O'REILLY®

Customizing Chef

GETTING THE MOST OUT OF YOUR INFRASTRUCTURE AUTOMATION

Early Release

RAW & UNEDITED

Jon Cowie

Customizing Chef

Jon Cowie

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Customizing Chef

by Jon Cowie

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Courtney Nash and Brian Anderson
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

November 2014: First Edition

Revision History for the First Edition:

2014-03-25: Early release revision 1

2014-05-12: Early release revision 2

See *http://oreilly.com/catalog/errata.csp?isbn=9781491949351* for release details.

ISBN: 978-1-491-94935-1

[?]

Table of Contents

Preface

Preface

As I sit here writing this Preface, I am very very tired. It's a feeling which will doubtless be very familiar to a number of you reading this book. Your brain feels sluggish, simple tasks seems to take just that little bit longer and you can never remember which way round the brackets and quotes go in awk expressions. This tiredness is something that we as Operations Engineers treat as a part of the job, albeit not a particularly welcome one - often due to the rigors of a stint in the on-call rotation.

It is when debugging complex issues in this state that I find myself most grateful for the modern incarnations of Configuration Management (CM) systems that we as Operations Professionals are able to make use of today. Although configuration management has been around for a long time, it's only in the last few years that CM systems like Chef, Puppet, CFEngine, Ansible and SaltStack have come along to evolve these tools from a loosely-couple collection of shell scripts and wiki pages into the rich orchestration frameworks we have today.

When we get paged at 3AM, we're quickly able to introspect into any changes that might have been made to the server or system in question and kickstart the late-night troubleshooting process. Imagine if instead of being able to run one command to look at the running config on one of your servers, you had to trawl through an internal wiki to find the install document that detailed that specific configuration. This very process may in fact be what some of you are using right now, and to you I say be strong, brave Ops Engineer. There is hope and its name is Configuration Management!

In this book, the CM system we'll be focussing in particular is Chef, which is developed and maintained by Chef Inc.. Chef Inc. themselves describe Chef as a *server and infrastructure automation tool*, but it could equally be described as a CM system or any of a number of other definitions.

Chef is an incredibly feature-rich and powerful tool, designed to provide a framework for users of any of the major Operating Systems to automate anything they want. Simply

put, Chef is a generic platform which provides a number of built in tools, resources and services to facilitate this automation, but typically Chef Inc. do not advocate any of these as "the one true way". And how could they? You, dear reader, are one of the greatest experts in the world on your particular infrastructure, if not **the** expert. You understand in detail the servers and operating systems you run, the software stacks which sit on top, and the unique structure, workflows and processes that your keep your business running. The chances are good that out of the box, Chef will allow you to automate a good deal of these systems, especially if you leverage the community cookbooks repository where Chef users upload cookbooks and recipes for their particular systems and software.

But what do you do when your specific needs require something which Chef doesn't provide out of the box? You customize, of course! One of Chef's greatest strengths is that it is nearly infinitely customizable. You can hook chef into your monitoring systems, you can modify the built in tooling and processes, you can add your **own** tooling. But where do you start? You may have a rough idea of what you'd like to achieve, but where in Chef should you add your custom piece? How do you make sure it will work nicely alongside your production environment? That's where this book comes in.

I'm going to take you on a journey through the internals of Chef, exploring how everything fits together, the various places you can hook in custom code, and when it's appropriate to use each one. We'll also look at whether you **should** customize - although this book is very much about helping you to customize Chef, it would be a fallacy for me to tell you that just because you **can** customize something, you automatically **should**. As someone once said, *with great power comes great responsibility*.

Going in to this book, you should already be familiar with writing Chef cookbooks and have a good grasp of at least the amount of Ruby needed to write cookbooks. If you're not quite at that stage yet, I'd recommend you take a look at "Learning Chef" by Seth Vargo and Mischa Taylor before going too far down the customization route. By the end of this book, my hope is that you will have a solid understanding of when and where to customize Chef, and the skills to dive in and start customizing!

PART I
Chef & Ruby 101

CHAPTER 1
Introduction

Before we dive into the internals of Chef and look at all the different things we can customize, let's take a step back, remind ourselves what we're dealing with here when we talk about Configuration Management and look at **why** you might want or need to customize Chef in the first place and what you need to know to get the most out of this book.

What is Configuration Management?

