interface. The floating IP is created in the public subnet (for our example, that is in 192.168.157.250 - 192.168.157.254). OpenStack is configured to with a virtual router to pass traffic between the private and public networks.
- `run-list "recipe [base]"`: This parameter is telling knife to use the run list "base" when it does the initial bootstrap chef-client run.
- `node-name api_vm01`: This parameter is telling OpenStack to name the new instance `api_vm01`. This will also be the name of the node on the Chef Server.
- `v`: Finally, this parameter is telling the knife command to provide verbose debugging output during the execution of the command.

 Note that it is the only parameter we used that has a single dash. You can increase the level of output to debug by adding a second V, making the parameter `-VV`.

Here is a look at the command with its output:

```
knife openstack server create \
> --availability-zone "nova" \
> --image "centos6" \
> --flavor "m1.small" \
> --sudo \
```

```
> --bootstrap-protocol ssh \
> --ssh-user "centos" \
> --identity-file "/Users/earlwaud/chef-repo/admin.pem" \
> --openstack-ssh-key-id admin \
> --openstack-floating-ip \
> --run-list "recipe[base]" \
> --node-name api_vm01 \
> -V

INFO: Using configuration from /Users/earlwaud/chef-repo/.chef/knife.rb

Waiting for server [wait time = 600]....
Instance ID      af367e7f-689a-4c1e-b086-da07da75e03b
Name             api_vm01
Private IP       10.0.0.12
Flavor           2
Image            2135dc5e-9c5a-473d-ace4-f1477a076836
Keypair          admin
State            ACTIVE
Availability Zone nova
Private IP Address: 10.0.0.12
[fog] [WARNING] Unrecognized arguments: openstack_identity_endpoint
Floating IP Address: 192.168.157.252
Private IP Address: 10.0.0.12
Bootstrapping the server by using bootstrap_protocol: ssh and image_os_type: linux

Waiting for sshd to host (192.168.157.252).....done
Doing old-style registration with the validation key at /Users/earlwaud/chef-repo/.chef/sdearl-validator.pem...
Delete your validation key in order to use your user credentials instead

Connecting to 192.168.157.252
192.168.157.252 ----> Installing Chef Omnibus (-v 12)
192.168.157.252 downloading https://www.opscode.com/chef/install.sh
. . .(truncated). . .
```

There you have it. You can now provision new OpenStack instances using the knife-openstack plugin. Go forth and provision!

References

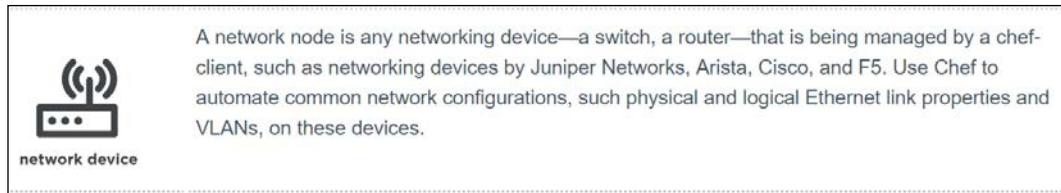
- Learn more about DevStack at this link <http://docs.openstack.org/developer/devstack/>.
- Download the Ubuntu Desktop install bits at <http://www.ubuntu.com/download/desktop>.
- Follow the knife-openstack plugin repo at <https://github.com/chef/knife-openstack>.
- Follow the status of the "Forbidden error" at <https://bugs.launchpad.net/devstack/+bug/1243075>.
- Download your OpenStack instance images from <http://docs.openstack.org/image-guide/obtain-images.html>.
- Read about the OpenStack configuration input settings at <http://docs.openstack.org/developer/devstack/configuration.html>.
- There is a nice OpenStack command cheat sheet http://docs.openstack.org/user-guide/cli_cheat_sheet.html.

Network automation

The **Software-defined Networking (SDN)** technology is still in its infancy. Since it is still evolving, it means different things to different people. It is like what the idea of Cloud Computing used to be; in the past, if you asked several different people what Cloud Computing means, you would have received several very different answers. Now, we have refined the definition of Cloud Computing and developed the specific technologies of **Platform as a Service (PaaS)**, **Infrastructure as a Service (IaaS)**, and **Software as a Service (SaaS)**.

Like Cloud Computing, SDN comprises many different technologies such as OpenFlow, Virtual Switching, and Data Center Network Fabrics. Because of this, there is no single answer to the question "What is Network Automation?". Every company has a unique network configuration (although with common elements), so every company will have a different need for their network automation. In fact, with the proper attention, this topic could be made into its own, thick, book.

So what can we cover in a few pages that will be valuable to the DevOps, or in this case the NetOps engineer? Let's work from Chef.io's point of view. They consider network devices as any other node that can be managed. Here is how Chef.io describes it:



OK, so let's take a look at how we turn the configuration of some of the more common data center elements into code with Chef. We will start with Juniper devices. Then, we'll explore Cisco devices, and finally, we'll learn about using Chef with F5 LTMs. Let's begin.

Automating juniper devices

As an example, let's look at how to manage a common data center switch from Juniper, the QFX5100 switch. This switch supports virtualized network environments so it will stand as a good representation for our purposes. If you are using the Junos Enhanced Automation software, the image comes with a Chef agent preinstalled. It is a ruby program that uses the NETCONF libraries.

The QFX5100 switch can be treated as a node for the Chef Server to manage. And with the built-in chef-client, we can converge the switch to our desired state. The first step is to register the node with the Chef Server. There is a specific bootstrap process to do this registration for us. You will need a specific bootstrap file for this process. You can use wget to download the bootstrap file with the following command:

```
wget https://github.com/opscode/junos-chef/blob/master/bootstrap/junos-minimal.erb
```

Once you have the bootstrap file, you will execute the knife bootstrap command. It will take the following format:

```
knife bootstrap <SWITCH IP ADDRESS> --template_file junos-minimal.erb -x root
```

For example, if the IP address of your switch is 172.0.0.10, you would use the command:

```
knife bootstrap 172.0.0.1 --template_file junos-minimal.erb -x root
```

You will be prompted for the root password, and when entered, the node will be bootstrapped and registered with the Chef Server. OK, now we have a new node, and we want to configure it for our environment. We are going to employ the cookbook netdev to handle the configuration of our switch. We need to add the cookbook to our repo. You can do this with the following commands:

```
knife cookbook site download netdev
tar zxvf ./netdev-2.0.0.tar.gz -C cookbooks
mkdir ~/chef-repo/cookbooks/netdev/recipes
```

Now you need to create the recipes you will use to configure the switch. Here are some example recipes. Create a `create_vlan` recipe, as follows:

```
vi ~/chef-repo/cookbooks/netdev/recipes/create_vlan.rb

netdev_vlan 'public' do
  vlan_id 100
  description 'public'
  action :create
end
netdev_vlan 'nonpublic' do
  vlan_id 200
  description 'nonpublic'
  action :create
end
```

Create a `create_access` recipe , as follows:

```
vi ~/chef-repo/cookbooks/netdev/recipes/create_access.rb

netdev_interface 'xe-0/0/14' do
  description 'access interface 14'
  action :create
end
netdev_interface 'xe-0/0/15' do
  description 'access interface 15'
  action :create
end

netdev_l2_interface 'xe-0/0/14' do
  description 'belongs to public VLAN'
  untagged_vlan 'public'
  vlan_tagging false
  action :create
end
```

```
netdev_l2_interface 'xe-0/0/15' do
  description 'belongs to nonpublic VLAN'
  untagged_vlan 'nonpublic'
  vlan_tagging false
  action :create
end
```

Create a `create_uplink` recipe, as follows:

```
vi ~/chef-repo/cookbooks/netdev/recipes/create_uplink.rb
```

```
netdev_l2_interface 'xe-0/0/10' do
  action :delete
end
netdev_l2_interface 'xe-0/0/11' do
  action :delete
end

netdev_lag 'ae0' do
  links ['xe-0/0/10', 'xe-0/0/11']
  minimum_links 1
  lacp 'active'
  action :create
end

netdev_l2_interface 'ae0' do
  description 'Uplink interface'
  tagged_vlans ['public', 'nonpublic']
  vlan_tagging true
  action :create
end
```

Now that you have the new recipes, you need to upload them to the Chef server with the following commands:

```
cd ~/chef-repo
knife cookbook upload netdev
```

Now you need to update the Chef server run list for the node that represents the switch. You can do this via the Chef server UI or using the `knife node edit` command. Edit the run list to include the new recipes you created earlier.

If you are using the `knife` command, you would enter the following:

```
knife node edit qfx5100
"run_list": ["recipe[netdev::create_vlan]", "recipe[netdev::create_access]", "recipe[netdev::create_uplink]"
```

Next, we need to update the switch, running the chef-client to have the device's configuration converged to our desired state. To do this, you need to log in to the switch and run the `ruby` command to execute the chef-client. Here is the command:

```
/opt/sdk/chef/bin/ruby /opt/sdk/chef/bin/chef-client -c /var/db/chef/client.rb
```

This will make the switch download the netdev cookbook and apply the recipes you added to the node's run list. This is the similar process that you are used to doing with your compute nodes, so it should seem very familiar now.

Be mindful of time drift

Although compute nodes are sensitive to small time differences between their clock and that of the Chef server, this sensitivity is greater for switches. It is vital that all elements are using NTP and are syncing to the same source for their clocks. The slightest drift between the clocks on the switches and the Chef server will result in failure to converge issues.

Now, let's take a look at some examples of automating Cisco systems.

Automating Cisco systems

In a way very similar to the automation of Juniper devices, Cisco has added Chef support to their Cisco NX-OS. The installation of the chef-client can be done manually or through automation. In the interest of space, I am going to review a high-level automated method of installing the client. Cisco NX-OS supports three environment types: bash-shell, guest-shell, and Open Agent Container (OAC). The chef client can be run in any of the three but only one at a time. I will focus on guest-shell as it is a CentOS environment that is enabled by default on most platforms.

 Note that the automated Chef client installation is not currently available when using the OAC environment, and if you wanted to use chef-client in that environment, you would need to do a manual installation.

There are two gems needed for the chef-client installation: the `chef-provisioning` and the `chef-provisioning-ssh` gems. The first one is installed when you install ChefDK, and the second you will need to install yourself. Use the following commands:

```
cd ~/chef-repo
chef gem install chef-provisioning-ssh
```

Now on the Cisco device, you need to do some setup. Here are the steps:

```
# Enter the guestshell environment using the 'guestshell' command
guestshell

# If using the management interface you must enter the management
namespace
sudo su -
chvrf management

# Set up hostname and DNS configuration
hostname n3k

echo 'n3k' > /etc/hostname

cat >> /etc/resolv.conf << EOF
nameserver 10.0.0.202
domain mycompany.com
search mycompany.com
EOF

configure terminal
  feature ssh
  username devops password password role network-admin
  username devops shelltype guestshell
end
```

Of course, you will need to adjust the values for nameserver, domain, search, and so on to match your systems.

Now you need to create a Chef provisioning input file. We will go into detail on the use of the Chef Provisioning feature in *Chapter 8, Using Chef Provisioning*. For now, just create an input file named `chef_provisioning.rb` and add the following as the contents of the file:

```
require 'chef/provisioning/ssh_driver'

with_driver 'ssh'

with_chef_server 'https://chefserver.example.com/organizations/chef',
  client_name: 'chefuser',
  signing_key_filename: '/etc/chef/chefuser.pem'
```

```

with_machine_options transport_options: {
    ip_address: '10.100.100.1',
    username: 'devops',
    options: {
        # BASH-SHELL options - uncomment if installing to bash-shell
        # prefix: 'sudo ip netns exec management ',

        # GUESTSHELL options - uncomment if installing to guestshell
        prefix: '/isan/bin/guestshell sudo ip netns exec management ',
        scp_temp_dir: '/bootflash',
    },
    ssh_options: {
        password: 'password',
    },
}

machine "n3k_100_1.example.com" do
    action [:ready, :setup, :converge]
    # Copy the trusted certificate to the newly provisioned node
    file '/Users/earlwaud/chef-repo/.chef/trusted_certs/chefserver.crt'
    converge true
end

```

Again, you need to update the values to match your systems. Once you have done this, save the file. Now run the `chef-client` command:

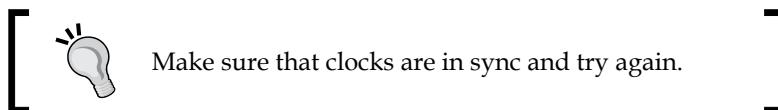
```
chef-client -z chef_provisioning.rb
```

This will install the chef-client into the guestshell environment of the Cisco device. However, you may encounter one of the following errors:

```
Unable to Connect ... or ... 401 "Unauthorized"
```

In the first case, you may have a stale SSH key in your `~/.ssh/known_hosts` file. You can remove it with the `ssh-keygen -R <hostname or IP>` command and then try again.

In the second case, you probably have a time sync issue between your system and the Cisco system.



Once the chef-client has been successfully installed into the guestshell environment of the Cisco device, you can begin to build recipes with the Cisco cookbook. Download and install the cookbook with these commands:

```
cd ~/chef-repo
knife cookbook site download cisco-cookbook
tar zxvf ./cisco-cookbook-1.1.2.tar.gz -C cookbooks
```

You are ready to create and edit recipes, upload them to the Chef server, and modify the run list for the Cisco devices in the same way you did for the Juniper devices described in the earlier section.

Now, let's take a look at automating F5 LTM's.

Automating F5 LTM's

There is an interesting article that discusses using Chef to automatically add web servers to an F5 LTM. It was written several years ago, and while the description of the process is accurate, the execution is no longer valid. We are going to re-engineer the process so that we can add this ability to our own Web Server role.

First, we need to download the cookbook from the article. Visit the link <https://devcentral.f5.com/articles/automating-web-app-deployments-with-opscode-chef-and-icontrol> and find the download link titled `f5-node-initiator.tgz` in the "installing the cookbook" section of the article. Click on the link and download the `tgz` file, saving it into your `~/chef-repo` folder. Now you can review the article for the description of what and why, but ignore the how, it is not relevant to today's versions of Chef. Here are the steps you need to set up and use this cookbook:

Start by extracting the cookbook. Use these commands:

```
cd ~/chef-repo
tar zxvf ./f5-node-initiator.tgz -C cookbooks
```

Unfortunately, the cookbook has a `metadata.rb` file that is missing an attribute for the cookbook name, which is required in current versions of chef. So, we need to edit the `metadata.rb` file to add that attribute:

```
cd ~/chef-repo
vi ./cookbooks/f5-node-initiator/metadata.rb
```

And a line at the top like this:

```
name          'f5-node-initiator'
```

Save the file. Now we need to upload the cookbook to our chef server. Use the following commands:

```
cd ~/chef-repo
knife cookbook upload f5-node-initiator
```

You can confirm the cookbook is available on the chef server with the `knife cookbook list` command. Now we need to create our web server role. Create or edit your `web_server.rb` file accordingly:

```
cd ~/chef-repo
vi ./roles/web_server.rb
```

Update the file contents as follows:

```
name 'webserver'
description 'The Web Server Role'
run_list([
  'recipe[base] ',
  'recipe[apache2] ',
  'recipe[f5-node-initiator] '
])
default_attributes({
  'bigip' => {
    'address' => '10.0.0.245',
    'user' => 'admin',
    'pass' => 'password',
    'pool_name' => 'chef_test_http_pool'
  }
})
override_attributes({
  'base_name' => 'Earl Waud',
  'user_name' => 'Earl Waud',
})
```

Replace the address, user, pass, and pool_name values with values appropriate for your environment. Save the file. Now we need to upload the role to our Chef server. Issue the following commands to upload the role:

```
cd ~/chef-repo  
knife role from file roles/web_server.rb
```

Next, we need to add the new role to our web server nodes. For example, if we have a web server node named web01, we would issue the following commands:

```
cd ~/chef-repo  
knife node run_list add web01 "role[webserver]"
```

The node web01 is now ready to re-run the chef-client and converge its configuration to that of our new webserver role that includes the f5-node-initiator.

The screenshot shows the Chef Web UI interface for a node named 'web01'. The top navigation bar has tabs for 'Details', 'Attributes', and 'Permissions', with 'Details' being the active tab. Below the tabs, there are three main sections: 'Last Check In' (5 Minutes ago, 2016-03-01), 'Uptime' (A Few Seconds, Since 2016-03-01), and 'Environment' (production). The 'Run List' section shows the node's configuration with four roles applied: 'base' (Version 0.1.0, Position 0), 'webserver' (Version ..., Position ...), 'apache2' (Version 3.1.0, Position 1), and 'f5-node-initiator' (Version 0.0.1, Position 2). The 'Tags' section indicates no items to display.

References

- Check out the `junos-chef` repo for Juniper switch configuration with Chef at <https://github.com/Juniper/junos-chef>.
- Learn about the Chef Junos integration at the Chef.io site at <https://docs.chef.io/junos.html>.
- Read about the QFX5100 switches from Juniper at <http://www.juniper.net/us/en/products-services/switching/qfx-series/qfx5100/>.
- There is the `netdev` repo on GitHub at <https://github.com/chef-partners/netdev>.
- There is the Cisco cookbook repo on GitHub at <https://github.com/cisco/cisco-network-chef-cookbook?files=1>.
- There is the manual chef-client installation instructions at <https://github.com/cisco/cisco-network-chef-cookbook/blob/develop/docs/README-agent-install.md>.
- There are the automated chef-client installation instructions at <https://github.com/cisco/cisco-network-chef-cookbook/blob/develop/docs/README-chef-provisioning.md>.
- Learn about the Chef to NX-OS integration at http://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/7-x/programmability/guide/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_7x/b_Cisco_Nexus_9000_Series_NX-OS_Programmability_Guide_7x_chapter_01110.html.
- Read an excellent article on automating WebApp deployments with Chef at <https://devcentral.f5.com/articles/automating-web-app-deployments-with-opscode-chef-and-icontrol>.
- Take a deep look at the F5 Big IP cookbook that was released into open source by Target: <https://github.com/target/f5-bigip-cookbook>.
- Download the F5 cookbook from the Chef supermarket at <https://supermarket.chef.io/cookbooks/f5/versions/0.2.4>.

Summary

In this chapter, we focused on provisioning in the traditional data center. With the widespread use of VMware vSphere, it is appropriate that we started with using the knife-vsphere plugin to deploy VMs into our vSphere environment. You learned how to provision and manage vSphere VM's configuration with Chef using this plugin. We also covered setting up a personal development environment using VMware's desktop hypervisors and Vagrant. We also provided references for you to do your own homework on several other VMware-related knife plugins, and with the information you have learned so far, you should be able to figure out how to make them work in your environment. Next, we took a look at provisioning into OpenStack environments. With the strong growth of OpenStack deployments and the ever-growing list of developers backing this technology, having a good foundation for provisioning into OpenStack is vital. You definitely have that knowledge now. And following the track of having a place for your development work, you learned how to stand up your own OpenStack development environment. Next, we took a dip into the waters of a NetOps engineer, exploring some areas for network automation. What a journey! This chapter is so full of information that you might have to read it twice!

However, if you are ready for more, then next up, we turn our attention to the clouds with a look at provisioning in Amazon AWS and other cloud based hosting providers.

6

Provisioning in the Cloud

This chapter will expand the previous chapter and explore the use of Chef for deployment and configuration in the cloud. Much like VMware is the 600 pound gorilla in the data center, Amazon is the 600 pound gorilla in the cloud. Amazon is the first provider that comes to mind when people ask about the cloud, and it has the biggest market share, at least so far. They currently host about 30 percent of cloud-based systems, which is a larger percentage than the next three providers combined. With that kind of dominance in the cloud market, you might ask why learn how to provision to any other cloud provider? Well, because Microsoft and Google are currently the fastest growing providers. Microsoft has already grabbed about 12 percent of the market share and is growing at a much faster rate than Amazon. And not one to be out done, Google is not too far behind Microsoft. Let the cloud provider wars begin! So, in this chapter, the reader will learn about provisioning to all three of the top cloud providers. This will help them to manage hybrid solutions for scale and high availability with whichever provider best meets their business needs. We'll begin by expanding the work we did in *Chapter 2, Knife Tooling and Plugins*, with provisioning EC2 instances. Then, you will learn how to leverage Chef to provision into Windows Azure and into Google Cloud Platform. The chapter will end with a look at using Chef to provision containers using Docker.

"I like rice. Rice is great if you're hungry and want two thousand of something."

– Mitch Hedberg

The following topics will be covered in this chapter:

- Provisioning AWS EC2 instances
- Provisioning Microsoft Azure instances
- Provisioning in the Google Cloud Platform
- Looking at an honorable mention – Linode
- Provisioning containers in Docker

Provisioning AWS EC2 instances

In *Chapter 2, Knife Tooling and Plugins*, you explored installing the knife-ec2 plugin and learned the knife configuration settings required to provision AWS EC2 instances. Now you are going to expand on this base of knowledge and learn more about AWS authentication. Then, we are going to see how easy it is to set up a Chef Server in an EC2 instance.

 Remember that you can set up an AWS account on Amazon and use the service for free, as long as you stay within the "Free Tier Eligible" guidelines. I will note where any of the examples go outside of the free services.

Understanding EC2 authentication

As we saw in *Chapter 2, Knife Tooling and Plugins*, you need to configure knife to allow it to use the EC2 plugin to create AWS EC2 instances. You can either add the credentials directly to your `knife.rb` file, or better, create an AWS configuration file with the account credentials in it and point your `knife.rb` file to that credentials file. The contents of the credentials file will look something like this:

```
cat /Users/earlwaud/.aws/credential_file
[default]
aws_access_key_id = TR42UIIKKN73BSROPZGA
aws_secret_access_key = eZMk1OXH2Ish056IkH50fLg7TQ9U2pDHzOh9OUF9
```

Then, you will add the pointer to the credentials in your knife file with a line like this:

```
knife[:aws_credential_file] = '/Users/earlwaud/.aws/credential_file'
```

Okay, we can create our EC2 instances with the knife-ec2 plugin. What else do we need? There are two more authentication areas we need to explore: the first is creating security key pairs, and the second is creating networking security groups. Let's take a deeper look into these two topics.

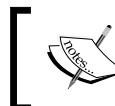
Creating key pairs

Access to the EC2 instances is handled using public key cryptography that encrypts and decrypts the login information. This method uses a public key to encrypt data like passwords. Then, the private key can be used to decrypt that data. These public and private key combinations are known as key pairs.

To log in to an Amazon EC2 instance, you need to provide the private key. The Linux-based instances are created without passwords, and the only access is via the use of the key pairs. Windows-based instances also use key pairs. But for Windows, you use the private key to obtain the administrator password, which can then be used to access the instance via RDP.