> Time stands still for no man
>
> — Anon

The term "Configuration Management" has been in common usage since the 1950s, when the US Air Force developed an approach to managing the extremely complex manufacturing processes involved in producing military equipment. The USAF needed a way to make sure that their equipment performed as required, while also functioning as expected and complying with the relevant military equipment standards. The term was codified as an actual standard in the early 1960s, and the ideas underpinning CM have since been adopted by a number of other industries such as civil and industrial engineering, and of course computing.

What we think of as Configuration management in the IT sense has been around in one form or another for as long as people have been running more than one computer, although it was not always recognized as such. Looking back through the mists of time to the era of Mainframes and Mini-computers, we find system administrators tending single monolithic systems, every tunable setting and parameter committed to memory or to multi-volume technical manuals. Computers of any description were far outside the reach of most companies - only the largest corporations and university labs could afford the asking price. Soon however, as computing hardware became more common,prices dropped to within the reach of more and more companies, and commodi-

tized "cloud" computing became commonplace the exploding usage of computers and the corresponding increase in sysadmins to look after them began to make this single repository of knowledge unsustainable.

The industry quickly recognized this problem, and started to document everything. Thus was born the era of internal configuration guides, document stores and wikis full of all the information a new hire might need to get up to speed on the infrastructure and systems at their employer. This approach, however, also had its limits. As the usage of computers expanded across the globe and more and more companies started developing software systems, manually updated documentation began to become more and more of a chore to keep relevant and up to date so people started automating configuration tasks using whatever scripting language they had to hand. These custom scripts worked perfectly well for a while. They made configuring systems and software much easier since people didn't have to copy and paste from the manual any more.

Eventually though, progress overtook the industry again. Everyone started using different architectures, different software versions and different configuration settings in the same infrastructure. All of a sudden, people found themselves having to maintain an increasing number of very complex scripts to manage all the different permutations they required, cater for all the possible ways these scripted processes could fail and so on. It was this very problem which led to the development of what we now recognize as CM in the Operations sense.

In 1993, a PostDoc at Oslo University in Norway called Mark Burgess decided that he'd had quite enough of managing all of these scripts, thank you very much. Mark wrote the very first version of CFEngine (configuration engine) to provide an architecture and OS independent way of managing the UNIX workstations he was responsible for at the time. This was the first of what we now recognize as configuration management systems.

In the years since CFEngine was first developed, a number of other CM systems were developed to satisfy various requirements that existing tools did not provide for. In 2005 Luke Kanies released Puppet, which for several years was the predominant alternative to CFEngine, until Chef came along in 2009. Since then, a number of other CM systems such as Ansible and SaltStack have been released but at time of writing, CFEngine, Puppet and Chef are the dominant players in the space.

So Why Chef?

Why, then, did Chef get written when there were already two well established and relatively mature CM systems to choose from?

The foundation of Chef can be traced back to a consulting company named HJK Solutions (now better known as Chef Inc.) who specialized in automating the configuration of infrastructure for startup companies. Through this work, HJK Solutions came to

realize that the utopia of fully automated infrastructures was becoming a real possibility, even for companies without a staff of highly experienced Operations Engineers. But things weren't quite at that stage yet - HJK observed a number of problems with what could be achieved with existing CM systems that needed to be solved first:

Service Oriented Architecture
: In order for fully automated infrastructure to become a possibility, Configuration Management systems needed to move towards a service oriented architecture instead of defining a canonical model of your infrastructure. Rather than defining your infrastructure configuration in stone, CM systems needed to be able to expose as much data as possible about the state of your infrastructure to allow you to better manage its automation.

Sharable Code
: For infrastructure as code to work effectively, people need to be able to **share** the code which builds their infrastructure. This is a principle found in traditional software development too - code should be modular, clean, and not repetitive. HJK found that the existing tools didn't provide the sort of generic, reusable building blocks necessary to allow people to effectively share and reuse their infrastructure code.

The folks at HJK didn't think that any other open source CM systems met the requirements they saw as necessary to achieve fully automated infrastructure, so they created Chef. But did they succeed? What makes Chef different from the other CM systems out there, and why might you want to use it? Since you're reading this book you likely already have an idea of the reasons Chef was chosen by your organization and the key differentiators between Chef and its contemporaries, but I'd like to highlight a few of them anyway.

Chef Frees Up People Where Possible
: Chef aims to help remove people from the process of automating infrastructure whenever possible. This may sound like the classic joke about replacing yourself with a very small shell script, but let's think about this - the less time you spend having to keep your server configurations in sync and installing the right combination of packages etc, the more time you have to spend doing far more interesting things! You still have to write your infrastructure automation code of course, but once you've done that Chef takes care of the rest.