You can create your key pairs through the Amazon user interface, or generate your own key pairs, and upload the public key to Amazon where it can be injected into your created instances. Here are the steps to creating your key pair using Amazon:

1. Browse the Amazon compute EC2 console and login.
2. Select the region that will use the key pair. For me, this is US West (Oregon).



It is important to remember that key pairs are only valid in the region where they are created or uploaded.

3. Expand the **NETWORK & SECURITY** accordion in the navigation pane on the left and select **Key Pairs**.
4. Click on the **Create Key Pair** button.
5. In the window that pops up, enter the name you want to use for your new key pair and then click on the **Create** button.
6. The private key file will be downloaded automatically via your browser. The filename will match the name you give the key pair and will have the extension of .pem.



This is the only time you will be given the private key file. Save it in a location that is persistent. I saved mine at the same location as my AWS credentials file, which is at /Users/earlwaud/.aws/credential_file. It is a real big pain if you lose a private key and need to access an instance that was created with the matching public key. You can find documentation for dealing with this on the Amazon site, but trust me, it's not fun. Keep your private keys safe and accessible.

7. In most use cases, as the last step, you need to change the permission on the private key file to be more restrictive. For example, to use the key with ssh, you must lock down the access to it. Use the command `chmod 400 ~/aws/us-west-oregon-earl01.pem`.

If you are accessing EC2 instances using PuTTY on your Windows system, you need to generate a private key file in a format that can be used by PuTTY. When you install PuTTY, there is an additional tool named PuTTYgen that is installed as well. PuTTYgen can be used to create an SSH-2 RSA version of your private key file, which PuTTY can use to access your instances. If you've never done this before, you can find the full instructions in the Amazon documentation. I will provide the direct link to that document in the references section later. You're now ready to use key pairs to create and access your new EC2 instances.

Creating security groups

Now, let's take a look at creating security groups. The EC2 security group is essentially a firewall for your instances. Each instance you create needs to have one or more security groups applied to provide the access rules for the instance. You can create a security group that has many rules for inbound or outbound traffic, or you can create multiple security groups where each has a specific rule for traffic, such as a rule to allow ICMP traffic. This multiple security group method is a better choice, allowing you to create protocol-specific security groups that can be applied on an as needed basis. If your application needs SSL access, you can create a security group for HTTPS and apply that group to your instance. Now it's very clear what traffic is permitted in or out of your systems.

For the examples in this book, I am going to describe the steps to create and configure a single security group with very open access. In a real production application, you will want to configure your security groups with very specific and limited access, allowing only the network traffic necessary to use your applications. With that detail noted, here are the steps to create the new security group, which we will use in the next section to deploy an EC2 instance for our new Chef Server.

1. Browse the Amazon compute EC2 console and login.
2. Select the region that will use the security group. For me, this is US West (Oregon). As with key pairs, security groups are only valid in the region where they are created.
3. Expand the **NETWORK & SECURITY** accordion in the navigation pane on the left and select Security Groups.
4. Click on the **Create Security Group** button.
5. In the popup window, enter a name and description.
6. Click on the **Add Rule** button one or more times and create the rules for the new group. In this security group, I added a rule for ICMP, SSH, HTTP, and HTTPS.
7. Once all the rules have been added, click on the **Create** button to create the new security group.

Take a look at the security group I created for the example in this chapter:

Type	Protocol	Port Range	Source
All ICMP	ICMP	0 - 65535	Anywhere
SSH	TCP	22	Anywhere
HTTP	TCP	80	Anywhere
HTTPS	TCP	443	Anywhere

You can see in the preceding image that I added four rules to my security group—one to allow ping traffic, one to allow SSH traffic, and one each for HTTP and HTTPS. Remember that it would be a better idea to create individual security groups for each protocol and then add those groups to your instances only if that protocol is needed for your application. Also note that I left the source of all these inbound rules to be wide open, allowing access from any IP address. You really don't want to do this for a production environment if you can avoid it. Of course, if the system is hosting a public web server, you may need to have an open access rule for HTTP or HTTPS, but you certainly wouldn't want to have SSH wide open like this.

Also, each security group has a default outbound rule that allows all network traffic to flow out of the instance. This too can be narrowed to limit what can exit your systems and should be modified to achieve a hardened environment.

Now that you have your AWS credentials setup to use knife to create EC2 instances and you've created a key pair to use for accessing those instances, and you have created your security group to control the network traffic to and from your instances, let's create a new instance. This time, we are going to use knife, along with our existing public Chef Server, to deploy and configure a new Chef Server in an EC2 instance.

Installing Chef Server into an EC2 instance

There are two ways you can arrive at the destination of having your Chef Server running in an EC2 instance. The first and easiest way is to just "purchase" a Chef Server AMI from the AWS Marketplace. There are different levels of the Chef Server AMIs based on the number of nodes that you plan to manage, starting at 25 nodes, all of which currently offer a 14-day free trial (which, by the way, is not exactly free, you still pay for the AWS infrastructure charges, just not for the hourly software charges). These Chef Server instances all come with an easy way to add additional Chef.io offerings, such as Chef Analytics, Management Console, and Chef Compliance, at an additional software cost, of course.

The second way, which is the way we are going to examine further, is to stand up the Chef Server in an EC2 instance ourselves. We are going to download the chef-server cookbook and its dependent cookbooks from the Chef marketplace. Then, we are going to upload the cookbooks to our current publicly accessible Chef Server. Then, we are going to use the knife-ec2 plugin to create our new instance and converge its configuration to that of a Chef Server. Finally, we are going to use some of the commands you learned in *Chapter 1, Setting Up a Development Environment on Your Workstation*, to round out the setup of our Chef Server. This sounds like a lot to do, but you'll see that it's not too bad. Let's start.

The first step is to download the chef-server cookbook and its dependencies. To do this, we are going to issue the following commands:

```
cd ~/chef-repo  
knife cookbook site install chef-server
```

This knife cookbook command will go out to the Chef supermarket and will download the chef-server cookbook archive file. It will extract the contents of that archive into our cookbooks folder and then delete the downloaded archive file. It will then look at the dependencies for the chef-server cookbook, and in turn, download, extract, and delete the archive for each of the dependent cookbooks. This process will be repeated for each of the dependent cookbooks until all the cookbooks necessary to use the chef-server cookbook have been added to our local repo.

Depending on the state of your repo, you may receive an error message telling you that you need to commit your exiting changes before installing new cookbooks. If you get this message, use the git add and git commit commands to "tidy up" your local repo as needed and then run the knife cookbook command again.

Now that we have the chef-server cookbook and all of its dependencies in our local repo, we need to upload them into our publically accessible chef server. In my case, I am using the free version of Chef Server available at <http://manage.chef.io>. Yes, this is a kind of recursion – we are using a Chef Server to deploy a Chef server. (I have a T-shirt that has a circle of text on the front that says "In order to understand recursion, one must first understand recursion." It is my favorite T-shirt.). To upload our cookbooks to the Chef Server, we will issue the following command:

```
cd ~/chef-repo
knife cookbook upload chef-server --include-dependencies
```

Now we are ready to create out new EC2 instance and converge it into a Chef Server. If you haven't done so yet, install the knife-ec2 plugin now with the following command:

```
chef gem install knife-ec2
```

Before we create our instance, let me point out that you cannot use the t2.micro flavor for your Chef Server instance. Chef Server's minimum memory requirements as stated in the Chef documentation are 4 GB. I have found that you can stand up a Chef Server (for testing and dev) using only 2 GB, which is configured when you use a t2.small flavor. The Chef Server cookbook will not converge successfully if you try to use it in a 1 GB server, which is what you get when you use the t2.micro flavor. So, t2.small is the smallest working environment flavor you have for testing this out. This is of importance because the t2.small flavor is not "Free Tier Eligible", and you will be incurring infrastructure charges when you deploy a t2.small instance. With that advisory said, here is the knife-ec2 command I used for deploying the EC2 instance I'll use for the Chef Server:

```
cd ~/chef-repo
knife ec2 server create \
--node-name ec2-chef-server \
--groups=chef-provisioning \
--region=us-west-2 \
--availability-zone=us-west-2a \
--image=ami-9abea4fb \
--server-connect-attribute=public_ip_address \
--flavor=t2.small \
--ssh-user=ubuntu \
--ssh-key=us-west-oregon-earl01 \
--identity-file=/Users/earlwaud/.aws/us-west-oregon-earl01.pem \
--run-list="recipe['chef-server']" \
-v
```

This will take a little while to complete, but when it is done, the new EC2 instance will be a registered node on your public-facing Chef Server, the chef-server cookbook will have been installed, and the instance will be ready for you to ssh into, to finish up the configuration. Of course, you could have added additional recipes to the run-list and completed some or all of the next steps, but for the sake of space in this chapter, I am going to just list out the steps manually a little later. Right now, let's review the server create command we used to provision our Chef Server;

- `knife ec2 server create`: This is the ec2 server create subcommand.
- `--node-name ec2-chef-server`: This parameter provides the name to use to register the node on the Chef Server. In this example, our node will be registered as "ec2-chef-server". This value will also be used for the instance name in AWS.
- `--groups=chef-provisioning`: The value used in this parameter is the name of the security group you have created in AWS. Remember that the security group provides the firewall settings for your instances. This time, we are referencing the security group "chef-provisioning" that we created earlier.
- `--region=us-west-2`: This parameter identifies the target region for our provisioning. This is one of the parameters that we can set in our knife.rb file so that we wouldn't need to add it to our commands like we did here. This VM will be deployed in the "us-west-2" region.
- `--availability-zone=us-west-2a`: Each region has multiple zones to provide for fault tolerance. This parameter specifies which of the zones to deploy into. In our example here, we are using the "us-west-2a" zone.
- `--image=ami-9abea4fb`: Here, we provide the name of the template image that we want to deploy. You can find the details of all of the available AMI images in the AWS Marketplace. The link for the Marketplace will be in the references section later. An easy way to find AMIs that are available in your target region is to log in to the AWS console. Select the region you are interested in. Then, select "AMIs" from the images navigation pane.



Remember to select "Public Images" in the search bar to see the images that are in the AWS Marketplace. In this example, we are using the AMI named "ami-9abea4fb". This is an Ubuntu 14.04 server image that uses the EBS root device type.

- `--server-connect-attribute=public_ip_address`: This parameter tells Knife that we want to have a public IP address assigned to our instance.

- `--flavor=t2.small`: The flavor parameter provides the resource size to use when provisioning this instance. The "t2" flavors are of general purpose, and "small" indicates that the VM will have 1 vCPU and 2 GB memory. Other sizing options include nano, micro, medium, and large. Note that most of the flavors are outside of the free tier subset, so be aware of this when selecting your flavors.
- `--ssh-user=ubuntu`: With many public AMIs available in the Marketplace, especially the OS-specific AMIs, access is granted with a specific username. In this case, we are using an Ubuntu server AMI, so the username is "ubuntu".
- `--ssh-key=us-west-oregon-earl01`: This parameter provides the name of the key pair we set up in AWS.
- `--identity-file=/Users/earlwaud/.aws/us-west-oregon-earl01.pem`: This parameter's value is the local path and filename for the key file associated with the key referenced in the `ssh-key` parameter.
- `--run-list="recipe['chef-server']"`: With this parameter, we provide the run-list to be used in the initial chef-client run. It is the state to which we wish to converge this node.
- `-v`: This is the standard verbose output option parameter. We can increase the level of output with a second V, like `-VV`.

Now that we understand the parameters used, here is a snippet of the output from the server create command used earlier. Make note of the Public DNS Name when you run your command, since we will use that to access the instance when it is deployed.

```
Waiting for EC2 to create the instance.....
Public DNS Name: ec2-54-201-38-94.us-west-2.compute.amazonaws.com
Public IP Address: 54.201.38.94
Private DNS Name: ip-172-31-18-207.us-west-2.compute.internal
Private IP Address: 172.31.18.207

Waiting for sshd access to become available.■
```

Let's take care of the final steps needed to turn our EC2 instance into a full-fledged Chef Server. The first of these final steps is to ssh into our new Chef Server. Remember that we need to use the private key that we created earlier along with the Public DNS name of the newly deployed instance.

Since we deployed an Ubuntu instance, the user name we need for our ssh connection is "ubuntu". This is the same user name that we used in the knife command earlier. If we had used a different OS provider for our EC2 instance, say CentOS, then we would be using a different username, that is, "centos" for these commands. And as you can see from the preceding image, the public DNS name is ec2-54-201-38-94.us-west-2.compute.amazonaws.com. To ssh into our new system, issue the following commands:

```
cd ~/chef-repo
ssh -i ./aws/us-west-oregon-earl01.pem ubuntu@ec2-54-201-38-94.us-
west-2.compute.amazonaws.com
```

You will receive a message about not being able to establish the authenticity of the host (a very normal ssh message). Enter yes to continue, and you will be logged into your new instance. Now we will issue the last few commands (these are much the same as we saw in *Chapter 1, Setting Up a Development Environment on Your Workstation*) to finish our Chef Server setup. Here are the commands:

```
sudo chef-server-ctl user-create earlwaud Earl Waud earl@sandiegoearl.com
MyPassword99 --filename ~/earlwaud-ec2.pem
sudo chef-server-ctl org-create sdearl "San Diego Earl" --association_
user earlwaud --filename ~/sdearl-validator-ec2.pem
sudo chef-server-ctl install chef-manage
sudo chef-server-ctl reconfigure
sudo opscode-manage-ctl reconfigure
```

You will want to download the two Chef Server .pem files that are created with the preceding commands, so they can be used later to work with this new Chef Server, but other than this, we are done. You can open up a web browser now and visit the URL for your new server. In my case, it is <http://ec2-54-201-38-94.us-west-2.compute.amazonaws.com>. You will be presented with the login for your Chef Server. Enter the username and password you provided in the user create command shown earlier and you are good to go.

Now, let's turn our attention to what Microsoft has been cooking up in the clouds with their hosting solution, Microsoft Azure.

References

- Read about the partnership between Chef.io and Amazon AWS at <https://www.chef.io/solutions/aws/>.
- Visit the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
- You can find the knife-ec2 plugin on github at <https://github.com/chef/knife-ec2>.

- Explore the many AMI images available to provision from in the AWS Marketplace at https://aws.amazon.com/marketplace/ref=csl_ec2_ami.
- Find the details for the Chef Server-specific AMIs offered on the AWS Marketplace at https://aws.amazon.com/marketplace/pp/B010OMO0UE/?ref=_ptnr_chef_website#product-details.
- Learn how to connect to your EC2 instances using PuTTY at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html?icmpid=docs_ec2_console.
- Amazon's documentation on key pairs is at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html?icmpid=docs_ec2_console.
- Amazon's documentation on security groups is at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs_ec2_console.
- The chef-server cookbook repo on Github is at <https://github.com/chef-cookbooks/chef-server>.
- Find all the available flavors that can be used to configure your instance's resource sizing at <https://aws.amazon.com/ec2/instance-types/>.

Provisioning Microsoft Azure instances

Now that we have a solid understanding of using Chef to turn our EC2 cloud infrastructure into code, it is easy to deploy and manage. Let's see how we can do that using Microsoft's cloud hosting provider service, Azure. As I mentioned earlier in the chapter, Azure is the fastest growing cloud provider, currently with about 12 percent of the cloud hosting market. As with Amazon, Microsoft offers a free trial of the service, so you can test out how things work for you with a real-life proof of concept.

At the time of writing this chapter, Microsoft Azure has multiple personalities. By this, I mean that Microsoft has created a newer fresh version of their Azure user interface, that is, Portal. Portal has a new domain name and a new cleaner, customizable UI. Along with this new UI, there is a new deployment model named Resource Manager. You may see this referred to as Azure Resource Manager. The older UI and deployment model, Azure Service Management, is now branded as Classic. Here are the URLs to access both versions of the UI:

- **Classic:** <http://manage.windowsazure.com/>
- **Portal:** <http://portal.azure.com/>

The newer Portal UI allows you to provision "Classic" VMs and will manage those VMs as well as any that are provisioned using the actual Classic UI. However, only Classic VMs (those created with the "Classic" setting using the Portal UI or those created using the Classic UI itself) are currently manageable using Chef. You cannot see or act on the Azure Resource Manager VMs created using the Portal UI.

[ So far, you can access both versions of the user interface when you create an account using either. So if you've already created an account via the Portal UI, you can still use all of the following information.]

With that framing, if you haven't done so yet, using either of the UIs, create your free trial account now, and let's explore Azure together.

Installing the knife-azure plugin

We begin, like we have several times before, by installing the plugin that will let us use Knife to work with a hosting provider, in this case, Microsoft Azure.

The current release of the knife-azure plugin is version 1.5.0. There is a release candidate of the plugin, version 1.6.0.rc.0, that was released in November 2015. This release candidate version is documented as the first version to support the newer resource model available via the Portal UI. The release candidate version provides access to the new Azure Resource Manager resource model with an additional unique keyword (azurerm), so your commands would start with:

```
knife azurerm ...
```

The keyword for working with the Classic model (azure) has not changed in the release candidate version, so those commands would still start with:

```
knife azure ...
```

Because the plugin will use separate keywords for each API, you can only execute in one model or the other in a single knife command.

Unfortunately, all of my attempts to get the knife azurerm commands to work have failed. Even the documentation says that the release candidate version of the plugin is not ready for production use. Because of this, the examples in this chapter will focus on Classic Azure and not Portal Azure. I am certain that in the near future, maybe by the time you read this book, the knife-azure plugin will get a new release version that also allows Chef Integration with the Portal Azure Resource Manager API to work.

Alright, let's install the plugin. You can easily install the plugin with the following commands:

```
cd ~/chef-repo
chef gem install knife-azure
```

Otherwise, if you want to give the release candidate version a try, use these commands:

```
cd ~/chef-repo
chef gem install knife-azure -v 1.6.0.rc.0
```

You can also use the more generic parameter:

```
cd ~/chef-repo
chef gem install knife-azure --prerelease
```

Either version works perfectly with the Classic model, so all of the commands will work as expected (except for the azurerm commands that I just pointed out earlier). If you are working in a production environment, I would strongly suggest that you stick with the latest release version and not use any release candidate versions).

By now, installing this plugin should seem very familiar and if you've been following along throughout the book, you probably could have installed it without looking at any of the commands shown earlier. Great!

Configuring the plugin

Now we are going to configure the knife-azure plugin. Configuring the plugin is also much like we have done previously. We are going to update the contents of our knife.rb file, adding a line for knife-azure. This time, however, the format is slightly different. This time, we need to have Azure generate a subscription file for us. We'll download the subscription file and save it in a safe location (I created a folder `~/.azure` to store the credentials file in so that name and location will be consistent with the name and location we used for AWS).

To have Azure create your credentials file, you need to visit <https://manage.windowsazure.com/publishsettings/index?client=xplat>. When you browse this link and you are already logged in to Azure, Microsoft will generate your subscription file and initiate the download automatically. If you are not logged in, you will be asked for your login credentials, and then you'll be redirected to the download page and your subscription file will download automatically. If the download does not start, you can click the link to restart the download. Note that you need to have your Azure account created for this to work. The webpage has a link to take you to the Sign Up page if you haven't done so yet.

Once you've saved and copied your downloaded subscription file into your safe place folder, you are ready to update your `knife.rb` file. I copied my subscription file to `~/.azure/credentials.publishsettings`. (You want to maintain the file type of `.publishsettings`, especially if you plan to use other tools like PowerShell to integrate with Azure.) Now, edit the `knife.rb` file and add the lines for the Azure plugin configuration:

```
cd ~/chef-repo  
vi .chef/knife.rb
```

Add the following lines:

```
#knife-azure  
knife[:azure_publish_settings_file] = '/Users/earlwaud/.azure/  
credentials.publishsettings'
```

Notice that I used the full path for the location/name of the subscription file. I found that if I use `~/azure...`, the plugin does not correctly find the credentials.

OK, we have our plugin and credentials installed, and we have our `knife` configuration file updated accordingly. Let's start working with Azure.

Verifying the configuration

We have our new `knife-azure` plugin ready to use. So, let's kick the tires on this thing. Start off with a simple command to list our exiting Azure hosted servers. Enter the following:

```
knife azure server list
```

Well, that was probably underwhelming, as you probably haven't created any Azure virtual machines yet. You should still have seen a column header line show up as a response to your command. It would look something like:

```
...  
DNS Name  VM Name  Status  IP Address  SSH Port  WinRM Port
```

This at least tells us that our plugin integration is working as desired. Now, let's try another command. This time, we want to list out all of the available images that we can use as a template to provision from. Try this command:

```
knife azure image list
```

Wow, that one gave us some output. You should have seen a very long list of images scroll across the screen. The information returned lists the name of each image, the base OS of each image, and the regions that each image can be deployed into. Much like AWS, many of the resources are associated to specific regions. Here are some of the possible regions you should see in your command results: Southeast Asia, Australia Southeast, West Europe, Central India, Japan East, Central US, East US, and South Central US. Currently, Microsoft provides Azure hosting services in 22 regional data centers.