Everything is Ruby
: Chef uses Ruby as its configuration language, as opposed to a custom DSL. On the surface, the language used to write Chef cookbooks may not always **look** like regular Ruby, but it is.

Chef is Infinitely Extensible
: Chef is infinitely extensible. Right from the outset, Chef was designed to integrate with any system you choose to use, and to allow any system to integrate with it. Chef Inc. don't see Chef as a carved stone tablet which represents your infrastructure, they see it as a service layer which exposes data about how your infrastructure is behaving and what it looks like. It is this extensibility and data service which allow the sorts of customizations we'll look at in this book.

Chef is Modular
: Chef was also designed from the ground up to allow you build your automated infrastructure using modular, reusable components. Chef cookbooks are formed out of discrete "chunks" of automation behaviour which can be extended, shared and reused across your infrastructure.

In Chef, Order Matters
: Chef guarantees you that it will run things in the same order, every time. If you define a run list in Chef, no matter how many Chef runs you perform the resources in that run list will **always** be applied in the same order. See Chapter 3 for more on how Chef runs are executed.

Thick Client, Thin Server
: Chef does as much work as it can on the node being configured rather than the server. Chef server is responsible for storing, managing and shipping out cookbooks, files and other data to client nodes which then run your infrastructure code.

Chef is a System State Service
: This *thin server* model allows chef to store a copy of the "state" of each node on the server, which includes data like the recipes applied to the server, when it last completed Chef run, its hardware configuration etc. By capturing this information in a central location, Chef server is able to provide us with a "system state" service which can tell us the current state of the node, what the state of the node was at the end of the **last** chef-client run and what the state of the node should be at the end of the **current** chef-client run.

Resource Signals
: Resources in Chef cookbooks are able to "signal" other resources to perform particular actions, allowing you to add conditional logic paths into your infrastructure code while still retaining the modularity and reusability that Chef was designed to provide. You are also able to re-use the same resource several times without redefining it each time - for example, you can define a "service" resource to control your MySQL server and then use that resource to stop and start your MySQL server by simply specifying the resource with different actions.

Chef gives you nearly limitless flexibility to automate your infrastructure as you see fit. It has never been positioned as a panacea to cure all your ills, or magically automate all

of the legacy cruft out of that 8 year old server in the corner. What Chef **is** however, is a framework designed to give you all of the tools you need to automate your infrastructure however you want.

Think Critically

Ultimately, as technology professionals our primary function is to add value to our businesses regardless of how much time we spend writing code or configuring servers. This is just as true for Operations Engineers and Developers as it is for front line sales staff and marketing. We might not be on the front lines shifting product, or selling to customers, but we code and configure the infrastructure and platforms that allow our businesses to function in the internet age. Every business is different, and only that business can make the decisions about what will add value and what won't. As a technical expert in your company, you know better than me, or Chef Inc., or anyone else for that matter what adds value to your particular company.

This is worth remembering, as it is the principle reason that Chef Inc. will not typically advocate particular workflows or tooling combinations as the "canonical" Chef way of doing things, and also the reason that I will not do so in this book. The instant that a methodology or technique is declared canonical, then when your particular requirements fall outside of canon you're considered to be doing it wrong - even if it might be what makes most sense for your particular use case. Chef is categorically **not** a system which requires you do do anything in a particular way, it provides you with everything that you need to make those decisions for yourself, for the good of your business. As with all systems however, there are some best practices - we'll look at some of those throughout the course of this book.

Simply put, Chef gives you the tools and flexibility to craft infrastructure code that is right for you, your team and your business. It won't get in the way and it won't try and force you down a particular path. My aim for this book is to help you gain a deeper understanding of how you can make use of this flexibility and extensibility so you can take what you learn and add even more value to your businesses. To this end, I want you to question everything you read in this book and ask yourself if it's right for you.

I can certainly guarantee that the content of this book will be technically accurate and as comprehensive as I can possibly make it, but just as Chef cannot automate your business needs automatically, I cannot tell you what is best for your company or what will solve your specific infrastructure automation problems. What I **can** do is give you the knowledge and techniques to make you better able to solve those problems for yourself.

In my day job, I work in Operations for Etsy, an online marketplace where people around the world connect to buy and sell unique goods. Etsy is fairly well known for its engineering culture which is rooted in the DevOps movement. We are not particularly driven

by procedures or rigid workflows. I usually don't have to deal with committees to get things done, and am given a high degree of autonomy to perform my job. I think it's important to frame the employment background I come from because yours may be very different.

You may work for a small startup where you are the only Operations engineer and have very little time for anything but the most crucial work. You may work for a large multinational corporation with a many-layered change control process and ITIL compliance to worry about. You may work for a company which stores credit card data and has to maintain PCI-DSS compliance with all the procedural and auditing headaches that entails. Every single one of you reading this book comes from a different employment background, and you're all using Chef to solve different problems.

So I propose a pact. I promise to do my best to write you an interesting, relevant book which is full of helpful information, useful code snippets and practical advise for how to approach customizing Chef. And in turn, once I've armed you with all of the knowledge and techniques that I can, you promise to carefully look at the problems you're trying to solve, and think critically about the best way to do that. What I'm asking you to consider when looking at customizing your Chef set up is not whether you **can** do something, but whether you **should**. And I have a couple of suggestions to help you do that…

Meet AwesomeInc.

To help with the process of critically examining the material covered in this book from the perspective of a business trying to solve real problems, I'd like to introduce AwesomeInc, a fictional company who will be working through a project to customize various aspects of their Chef infrastructure.

AwesomeInc is a midsize company of around 200 staff based in California who produce and sell custom car parts. It was founded in 2005 by two siblings, Chad and Kate Awesome, who started a small business customizing their friends cars from their parents driveway. The business grew rapidly as word spread, and AwesomeInc now ships its own line of custom car parts all across the US and Canada. Business has been so good recently that the executive team have decided that this is the year to go international!

AwesomeInc has 3 dedicated Operations Engineers and 15 full time development staff all led by Mike, the straight-shooting, hard negotiating but secretly lovable Director of Engineering. They've already rolled out Chef across their suite of 150 servers, a mix of both physical hardware and virtualized servers in the cloud. They're using the open sourced version of Chef server, hosted internally.

Their Ops staff and developers are well versed in writing cookbooks, and have been through Chef Inc.'s "Chef fundamentals" training. But now the word has come down from on high that with the push to go international, AwesomeInc's infrastructure will

be moving from a single system to separate geographically diverse systems, linking back to a central stock database in the US.

Spotting a chance to get ahead of the game, Mike instructs his team to do an audit of their servers and make sure that all of their cookbooks are up to date prior to commencing the work to support multiple locations. When the team complete their audit however, it turns out that things weren't quite as rosy as they'd hoped.

Although the majority of AwesomeInc's cookbooks are working perfectly and up to date, there are a few cookbooks which had been modified and not tested properly, which had been causing Chef runs to silently fail on a number of AwesomeInc's servers. Additionally, a recent project to migrate some of the infrastructure from Solaris to CentOS had resulted in a bunch of hacks dotted throughout the Chef repository to cope with the differences in packaging systems and repository management which had never been cleaned up.

Mike decides to get a grip on the situation and calls a team meeting to ticket up the cleanup work that's needed to bring their cookbooks back up to scratch, and look at ways to avoid the issues they've been seeing. During the meeting, a number of his Operations team express concern about how they'll cope with the increased headcount that will follow AwesomeInc's international expansion. It turns out that Operations staff and Developers are already beginning to find themselves treading on each others toes when they make Chef changes, and it's becoming increasingly hard to keep track of what has been changed by who. The team are worried that with the increased frequency of changes that will likely follow the hiring of more engineering staff, it will become harder and harder to maintain stable systems.

They mull over these issues for a few days, and quickly come to the consensus that things need to change. But then Mike asks the million dollar question: How do we fix this? The team rapidly realize that out of the box, Chef won't fix these problems for them so their only option is to customize chef to give them more visibility into how Chef is running on their servers, who is making changes and when, and what those changes are doing.

As we work through the material in this book, we'll follow AwesomeInc as they learn about the possibilities for customizing Chef, and what they can do to solve the specific problems they've encountered in the past. We'll focus on **what** solutions they can make use of as well as **why** each might be a good or bad idea. In addition we'll look at what members of the Chef community have done to solve the same problems in the real world - these techniques are no use, after all, if they don't transfer to real life.

Criteria for Customization

> With great power comes great responsibility.
>
> — Voltaire (François-Marie Arouet)

You could be forgiven for thinking that having written a book called "Customizing Chef", my advice to you would always be to customize Chef. It isn't. I'd much rather you carefully consider the customization you've planned to make and decide that maybe it's not the best idea, than make a whole bunch of customizations that don't really help you.

One of the recurring problems that our industry as a whole suffers from is a terminal case of the "New Shinies". We're all of us somewhat prone to taking a new methodology or tool, and then trying to make it do everything, regardless of whether or not it is actually best suited to those things.

With a system as flexible and customizable as Chef, you're more limited by your imagination than any technical limitation, and that carries with it the risk of new-shiny overtaking a rational decision making process. So how do you make sure that when you customize Chef, you're doing it for well thought out reasons that will ultimately add value to your business? We've met AwesomeInc and began to explore the challenges they're dealing with, but what happens when your specific challenges don't mesh nicely with my carefully selected examples?

Let's look at some more general criteria you can use to vet proposed customizations. When you're thinking about developing or implementing customizations to your Chef setup, I want you to come back to this list and mentally check off how many criteria are satisfied. If the answer is none, it's probably time to take a really careful look at why you're doing it - you might be doing it because you can, rather than because you should. If you're like me and enjoy both symmetry and easy-to-remember acronyms, remember SMVMS:

Simplicity
: Will the customization you're considering make something simpler? This could range from simplifying your deployment process by automating cookbook upload and testing, through cleaning up an old legacy recipe full of labyrinthine control statements, all the way to making it easier for a new hire to work with your apache configurations.

Modularity
: Will the customization you're considering make your infrastructure code more modular and reusable? Are you taking multiple slightly-modified chunks of "copy-pasta" code and refining that into a more generic modular resource which can be defined once and then called from the various places which need it?

Visibility
: Will the customization you're considering increase your visibility into your Chef infrastructure? Will it let you introspect deeper into your Chef runs than you've ever been able to before? Will it generate metrics for you so you can tell at a glance whether or not your chef runs are getting slower or faster? Will people be able to

gain greater awareness of when Chef changes have gone live and what exactly they did?

Maintainability
: Will the customization you're considering make it easier for your Chef users to maintain your infrastructure codebase? Will it make people less afraid to change that one cookbook that runs on all of your servers? When a new hire starts, are they going to be able to tear up the manual of voodoo incantations previously required to work with this code?

Scalability
: Will the customization you're considering help your infrastructure scale to meet your businesses growth needs? Have you hit the point where you need to diversify your infrastructure from a single Datacenter and all of a sudden the assumptions you made in your recipes about server naming conventions break down? Are you building out a small software stack into a much larger n-tier cluster?

Let's look in a little more detail at some questions AwesomeInc asked themselves when looking at how to solve the problems they identified and how they map to the "Criteria for Customization" discussed above - all are excellent candidates for a sound decision to customize but for different reasons.

How do we find out when our Chef runs are failing and why?

AwesomeInc are looking to improve the visibility they have into Chef and how it's performing. Out of the box, Chef will expose some information to you through both the WebUI and the command life on when a particular node last checked in with the Chef server and what its current run list is, but it won't tell you whether or not the last Chef run failed or the cause of any failures. This might sound like somewhat of an omission, but look at it this way. Bearing in mind the wide variety of notification and alerting systems out there, how should Chef expose this information to a Windows user? A Linux user? Should it **always** alert on every failed run?

Rather than trying to solve all possible use cases, Chef Inc. have made it extremely easy to capture this information in Chef so that you can handle it in a way that works well with your particular choice of monitoring and alerting software. We'll look in more detail about how you can capture and make use of this information in Part II.

Some chef setups such as chef-solo do not use a centralized Chef server and will not support some types of customization, such as those making use of persistent node attributes. Please see "Chef Installation Types and Limitations" on page 18 for more details

How do we simplify our recipes to reduce the amount of replicated code?

AwesomeInc are looking to improve both the modularity and simplicity of their infrastructure code. As we've already seen, Chef provides you building blocks and tools to let you automate your infrastructure however you want. Out of the box it provides resources for a number of operations, from configuring users through downloading remote files to managing packages and services. But as with Chef itself, these resources are designed to be as simple and generic as possible. Chef doesn't provide a built in resource for configuring a virtual host in Apache for example, although this can still be done with a combination of out-of-the-box resources.

Again though, this is where the extensibility and flexibility of Chef really comes in handy. There is nothing to stop you writing your own resource to configure Apache virtual hosts, or set up RVM on your development workstations, or configure access to your MySQL databases. The sky is the limit here - Chef gives you the framework, and lets you decide for yourself. We'll look at how to write your own resources and recipe logic in Part III.

How do we stop our developers and Ops staff treading all over each others changes?

AwesomeInc are looking to improve the visibility they have into Chef changes and the scalability of their engineering organization so that they can increase their headcount without compromising the stability of their infrastructure with a sudden increase in the volume of Chef changes. Out of the box, Chef comes with knife, a command line tool which lets you perform a number of operations such as uploading cookbooks, configuring node run lists, running commands across a group of your servers and running search queries against your infrastructure - AwesomeInc have been using these built in knife commands to work with Chef.

Knife ships with a powerful set of features but once again, it is specifically designed to be as generic as possible. Knife does not ship with built-in functionality to spin up nodes on Amazon EC2. It does not ship with functionality to stop teams with large numbers of Chef users from treading on each others toes - although it does have a flag to stop you uploading the same cookbook version twice. But what knife **does** ship with is a design that allows it to be customized easily and extensively. We'll look at how to customize Knife in Chapter 8.

Some chef setups such as chef-solo do not use a centralized Chef server and will not work with some knife commands or plugins. Please see "Chef Installation Types and Limitations" on page 18 for more details

State of the Customization Nation

Now that we've examined the scenarios we'll be working through in this book and looked at some general criteria to weigh your customization ideas against, let's get those creative juices flowing and have a brief look at some of the customizations the Chef community has already produced and the problems they were trying to solve. This by no means an exhaustive list, and my inclusion (or exclusion) of a tool by no means implies I'm passing judgement on it or recommending it - my aim is simply to take you on a whistle-stop tour around some of the chef customizations to be found in the wild today to get you thinking about what is possible - it may also help you identify particular chapters of this book to read first.

Community Cookbooks

One of the most comprehensive and extensive Chef resources available today is the Chef Inc. Community Cookbooks site. It is a publicly accessible website hosted by Chef Inc. where Chef users can upload and share their cookbooks with the wider community. You'll find cookbooks for a decent proportion of the software packages systems you're likely to want to use, varying from simple cookbooks containing one or two recipes all the way through to more advanced cookbooks containing custom providers and resources for a wide range of tasks (see Chapter 4 for more details on how to write resources and providers)

It's important to remember that the majority of the cookbooks are open source code uploaded by members of the Chef community. They are not supported or warrantied by Chef Inc., and their presence on the community cookbooks site is not an indication of quality or correctness - as we've already discussed, as with all solutions it's very important that you make sure these cookbooks are right for your infrastructure. For example, some cookbooks may only support a particular Linux distribution, or make certain assumptions about how you configure your software. Community members are able to review and comment on cookbooks however, which can be a good starting point for evaluating suitability.

Development Tooling

Although more strictly tooling built **around** Chef than a direct customization, a number of tools have been written to help support the Chef development process. One of the benefits of "Infrastructure as Code" is that we get to benefit from the expertise and experience of the wider development community, and adopt software engineering best practices to write awesome infrastructure code. These tools are some of the fruits of this way of thinking.

Foodcritic
: Foodcritic is a lint checker which checks your cookbooks against a set of "best practice" rules as well as letting you define your own rules. Foodcritic can check for coding style as well as the actual correctness of your cookbooks.

Test Kitchen
: Test Kitchen is a tool provided by Chef Inc. which lets you write integration tests for your Chef cookbooks. It uses the Vagrant virtualization software to spin up "test" nodes which run your integration tests and produce a report afterwards.

ChefSpec
: Chef Spec is a unit testing framework which allows you to write RSpec style tests for testing Chef cookbooks and verifying that they behave as expected.

Leibniz
: Leibniz is an acceptance testing tool which leverages the provisioning features of test-kitchen to allow you to run acceptance tests against your infrastructure using Cucumber / Gherkin features.

Hudson chef plugin
: Chef Hudson Plugin is a plugin for the Hudson continuous integration tool which allows you to initiate a Chef run on a remote host with the specified config, and report its status.

Workflow Tooling

Cookbooks aside, some of the most popular tooling written for Chef has been to support different workflows for working with Chef. As we've already seen, Chef Inc. intend Chef to be as generic as possible and correspondingly will not generally advocate a particular workflow or methodology - if they did, as soon as your requirements fall outside of that "one true way", you're doing it wrong. The workflow tools described in this section are good examples of members of the Chef community producing a tool to fulfil business requirements that was were not satisfied by Chef out-of-the-box. In the three particular cases mentioned here, it turns out that a decent chunk of the community had the same requirements and adopted one of the other of the below