Generating public/private key pairs

We now know that our knife-azure plugin is working, so it's almost time to use it to provision a new VM. We have one more step before we give provisioning a try though. We need to create a key pair to use for authentication with our Knife commands. I've found it, let's say "challenging" to get the authentication to work with username and password credentials, so I recommend you to focus on using key pairs for authentication and save yourself that headache. So, let's create a key pair for our Azure provisioning purposes. You can use the ssh-keygen command to create the key pair you need. Here is what it looked like when I created my key pair:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/earlwaud/.ssh/id_rsa): /
Users/earlwaud/.ssh/azure_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/earlwaud/.ssh/azure_rsa.
Your public key has been saved in /Users/earlwaud/.ssh/azure_rsa.pub.
The key fingerprint is:
7c:ed:c9:84:ee:7a:ea:48:65:00:d1:2a:ca:d3:89:c1 earlwaud@Earls-Mac.local
The key's randomart image is:
+--- [ RSA 2048] ---+
|   oo          |
|   ..          |
| .  ..         |
| E. .  o  o   |
| ...+..  S o o |
| ..+ o  o o + .|
|   .  .  . +    |
|     . . .      |
|     ..++.     |
+-----+
```

Provisioning virtual machines

OK, now that we have our key pair ready, let's create a new Azure compute resource. Here is the knife-azure plugin command to create a new server named earl-vm01:

```
knife azure server create \
--azure-dns-name earl-dns01 \
--azure-vm-name earl-vm01 \
--azure-source-image
112500ae3b842c8b9c604889f8753c3_OpenLogic-CentOS-71-20160308 \
--azure-service-location 'Central US' \
--ssh-user earlwaud \
--identity-file ~/.ssh/azure_rsa \
--bootstrap-protocol ssh \
--node-name earl-node01 \
--run-list 'recipe["base"]' \
-v
```

The parameters used here should look quite familiar to you because they are very much like the parameters used for our other knife plugin server creates. Let's review the parameters for certainty.

The first part is the command itself:

- `knife azure server create`: This is the normal knife plugin format for a create server command. The keyword is `azure`, and the subcommand is `server create`.
- `--azure-dns-name 'earl-dns01'`: This parameter provides the name that will be assigned as the public DNS name. The domain used by Azure is `.cloudapp.net`. This implies that you must create a unique name to use as the value for this parameter, as it will create a public DNS name that must not conflict with any other public names in the same domain. You can actually create more than one server that uses the same DNS name. This supports Azure availability groups. In our example, we will have a public name of `earl-dns01.cloudapp.net`.
- `--azure-vm-name earl-vm01`: This parameter supplies the friendly name of the server. It is what you will see when you use the `knife azure server list` command and must be of 15 characters or less.

- `--azure-source-image 112500ae3b842c8b9c604889f8753c3`
OpenLogic-CentOS-71-20160308: This parameter tells the plugin the name of the image file to provision our new system from. Again, you can get a full list of the available images with the command `knife azure image list`. I found it helpful to pipe the output into a `grep` command to "filter" the list. I use `knife azure image list | grep -i "centos-71"` to find the CentOS 7.1 images available.
- `--azure-service-location 'Central US'`: This parameter provides the name of the region to create the new system in.

 I believe that "West US" is not currently accessible with the `knife-azure` plugin. I would normally have selected that region, but due to the lack of availability for West US, I selected "Central US".

- `--ssh-user earlwaud`: This parameter tells `knife` what we want our username to be. In my case, I opted with the ever narcissistic "earlwaud" option.
- `--identity-file ~/.ssh/azure_rsa`: Do you remember the key pair we created earlier? This parameter is the path and name of that key file. This file will be injected into the new server so that the public key can be used to provide password-less access.
- `--bootstrap-protocol ssh`: This parameter tells `knife` to use the `ssh` protocol to bootstrap the new system. If we had selected a Windows OS-based image file to provision from, then we would have told `knife` to use the protocol of "winrm". You will usually use a value of "ssh" for deploying Linux systems.
- `--node-name earl-node01`: This parameter tells `knife` the name you want to register this new server as within your Chef Server. In this example, we will have a node named "earl-node01".
- `--run-list 'recipe['base']'`: This parameter provides the run-list value. As you already know, it supplies the list of recipes to converge the new system to. The value is a comma-separated list of roles or recipes to apply to the new system.
- `-v`: Finally, this is the verbose option, which can be made even more so with a second V as in `-vv`.

When the command completes, we will have a new compute resource hosted in Microsoft Azure named earl-vm01. This new system will be registered with our Chef server with the node name earl-node01 and be bootstrapped to the provided run list, in this case, "base". The system will be configured to allow key pair access using the username "earlwaud" and the public dns name of earl-dns01.cloudapp.net, and the access key in this case will be azure_rsa. So, we can ssh into our new system with the following command:

```
ssh -i ~/.ssh/azure_rsa earlwaud@earl-dns01.cloudapp.net
```

Provisioning a new Chef server into Azure

Well that was pretty easy, right? So let's try one more and see if we can deploy an Ubuntu image to step up a new Chef server in Microsoft Azure, as we did with our Amazon EC2 instance. We are going to need to change a few of the parameters in our server create command. First, we need an Ubuntu image name for the source image parameter. Then, we need to update the run-list to tell the command to converge on a Chef server configuration, and finally, we need to increase the resources allocated to the system because the default size of "Small" is too small for a Chef Server. With those changes identified, here is the new server create command:

```
knife azure server create \
--azure-dns-name earl-chef \
--azure-vm-name earl-chefvm01 \
--azure-vm-size Medium \
--azure-source-image b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-
LTS-amd64-server-20150123-en-us-30GB \
--azure-service-location 'Central US' \
--ssh-user earlwaud \
--identity-file ~/.ssh/azure_rsa \
--bootstrap-protocol ssh \
--node-name earl-chefvm01 \
--run-list 'recipe["chef-server"]' \
--tcp-endpoints 80,443,9683 \
-v
```

Let's take a quick look at the additional parameters we used this time:

- `--azure-vm-size Medium`: With this parameter, we are asking Azure to create our new system with the resources associated with a "Medium" environment (2 cores, 3.5 GB memory). The other options available as values for this parameter include ExtraSmall (shared core, 768 MB memory), Small (1 core, 1.75 GB memory), Large (4 cores, 7 GB memory), and ExtraLarge (8 cores, 14 GB memory).
- `--tcp-endpoints 80,443,9683`: If you recall from the AWS examples, you create Security Groups with permissions for different network protocols. The `tcp-endpoints` parameter is an Azure shortcut for this same type of feature. In this example, I am creating a new Chef server, and a Chef server needs the TCP ports 80, 443, and 9683 open and accessible to external connections. This parameter is configuring our new server to have that accessibility.

Executing the `server create` command given previously creates our Chef server successfully. This command does take a long time to complete, and at one point, it may seem to have frozen. Be patient. Allow it time to run to completion, and it will be successful. Note that if you use a larger `azure-vm-size` value, the execution will complete more quickly.

Okay, now we can go about our business of sshing into our new Chef server using the same format as the `ssh` command shown earlier:

```
ssh -i ~/.ssh/azure_rsa earlwaud@earl-chef.cloudapp.net
```

Then, we can complete the setup of our Chef server with these commands (which are nearly identical to the ones we used for our Chef server in the EC2 instance):

```
sudo chef-server-ctl user-create earlwaud Earl Waud earl@sandiegoearl.com  
MyPassword99 --filename ~/earlwaud-azure.pem  
sudo chef-server-ctl org-create sdearl "San Diego Earl" --association_  
user earlwaud --filename ~/sdearl-validator-azure.pem  
sudo chef-server-ctl install chef-manage  
sudo chef-server-ctl reconfigure  
sudo opscode-manage-ctl reconfigure
```

Our new Chef server is online and ready for us to log in at `https://earl-chef.cloudapp.net`. This is getting easy, right? Now, with this work done, we can begin to manage our infrastructure using our new Chef server hosted in our Azure cloud.

Several of the parameters used in the earlier server create command can be added as knife.rb configuration parameters. Setting them in the knife configuration file will allow you to issue your server create commands without need to supply the parameter. Here is a subset of the common parameters available for your knife.rb file. Remember to use fully qualified paths for filename configuration parameters:

```
knife[:azure_publish_settings_file] = '/Users/earlwaud/.azure/
credentials.publishsettings'
knife[:azure_dns_name] = 'earl-dns01'
knife[:azure_service_location] = 'Central US'
knife[:azure_source_image] = '112500ae3b842c8b9c604889f8753c3_'
OpenLogic-CentOS-71-20160308'
knife[:azure_vm_name] = 'earl-vm01'
knife[:ssh_user] = 'earlwaud'
knife[:identity_file] = '/Users/earlwaud/.ssh/azure_rsa'
knife[:tcp-endpoints] = '80,443,9683'
knife[:udp-endpoints] = '123:124'
```

Once we've finished using our new Azure compute resources and they are no longer desired, we can use the knife-azure plugin to delete the virtual machines and remove the nodes from our Chef server. Here is an example of a command to do this:

```
knife azure server delete earl-vm01 -N earl-node01 --purge -y
```

Now that we have the Chef chops to provision into the top two major cloud hosting providers, let's swing for a triple and learn how to provision into the Google Cloud Platform. Are you ready?

References

- Create a Microsoft Azure account using the Classic UI at <https://account.windowsazure.com/SignUp>.
- Or create your account using the Portal UI at <https://azure.microsoft.com/en-us/>.
- Learn about the Chef and Microsoft partnership at <https://www.chef.io/solutions/azure/>.
- Find the knife-azure plugin repo on Github at <https://github.com/chef/knife-azure>.
- Find the azure-cookbook repo on Github at <https://github.com/chef-partners/azure-cookbook>.
- Link to download (Classic UI) Azure PublishSettings file from Microsoft <https://manage.windowsazure.com/publishsettings/index?client=xplat>.

- If you want to do some DevOps work using Microsoft PowerShell and Azure, you can find the instructions at <https://azure.microsoft.com/en-us/documentation/articles/powershell-install-configure/>.

Provisioning in the Google Cloud platform

By now, you should be seeing the similarities between using Chef with Amazon and Microsoft, and as you might expect, the similarities don't end there. As you are about to see, we are going to do much the same as we explore provisioning and managing systems hosted on the Google Cloud Platform in the **Google Compute Engine** (GCE). One of the biggest advantages that Google's Engine offers is speed of deployment, so if you need to stand up images quickly, this might be the hosting platform for you. The Google Cloud Platform is feature-rich and offers automation for many of the services under the platform umbrella. Unfortunately, not all of the automation features are available using the associated Knife plugin. Still, there is plenty we can do, so let's explore what Chef provisioning options are available with the GCE and the knife-google plugin.

Installing and configuring the knife-google plugin

We are going to install the knife-google plugin to allow us to provision and manage our GCE hosts. The installation is via a gem, like the previous plugins we have explored. Use the following command for the installation:

```
chef gem install knife-google
```

This gem has many dependencies, and you will see them installed along the knife-google plugin gem. In my case, the command installed 15 gems in total, but other than the additional dependencies, there is nothing unusual here.

Now we need to set up authentication. This is where we get into something different. All of the previous Knife plugin integrations had a parameter that you add to the `knife.rb` file to supply a credential or point to a file to supply the credential for authentication to the specific hosting platform. Google handles this differently. The Google Knife plugin expects a credential file with a specific name to be in a specific location on your workstation and will look there when you run the knife-google plugin commands.

What's more, the credentials file has to be created with the Google Cloud SDK. It is not something that you log in to the Google Cloud Platform UI and download. And, there is no place in the UI where you can gather all of the values you need to build the file yourself. You are literally forced to install the Google Cloud SDK on your Chef workstation, and once it is installed, issue an SDK command that generates a `.json` file in this location:

```
~/.config/gcloud/application_default_credentials.json
```

You can store this credential file in a different location and create an environment variable to point to it, but there does not seem to be the typical `knife.rb` parameter that you can use to point to a different location or name for the file.

This requirement, having to install the Google Cloud SDK on your Chef workstation to be able to use the `knife-google` plugin, seems to be the least Chef-friendly installation of any of the Cloud integrations explored in this book. This both surprised and disappointed me a little. So with that said, how do you create the credential file you need for the `knife-google` plugin? Here are the steps for an OSX workstation.

Open a Terminal window and enter the following command:

```
curl https://sdk.cloud.google.com | bash
```

This will download the SDK and execute its installation. Now, restart your shell with the following command:

```
exec -l $SHELL
```

Next, initialize the `gcloud` environment with the following command:

```
gcloud init
```

This command will launch a browser window, so you can log in to the Google Cloud Platform and it will ask you to provide various selections to set up an environment for using the SDK. When the command finishes, it will have created the credentials file you need to use the `knife-google` plugin to integrate Chef with the Google Cloud Engine. The resulting `.json` file will have four parameters—`client_id`, `client_secret`, `refresh_token`, and `type`. Okay. That was a long process, but it seems to be the only way to get our needed credential file. And now, with this done, we can get back to the more "normal" Knife integration type stuff.

It is time again to open up our `knife.rb` file and add more configuration data to it. This time, we need to add a value for Google's `gce_project` parameter. Adding a value for the `gce_zone` parameter makes it so that we don't have to type it repeatedly with each of the `knife-google` commands. We can also add a parameter for the compute zone. The values of these settings can be determined from the Google Cloud Platform UI, and they were shown when we ran the `gcloud init` command to create our credential file. Here are the values I used in my `knife.rb` file:

```
#knife-google
knife[:gce_project] = 'api-project-167666703318'
knife[:gce_zone] = 'us-central1-a'
```

Now we are almost ready to start issuing `knife-google` plugin commands that we expect to work. We've got one more step to complete first, that being creating and registering our key pairs.

Creating and registering your key pairs

First, we need to create a key pair that we will then post to our user settings in the Google Cloud Platform. To create a new key pair, follow the instructions shown in the earlier Azure section. Here is the command for reference:

ssh-keygen

Save the keys in your `.ssh` folder. Give them a name that you can remember, letting you identify the key later so you can upload it to Google. I used the following:

`/Users/earlwaud/.ssh/gce_rsa`

With the keys generated, we now need to upload the public key to Google. Here are the steps required to successfully upload your key:

1. Visit the Google Cloud Platform UI at <https://cloud.google.com/>.
2. Log in to your account.
3. Go to console view by clicking on **My console** to the top-right of the page.
4. Select your project from the project list dropdown in the menu bar (you may only have one project at this point, so it may be preselected for you).
5. Using the **Products and Services** menu button, go to **Compute Engine**.
6. Select the **Metadata** functions from the **Compute Engine** menu options.
7. Change to the **SSH Keys** tab.

8. Click on the **Add item** button.
9. Now copy and paste in the entire contents of your public key into the "username" input field. To change the username automatically detected for the key, prefix your key with the username you plan to use as the --ssh-user value when creating a server with Knife. For example, if you plan to connect as "chef-user01", your key should look like: chef-user01:ssh-rsa AAAAB3NzaC1yc2EAAA...
10. Finally, click on **Save** to complete the key entry. If you need to add more than one key, maybe for different users, then you need to hit the **Edit** button and then you get the **Add item** button again.

Here is what the uploaded public key file will look like in the GCE UI:

Username	Key
earlwaud	ssh-rsa AAAAB3NzaC1yc2EAAA...TrnnTWy5mx6X9 earlwaud@Earls-Mac.local

That rounds out our setup steps. Now we are ready to start provisioning using the knife-google plugin.

Provisioning virtual machines with the GCE

Let's warm up the engine so to speak, with a couple of knife-google plugin commands, as we have done for the previous plugins. We'll start with the server list command, which should provide us with a server list header and little else, since we haven't provisioned any servers yet. Of course, if you have pre-existing GCE instances, then you should see them listed with this command:

```
knife google server list
```

Now, let's give the image list command a shot. Issue the following command:

```
knife google image list
```

Well, that was unexpected. We have already found a common plugin command that is not available with the knife-google plugin. Using our other plugins, this command provided a list of the images available to provision from. Okay. We will figure out what our available images in the GCE are later. What we did get was a list of the knife-google commands we have available. Here is the list:

```
Available google subcommands: (for details, knife SUB-COMMAND --help)
** GOOGLE COMMANDS **

knife google disk create NAME --gce-disk-size N (options)
knife google disk delete NAME [NAME] (options)
knife google disk list
knife google project quotas
knife google zone list
knife google region quotas
knife google server create NAME -m MACHINE_TYPE -I IMAGE (options)
knife google server delete INSTANCE_NAME [INSTANCE_NAME] (options)
knife google server list
knife google server show INSTANCE_NAME (options)
knife google zone list
```

Okay, the first thing I noticed was that the help text listed the "zone list" command twice. Well, that is surely a typo in the help command output. That first zone list should probably say "region list" instead. So, let's try that command to see if there is a "region list" command:

```
knife google region list
```

Yes. It works. You should get back a list of the regions and their status and subzones. It should look something like this:

Region	Status	Zones
asia-east1	up	asia-east1-a, asia-east1-b, asia-east1-c
europe-west1	up	europe-west1-b, europe-west1-c, europe-west1-d
us-central1	up	us-central1-a, us-central1-b, us-central1-c, us-central1-f
us-east1	up	us-east1-b, us-east1-c, us-east1-d

Okay, now let's decide on an image we want to deploy. The GCE images list can be found in the UI. Log in to the Google Cloud Platform and select the Compute Engine console. From there select the "Image" views from the menu options on the left. Then, look through the available images to the right. For the examples in the rest of this section, I will select the image: centos-7-v20160301. So, let's build our server create command. On our Chef workstation, enter the following command:

```
knife google server create earl-gce101 \
--gce-machine-type g1-small \
--gce-image centos-7-v20160301 \
--identity-file ~/.ssh/gce_rsa \
--gce-can-ip-forward \
--ssh-user earlwaud \
-v
```

This command will create a new VM instance in GCE, and it will install the chef-client, registering the node with the Chef server. It will also converge the node to the desired state, but in this case, we didn't provide a run-list, so there were zero changes converged. Let's review this command in detail now. Starting with the command itself:

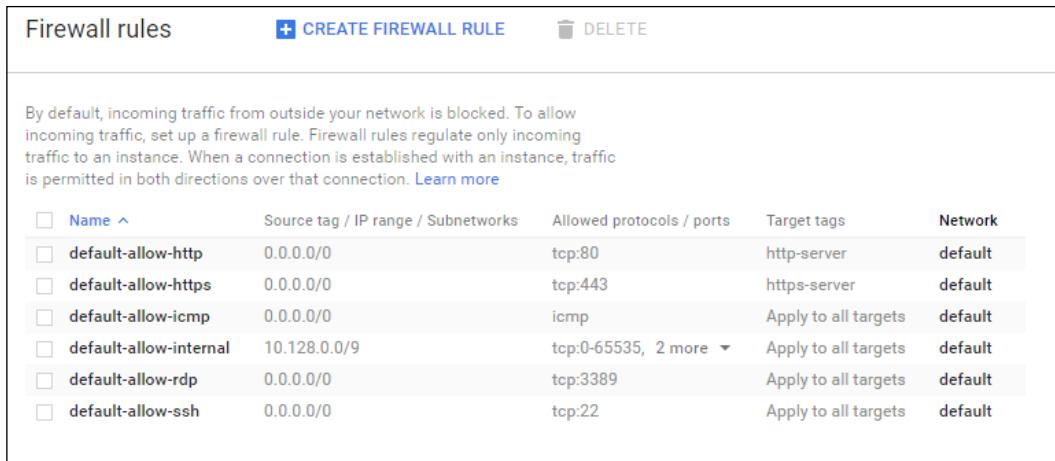
- `knife google server create earl-gce101`: Okay, this is the Knife command for creating our instances. We have the subcommand itself, "server create", and the first parameter, which is the name of our VM instance to be created. In this example, the VM instance name is "earl-gce101".
- `--gce-machine-type g1-small`: The value of this parameter provides the size of the instance to provision. In our example, we are creating a "g1-small" instance. This is one of the two partial vCPU flavors, with the other small one being "f1-micro". We will use a larger, nonpartial vCPU flavor in our next example, and you can read about all of the Machine Types available by visiting the relevant link provided later in the references section.
- `--gce-image centos-7-v20160301`: This parameter's value provides the name of the template image that we want to use to provision our VM instance. As mentioned already, you can discover all of the available images via the Google Compute Engine UI.
- `--identity-file ~/.ssh/gce_rsa`: The value for the identity-file parameter provides the path and filename of the private key of our key pair. The public key was uploaded to the SSH Keys section of the Google Compute Engine UI earlier.

- `--gce-can-ip-forward`: This parameter is a toggle that enables the forwarding of traffic from the public IP address to the private address. It is essentially the gate that either isolates or exposes the VM instance. The VM instance is isolated by default, and you can call that out explicitly with this parameter's opposite `--no-gce-can-ip-forward`.
- `--ssh-user earlwaud`: This parameter and value tell knife the username to use in ssh connections. This should match the username associated with the key generated and called out in our `--identity-file` parameter.
- `-v`: Once more, this is the verbose option, which can be made even more so with a second V as in `-vv`.

Okay, let's repeat our Chef server deployment, this time into GCE, to see what's different and what's same as our previous Chef server deployments.

Deploying Chef server into a GCE instance

Before we can deploy Chef server onto a GCE VM instance, we need to set up some firewall rules to allow the proper network access. To accomplish this, we are going to visit the Google Cloud Platform console. Once you've logged in, bring up the "Products & Services" menu, and choose "Networking". Then, using the navigation pane on the left, select **Firewall rules**. Now, to the right, you will see the list of firewall rules. Here is a look at what the default firewall rules look like:



The screenshot shows the Google Cloud Platform Firewall rules interface. At the top, there are buttons for 'CREATE FIREWALL RULE' and 'DELETE'. Below this is a descriptive text about allowing incoming traffic. A table lists six default firewall rules:

Name	Source tag / IP range / Subnetworks	Allowed protocols / ports	Target tags	Network
default-allow-http	0.0.0.0/0	tcp:80	http-server	default
default-allow-https	0.0.0.0/0	tcp:443	https-server	default
default-allow-icmp	0.0.0.0/0	icmp	Apply to all targets	default
default-allow-internal	10.128.0.0/9	tcp:0-65535, 2 more	Apply to all targets	default
default-allow-rdp	0.0.0.0/0	tcp:3389	Apply to all targets	default
default-allow-ssh	0.0.0.0/0	tcp:22	Apply to all targets	default

One of the key bits of information shown in the previous screenshot is the **Target tags** column. The data in that column shows the tag name we will use to enable the associated firewall rule with the instances we provision. For example, to enable SSL, we will use the tag name "https-server" in our knife server creates commands.

For Chef server, we need to open up TCP port 9683. To do this, we want to use the "Create Firewall Rule" button at the top of the window. Let's add the rule for port 9683 now. When you click on the "Create Firewall Rule" button, you are shown an input form. In that form, provide a name for the rule. I am using "chef-server". The name must be all lower case characters with no spaces. Next, we can provide an optional description. I am using "TCP port 9683 required for a Chef server". If you have created additional networks beyond the "default" provided, you can select the network to which you want to add this rule. I am using the default network, so no change is necessary for me. The next value, "Source filter" allows you to specify the systems whose traffic will be allowed through this port.



Best practices would be to set up specific subnets to allow. If the rule will be for all possible servers, you can supply a value of "Allow from any source (0.0.0.0/0)". This is the option I am setting, knowing that this server is only a temporary test system.



The next input will provide the protocol and port that we are opening with this rule. In our example, we are entering "tcp:9683". If we were to enter just "tcp", it would open all ports, which is really not the desired state for a production system. Finally, we provide a tag name for this rule. While the form suggests that this value is optional, it is actually a requirement for us, since it is how we tell the `knife server create` command to use this rule. In the example for the Chef server rule, I am using the tag name "chef-server".

Okay, we're ready to provision our new GCE Chef server instance. With this example command, we are going to deploy the new instance, register the new node with our public Chef server, and converge the state to a Chef server with the initial chef-client run. Here's the command:

```
knife google server create earl-gce-chef \
--node-name earl-gce-chef \
--gce-machine-type n1-standard-1 \
--gce-image ubuntu-1404-trusty-v20160314 \
--bootstrap-protocol ssh \
--ssh-user earlwaud \
--identity-file ~/.ssh/gce_rsa \
--user earlwaud \
--gce-can-ip-forward \
--gce-tags http-server,https-server,chef-server \
--run-list="recipe['chef-server']" \
-v
```

Let's take a closer look at the command parameters, focusing on the few parameters that are new or different from what we used in the previous command;

- `knife google server create earl-gce-chef`: This is the same command we used before; the only difference this time is the name is now "earl-gce-chef".
- `--node-name earl-gce-chef`: Again, this is the same parameter, but this time, we are using a node name of "earl-gce-chef".
- `--gce-machine-type n1-standard-1`: We need to create a VM that is a little more beefy than that provided with a "g1-small" flavor, so this time, we are using an "n1-standard-1". This will create our VM with 1 vCPU and 3.75 GB of memory (and 10 GB system drive).
- `--gce-image ubuntu-1404-trusty-v20160314`: This time, we requested an Ubuntu 14.04 server image for our Chef server provision.
- `--user earlwaud`: This parameter provides the API Client Username.
- `--gce-tags http-server,https-server,chef-server`: This parameter and its values list the firewall rules we want to put into place for the provisioned VM. The values are the tags we saw (and created) in the Google Cloud Platform Networking UI. This example is applying the http rule, the https rule, and our new chef-server rule. The resulting VM will have the TCP ports 80, 443, and 9683 open from all IP addresses.
- `--run-list="recipe['chef-server']"`: This parameter provides the now familiar run-list for our first chef-client run.

Once again, let's ssh into our new server, and issue the few commands needed to complete the Chef Server setup:

```
sudo chef-server-ctl user-create earlwaud Earl Waud earl@sandiegoearl.com  
MyPassword99 --filename ~/earlwaud-gce.pem  
sudo chef-server-ctl org-create sdearl "San Diego Earl" --association_  
user earlwaud --filename ~/sdearl-validator-gce.pem  
sudo chef-server-ctl install chef-manage  
sudo chef-server-ctl reconfigure  
sudo opscode-manage-ctl reconfigure
```

We now have a new Chef server ready to use, provisioned into our Google Cloud Platform. Well done! Now, let's delete it. As with the plugins we used before, we can issue a server delete command to delete the instances and unregister it from our Chef server. Here is an example command:

```
knife google server delete earl-gce-chef --purge
```

That wraps up our look into provisioning in the Google Cloud Platform. You should now have enough detail to be able to provision successfully using the GCE for whatever needs you may have. So, now we've reviewed using Knife for provisioning in the top three Cloud hosting providers. Before we switch gears a little, let's take a quick look at an honorable mention Cloud hosting provider – Linode.com.

References

- Create your account and start a free 60-day trial at <https://cloud.google.com/>.
- Chef and GCE information on Chef.io: <https://www.chef.io/solutions/google-cloud-platform/>.
- Learn about the knife-google plugin on Github at <https://github.com/chef/knife-google>.
- Learn about the GCE cookbook LWRP on Github at <https://github.com/chef/knife-google>.
- Learn about the Google Cloud Storage (GCS) cookbook LWRP on Github at <https://github.com/chef-partners/google-cloud-storage>.
- Watch some demo screencasts to learn more about Google DevOps with Chef at <http://googlecloudplatform.github.io/compute-video-demo-chef/>.
- Instructions to install the Google Cloud SDK <https://cloud.google.com/sdk/>.
- The list of Google Compute Engine Machine Types (instance sizes) is at https://cloud.google.com/compute/docs/machine-types?hl=en_US&ga=1.97972955.533487120.1458267295.

Looking at an honorable mention – Linode

Here is a quick look at the Chef integration to provision to the hosting provider Linode.com. While Linode has nowhere near the market share of any of the top hosting providers we've covered so far, there is a strong user base of passionate developers (like my colleague Barry Ruffner who introduced me to Linode) and many happy companies who use the hosting service to fill their Linux-based hosts needs with distros from CentOS, Debian, Fedora, openSUSE, Ubuntu, and Slackware, to name a few. And just like with the other hosting providers we discussed, you have the ability to create and manage your servers hosted there with a knife plugin, knife-linode. You can install the plugin with the now familiar Chef command:

```
chef gem install knife-linode
```

As with our other plugins, you will want to add a linode configuration parameter to your `knife.rb` file so you don't have to provide the parameter with each command. It will contain your API key, which can be generated from the API section of your [linode.com profile settings page](#). Here is what the `knife.rb` configuration parameter will look like:

```
#knife-linode
knife[:linode_api_key] =
'07wCP4TVURELEDtG6uF0mp6Gxf1BVZtEc2jm6uw1V63Yc3UodwztxCuk4EU8uPvQ'
```

With that in place, you're ready to provision and manage your Linode hosts. I'm not going into any real detail on the Linode commands here because I think that with the data provided in the previous sections of this chapter, you can handle those details easily. What I will do is share a list of the subcommands available and then let you take it from there. So, without further ado, here are the `knife-linode` plugin subcommands:

- The knife linode server list
- The knife linode image list
- The knife linode server create
- The knife linode server delete
- The knife linode server reboot
- The knife linode data center list
- The knife linode flavor list
- The knife linode kernel list
- The knife linode stackscript list

In the references section later, you can find the links you'll need to set up your own Linode account and to learn more about the `linode` plugin. Note that at this time, Linode does not offer a free trial, but they do have a 7-day money-back guarantee.

References

- The Linode website: <https://www.linode.com/>
- The `knife-linode` plugin repo on Github: <https://github.com/chef/knife-linode>

Provisioning containers in Docker

Now you are going to learn about something a little different, Docker. Docker is a container provider, and although it is not the only container game in town, it is by far the most widely used. You will be able to leverage what you learn in this section to other container platforms as desired. Of course, you don't have to use Docker in the cloud. It is equally viable in the traditional data center, but since it is a fresh re-imagining of virtual environments, I thought it would better fit in a chapter in the clouds.

Containers offer much benefit as an alternative to "traditional" virtual machines. For example, containers are lightweight and can be deployed much more densely than traditional VMs. Containers contain fewer overhead processes, and so they can start up much faster than VMs. There are many advantages that might compel you to want to create your own containers, so let's give provisioning a try with Chef. Of course, you can use a commercial provider like dotCloud, Tutum, or Google's Container Engine. But we are going to set up our own Docker host to serve our containers.

Setting up a Docker server to host our containers

We want a Docker server to provision our containers, and since this is a book about Chef Provisioning, we're going to leverage Chef to set up our Docker server. We will download the Docker cookbook from the Chef Marketplace and then upload the cookbook to our Chef server. Once it has been uploaded, we will deploy a new server using knife and converge the server into a Docker server with the cookbook and chef-client. Let's get started. Here is the command to download the Docker cookbook:

```
knife cookbook site install docker
```

This command will download the Docker cookbook and all of its dependencies from the Chef Marketplace and then install them into the cookbooks folder of our current repo.

The docker cookbook has a dependency on the compat_resource cookbook, specifically version 12.7.1, so let's make sure that we have the desired version installed in our repo with the following command:

```
knife cookbook site install compat_resource 12.7.1
```

Now, let's upload the docker cookbook and its dependencies with the following command:

```
knife cookbook upload docker --include-dependencies
```

We need to do one more thing to finish our cookbook setup. The docker cookbook is a library-only cookbook. It does not have recipes and is intended to be used by other cookbooks. So we are going to create the most basic cookbook, named my-docker. It is going to have a default recipe that will install the docker server for us. Let's create the new cookbook structure with Chef, using this command:

```
cd ~/chef-repo
chef generate cookbook cookbooks/my-docker
```

Now, let's update the metadata for our new cookbook by editing the metadata.rb file:

```
cd ~/chef-repo
vi cookbooks/my-docker/metadata.rb
```

Update the contents of the metadata.rb file, filling in the maintainer and maintainer e-mail values, and then, most importantly, add a "depends" line for docker. The resulting file should look something like this:

```
name 'my-docker'
maintainer 'Earl Waud'
maintainer_email 'earl.waud@gmail.com'
license 'all_rights'
description 'Installs/Configures my-docker'
long_description 'Installs/Configures my-docker'
version '0.1.0'

depends 'docker', '~> 2.0'
```

Now, let's create a my-docker recipe by creating a default.rb file in the recipes folder of our new cookbook:

```
cd ~/chef-repo
vi cookbooks/my-docker/recipes/default.rb
```

Insert the following recipe code:

```
docker_service 'default' do
  action [:create, :start]
end
```

Save the file, and we can upload the cookbook to our Chef server with the following command:

```
cd ~/chef-repo  
Knife cookbook upload my-docker
```

Great! We are now ready to deploy our new server and converge it into a Docker host. We have several options for where we want to deploy this new server. We can go with something in our own data center, or we can deploy it in the cloud using any of the hosting providers we've looked at in this chapter. Somewhat arbitrarily, I am going to choose the Microsoft Azure service to host our Docker server. With that decision, here is the command:

```
knife azure server create \  
  --azure-dns-name earl-docker \  
  --azure-vm-name earl-docker \  
  --azure-vm-size Medium \  
  --azure-source-image b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-  
    LTS-amd64-server-20150123-en-us-30GB \  
  --azure-service-location 'Central US' \  
  --ssh-user earlwaud \  
  --identity-file ~/.ssh/azure_rsa \  
  --bootstrap-protocol ssh \  
  --node-name earl-docker \  
  --run-list 'recipe["my-docker"]' \  
  --tcp-endpoints 80,443,3306 \  
  -v
```

This command does as we desire. It provisions a new server into Azure classic, registers the node with our public Chef server, and then running the chef-client, it converges the host into the state of a Docker server. Most of this command should be very familiar, as it is nearly identical to the command we used to provision a Chef server into Azure earlier in this chapter. The only two significant differences are the run list used and the port 3306 specified in the tcp-endpoints parameter. The 3306 port is used by the Docker engine.

While this method of deploying Docker onto a server might be the long way around, it is still important because it follows our DevOps methodology and it is OS agnostic. In other words, I could have just deployed my Ubuntu server, logged in and done an appropriate `wget` command, and achieved the same state. But by using a Chef cookbook, we have achieved our infrastructure as code goal. Also, if we had used a CentOS host instead of Ubuntu, our cookbook would have still done the right thing to install the Docker engine. Likewise, if it were a Windows-based host, it would have properly installed the Docker engine.

Alright, we've got our Docker server set up and running the Docker engine. We can confirm this by using `ssh` to log in to the server and running a `docker` command. For example, we can use the following command:

```
sudo docker ps
```

This command will list the containers running on the Docker host. We are now to use Docker from our Azure instance.

Installing and using Docker on our Chef Workstation

Since we've been primarily using OS X for our Chef workstation examples, we are going to use the Docker toolbox as the local host for our Docker containers example. This is the same tool you would use for a Windows Chef Workstation.

If you are using Ubuntu or another Linux flavor, you can install `docker.io` and execute the remainder of the examples with a direct Docker install. Frankly, this is a much easier and better solution to create local Docker containers than the OS X option. If you can do your container work in an Ubuntu workstation, I would highly recommend it.

So, if you are still using an OS X workstation, the next step is to install the toolbox. You can download and install the toolbox for free from www.docker.com/products/docker-toolbox.

Once the toolbox is installed, run the "docker quickstart terminal" to open up a Terminal session with the docker environment preconfigured. This actually spins up a virtual machine that is running Docker. Fun, right?!

Now we can use the knife-docker plugin (as well as any standard docker commands, such as `docker ps`) in our Terminal window.

Now can give some Docker commands a try. Here are some commands for you:

```
docker ps  
docker images  
docker run -t -i ubuntu:14.04 /bin/bash
```

Using the knife-docker plugin

We are going to switch gears a little bit now. The knife-docker plugin does not successfully execute the chef client run when run on OS X. There is an unreleased version 0.0.3 that is working to address this issue, but at the time of writing this, it was not working. So, we are going to execute this example using Ubuntu for our workstation.

Let's start by installing the knife-docker plugin for working with Docker. Issue the following command:

```
chef gem install knife-docker
```

For our docker container examples, we will work with publicly downloadable docker images. Let's start by listing the docker images we have currently available on our workstation. Open up a terminal session. From within the terminal session, issue the following command:

```
sudo docker images
```

You should have an empty list of images at this point if you have just recently installed your local docker environment. Next, let's see a list of some of the publicly available images. We can do this by issuing a docker search command like this:

```
sudo docker search ubuntu
```

From that command, you should get a list of public docker images related to the search key "ubuntu". In my list, there is an image named `rastasheep/ubuntu-sshd`. I want to use that for our test of the knife-docker plugin, so I am going to download that image with the following command:

```
sudo docker pull rastasheep/ubuntu-sshd
```

Many docker images do not pre-install sshd, so we want to make sure we are working with one that does. The `rastasheep/ubuntu-sshd`, as the name suggests, is an image that runs sshd. Okay, let's issue the `image` command again to see if the downloaded image shows up in our list as expected:

```
sudo docker images
```

Yep, there it is. Excellent. Now we are going to use that image to deploy a docker container using the knife-docker plugin. Here is the basic command:

```
sudo knife docker create -I rastasheep/ubuntu-sshd --node-name docker01
```

This command will deploy the new container using the image `rastasheep/ubuntu-sshd` as the template. The container uses the credentials of `username`, `root`, and `password`, `root`, so when you are prompted for the password, enter `root`.

The command deploys the container and registers it with the Chef server as configured in our `.chef/knife.rb` file using the value in the `--node-name` parameter, in this case, `docker01`.

The other knife-docker command allows us to use the knife-docker plugin to delete our containers and clean up their artifacts from the Chef server. Here is an example command:

```
sudo knife docker delete f5e483540f3d -P
```

The two parameters used here are the container ID and the `-P` switch, which instructs knife to delete the node and client as well as the container.



I would like to point out that using the knife-docker plugin as a container provisioning method while functional is not really the way to go. We will examine the much more favored method of deploying containers using Chef in *Chapter 8, Using Chef Provisioning*, so stay tuned.

References

- The link to the Docker home page is <https://www.docker.com/>.
- You can find the Docker repo at <https://github.com/docker/docker>.
- You can find the Chef cookbook for Docker at <https://github.com/chef-cookbooks/docker>.

- Sign up for a commercial Docker hosting service at one of these providers, Tutum (at <https://www.tutum.co/>), dotCloud (at <https://cloud.docker.com/>), or use the Container Engine system with the Google Cloud Provider (at <https://cloud.google.com/container-engine/>).
- The link to Chef.io information on using Chef for containers is <https://docs.chef.io/containers.html>.
- Find the knife-docker plugin on Github at <https://github.com/ema/knife-docker>.
- Download and install the Docker Toolbox from <https://www.docker.com/products/docker-toolbox>.
- There is an interesting Ubuntu base image for your containers at <https://github.com/phusion/baseimage-docker>.
- There is another interesting base image container at <https://github.com/phusion/passenger-docker>.
- Read through a tutorial on using Docker at <http://www.dedoimedo.com/computers/docker-networking.html> and <http://www.dedoimedo.com/computers/docker-guide.html>.

Summary

In this chapter, you learned how to turn our cloud-based servers (and containers) into code, so they can benefit from all the methodologies we are used to using for our application projects. We solidified our knowledge of using Amazon AWS EC2 instances. You learned how to leverage the next big players in the cloud provider arena, Microsoft and Google. We took a brief look at provisioning hosts on Linode.com, and you learned how to use Chef to provision our containers in Docker. Adding all this new cloud knowledge to what you learned about doing the same things in our traditional data center, we are now armed with the skills to easily create a hybrid hosting solution that lets us target our deployments in-house or in the cloud and lets us burst from one to the other depending on the needs of our business. With so many options available to provision our workloads, let's take a quick look at making sure that those workloads are as error free as possible by examining test-driven development in the next chapter. Turn the page, and I'll see you there!

7

Test-Driven Development

In this chapter, we're going to delve into the Test-driven Development model, showing how to maximize development efforts and quickly deliver highly functional and well-tested environments. The reader will learn how to transform their model for Chef from one where cookbooks are code to one, where cookbooks are artifacts, with test practices built into both models. Initially, it may seem difficult and time consuming to build the skills and habits of test-driven development. However, once the skills and habits are acquired, the development cycle becomes accelerated and the resulting infrastructure is much more stable. What's more, DevOps Engineers will have greater certainty that the infrastructure will be fully functional. It will no longer be necessary to do ad hoc tests and spot checks on infrastructure deployed into production environments. The infrastructure code will be continually tested through the development process, and the artifact deployments will be equally well-vetted before anything is ever pushed into production. The result is better deliverables and shorter development cycles.

"This is my advice to people: Learn how to cook, try new recipes, learn from your mistakes, be fearless, and above all have fun" – Julia Child

The following topics will be covered:

- Analyzing cookbooks with RuboCop
- Analyzing cookbooks with Foodcritic
- Unit testing with ChefSpec
- Integration testing with Kitchen and ServerSpec

Analyzing cookbooks with RuboCop

We begin with a look at tools working within the cookbooks-as-code model. These tools are considered static analysis test tools, and they examine the code itself to verify syntax, structure, and alignment with desired coding practices. The first tool we'll explore is named RuboCop.

Simply put, by using RuboCop, you will produce better cookbooks. RuboCop is included as part of the ChefDK installation. It is a command-line tool used to test the syntax and style of individual cookbooks. RuboCop uses a set of rules, known as cops, to test specific considerations, such as syntax, style, and metrics for Ruby best practices. These cops deliver results as warnings, deviations from convention, and different levels of errors. The main benefits of using RuboCop are that you identify errors early in the development process and the resulting code conforms to an agreed upon style and formatting practice. So, how do we use RuboCop? Let's take a look now.

Using RuboCop

The expectation to use RuboCop is that you provide the path and name of a cookbook to test or you execute the tool within the folder of a specific cookbook to test that cookbook. For example, two different ways to execute RuboCop to test the apache cookbook that is part of your chef-repo are as follows:

```
cd ~/chef-repo/cookbooks/apache  
rubocop .
```

We also have the following:

```
cd ~/chef-repo  
rubocop cookbooks/apache
```

RuboCop is a Ruby test tool, and as such, it will examine all of the Ruby files in your cookbook. By default, there are many RuboCop cops patrolling the cookbooks for violations. They are grouped into categories based on the type of rule they enforce, including lint, style, metrics, Ruby, and so on. You can see all of the cops enforced by RuboCop with the following command:

```
rubocop --show-cops
```

This command will list out all of the cops, and it includes URLs to the RuboCop documentation for the rules enforced by that cop.

With this information, you can use RuboCop to evaluate the rules of a single cop by using the parameter `--only` and the name of the cop. For example, to see if you have any lines that are longer than the standard 80 characters, you can issue the following commands:

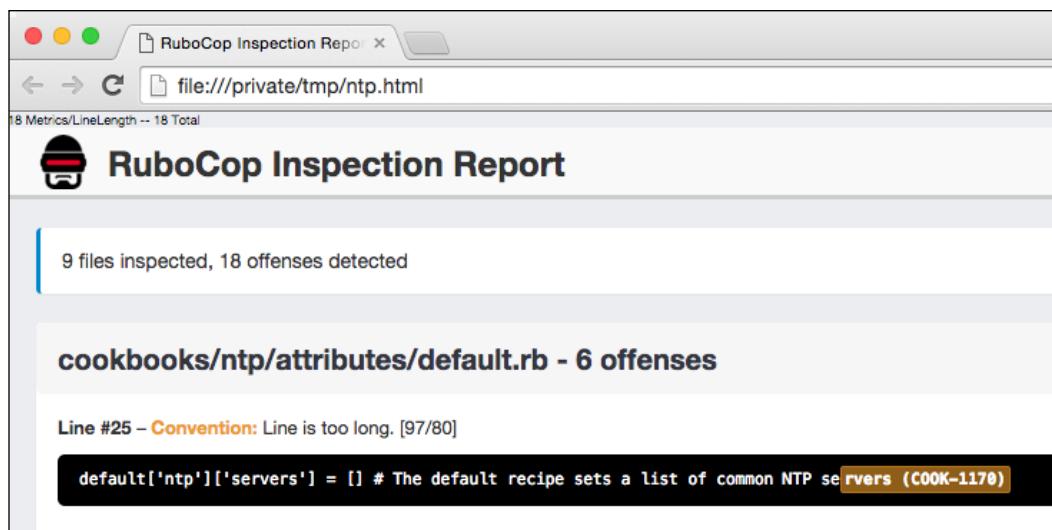
```
cd ~/chef-repo/cookbooks/apache
rubocop --only Metrics/LineLength
```

Examining the RuboCop output

By default, RuboCop displays the results in what is known as the progress format. The available output formats include [p]rogress (default), [s]imple, [c]lang, [j]son, [h]tml, [o]ffenses, and so on. These output formats can be called upon with the `--format` (or `-f`) parameter. For example, to see the results for a RuboCop test of the NTP cookbook in the [o]ffences and [h]tml formats, issue the following commands:

```
cd ~/chef-repo
rubocop -f o -f h cookbooks/ntp > /tmp/ntp.html
```

Now you can open the output in your browser, and the output is nicely formatted and easy to review. It will look something like this:



Each Ruby file is tested for offences, and the output provides the specifics of the violations found. The first thing reported is a summary of how many files were tested and how many total violations were found. From there, the results of each file examined are provided, starting with the filename and number of violations in that file. Then, the individual violations are listed with each providing the violation details, including the type of issue. The following abbreviations are used to indicate the type of issue found:

```
'.' means the file contains no issues  
'C' means there is an issue with convention  
'W' means there is a warning  
'E' means there is an error  
'F' means there is a fatal error
```

Additional information provided in the violation details includes the line number and position within the line where the issue was found, followed by the actual line of code from the Ruby file that contains the issue. Using this information, you should be able to easily remedy the issues reported on by RuboCop.

With the multitude of cops and the extensive violation data they watch for, it can become overwhelming to keep your code in obedience with the laws. Because of this, RuboCop provides a way to customize your cops and rules so that you can focus in on the ones that are most important to you and your team. Let's look at how we can configure RuboCop to use a subset of cops without having to call them out on the command line.

Customizing RuboCop

RuboCop customization is handled via a `.rubocop.yml` file in the cookbook's folder. The easiest way to create this file is to let RuboCop help you. For example, you can issue the following commands:

```
cd ~/chef-repo/cookbooks/apache  
rubocop --auto-gen-config
```

This will create a new file named `.rubocop_todo.yml` in the cookbook folder. Now you have a place to start. This file contains the minimum set of cop configurations that, when used by RuboCop in subsequent runs, allow your cookbook to pass all cop evaluations. For example, if the longest line of code in your cookbook is 93 characters long, the cop for line length checks will be configured to all lines to be up to 93 characters long.

With the new todo file, you can proceed in one of the three ways. You can rename your `.rubocop_todo.yml` file to `.rubocop.yml`. You can also add a parameter to your command like this:

```
rubocop --config .rubocop_todo.yml
```

Or, you can create your own `.rubocop.yml` file and add the line `inherit_from: .rubocop_todo.yml` to it. With any of these shortcuts, all existing violations will be ignored in your future RuboCop runs.

With any of those options in place, you can edit the configuration file and re-enable the cops that you want to address one by one. For example, if you care about maximum line lengths in your cookbooks, you can re-enable that rule check by commenting out the Metrics/LineLength rule in the configuration file or by adjusting the rule's setting value to the desired line length max value, that is, 80. Once this is done, violations of that rule will again be reported by the LineLength cop the next time RuboCop is executed. This will provide the data that needs to fix any violations of that rule. Using this one by one rule check and fix technique, you implement corrections to the rules that matter most to you, without being overwhelmed by the volume of violations that RuboCop can find.

One final note about RuboCop is that it offers an "auto correct" option that will fix minor violations automagically for you. Add the parameter `-a` or `--auto-correct` to your command line to use this option. While this is a very handy feature, be aware that it is not perfect, and you should review its changes for sanity. Now, let's continue our testing with the tool, Foodcritic.

References

- You can find the RuboCop home page at <http://batsov.com/rubocop>.
- Learn more about RuboCop on the Chef.io site, <https://docs.chef.io/rubocop.html>.
- There is the RuboCop repo on GitHub at <https://github.com/bbatsov/rubocop>.
- You can find the list of RuboCop cops that are enabled by default at <https://github.com/bbatsov/rubocop/blob/master/config/enabled.yml>.
- The link for the cops that are disabled by default is <https://github.com/bbatsov/rubocop/blob/master/config-disabled.yml>.

- If you want to add RuboCop support directly to your editor, visit one or more of these GitHub repos—Sublime Text at https://github.com/pderichs/sublime_rubocop, Atom at <https://github.com/yujinakayama/atom-lint>, or VIM at <https://github.com/ngmy/vim-rubocop>.

Analyzing cookbooks with Foodcritic

Let's take a quick look at another static analysis test tool that is included with ChefDK. That tool is named Foodcritic. Like RuboCop, Foodcritic examines the code in your cookbooks for violations to its rules. However, where RuboCop is a generic Ruby linter, Foodcritic is a Chef-specific linting tool. The difference being that using RuboCop helps you to keep your cookbooks in line with Ruby best practices, and Foodcritic helps to keep your cookbooks in line with Chef best practices. Foodcritic currently has 61 built-in rules to evaluate your cookbooks and allows the creation and use of custom rules as well. Both tools have their value, and I suggest that there is a place for each one in your DevOps workflow.

Using Foodcritic

You will find using Foodcritic to be very much like using RuboCop. It is expected that you will be testing individual cookbooks and as such will execute Foodcritic at the command line in the folder of the cookbook being tested, or you can execute Foodcritic in your chef repo with a parameter for the path to the cookbook being tested. For example, if you want to test the apache cookbook that is a part of your chef-repo, you would execute the following command:

```
cd ~/chef-repo/cookbooks/apache
foodcritic .
```

One of the optional parameters you can include will provide context data when rule violations are found. The parameter is -c or --context. I've found that using this parameter makes correcting the issues much easier. Here is an example command, which uses the context parameter with Foodcritic to test a work-in-progress apache cookbook:

```

1. exit (bash)
Earls-Mac:~ earlwaud$ cd ~/chef-repo
Earls-Mac:chef-repo earlwaud$ foodcritic --context cookbooks/apache
cookbooks/apache/metadata.rb
FC008: Generated cookbook metadata needs updating
1|name          "apache"
2|maintainer    "YOUR_COMPANY_NAME"
FC008: Generated cookbook metadata needs updating
3|maintainer_email "YOUR_EMAIL"
4|license        "All rights reserved"
5|description    "various apache server related resource provides (LWRP)"
6|long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
cookbooks/apache/templates/default/fast-cgi-vhost.erb
FC034: Unused template variables
1|<VirtualHost *:<%= @params[:ssl] == false ? 80 : 443 %>>
2|
3|  ServerName <%= @params[:server_name] %>
4|  <%- @params[:server_alias].each do |a| %>
cookbooks/apache/providers/fastcgi.rb
FC043: Prefer new notification syntax
41| case node.platform # sorry for this case, but gentoo still not supported in
42|           # http://tickets.opscode.com/browse/COOK-817
43| when 'gentoo'

```

Let's take a closer look at the output provided by Foodcritic.

Examining the Foodcritic output

Foodcritic has a set of rules built in that checks for specific issues in your cookbooks. Each rule has an identifier, such as FC008 or FC043. When a rule violation is found, Foodcritic will output the rule ID with the rule description. If you use the optional context parameter as shown earlier, the actual code that violates the rule will also be included in the output displayed. The nature and detail of each rule can be found on the Foodcritic website, <http://www.foodcritic.io/>. Clicking on the rule ID will take you to a description of the issue found by the rule, and it provides guidance on how to correct that issue.

Customizing Foodcritic

Foodcritic offers a "filter" to fine tune the rules that are checked, much like RuboCop. It is applied in one of two ways. Either via a parameter on the command line or as a parameter in a special .foodcritic configuration file. To use the parameter on the command line, use a -t followed by the rule ID. For example, to check the apache cookbook for violations of only rule ID FC008, you would use a command like this:

```
cd ~/chef-repo
foodcritic -t FC008 cookbooks/apache
```

You can also check for violations of every rule except the rule specified by preceding the rule ID with a tilde. Thus, to check the apache cookbook for violations of every rule except FC008, you would use a command like this:

```
cd ~/chef-repo
foodcritic -t ~FC008 cookbooks/apache
```

Remember that you can put these parameters into a configuration file in the cookbook folder, and they will be considered in every Foodcritic run for that cookbook without needing to add them to the command line. For example, if you want to exclude the rules FC008, FC034, and FC043 every time you run Foodcritic to test the apache cookbook, you can create the configuration file using this command:

```
cd ~/chef-repo/cookbooks/apache
echo "~FC008,~FC034,~FC043" > .foodcritic
foodcritic .
```

References

- The link to the Foodcritic home page is <http://www.foodcritic.io>.
- You can read more about Foodcritic on the Chef.io site, <https://docs.chef.io/foodcritic.html>.
- The Foodcritic repo on GitHub is <https://github.com/acrm/p/foodcritic>.
- Additional custom Foodcritic rules can be found at <https://github.com/etsy/foodcritic-rules>.

Unit testing with ChefSpec

Now that we have cookbooks that are adhering to the best practices of both Ruby and Chef, it is time to turn our attention away from static analysis and toward runtime testing. This is where we begin to move from a model of cookbooks-as-code to one of cookbooks-as-artifacts. Unit testing is a kind of middle ground, where we are doing a combination of code testing and artifact testing. This is also the point at which we move into a test-driven development model, where you write the tests before you write the code.

Ruby is a dynamically typed language. Ruby code is not type checked until it is executed. So in order to actually test our cookbooks, they have to be run. Before ChefSpec, this meant that you had to make the code changes to your cookbooks on your workstation, upload your modified cookbooks to your Chef server, then finally converge a node that is assigned that cookbook in order to be able to actually test your changes. This was a work- and time-intensive process that, with a complex set of cookbooks, could easily take 20 minutes or more. Not cool.

That was all before ChefSpec. ChefSpec is a unit testing tool that simulates a chef-client run to actually execute your cookbook code locally, saving you the time and effort of uploading the cookbook to your Chef server and executing a chef-client run on some node to test your changes. Using ChefSpec, you can accomplish your tests in a fraction of the time that was required before. ChefSpec is built on RSpec, which is a Ruby-specific unit test tool. ChefSpec is to RSpec what Foodcritic is to RuboCop. That is, ChefSpec takes the same unit test concepts used for Ruby in RSpec and adapts them to be Chef-specific. Both RSpec and ChefSpec are installed as part of ChefDK.

ChefSpec allows you to define what you expect the resources of your cookbooks to do. Then, you can use ChefSpec to test the cookbooks to see if they accomplish your expectations.

Using ChefSpec

Getting started doing unit testing with ChefSpec is easy. When you begin by creating your cookbooks and recipes using the generate feature of Chef, you automatically get a ChefSpec test spec template file created for you. Let's say that you want to create a new web app cookbook. Using Chef, we can create the cookbook with the following command:

```
cd ~/chef-repo/cookbooks
chef generate cookbook mywebapp
cd mywebapp
berks init
```

Allow the `berks init` command to overwrite any of the configuration files with a response of "Y" at each prompt. Now take a look at what was created:

```
Earls-Mac:mywebapp earlwaud$ tree
.
├── Berksfile
├── Gemfile
├── Policyfile.rb
├── README.md
├── Thorfile
├── Vagrantfile
├── cheffignore
└── metadata.rb
├── recipes
│   └── default.rb
└── spec
    ├── spec_helper.rb
    └── unit
        └── recipes
            └── default_spec.rb
└── test
    └── integration
        ├── default
        │   └── serverspec
        │       └── default_spec.rb
        └── helpers
            └── serverspec
                └── spec_helper.rb

10 directories, 13 files
Earls-Mac:mywebapp earlwaud$
```

We have two directories that have been created for us that relate to testing. The first is named `spec`, and the second is named `test`. We will take a closer look at the folder named `test` in the next section of this chapter. Looking closer at the folder named `spec`, we can see that we have a subfolder named `unit`. This implies that we will be using this for our unit tests. We have a folder named `recipes` with a single file named `default_spec.rb` later. This file, `default_spec.rb`, is where we will create the unit tests for our corresponding `default.rb` cookbook recipe file, which as you can see in the image is found in the normal location under the cookbook's `recipes` folder.

In order to better understand how to use ChefSpec, let's create a very simple default recipe for our `mywebapp` cookbook. We will add some resources to the default recipe that installs and starts `httpd`, and provides a super simple `index.html` file to serve up. Start by editing the recipe file:

```
cd ~/chef-repo/cookbooks/mywebapp
vi recipe/default.rb
```

Add the following resources:

```
package 'httpd'

file '/var/www/html/index.html' do
  content '<h1>Hello MyWebApp User</h1>'
end

service 'httpd' do
  action [:enable, :start]
end
```

Now save and exit the editor. We have three resources; the first installs the httpd package, then we create an `index.html` file with the Hello Users content, and we have the service that starts httpd and makes sure that it starts on boot. It is simple, but it does enough to explore our use of ChefSpec.

At this point, we can actually run a test. Remember that ChefSpec is built upon RSpec, so we will be using RSpec to execute our test. The ChefSpec test specs are in the `spec/unit/recipes` folder, and they will have a name that matches our recipe file. That is, for a recipe file named `default.rb`, the ChefSpec test file will be named `default_spec.rb`. So, let's give it a try with the following command:

```
cd ~/chef-repo/cookbooks/mywebapp
chef exec rspec spec/unit/recipes/default_spec.rb
```

This is going to execute the unit test defined in the `default_spec.rb` file to test the cookbook default recipe found in `recipes/default.rb`. If you've been following along, and you run the command given previously, you will get results that look like this:

```
cd ~/chef-repo/cookbooks/mywebapp
chef exec rspec spec/unit/recipes/default_spec.rb
.

Finished in 3.84 seconds (files took 2.41 seconds to load)
1 example, 0 failures
```

Well, we can see that the unit test took about 4 seconds to complete, and that it had zero failures, but what did it actually test? To find out, we want to examine the contents of our `default_spec.rb` file, the one found in `cookbooks/mywebapp/spec/unit/recipes` folder:

```
cd ~/chef-repo/cookbooks/mywebapp
cat spec/unit/recipes/default_spec.rb

require 'spec_helper'

describe 'mywebapp::default' do
  context 'When all attributes are default, on an unspecified platform' do
    let(:chef_run) do
      runner = ChefSpec::ServerRunner.new
      runner.converge(described_recipe)
    end

    it 'converges successfully' do
      expect { chef_run }.to_not raise_error
    end
  end
end
```

Okay, that is probably a bit confusing at first glance, so let's break it down. We start off with a require statement. That is your standard require gem declaration; in this case, we require the `spec_helper` gem. Next, we have the describe line. This is identifying the name of the cookbook and recipe that we want to test. In this case, it is the default recipe of the `mywebapp` cookbook. Next, we have a context block. This block is what tells ChefSpec to set up a temporary node that will be converged to the state described in the recipe. The temporary node is named `runner`, and it is going to converge the recipe called out in the describe line earlier. There are two types of nodes or runners that can be used, a `SoloRunner` or a `ServerRunner`. `ServerRunner` is the default type of runner. The next block is defining our expectation. This is the unit test we hope to see pass. In this case, it is simply saying that when we do the Chef run, we don't want to see any errors.

Expanding our tests

So far, we have created a pretty basic unit test. It just runs and checks to see whether anything had an error. What if we want to be more specific? Let's begin to expand our unit tests. This time, we want to make sure that our recipe installs the package httpd. Editing our `default_spec.rb` file, we add another expectation. Here is the resulting file content:

```
cd ~/chef-repo/cookbooks/mywebapp  
vi spec/unit/recipes/default_spec.rb
```

Insert the following in the file:

```
require 'spec_helper'  
  
describe 'mywebapp::default' do  
  context 'When all attributes are default, on an unspecified  
  platform' do  
    let(:chef_run) do  
      runner = ChefSpec::ServerRunner.new  
      runner.converge(described_recipe)  
    end  
  
    it 'converges successfully' do  
      expect { chef_run }.to_not raise_error  
    end  
  
    it 'installs the package httpd' do  
      expect(chef_run).to install_package('httpd')  
    end  
  end  
end
```

I've highlighted the new section for easy identification. Here, we are declaring our expectation that the recipe will install the package httpd. Now, before you start to panic, wondering how in the world are you supposed to know what needs to be added to a unit test file to set the expectations, realize that the very smart people who developed ChefSpec have already thought of that. They have provided a GitHub repo that has a very well-developed set of examples. The examples provided have both the cookbook recipe resources and the corresponding unit test expectation specs. You can find these examples at <https://github.com/sethvargo/chefspec/tree/master/examples>.

So, let's run our unit test again now that we have added our new expectation:

```
cd ~/chef-repo/cookbooks/mywebapp
chef exec rspec spec/unit/recipes/default_spec.rb
..
Finished in 0.77693 seconds (files took 2.31 seconds to load)
2 examples, 0 failures
```

Now, there are zero failures. So, both of the expectations passed successfully. You may have noticed that the output from our command started off with two dots this time. There was one dot last time. The dots represent the progress of our tests. Two expectations provide two dots. This format of output is appropriately named the progress format. There are other formats in which you can have our output delivered. For example, there is a format named document format. If you add a `-fd` parameter to your command, it will deliver the output in the document format. Go ahead and give that a try now, as follows:

```
cd ~/chef-repo/cookbooks/mywebapp
chef exec rspec -fd spec/unit/recipes/default_spec.rb

mywebapp::default
When all attributes are default, on an unspecified platform
  converges successfully
  installs the package httpd

Finished in 0.65813 seconds (files took 2.3 seconds to load)
2 examples, 0 failures
```

With the document format, the output is changed from a sequence of dots to a human-readable description of what is being tested. You can take it a step further and add the parameter `-c` to the command, and the output will be color coded for easy interpretation.

Another thing that makes this process very easy is knowing that you don't have to know how to write every expectation at the time you add them to your spec file. You can add your expectations in a "pending" form first by just inserting some `it '...'` lines into your spec file. For example, I could have inserted the following pending expectations into my `default_spec.rb` file:

```
it 'installs a package named httpd'
it 'makes me a cup of Earl Grey tea'
```

When you run your ChefSpec test, you will get a result that shows you have some pending expectations to complete in your spec file:

```
Earls-Mac:mywebapp earlwauds$ chef exec rspec -c -fd spec/unit/recipes/default_spec.rb
mywebapp::default
  When all attributes are default, on an unspecified platform
    converges successfully
      installs a package named httpd (PENDING: Not yet implemented)
      makes me a cup of earl grey tea (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) mywebapp::default When all attributes are default, on an unspecified platform installs a package nam
ed httpd
      # Not yet implemented
      # ./spec/unit/recipes/default_spec.rb:24

  2) mywebapp::default When all attributes are default, on an unspecified platform makes me a cup of earl
grey tea
      # Not yet implemented
      # ./spec/unit/recipes/default_spec.rb:25

Finished in 0.40847 seconds (files took 2.23 seconds to load)
3 examples, 0 failures, 2 pending
Earls-Mac:mywebapp earlwauds$
```

 Remember that the best practice for test-driven development is that you create your tests first and then you create your recipe code to pass the tests you defined. With this in mind, you can create your expectations in your spec files for each recipe before you write the recipe itself.

When you initially run your ChefSpec test, all of the tests will fail, and that is okay. Then, when you create your recipe and rerun the ChefSpec test, you will see the tests passing and know that you are delivering a recipe that is converging to the state defined in your expectations.

Remember to use `chef generate` to create any additional recipes you need in your cookbooks so that the corresponding test files get created for you automatically. For example, if you want to refactor your cookbook so that it is more modular, and if you want to move the package install code into its own recipe, you can use the `chef generate` command to help you out. Here is the command for reference:

```
cd ~/chef-repo/cookbooks/mywebapp
chef generate recipe install
```

This command will create your `install.rb` recipe file in the `recipes` folder, and it will create your unit test `install_spec.rb` file in the `spec/unit/recipes` folder. From there, you can insert your expectations and begin unit testing the new recipe in the same way you did for your default recipe.

This form of unit testing is very powerful and provides a very fast method of testing your cookbook recipes. You can accomplish in seconds what it would otherwise take many minutes to achieve with full integration testing. Still, there is a place for integration testing, so let's now take a look at another testing tool, Test Kitchen, which allows us to do full integration tests with ease.

References

- There is the ChefSpec documentation on the Chef.io site at <https://docs.chef.io/chefspec.html>.
- You can find the ChefSpec repo on GitHub at <https://github.com/sethvargo/chefspec>.
- The expectation examples in the ChefSpec repo are at <https://github.com/sethvargo/chefspec/tree/master/examples>.
- There is an exceptional webinar on using ChefSpec at <https://www.chef.io/webinars/?commid=194483>.

Integration testing with Kitchen and ServerSpec

Now we are going to move to the cookbooks-as-artifacts model of testing. This is full integration testing, and we are going to look to the Test Kitchen framework as the tool for this type of test. Kitchen is another tool that is included as part of the installation of ChefDK, so there is no additional work to install it. Kitchen allows you to easily deploy a new system, install Chef, download the run-list cookbooks, and converge the node instance to fully test the functionality of our policies, roles, cookbooks, and so on.

To explore using Test Kitchen, let's imagine that we want to create a cookbook that will set up a web server to present our new application's website to users. The best way to approach this from a test-driven development aspect is to consider the desired results from the user's point of view. The user does not care that we have a web server installed and running; they only care about getting an appropriate response from our app when they visit its website. With this in mind, we will craft our tests to validate the results from our user's point of view. Does our instance deliver an appropriate response when the user visits the app's website? Realize that by testing this result in this way, we are in fact also testing the results of everything it takes to get there, that is, standing up a new instance, installing and running the web server application, setting up the web page content, and so on. This is all confirmed by a single, simple test, to make sure that the user gets the response they desire. This result-focused testing is known as behavior-driven development, or BDD. We will talk more about our BDD test shortly.

Preparing our tests

As I mentioned already, Kitchen is installed as part of ChefDK, so it should already be installed on your Chef workstation. Kitchen leverages other tools that we have talked about in earlier chapters. For example, as a default configuration, it will use the tool Vagrant to create our test nodes. Vagrant can be configured to use any number of hosting platforms as the place it launches our test nodes, but the "out-of-the-box" configuration for Kitchen is to use the open source tool VirtualBox as a local hosting platform. If you have not done so already, I would strongly suggest that you download and install both of these tools now. Once you have the tools installed, the rest of this section will be easy and make more sense as you run through the examples.

Let's take a look at a default Kitchen configuration file. To make sure that we have a "default" file, we will actually generate a new cookbook using the `chef generate` command (note that you may have already done this if you have been following along in this chapter):

```
cd ~/chef-repo/cookbooks
chef generate cookbook mywebapp
cd mywebapp
berks init
```

This will create a template cookbook named mywebapp in your `chef-repo/cookbooks` folder. As we saw in the previous section, the files created by the `chef generate` command include test input files we can build on. From these test files, we want to begin by examining the Kitchen configuration file. It is found in the root of the cookbook and has the name `.kitchen.yml`. Here is what that file looks like after you've run the preceding commands (note that the `berks init` command overwrites the `.kitchen.yml` file created by the `chef generate` command):

```
cd ~/chef-repo/cookbooks/mywebapp
cat .kitchen.yml
---
driver:
  name: vagrant
provisioner:
  name: chef_solo
platforms:
  - name: ubuntu-14.04
  - name: centos-7.1
suites:
  - name: default
    run_list:
      - recipe[mywebapp::default]
attributes:
```

As you can see in the file contents shown earlier, we have four sections of configuration info: driver, provisioner, platforms, and suites. Beginning with the driver section, the default driver name is Vagrant. What this setting means is that Kitchen will use Vagrant as the mechanism to create and manage our test instances. Several other options for drivers are available to be installed and used, such as Docker, EC2, Google, and OpenStack, to name a few. The Kitchen-Vagrant driver is an excellent option for our use, as it is installed as part of ChefDK, and it is quick and easy to integrate with an installation of VirtualBox. This option keeps all of the provisioning local to your workstation, which allows for quick integration testing.

The next section in the configuration file calls out the type of provisioner that will be used. There are a few choices for provisioner, with `chef_solo` and `chef_zero` as the most common. The provisioner determines how `chef-client` is run, and for the tests in this set of examples, leaving the value of `chef_solo` will be fine. The `chef_zero` provisioner is another common option as it is more like a full Chef server, but a Chef server in a local mode. There is another provisioner that may quickly become common, the `policyfile_zero` provisioner. This is provisioner that should be used when your run list is a policyfile.

Next, we have the platforms section. The platforms section defines the operating systems that we want to provision as test nodes. By the way, another benefit of using the Vagrant driver is that we can use abbreviated names for the platforms. For example, you can see in the configuration file contents earlier that we have the names `ubuntu-14.04` and `centos-7.1`. This is identifying our platforms to specific versions of those OSes, without needing to fully identify the actual templates or other parameters that would be required if we were using a different driver.

Finally, we have the suites section. This defines the what-to-test configuration. In the suites section, you provide a name for each suite you wish to test and define what you want to be tested. In this case, we have just one suite named "default", and it is going to test the default recipe in the `mywebapp` cookbook. There is a virtually unlimited number of things that you can include in your suites configuration section, and it is way beyond the scope of this chapter to provide you with a significant sampling, so I urge you to visit the links provided in the references section later, specifically for the `.kitchen.yml` configuration settings and the getting started guide. For our tests, just leave this set to the default value, which is to test the `mywebapp` cookbook's default recipe.

There is one more comment about the platforms and suites sections of the `.kitchen.yml` file. The values provided in these two sections create a matrix of test scenarios. That is, if you have one platform value and one suite, those combine to create one test ($1 \times 1 = 1$). If you have two platforms and one suite, you will have two test scenarios – one suite tested in each of the two platforms ($2 \times 1 = 2$). If you have two platforms and two suites, you will have four test scenarios, with each suite tested in each platform ($2 \times 2 = 4$). Add the third platform, and you now have six test scenarios ($2 \times 3 = 6$), and so on. Make sense? To simplify the Kitchen examples that follow, I am going to comment out the Ubuntu platform line in our `.kitchen.yml` file so that we are focused on a single centos platform with a single suite to test. This will result in one test scenario.

If you did not create the contents for the default recipe described in the previous section of this chapter, you should add that to the default recipe file now:

```
cd ~/chef-repo/cookbooks/mywebapp  
vi recipes/default.rb
```

Add the following to the file:

```
package 'httpd'  
  
file '/var/www/html/index.html' do  
  content '<h1>Hello MyWebApp User</h1>'  
end
```

```
service 'httpd' do
  action [:enable, :start]
end
```

Great! That was a lot of preparation— mostly just a lot of description of what was autogenerated for us—but we are now ready to try out Test Kitchen.

Kitchen create

The first command we will try is the create command. When you execute the command, Kitchen will read the contents of the `.kitchen.yml` file:

```
cd ~/chef-repo/cookbooks/mywebapp
kitchen create
```

Based on its contents, Kitchen will use the selected driver to stand up one or more new servers as defined by the platforms configuration section. Based on our `.kitchen.yml` file, Kitchen will use Vagrant to deploy a new server based on the CentOS 7.1 platform. Since the default for the Vagrant driver is to use VirtualBox as the hosting provider, the server will be deployed as VirtualBox VM.

If this is the first time you have run the `kitchen create` command for a specified platform, the template image will be downloaded to your workstation before being deployed. Templates are cached so that when you run future `kitchen create` commands, they do not have to be downloaded again.

There is a fair amount of output generated from this command, but you can look for some specific bits of information. Based on the name value in the suites section (in our case "default") and the platforms listed in the platforms section (in our case "centos-7.1"), the `kitchen create` command will name our test server `default-centos-71`. In the `create` command output, you will see some lines that look like the following:

```
Vagrant instance <default-centos-71> created.
Finished creating <default-centos-71> (0m41.08s).
```

This is indicating that the new server instance has been deployed, and it is ready to run chef-client to prepare for the tests to follow. You should see a line like this for each of the test scenarios in the matrix defined by the `.kitchen.yml` file. Now let's use `kitchen converge` to run chef-client and converge our new server instance to the desired state.

At this point, if you so desire, you can log in to the instance with the command `kitchen login`. Type "exit" to log out of the instance.

Kitchen converge

To converge our server instance to the desired state, execute the following command:

```
cd ~/chef-repo/cookbooks/mywebapp
kitchen converge
```

This will instruct Kitchen to read the `.kitchen.yml` file, and then, based on the selected provisioner and the values defined in the suites section of the configuration file, Kitchen will install and run chef-client with specific settings in the deployed server instance.

By the way, you can actually skip calling the `kitchen create` command, and just call the `kitchen converge` command. This will automatically execute the `kitchen create` function first, if the server instances have not already been deployed with a `kitchen create` command.

Here is a portion of the type of output you should expect from running the `kitchen converge` command:

```
Installing chef
installing with rpm...
warning: /tmp/install.sh.12447/chef-12.8.1-1.el7.x86_64.rpm:
Header V4 DSA/SHA1 Signature, key ID 83ef826a: NOKEY
Preparing... #####
##### [100%]
Updating / installing...
#####
##### [100%]
Thank you for installing Chef!
Transferring files to <default-centos-71>
Starting Chef Client, version 12.8.1
Creating a new client identity for default-centos-71 using the
validator key.
resolving cookbooks for run list: ["mywebapp::default"]
Synchronizing Cookbooks:
- mywebapp (0.1.0)
Installing Cookbook Gems:
Compiling Cookbooks...
Converging 3 resources
```

You can see in this output snippet that the kitchen converge command installs chef, synchronizes the cookbooks listed in the suite's run-list, and converges the cookbook resources. Now, let's finally do some actual testing.

Kitchen verify

Now we have a server instance running, and it's been converged to the desired state, that is, our kitchen converge completed without errors. However, this does not mean that we have actually tested anything. To test the state, we need to run kitchen verify. Kitchen verify will look at our test spec files and will validate our converged instance to verify that it passes our defined tests. So, run the verify command now:

```
cd ~/chef-repo/cookbooks/mywebapp
kitchen verify
```

This will actually test our converged instance to see if it delivers the results we desire. Kitchen will be using ServerSpec to do our tests. So, one of the first things you will see in the output from the verify command is about installing ServerSpec and its required gems. Then, you will see the ServerSpec command used to run the tests, which will be followed by the results of the tests. Here is part of what that output will look like:

```
----> serverspec installed (version 2.31.0)
      /opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.4.1/lib:/tmp/verifier/gems/
gems/rspec-core-3.4.4/lib /opt/chef/embedded/bin/rspec --pattern /tmp/
verifier/suites/serverspec/*/*/*_spec.rb --color --format documentation
--default-path /tmp/verifier/suites/serverspec

mywebapp::default
  does something (PENDING: Replace this with meaningful tests)

  Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) mywebapp::default does something
     # Replace this with meaningful tests
     # /tmp/verifier/suites/serverspec/default_spec.rb:6
```

```
Finished in 0.00052 seconds (files took 0.31508 seconds to load)
1 example, 0 failures, 1 pending

Finished verifying <default-centos-71> (0m13.90s).
-----> Kitchen is finished. (0m14.59s)
```

You can see that one test (example) was performed and that the tests took just under 15 seconds to complete. But we didn't actually perform the BDD test we discussed earlier, which was to verify that the user gets an appropriate response when they visit our new app's website. This is because we didn't write that test yet. So, let's do that now. Edit our test file:

```
cd ~/chef-repo/cookbooks/mywebapp
vi test/integration/default/serverspec/default_spec.rb
```

The file contents will look like this:

```
require 'spec_helper'

describe 'mywebapp::default' do
  # Serverspec examples can be found at
  # http://serverspec.org/resource_types.html
  it 'does something' do
    skip 'Replace this with meaningful tests'
  end
end
```

We are going to replace the inner block test `it 'does something'` with a new test that verifies that the user gets an appropriate response when they visit our app's website. In this oversimplified example, the appropriate response is going to be Hello MyWebApp User. To create this test, replace the inner block of test code, so the resulting file looks like this:

```
require 'spec_helper'

describe 'mywebapp::default' do
  describe command('curl http://localhost') do
    its(:stdout) { should match(/Hello MyWebApp User/) }
  end
end
```

Now save the file and run the verify command again:

```
cd ~/chef-repo/cookbooks/mywebapp
kitchen verify
```

This time, you should see that we executed our test successfully, and in fact, the results prove that the user is getting an appropriate response when they visit our new app's website:

```
-----> Running serverspec test suite
      /opt/chef/embedded/bin/ruby -I/tmp/verifier/suites/serverspec
-I/tmp/verifier/gems/gems/rspec-support-3.4.1/lib:/tmp/verifier/gems/
gems/rspec-core-3.4.4/lib /opt/chef/embedded/bin/rspec --pattern /tmp/
verifier/suites/serverspec/*/*/*_spec.rb --color --format documentation
--default-path /tmp/verifier/suites/serverspec

mywebapp::default
  Command "curl http://localhost"
    stdout
      should match /Hello MyWebApp User/

Finished in 0.20631 seconds (files took 0.31911 seconds to load)
1 example, 0 failures

Finished verifying <default-centos-71> (0m3.20s).
-----> Kitchen is finished. (0m3.95s)
```

If you have not previously run `kitchen create` or `kitchen converge` and you run the `kitchen verify` command, kitchen will execute both the `create` and `converge` steps automatically before running the `verify` step.

However, it is important to note that running the `verify` command does not automatically rerun the `converge` command if it has already run. If anything related to the cookbook is changed between calls to `kitchen verify`, it is necessary to run `kitchen converge` in between. Often, you'll just want to issue the commands like this:

```
cd ~/chef-repo/cookbooks/mywebapp
kitchen converge && kitchen verify
```

With that command, you can be certain that your instance is converged to the currently defined state before you try to verify your test results. Running the `converge` and `verify` commands like this is a quick way to test changes to the cookbook and ServerSpec tests. Just be aware that if a cookbook resource has already been converged, it will result in a no-op for the `kitchen converge` command and will just report "resource up to date" in the output. This is a valid test case for verifying instance upgrade scenarios. However, if your goal is to verify the new instance deployment scenarios, you will want to add a `kitchen destroy` command in between your test runs. This will shut down and delete your deployed test instance.

The command for this is as you would expect:

```
cd ~/chef-repo/cookbooks/mywebapp  
kitchen destroy
```

Now when you run your kitchen verify command, a new instance will be deployed, converged, and verified. Of course, the wise people who developed Test Kitchen realize that a common test pattern is to issue the create, converge, verify, and destroy commands in sequence to do a full new deployment integration test. So, they have provided us with a single command that will do all of this for us. It actually will start with a destroy command to make sure that any existing running instances are cleaned up before we start the new tests. To execute this, you issue the command as follows:

```
cd ~/chef-repo/cookbooks/mywebapp  
kitchen test
```

If the verify tests are successful, a trailing kitchen destroy command will be executed to clean up your test instance. If the verify tests result in an error, the trailing destroy command will not be executed. This affords you the opportunity to investigate the test image to better debug the failure. Pretty sweet, right? You bet!

So, you now have the necessary skills to do full integration testing in your test-driven development cycle. Remember that what you have shown in this chapter is just the tip of the iceberg of what is available to you for your chef testing. Visit the links provided in the references and learn how to expand on what you have already learned. Your customers will be glad you did!

References

- The link to the Chef.io information about Test Kitchen is <https://docs.chef.io/kitchen.html>.
- You can find the Test Kitchen repo on GitHub at <https://github.com/test-kitchen/test-kitchen>.
- The link to the Test Kitchen home page is <http://kitchen.ci>.
- The link to the ServerSpec site is <http://serverspec.org>.
- The link to the home page for the Vagrant tool is <https://www.vagrantup.com>.
- The VirtualBox software can be found at the link <https://www.virtualbox.org/wiki/Downloads>.
- Learn about many options that can be used in the .kitchen.yml configuration file https://docs.chef.io/config_yml_kitchen.html.

- An excellent Test Kitchen getting started guide is found at <http://kitchen.ci/docs/getting-started>.
- Find information about the various ServerSpec resource tests available to you at http://serverspec.org/resource_types.html.

Summary

In this chapter, you learned how to use several of the tools available to implement a test-driven development cycle for infrastructure code development. We looked at tools that focus on cookbooks-as-code, validating that the code is syntactically correct, that it is structurally correct, and that it follows the coding standards we want all of our source code to employ. These tools include RuboCop and Foodcritic. Then, we looked at tools that fit into the cookbooks-as-artifacts model and test the functionality and deployment aspects of our cookbooks. These tools include ChefSpec and Test Kitchen. We learned that when tests are performed on our cookbooks in both models, the result is better infrastructure delivered faster than ever before. Remember that what you learned in this chapter is only the beginning. In fact, we barely scratched the surface. It is up to you, the reader, to persist in your education and the use of these test tools. Continue to the point where your normal workflow includes using these tools as a matter of habit. Like driving a car for the first time, there seems to be too much to do, too many controls, and so much at stake. But with time and practice, it all becomes natural. Everything is executed by habit, arriving at the destination safely and efficiently, almost without thinking about it at all. So it will be with your test-driven development journey. We are now very well equipped to handle our Chef Provisioning needs and deliver solid infrastructure across a wide variety of environments, but there is one more feature provided by Chef that we can use to accelerate our provisioning. Let's wrap all of this up in a nice bow, with a look at using Chef Provisioning.

8

Using Chef Provisioning

In this chapter, you're going to learn to use Chef provisioning.

Wait... what? This is the last chapter in the book titled Chef Provisioning, and now you are going to learn to use Chef provisioning? What were we doing in Chapters 1-7?

Yes, in Chapters 1-7, we were doing Chef provisioning, but ChefDK actually comes with a feature named Chef Provisioning. In the earlier chapters, you learned how to use Chef to provision into a variety of hosting solutions, both in the data center and the cloud. However, most of the work we did to provision our environments was done in single-instance bites, creating one server instance at a time. All perfectly acceptable and very valuable. However, in the world of a DevOps Engineer, it is a rare situation when a single instance is all that is needed for an application. These days, distributed applications are built in clusters. An app will require multiple web servers, databases, app servers, networking and load balancers, and so on. Wouldn't it be great if there was way to create a cookbook to describe our application's entire cluster as code? Well, there is, and it is called Chef provisioning.

"The devil came to me last night and asked what I wanted in exchange for my soul. I still can't believe I said pizza. Friggin' cravings." – Marc Ostroff

The following topics will be covered:

- What is Chef provisioning?
- How to use Chef provisioning

What is Chef provisioning?

If you have been using Chef for a while, you may have heard about a feature named Chef Metal. Chef provisioning is the rebranded and evolved tool formerly known as Chef Metal. Chef provisioning is the Chef of all things. It serves the purpose of turning your application-run books into code. Now, all the benefits of infrastructure as code that you have come to enjoy in order to manage your server instances can also be enjoyed in managing your entire application cluster. Chef provisioning lets you describe your entire app the way you currently describe a single machine, using resources. Because of this, Chef provisioning allows you to test an entire distributed application at once.

It's highly idempotent

Chef Provision is highly idempotent, meaning that you can run the same Chef provisioning command repeatedly, and only the specific resources that have changed since the previous run will be affected.

For example, when you first run the Chef Provisioning command to create a new application cluster, all the machines described will be deployed, and they will be converged to their desired state. If you're testing changes to a recipe used in the run-list of a specific role and you run the Chef Provisioning command again, Chef Provisioning will not redeploy the machines already running. It will just load the modified recipe to the instances using the modified role and reconverge only those instances to the new desired state.

This is the same idempotency behavior you have learned to depend on when using Chef for your single-instance work.

Consider the benefit of being able to rerun your Chef Provisioning commands, where they only affect changes in an application's cluster if something has diverged from the fully described cluster recipe.

A hypothetical example

Imagine that you have a new distributed web application that you need to deploy. Here is an example of how you could use Chef Provisioning to do this for you.

1. You can create a cluster recipe that, when converged, deploys your app's fully realized cluster. The recipe includes all the required web servers, DBs, load balancers, and so on.

2. You create a recipe for an "app provisioner" instance that, when converged, runs the Chef Provisioning command with the cluster recipe created in step 1.
3. You deploy your app provisioner. The app provisioner converges, and as part of that convergence, it deploys your entire app cluster. All the web servers, the DBs, the load balancers, and so on, all get stood up and configured, and they are ready to be used by your customers. And by the way, when app provisioner converges, it is configured to re-run Chef Provisioning with the app cluster's recipe every 15 minutes. Because of the idempotency of Chef Provisioning, each run is a no-op until...
4. ...Time passes with your app running and serving your happy customers, but then some unforeseen event causes a web server to crash and burn. Yikes! No need to worry. It's okay. Because the next time your app provisioner runs the Chef Provisioning command, it will see that one of the web servers you defined is no longer "as defined". Chef provisioning will deploy a new web server based on your cluster recipe, converging it to the desired state, and voilà, you are back to 100% functionality.
5. (Bonus step) If you really wanted to go all inception-like on this idea, you could add the ability to deploy your app provisioner to the instances in your app cluster so that they would monitor, and if necessary redeploy the provisioner. They watch the watcher. Where's my spinning top?

References

- Learn more about Chef provisioning on the [chef.io site, https://docs.chef.io/provisioning.html](https://docs.chef.io/provisioning.html).
- Review the code for Chef provisioning in the GitHub repo at <https://github.com/chef/chef-provisioning>.

How to use Chef Provisioning

Now that you have a picture of what Chef provisioning is and how you might use it to turn an application's cluster infrastructure into code, let's take a look at how to actually use it. Like other aspects of Chef, we create recipes for Chef provisioning. And like the recipes in our other cookbooks, these recipes will contain resource definitions that will describe the desired state of our application cluster.

So, let's begin by learning about the resources that we can create in our Chef provisioning recipes.

Chef Provisioning resources

A Chef Provisioning recipe can contain the definitions of the following resources:

- **with_driver**: This resource tells Chef Provisioning the name of the driver to use with subsequent machine resources. The driver name corresponds to the hosting provider, such as Vagrant, AWS, and Docker. More than one with_driver resource can be used in a recipe, providing for the ability to deploy subsets of machines to different hosting providers within the same application cluster. For example, with one recipe, you can deploy instances to both vSphere and AWS.
- **machine_image**: This resource allows you to define a template to be used when describing your machine resources.
- **machine**: This is the primary resource for your Chef Provisioning recipes. This is what you use to describe your instances.
- **machine_options**: This resource is used as a helper resource to define your machine resources.
- **machine_batch**: The machine_batch resource provides a way of describing multiple machine resources that can be deployed concurrently. When Chef Provisioning is executing recipes, it does so sequentially, in the order described. However, using the machine_batch resource tells Chef Provisioning that the resources described within can be executed concurrently.

Each of the drivers associated with specific hosting providers may support additional resource types. For example, when you are using the AWS driver, some additional resources that are available include aws_key_pair, aws_launch_configuration, and aws_vpc. With the Vagrant driver, you have additional resources such as vagrant_box. Please visit the repo associated with the driver you are using to get the full list of driver-specific resources available. Links to the most common drivers are provided in the References section later.

A Vagrant example

Let's put this information to use. Since the Vagrant driver is included in ChefDK, and we don't have to do any additional installation work to use Chef Provisioning with Vagrant and VirtualBox, we'll start there.

Here is the requirement: Deploy the new "cool-app" cluster, which needs three instances—one DB and two web servers.

Here is the solution: we will create a cluster recipe to deploy three instances – one database server and two web servers.

Let's create a new app in our chef-repo:

```
cd ~/chef-repo
chef generate app cool-app
```

Now, let's create our recipe file:

```
mkdir -p cool-app/provision/recipes
touch cool-app/provision/recipes/app-cluster.rb
```

And now, let's edit our recipe file:

```
vi cool-app/provision/recipes/app-cluster.rb
```

Enter the contents shown later in the recipe file:

```
require 'chef/provisioning/vagrant_driver'
with_driver 'vagrant'

vagrant_box 'centos-7.1' do
  url 'https://github.com/CommanderK5/packer-centos-template/releases/
download/0.7.1/vagrant-centos-7.1.box'
end

with_machine_options :vagrant_options => {
  'vm.box' => 'centos-7.1'
}

machine 'db' do
  recipe 'postgresql'
  converge true
end

num_webservers = 2
machine_batch do
  1.upto(num_webservers) do |i|
    machine "web#{i}" do
      recipe 'apache'
      converge true
    end
  end
end
```

Save the file, and we will be ready to run our Chef Provisioning command. But before we do that, let's go over the recipe a bit.

1. We start it off with a require command that is calling out the dependency on the Vagrant driver.
2. Then, we actually describe the driver we are using in the `with_driver` resource.
3. Next, we use the `vagrant_box` resource, one that is obviously unique to the Vagrant driver, to define our centos-7.1 image.
4. In the next resource, `with_machine_options`, we are describing which image we want to use for our machine commands.
5. Next is our first `machine` resource. This one is describing our database instance. You can see that we are using the PostgreSQL recipe for our DB. Our Chef Provisioning cluster recipe will actually be using our public Chef server as defined in the `.chef/knife.rb` file, so our nodes will be registered to that Chef server as well. As such, the cookbook recipes that we use for our cool-app instances will need to have been uploaded to the Chef server before running the Chef Provisioning command.
6. The last resource block defined in our `app-cluster.rb` file is the `machine_batch` resource. This resource is describing the resources that can be deployed synchronously, in this case our two web servers, `web1` and `web2`. These instances will have the Apache recipe applied, which, like the PostgreSQL cookbook, needs to be uploaded to the Chef server before the execution of our Chef Provisioning command.

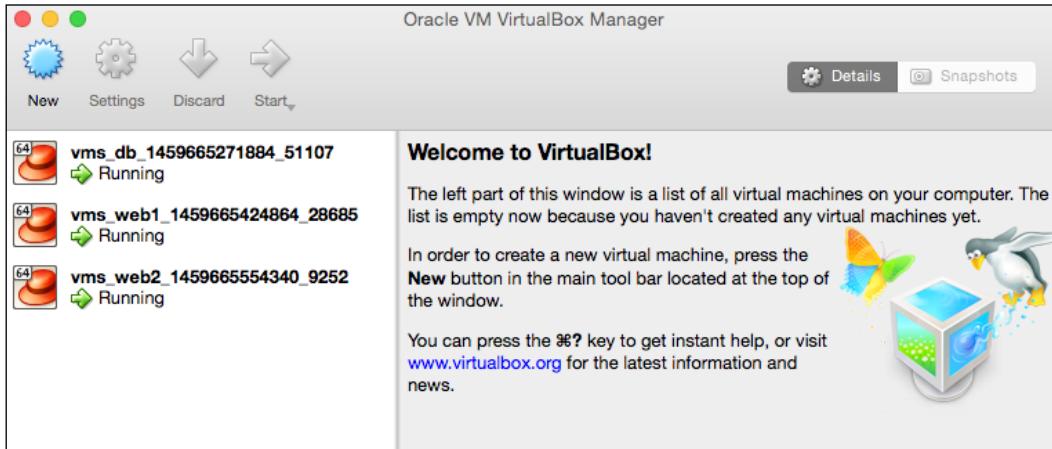
Okay, provided we have uploaded the necessary cookbooks to the Chef server, we are ready to run the Chef Provisioning command. So, let's do so.

There are three modes you can employ when using Chef Provisioning; two modes utilize policyfiles and one does not. For this example, we will be using the non-policy mode, so the commands to run Chef Provisioning are:

```
cd ~/chef-repo  
chef provision --no-policy --cookbook ./cool-app/provision -r app-cluster
```

We are telling Chef Provisioning that we are using the non-policy mode with the `--no-policy` parameter, then using the `--cookbook` parameter, we say where to find our cluster recipe, and finally using the `-r` parameter, we call out the name of our cluster recipe.

There will be a lot of output during the execution of this command, far too much to show in this space, but if everything goes as planned, your VirtualBox will have three new instances running in it. It will look something like this:



In addition, your Chef server will have three new registered nodes in the Nodes list: db, web1, and web2. It will look something like this:

The screenshot shows the Chef Manage interface, specifically the Nodes page. The left sidebar has a 'Nodes' section with options like Delete, Manage Tags, Reset Key, Edit Run List, and Edit Attributes. The main area shows a table titled "Showing All Nodes". The table has columns for Node Name, Platform, FQDN, IP Address, Uptime, Last Check-In, and Environment. There are three entries in the table:

Node Name	Platform	FQDN	IP Address	Uptime	Last Check-In	Environment
web2	centos	web2	10.0.2.15	2 minutes	14 minutes ...	_default
db	centos	db	10.0.2.15	a minute	20 minutes ...	_default
web1	centos	web1	10.0.2.15	3 minutes	19 minutes ...	_default

Okay, we've done a test deployment of our app cluster using Chef Provisioning, and we have these three instances and associated nodes in our Chef server that we don't really want hanging around. So what do we do?

Fortunately, Chef Provisioning has resource actions for cleanup time too. We can create a destroy-all recipe that can be called upon by Chef Provisioning that will "destroy all".



Be careful! This is a quick example script to clean up our test environments. It will unceremoniously deregister all your nodes and delete all the associated instances. I am leaving it to you, the reader, to modify the recipe to be more focused in its clean-up actions.

With that advisory said, here is the destroy-all recipe:

```
cd ~/chef-repo  
vi cool-app/provision/recipes/destroy-all.rb
```

Enter the contents as follows:

```
require 'chef/provisioning'  
machine_batch do  
  machines search(:node, '*:*').map { |n| n.name }  
  action :destroy  
end
```

Save the file and run the following command:

```
cd ~/chef-repo  
chef provision --no-policy --cookbook ./cool-app/provision -r destroy-all
```

Now all of the nodes and associated instances are gone!

A cloud example with AWS

Now that we have successfully deployed our cool-app cluster onto our local workstation with Vagrant and VirtualBox, it's time to try to deploy the app to the cloud. Let's use AWS for this example, creating EC2 instances using Chef Provisioning. This time, however, we do need to install the AWS driver. To do this, enter the following commands:

```
cd ~/chef-repo  
chef gem install chef-provisioning-aws
```

I've found that the AWS driver is a little finicky about a few configuration settings. Follow these three steps to make sure that your experience is a successful one.

Create a "config" file in your `~/.aws` folder. The contents of the file provide a default section with the AWS region you want to work with. Here are the steps:

```
cd ~/.aws  
vi ./config
```

Enter contents like the following:

```
[default]  
region = us-west-2
```

It is expecting your credentials to be in the `~/.aws/credentials` file. The file has to have the name `credentials`, and it has to be in that exact location.

Your key pair files need to be in the `~/.ssh/` folder. Even though there is a parameter available in the `with_machine_options` resource for `key_path` that you can fill with a different path for your keys, it seems that the AWS driver only looks in `~/.ssh`. So, just make sure to save your key files in the `~/.ssh` folder.

Okay, now let's create our new AWS cluster recipe file:

```
cd ~/chef-repo  
touch cool-app/provision/recipes/aws-cluster.rb
```

Edit the new recipe file:

```
cd ~/chef-repo  
vi cool-app/provision/recipes/aws-cluster.rb
```

Enter the following as the contents in the file:

```
require 'chef/provisioning/aws_driver'  
with_driver 'aws'  
  
with_machine_options ({  
  bootstrap_options: {  
    image_id: 'ami-8d5b11bd',  
    instance_type: 't1.micro',  
    key_name: 'chef-server-ec2',  
    security_group_ids: 'chef-provisioning',  
  },  
  use_private_ip_for_ssh: false,  
  transport_address_location: :public_ip,  
  ssh_username: 'root',
```

Using Chef Provisioning

```
availability_zone: 'us-west-2a',
})

machine 'db' do
  recipe 'postgresql'
  converge true
end

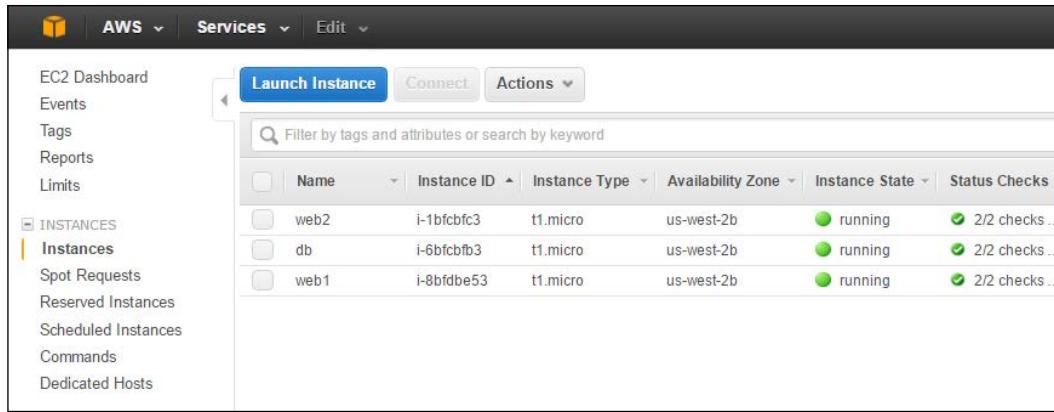
num_webservers = 2
machine_batch do
  1.upto(num_webservers) do |i|
    machine "web#{i}" do
      recipe 'apache'
      converge true
    end
  end
end
```

Now save the file, and let's talk about some of the differences between this recipe and the previous one that we used to provision with the Vagrant driver. The first two directives, `require` and `with_driver`, are specific to our desire to provision using AWS. Then, we have a `with_machine_options` resource. This is the same type of resource we used in the Vagrant example, but this time, we have added several additional parameters that are needed for deployment to AWS, all of which should be very familiar to you if you read *Chapter 6, Provisioning in the Cloud*. After this, we have the `machine` and `machine_batch` resources, which are identical to the ones we used in the earlier example.

Okay, now let's run the Chef Provisioning command to deploy and converge our cluster:

```
cd ~/chef-repo
chef provision --no-policy --cookbook ./cool-app/provision -r aws-cluster
```

As before, our cluster recipe is used by Chef Provisioning to deploy our cool-app cluster, and we have fully realized deployment with a single command. The results on AWS should look something like this:



A Docker example

Okay, we are able to use Chef Provisioning to deploy our cluster to our local workstation using the Vagrant driver, and we can deploy the cluster to the cloud with the AWS driver. Let's turn our attention back to Docker and see how we can deploy our cluster as Docker containers.

We need to install the Docker driver for Chef Provisioning. To accomplish this, issue the following command:

```
cd ~/chef-repo
chef gem install chef-provisioning-docker
```

Excellent. Now let's create a new cluster recipe file:

```
cd ~/chef-repo
touch cool-app/provision/recipes/docker-cluster.rb
```

Fill the new recipe file with the following:

```
require 'chef/provisioning/docker_driver'
with_driver 'docker'

machine_image 'ubuntu' do
  machine_options :docker_options => {
    base_image: {
      name: 'ubuntu',
      repository: 'ubuntu',
      tag: '14.04'
    }
  }
}
```

Using Chef Provisioning

```
end

machine 'db' do
  from_image 'ubuntu'
  recipe 'postgresql'
  converge true
end

num_webservers = 2
machine_batch do
  1.upto(num_webservers) do |i|
    machine "web#{i}" do
      from_image 'ubuntu'
      recipe 'apache'
      converge true
    end
  end
end
end
```

This recipe should now seem pretty familiar. Really the only thing that is new here is that we've used a `machine_image` resource to define what our Docker image should look like, and in the machine resources, we've included the `from_image` directive. Neat, right?

Since this is a section on using Chef Provisioning with Docker, if you haven't installed your Docker environment yet, now this would be a good time to do so. Visit the Docker Toolbox site so you can download and install the toolbox.

Take a look at the URL, <https://www.docker.com/products/docker-toolbox>. Once your toolbox is installed, you'll want to launch both the "Kitematic UI" and the "Docker Quickstart Terminal". These are the tools provided by the Docker Toolbox:



Now, within the Quickstart Terminal, you will run your Chef Provisioning commands. Let's go ahead and run the command:

```
cd ~/chef-repo
chef provision --no-policy --cookbook ./cool-app/provision -r docker-cluster
```

There will be significant output from this command, but it is interesting, so here is what you should see:

```
$ cd ~/chef-repo
$ chef provision --no-policy --cookbook ./cool-app/provision -r docker-cluster
Installing Cookbook Gems:
Compiling Cookbooks...
Recipe: provision::docker-cluster
  * machine_image[ubuntu] action create (up to date)
  * machine[db] action converge
    - create node db at https://api.chef.io/organizations/sdearl
    - add normal.tags = nil
    - add normal.chef_provisioning = {"from_image"=>"ubuntu",
"reference"=>{"driver_url"=>"docker:tcp://192.168.99.100:2376", "driver_version"=>"0.9.0", "allocated_at"=>"2016-04-04 04:07:32 UTC", "host_node"=>"https://api.chef.io/organizations/sdearl/nodes/", "container_name"=>"db", "image_id"=>nil, "docker_options"=>nil, "container_id"=>nil}
    - update run_list from [] to ["recipe[postgresql]"]
    - create container to converge db
    - start converge container chef-converge.db
    - commit and delete converged container for db
    - update node db at https://api.chef.io/organizations/sdearl
    - update normal.chef_provisioning.reference.container_id from nil to "43ac4f550657946e9205315d4bddde049ca2ba0ad769854075f9bfd74d3df23"
    - create final container for db
    - start container db
  * machine_batch[default] action converge
    - [web1] create node web1 at https://api.chef.io/organizations/sdearl
    - [web1] add normal.tags = nil
```

Using Chef Provisioning

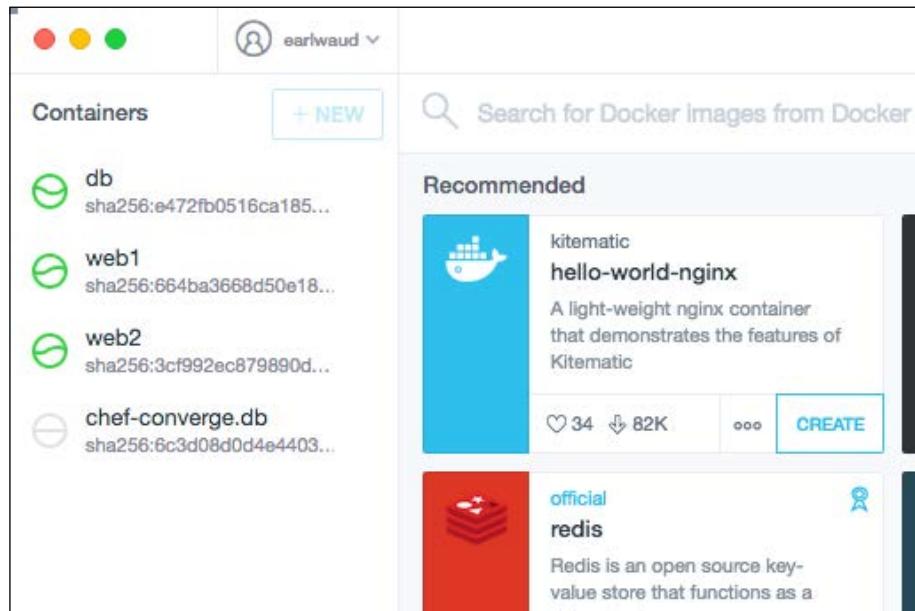
```
- [web1]   add normal.chef_provisioning = {"from_image"=>"ubuntu",
"reference"=>{"driver_url"=>"docker:tcp://192.168.99.100:2376", "driver_
version"=>"0.9.0", "allocated_at"=>"2016-04-04 04:07:36 UTC", "host_
node"=>"https://api.chef.io/organizations/sdearl/nodes/", "container_
name"=>"web1", "image_id"=>nil, "docker_options"=>nil, "container_
id"=>nil}}
- [web1]   update run_list from [] to ["recipe[apache]"]
- [web2]  create node web2 at https://api.chef.io/organizations/sdearl
- [web2]  add normal.tags = nil
- [web2]  add normal.chef_provisioning = {"from_image"=>"ubuntu",
"reference"=>{"driver_url"=>"docker:tcp://192.168.99.100:2376", "driver_
version"=>"0.9.0", "allocated_at"=>"2016-04-04 04:07:36 UTC", "host_
node"=>"https://api.chef.io/organizations/sdearl/nodes/", "container_
name"=>"web2", "image_id"=>nil, "docker_options"=>nil, "container_
id"=>nil}}
- [web2]  update run_list from [] to ["recipe[apache]"]
- [web1]  create container to converge web1
- [web2]  create container to converge web2
- [web1]  start converge container chef-converge.web1
- [web2]  start converge container chef-converge.web2
- [web1]  commit and delete converged container for web1
- [web2]  commit and delete converged container for web2
- [web1]  update node web1 at https://api.chef.io/organizations/sdearl
- [web1]  update normal.chef_provisioning.
reference.container_id from nil to
"fcbdc9eb4de28474237c4ec146f3852ce796ff998af23c5bff5792a3a224d119"
- [web1]  create final container for web1
- [web2]  update node web2 at https://api.chef.io/organizations/sdearl
- [web2]  update normal.chef_provisioning.
reference.container_id from nil to
"c6c89b0c946cb76156fbaefb32a3f54bdbale85f3a25025305ab6aa56134160d"
- [web2]  create final container for web2
- [web2]  start container web2
- [web1]  start container web1
```

Some things to note in the output include the following:

1. The machine image "Ubuntu" is created first.
2. Then, we converge the DB machine. This phase ends with the starting of the DB container.
3. After this, we get to the `machine_batch` resource, which will converge two machines – web1 and web2.

Note how the output from those actions are intermixed. This is because of the synchronous nature of using the `machine_batch` resource type. Here too, this ends with the starting of our two web containers. Success!

Now we have our cool-app cluster running in Docker containers. It should look something like this:



References

- Here are the GitHub repos for a few of the Chef Provisioning drivers, each of which provides examples for using Chef Provisioning as its named provider:
 - AWS driver: <https://github.com/chef/chef-provisioning-aws>
 - Azure driver: <https://github.com/chef/chef-provisioning-azure>
 - Docker driver: <https://github.com/chef/chef-provisioning-docker>
 - Vagrant driver: <https://github.com/chef/chef-provisioning-vagrant>

- Google driver: <https://github.com/chef/chef-provisioning-google>
- VSphere driver: <https://github.com/CenturyLinkCloud/chef-provisioning-vsphere>
- Here is the link to the public Vagrant boxes: <http://www.vagrantbox.es>
- Learn all of the AWS create instance options at http://docs.aws.amazon.com/sdkforruby/api/Aws/EC2/Resource.html#create_instances-instance_method
- You can download the Docker Toolbox at <https://www.docker.com/products/docker-toolbox>

Summary

In this chapter, we have learned how to master Chef Provisioning Finally.

Index

A

affected nodes
displaying, before uploading cookbooks 255, 256

Amazon EC2
about 48
authentication, setting up 48, 49
Chef node, removing 52
instance, bootstrapping 50, 51
instance, provisioning 49, 50
instance, terminating 51
knife plugin, installing 48
working with 48

Amazon EC2 authentication
setting up 48, 49

Amazon EC2 console
URL 602

Amazon EC2 instances
managing 436-439

Amazon Elastic Compute Cloud (EC2) 436

Amazon Machine Image (AMI) 438

Amazon Web Services (AWS) 436

AMI ID
URL 49

Apache Solr documentation
URL 155

application
configuring 117

application run
maintaining 118

application template
defining 142-144

APT repositories 345

arbitrary node attributes
finding 319

arguments
passing, to shell commands 312, 313

Atom editor
URL 488

attribute precedence hierarchy
about 510
attribute precedence factors 510, 511
resulting precedence hierarchy 512

attributes
about 24, 62-64
configuration values 62
multiple files 64
overriding 314-316
URL 317
used, for configuring recipes 278-280
using 66

authorized keys, SSH keys
templating 136, 137

automation, Chef 188

AWS Access Key ID 436

AWS Console
URL 436

AWS create instance options
URL 672

AWS driver
URL 671

AWS EC2 instances
Chef Server, installing 598-602
EC2 authentication 594
key pairs, creating 594-596
provisioning 594
security groups, creating 596, 597

AWS Identity and Access Management (IAM) 436

AWS Management Console 436

AWS Marketplace

URL 603

AWS Secret Access Key 436**Azure driver**

URL 671

B**Base64 encoding** 150**base image container**

URL 630

baseline role 108**base server role**

creating 119

BASH Automated Testing System

(BATS) 182

Berkshelf

about 457

cookbook dependencies, managing
with 213-217

URL 217

Blueprintcookbooks, creating with 409-412
URL 412**Boolean operators**

using, in search 319

bootstrap 24**bootstrapping data**

URL 487

bootstrapping nodes, with knife

URL 430

bootstrap script 24**breakpoint resource, Chef shell**

using 175, 176

Bundler, ChefSpec 94**Busser** 253**C****Capistrano**

Chef, using with 187

Capistrano

about 395

URL 358

CentOS instance image

adding, to provision 576, 577

Certified Culinary Educator (CCE) 479**Chef**

about 23, 446, 447

attributes 24

automation 188

benefits 4, 45

bootstrap 24

bootstrap script 24

Chef Development Kit (ChefDK) 452

chef-solo, installing 5

chef-solo, verifying 6-8

cookbook 24

data bags 25, 451

dependencies, managing 115

environment 25

environments 451

extending 184

gems, managing 5, 6

integration 188

node 23

Ohai 449

Ohai plugin, writing 184

on-premise (private) Chef Server,

setting up 452-456

provider 25

recipe 24

references 447, 456

resource 24

role 24

roles 450

Ruby gem mechanism 5

run List 24

run lists 449

supermarket 452

terminology 3

updates, automating 188

used, for software deployment 106

using, with Capistran 187

version, determining 446

working with 4, 5

workstation 23

chef-apply

about 475

advantages 475, 476

references 476

Chef client

installing 194

running, as daemon 235, 236

chef-client installation
automated, URL 591
manual, URL 591

Chef client runs
debugging 260, 262
results, inspecting 262, 263

Chef community
URL 108

Chef, components
Chef server 448
nodes 448
workstations 448

Chef cookbook
for Docker, URL 629
problems, flagging 242-244
testing 240-242

Chef Development Kit (ChefDK)
about 445, 452, 457
community tools 457
downloading 458
installing 458, 459
installing, on Ubuntu 460
installing, on Windows workstation 464
Knife 457
references 458, 464
URL 458
verifying 460-464

ChefDK, on GitHub
URL 198

Chef Domain Specific Language (DSL)
Ruby, mixing with 284-286

Chef environments
_default environment 506
Chef.io documentation, URL 508
cookbook versions constraints, URL 508
cookbook versions, specifying 507
creating 506
using 506

chef-full bootstrap script
URL 430

Chef infrastructure
loading, from file with knife 440, 442
loading, from file with spiceweasel
tool 440, 441

Chef.io
and Amazon AWS, URL 602

Chef.io slide deck
URL 537

Chef Junos integration
URL 591

Chef LWRP, component
provider 145
resource 145

Chef Management Console
URL 509

Chef node, Amazon EC2
removing 52

Chef node, Rackspace Cloud
removing 57

Chef organizations
access control 509
Chef.io reference, URL 509
creating 508
using 508

Chef products
URL 202

Chef provisioning
about 658
chef.io site, URL 659
cloud example, with AWS 664-666
Docker example 667-670
hypothetical example 658, 659
idempotent 658
in GitHub repo, URL 659
resources 660
used, for creating infrastructure 406-408
using 659
Vagrant example 660-664

Chef provisioning documentation
URL 409

Chef repo
.chef folder files 467, 468
Chef tool, using 465, 466
references 471
self-signed certificates 469, 470
setting up 464
starter kit 466, 467
tree script, creating 468

Chef roles
Chef.io documentation, URL 505
creating 500-502
downside 505
nested roles, using 504

per-environment run-lists, using 502, 503
using 499, 500

Chef run
URL 266

Chef run lists
URL 258

Chef server
about 194, 448
configuring 13
files, inspecting on 208, 209
installing 9
installing, into EC2 instance 598-602
interacting, Chef shell used 166, 167
node, deleting from 224, 225
URL 11, 409

Chef server configuration
about 13
chef-server-ctl, working 14
host server actions, monitoring 14, 15
services, verifying 16

chef-server cookbook
URL 603

chef-server-ctl, Chef server
working 14

Chef server, installing on Red Hat Enterprise Linux
package, downloading 12
package, downloading, URL 12

Chef server, installing on Ubuntu
about 11
package, downloading 12
package, downloading, URL 12
package, installing 12

Chef server interaction, with Chef shell
data, editing 168
data, interacting with 167
data, searching 168
data, transforming 168, 169

Chef Server-specific
URL 603

Chef service
about 4
knife configuration 19
validating 17-19

chef-shell
URL 238
using 236-238

Chef shell
about 165
breakpoint resource, using 175
client mode 166
recipe, creating in 170
solo mode 165
standalone mode 165
used, for debugging 175
used, for executing recipes 169
used, for interacting with Chef server 166, 167
using 165

chef-solo
installing 5
verifying 6-8

Chef-solo
about 157
configuring 162
cookbooks_path 162
data_bags_path 162
data bags, using with 163, 164
limitations 161
roles_path 162
roles, using with 163, 164
run directions, providing 163
Vagrant, combining with 161-163

ChefSpec
about 90, 458
and RSpec 91
dependencies, locking in Ruby 94, 95
installing 94
on Chef.io site, URL 646
overview 93, 94
repo on GitHub, URL 646
repo, URL 646
simple recipe, creating 95
testing basics 92
tests, expanding 643-645
unit testing with 639
using 93, 639-642

ChefSpec README
URL 248

ChefSpec repository, on GitHub
URL 249

ChefSpec test
expanding 100
multiple examples 100, 102

testing, for multiple platforms 102, 103
writing 96, 97

Chef Supermarket
URL 31

Chef + VMware
URL 558

Chef + VMware integration
URL 558

chef-zero
URL 227

Cisco cookbook repo
on GitHub, URL 591

Cisco systems
automating 585-588

classic
URL 603

Classic UI
URL 612

client mode, Chef shell 166

cloud
leveraging 47

cloud platform providers
Amazon EC2 48
Rackspace Cloud 53

command
running, after file updation 353-355
running, on many machines
at once 412, 413

community Chef style
using 277, 278

community cookbooks
extending, application wrapper
cookbooks used 298-300

community exception
using 268, 270

community tools 457

components, cookbooks
attributes 61
definitions 61
providers 61
recipes 61
resources 61
Ruby libraries 61
support files 62
templates 61
tests 62

components, for Chef server installation 10

Compute Engine (GCE)
Chef server, deploying 619-622
on Chef.io, URL 622
virtual machines, provisioning
with 616-619

conditional execution
URL 338
used, for making recipes
idempotent 337, 338

configuration blocks
using 173

configuration data
attribute data example 42
data bags 43
organizing 40, 41
sources 40

configuration files
creating, templates used 340-342

Configuration Management 281, 340

containers
Docker, installing on Chef Workstation 627
Docker, using on Chef Workstation 627
hosting, by setting up Docker
server 624-627
knife-docker plugin, using 628, 629
provisioning, in Docker 624
URL 630

continuous integration (CI) 244

cookbook dependencies
defining 211, 212
managing, with Berkshelf 213-217

cookbooks
about 4, 24, 32, 59, 60, 449
application, configuring 117
application deployment cookbook,
viewing 110-112
application run, maintaining 118
components 59
creating 206, 207, 208
creating, from running system
with Blueprint 409-412
database recipe, viewing 109, 110
dependencies, installing 115
diff-ing, with knife 266, 267
directories, preparing 112, 113
downloading 108, 109, 217-221
freezing 233, 234

installing 32
integrating 217-222
managing 108
Python's requirements file, using 116
Python virtual environment,
 constructing 113
source code, checking out 114
URL 108
URL, for downloading 108
using 206-208

cookbook, Test Kitchen
 created instance, converging 181
 instance, provisioning 180
 summarizing 184
 test, writing 182, 183

cross-platform cookbooks
 case statements, avoiding 333
 support, declaring for specific
 operating system 334
 writing 331, 332

custom bootstrap scripts
 using 428, 429

custom definition, developing
 about 139
 application template, defining 142, 144
 code, organizing 139
 definition, writing for using PIP 139-142

Customer Resource
 example use case, creating 526

custom extensions
 resource, building 145
 writing 139

custom handlers
 creating 270-272

custom knife plugins
 creating 304-307
 URL 308
 using 222, 223

custom Ohai plugins
 creating 300-303

Custom Resource
 about 525
 Chef.io slide deck, URL 537
 converting, into library module 541, 543
 creating 531, 532
 DSL, URL 541

library module Custom Resource,
 subclassing 543-545
prior to version 12.5, URL 525
subclassing 541
URL 525
using 531-537
website Custom Resource, creating 532-535

Custom Resource, making
 idempotence, handling 539, 541
 more reusable 537
 mywebapp example, code reusing 538
 only cookbook 538
 reusable Custom Resource only cookbook,
 using 538, 539

D

data

 decrypting 152
 encrypting 151
 querying 153, 154
 searching 131, 153
 storing, in data bags 129

data bags

 about 25, 43, 281, 320, 451
 creating, for users 129, 130
 data, searching 153
 data, storing in 129
 multiple machines, managing with
 search queries 155
 searching, from recipes 153
 searching, with knife tool 153
 securing 150
 URL 323
 users, creating from 364-367
 using 43, 44, 320-322
 working with 150

data bag search

 URL 324

data bags, securing

 about 150
 data, decrypting 152
 data, encrypting 151
 keys, storing on nodes 152
 secret keys 150

data bags, using with Chef-solo

 about 163, 164

custom JSON data, injecting 164
custom node name, providing 164
data bag values
 accessing, from external scripts 327-329
database cookbook
 about 385
 URL 389
database host
 configuring 123
database recipe
 viewing 109, 110
database server role
 creating 120
data, Chef server
 editing 168
 interacting 167
 searching 168
 transforming 168, 169
data, searching
 data bags, searching from recipes 153
 data bags, searching with knife tool 153
 data, querying 153, 155
default action 73
default behavior, resource
 overriding 73
definition
 about 61, 81, 82, 139, 291
 using 291, 293
dependencies
 installing 115
 managing 116
 managing, in Chef 115
deployment keys, SSH keys
 adding 138, 139
deploy users 138
DevStack
 downloading 563-566
 installing 563-566
 URL 581
DevStack Host VM
 deploying 560-563
DevStack OpenStack all-in-one environment
 setting up 560
DevStack public key
 setting up 571, 572

different files
 distributing, based on target platform 360-362
directories
 preparing 112, 113
directory resource
 URL 360
directory trees
 distributing 355-357
Docker
 containers, provisioning 624
 installing, on Chef Workstation 627
 knife-docker plugin, using 628, 629
 server setting up, to host containers 624-627
 tutorial, URL 630
 URL 629
 using, on Chef Workstation 627
Docker driver
 URL 671
Docker repo
 URL 629
Docker Toolbox
 URL 630, 672
dotCloud
 URL 630
driver plugins 177
DRY (Don't Repeat Yourself) 248
dry-run mode 148
DSL (Domain-specific Language) 61, 204, 276, 524

E

EC2 instances
 database host, configuring 123
 provisioning 122
echo off state 170
echo on state 170
encrypted data bag items
 private key file, using 326, 327
 URL 327
 using 324-326
encryption key
 storing, on nodes 152
EnterpriseDaddy
 URL 548

environment
about 25, 451
defining 330, 331
example 39
using 229-233

environment variables
setting 310, 311

equal sign 77

ERB primer
about 76
Ruby code, executing 76
Ruby code execution, variable replacement 77
URL 76

Erubis 283

examples, knife show
URL 210

example use case, Customer Resource
about 526
chef-client documentation, URL 531
cookbook, creating 526, 527
default recipe, creating 527-530
goal 526
recipe 531

Exception and Report handlers
URL 270

exceptions
logging, in recipes 264, 265
raising, in recipes 264, 265

external monitoring services
SNMP, setting for 414, 415

F

F5 Big IP cookbook
downloading, URL 591
URL 591

F5 LTMs
automating 588, 589

fail2ban cookbook
URL 436

fail2ban tool
about 433
managing, to ban malicious IP addresses 433-435
URL 436

fatal! method
URL 266

Fauxhai 247
URL 250

files
inspecting, on Chef server with knife 208-210

firewalls
managing, with iptables 430-433

Flowdock
about 270
URL 270

Foodcritic
about 457
cookbooks, analyzing with 636
customizing 638
on Chef.io site, URL 638
output, analyzing 637
repo on GitHub , URL 638
rules, URL 638
URL 637, 638
using 636

Forbidden error
URL 581

Free Ruler 402

Fully Qualified Domain Name (FQDN) 283

G

gems
managing, with RVM 5

gemset 6

Git
URL 197, 395

GitHub
URL 197

Google Cloud platform
Chef server, deploying into GCE instance 619-621
key pairs, creating 615, 616
key pairs, registering 615, 616
knife-google plugin, configuring 613-615
knife-google plugin, installing 613
provisioning 613
virtual machines, provisioning with GCE 616-618

Google Cloud SDK
 URL 622
Google Compute Engine (GCE) 613
Google Compute Engine Machine Types
 URL 622
Google driver
 URL 672

H

handlers
 URL 268
HAProxy
 about 425
 used, to load-balance multiple
 web servers 425-428
HA services
 building, heartbeat used 420-424
heartbeat
 used, for building HA services 420-424
heartbeat configuration
 URL 425
heartbeat cookbook
 about 424
 URL 425
Heavyweight Resource Providers (HWRP) 524
high availability (HA) 420
hosted Chef 194
Hosted Chef
 URL 5
hosted Chef platform
 using 199-201
htpasswd generator
 URL 417
HTTP Upload Progress Module
 URL 381

I

Image-O-Rama 31
image-processing role
 defining 28
image search role
 defining 28
implications
 URL 371
Infrastructure as a Service (IaaS) 581

infrastructure modeling
 about 25-27
 Chef, using 26
 environment 39, 40
 nodes, converging 38
 recipes, applying to role 33
 recipes, determining 31
 role 27
 role, implementing 30
 role, mapping to node 36
 role, mapping to nodes 38
 service-oriented architecture,
 advantages 26
 service-oriented web application,
 services 26
 steps 25
installation
 Vagrant 158
installation, Chef server
 about 9
 components 10
 omnibus installer, obtaining 11
 on Red Hat Enterprise Linux 12
 on Ubuntu 11
 requisites 9
instance, Amazon EC2
 bootstrapping 50, 51
 provisioning 49, 50
 terminating 51, 52
instance, Rackspace Cloud
 provisioning 53-55
 terminating 55, 56
instance resource sizing
 URL 603
instance types
 URL 438
integration, Chef 188
integration testing
 about 177
 Test Kitchen, using 177
 tests, preparing 647-650
 with Kitchen and ServerSpec 646
integration testing, Chef cookbooks
 with Test Kitchen 250-254
Interactive Ruby shell (IRB) 336
iptables
 firewalls, managing with 430-433

using 301

iptables cookbook
about 433
URL 433

J

juniper devices
automating 582-585

junos-chef repo
URL 591

K

key pairs
creating 594-596
URL 603

Kitchen
converge 651
create 650
integration testing with 646
verify 652-655

Kitchenplan
about 402
URL 403

knife
used, to search nodes 319

knife-azure plugin
installing 604, 605
URL 612

Knife Block
URL 509

Knife Bootstrap 481-483

knife, Chef command-line tool 47

Knife command
URL 487

knife diff
URL 267

Knife Diff 483

knife-docker plugin
URL 630
using 628, 629

knife-ec2 plugin
URL 602

Knife EC2 plugin
about 491
EC2 Create Server plugin 496, 497

EC2 instances, URL 494
existing EC2 instance, bootstrapping 493
GitHub Knife EC2 plugin, URL 494
installing 491, 492
instance, creating 492, 493
instances, deleting 494
Knife Cloud plugins, URL 494
on GitHub page, URL 494
URL 494, 497

knife-esx plugin
URL 559

Knife Exec 484

knife-google plugin
configuring 613, 615
installing 613, 615
on GitHub, URL 622

Knife help
about 479, 480
URL 481

Knife Node 483

knife node list 166

knife-openstack plugin
URL 581

knife OpenStack plugin
installing 571

knife-openstack plugin knife settings
configuring 573-575

knife plugin, Amazon EC2
installing 48

Knife plugins
basic plugin file format 495, 496
creating 495

knife-preflight plugin
URL 256

knife-rackspace plugin 53

Knife Search
about 485, 486
URL 487

Knife SSH 485

Knife subcommands 481

Knife to editor, integration
about 487
wait option, uses 488
wait option, using 487

knife tool
about 17
used, for searching databags 153

knife-vsphere plugin
installing 549-551
URL 558
used, for provisioning VMs 551-554

Knife Windows plugin
about 489
GitHub repo, URL 490
installing 489
URL 490
Windows node, bootstrapping 489, 490
Windows node data, searching for 490

L

large-scale infrastructure 45

libraries
Chef documentation, URL 545
using 288-290

Lightweight Resource Provider (LWRP)
about 270, 275, 366, 524
creating 293-298
URL 298

Linode
about 622, 623
knife-linode plugin, URL 623
URL 623

list of operating systems
finding, for using in cookbooks 334-336

load_current_resource method 149

local environment
configuring 107

local mode
hosted Chef, moving 227
knife, running 227
recipes, developing with 225, 227
URL 227

local workstation
managing 400-403

M

malicious IP addresses
banning 433-435

Mash 331

matching ChefSpec test
creating 95

metadata 67, 68

Microsoft Azure instances
configuration, verifying 606
knife-azure plugin, installing 604, 605
new Chef server, provisioning into
Azure 610-612
plugin, configuring 605, 606
provisioning 603, 604
public/private key pairs, generating 607
virtual machines, provisioning 608, 609

Microsoft PowerShell

URL 613

Mina

URL 358

mock methods 91

multiple attribute files

about 64

external attributes, loading 65
multiple platforms, supporting 65

multiple machines

managing, with search queries 155

mysql 110

mysql cookbook

about 385

URL 389

MySQL databases

creating 385-388

MySQL users

creating 385-388

N

Nagios 416

nagios cookbook

URL 420

Nagios monitoring server

deploying 416-420

netdev repo

on GitHub, URL 591

network automation

Cisco systems, automating 585-588

F5 LTM, automating 588, 589

juniper devices, automating 582-585

new host

provisioning, with Vagrant 158, 159

nginx

building 377

installing 406

managing 377-380
nginx cookbook
 URL 381
nginx virtual hosts
 creating 381-385
node
 about 23
 converging 38
 deleting, from Chef server 224, 225
 role, mapping to 36, 37
node attributes
 accessing 172, 173
 defining 171
 setting 172
node delete command 57
node hash 66
NTP
 managing 374-376
NX-OS integration
 URL 591

O

Ohai
 about 449
 URL 304, 331
Ohai plugin
 attributes 184
 URL 304
 writing 184-186
old files
 cleaning 358, 359
omnibus installation package, Chef 9
omnibus installer 198
omnibus installer, Chef server
 obtaining 11
 outline 11
 steps 11
on-premise (private) Chef Server
 setting up 452-456
openssh cookbook
 URL 371
OpenSSH service role
 defining 29
openssl 326
OpenStack command cheat sheet
 URL 581

OpenStack configuration input settings
 URL 581
OpenStack instance images
 URL 581
OpenStack provisioning
 about 559
 CentOS instance image, adding
 to provision 576, 577
 demo project rc file, applying 567
 demo project rc file, downloading 567
 DevStack, downloading 563-566
 DevStack Host VM, deploying 560-563
 DevStack, installing 563-566
 DevStack OpenStack all-in-one
 environment, setting up 560
 DevStack public key, setting up 571, 572
 floating IP addresses, allocating
 to project 572, 573
 knife OpenStack plugin, installing 571
 knife-openstack plugin knife settings,
 configuring 573-575
 new instance, provisioning 577-581
 security rules, adding to allow ping and
 ssh traffic 567-571

P

packages
 installing, from third-party
 repository 345-349
passwordless sudo
 enabling 371-373
PIP
 used, for writing definition 139-142
Platform as a Service (PaaS) 581
platforms 335
platform values
 URL 336
policies
 about 513
 Chef.io blog post, URL 521
 Chef.io webinar, URL 521
 chef-zero repo, URL 521
 example 516-519
 Gotchas 520, 521
 policyfile documentation, URL 521

source options, specifying for cookbooks 519, 520
URL 521
uses 514
ways, for using 514, 515

portal
 URL 603

Portal UI
 URL 612

postgresql 110

PostgreSQL service role
 defining 29

precedence, for attributes
 URL 279

provider
 about 25, 61, 145
 implementing 146, 148

public SSH key
 URL 364

PuTTY
 URL 603

Python application
 modelling 107, 108

Python requirements file
 using 116

Python's requirements file
 using 116

Python virtual environment
 constructing 113, 114

Q

QFX5100 switches
 URL 591

R

Rackspace Cloud
 about 52, 53
 Chef node, removing 57
 instance, provisioning 53-55
 instance, terminating 55, 56

read-eval-print-loop (REPL) 165

recipe
 about 4, 24, 61, 69, 70, 83, 449
 applying, to role 33-35
 attributes 450
 building 97

cookbook, installing 32
complicated recipe, actions 86-88
convergence 451
creating, in Chef shell 170
databags, searching from 153
determining 31
developing 83, 84
developing, with local mode 225-227
enhancing, with search method 132
executing 84, 85
failures 98, 99
idempotence 451
order 451
need for 31
simple service, installing 85, 86
starting states 69
testing 90
testing, with ChefSpec 90
tests, executing 98
writing 84

recipes, executing with Chef shell
 about 169, 173, 175
 attribute mode 170
 attributes, accessing 172, 173
 attributes, setting 172
 configuration blocks, using 173
 node attributes, defining 171
 recipe mode 170

Red Hat Enterprise Linux
 Chef server, installing on 12

Region 438

report handlers
 using 268, 270

resources
 about 24, 61, 70, 71, 450
 building 145
 built-in resources 70
 default behavior, overriding 73
 defining 145, 524
 existing resource, loading 149
 modifying 148
 package 450
 package resource 145
 provider, implementing 146, 148
 service 450
 template 450
 updated status, declaring 150

using 71, 72

reusable resources, defining in Chef

- benefits 145

role, defining

- about 28
- image-processing role 28
- image search role 28
- OpenSSH service role 29
- PostgreSQL service role 29
- Solr service role 29
- web application service role 28

roles

- about 24, 27, 450
- applying, to recipes 33-35
- base server role, creating 119
- database server role, creating 120, 121
- defining 28, 119
- implementing 30
- mapping, to node 36
- URL 317
- using 228
- web server role, creating 121

roles, using with Chef-solo

- about 163, 164
- custom JSON data, injecting 164
- custom node name, providing 164

RSpec

- about 90
- and ChefSpec 91
- comparing, with testing libraries 92, 93
- failures 98, 99
- testing levels 92
- URL 250

RuboCop

- cookbooks, analyzing with 632
- cops, URL 635
- customizing 634
- on Chef.io site, URL 635
- output, examining 633, 634
- repo on GitHub, URL 635
- URL 635
- using 632

Ruby

- dependencies, locking 94, 95
- mixing, with Chef DSL 284-286
- used, for conditionals 342-345
- used, for iterations 342-345

Ruby gem mechanism 5

Ruby gems

- installing 287
- using, in recipes 287

Ruby libraries 61

Ruby on Rails applications

- managing 392-397

Ruby Version Manager (RVM)

- about 5
- URL 5

runit cookbook

- URL 334

run list, node

- overriding, to execute single recipe 256, 258

run lists 24, 449

runner 103

S

s3cmd package

- URL 349

search

- URL 319
- used, for finding data bag items 323, 324
- used, to find nodes 317, 318

search method

- used, for enhancing recipes 132

search queries

- used, for managing multiple machines 155

secret keys, databags

- managing, with search queries 150

Secure Shell Daemon(SSHD)

- securing 368-370

Secure Shell (SSH) 428

Secure Shell (SSH) key

- URL 195

security groups

- creating 596, 597
- URL 603

self-signed certificates 469, 470

Semantic Versioning

- URL 277

server list subcommand 56

ServerSpec

- URL 655

ServerSpec resource tests

- URL 656

Service Level Agreement (SLA) 199
service-oriented architecture (SOA) 157
setup
 describing 106
shell script 412
shell user recipe
 evolution 128, 129
Simple Network Management Protocol (SNMP)
 setting up, for external monitoring services 414, 415
snmp cookbook
 URL 416
software
 deploying 124
 deploying, with Chef 106
 installing, from source 349-353
Software as a Service (SaaS) 581
Software-defined Networking (SDN) 581
software deployment
 automating 125
 updates, deploying manually 125
solo mode, Chef shell 165
Solr service role
 defining 29
source code
 checking 114, 115
spiceweasel
 URL 442
ssh DDos attacks 434
SSH keys
 authorized keys, templating 136, 137
 deployment keys, adding 138
 distributing 136
standalone mode, Chef shell 165
strainer
 URL 244
Sublime Text
 URL 636
Sublime Text editor
 URL 488
sudo cookbook
 URL 374
supervisord service
 URL 118
support files 61
symlink 395

T

templates
 about 61, 74
 computed configuration, accessing 79, 80
 ERB primer 76
 need for 74-76
 searching for 80, 81
 search order 80
 template resource 77, 78
 URL 284, 342
 used, for creating configuration files 340-342
 using 281-283
 variables 78
 variables, passing to 78, 79
terminology, Chef
 Chef service 4
 cookbook 4
 node 3
 recipe 4
 workstation 4
test-driven-development (TDD)
 about 92
 for cookbooks, ChefSpec used 245-249
Test First approach 247
Test Kitchen
 about 457
 cookbook, building 178
 cookbook, preparing for 179, 180
 cookbook, testing 180
 home page 655
 installing 178
 integration testing, Chef cookbooks with 250-254
 tasks 178
 URL 254-256
 used, for testing 178
 using 177
tests 62
text editor
 using 471-473
TextMate
 URL 488
tree script
 creating 468

Tutum

URL 630

twitter gem

URL 308

U**Ubuntu**Chef Development Kit (ChefDK),
installing 460

Chef server, installing on 11

Ubuntu Desktop

URL 581

unit testing

with Chefspec 639

Unix environment variables, in Chef

URL 312

updated_by_last_action method 150**user cookbook**

enhancing 133-135

user management

Chef 127

data, storing in data bags 129

recipe, enhancing with search method 132

shell user recipe, evolution 128, 129

SSH keys, distributing 136

user cookbook, enhancing 133-135

users

adding 122

creating, from data bags 364-367

managing 127

users cookbook

URL 368

V**Vagrant**

about 157

combining, with Chef-solo 161-163

image, booting 159

installing 158

URL 158, 202, 474

used, for provisioning new host 158, 159

using 474

virtual machines, managing with 202-206

Vagrant base boxes

URL 559

Vagrant box

URL 559

Vagrant Butcher plugin

URL 206

Vagrant Cloud

URL 159

Vagrant documentation

URL 206

Vagrant driver

URL 671

Vagrant image

booting 159, 160

Vagrant machine

creating 406

Vagrant Plugin Butcher

URL 559

Vagrant plugin, for Amazon AWS

URL 206

Vagrant plugin, for VMware

URL 206

Vagrant tool

URL 655

Vagrant users 372**validators** 201**values**

calculating, in attribute files 280, 281

Varnish

installing 397

managing 397-399

URL 400

varnish cookbook

URL 400

vCloud Director (vCD) 549**version control**

using 194-197

version control system (VCS)

about 194

using 473

VirtualBox

URL 202, 474

VirtualBox software

URL 655

virtualization hypervisor 474**Virtual Machine (VM)**

about 202

managing, with Vagrant 202-206

Virtual Private Network (VPN) 370

VMware Fusion
URL 559
VMware provisioning
about 547-549
Chef, using 555-558
knife-vsphere plugin, installing 549-551
knife-vsphere plugin used 551-554
vSphere and Chef environments,
cleaning up 554, 555
VMware vCloud Automation Center
(vCAC) 549
VMware Workstation
URL 559
vSphere 6 lab
URL 558
VSphere driver
URL 672

W

wait option 487
WebApp deployments
URL 591
web application service role
defining 28
webinar
URL 646
web server
configuring 123, 124

web server role
creating 121
why-run mechanism 148
why-run mode
URL 260
using 258, 260
Windows workstation
Chef Development Kit (ChefDK),
installing 464
wordpress cookbook
about 391
URL 392
WordPress sites
managing 389-391
workstation
about 4, 23, 448
references 475
setting up 471
text editor, using 471-473
Vagrant 474
version control system 473
virtualization hypervisor 474

Y

YAML 106

Bibliography

This course is a blend of different projects and texts all packaged up keeping your journey in mind. It includes the content from the following Packt products:

- *Chef Essentials by John Ewart*
- *Chef Infrastructure Automation Cookbook Second Edition by Matthias Marschall*
- *Mastering Chef Provisioning by Earl Waud*



**Thank you for buying
Chef: Powerful Infrastructure Automation**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles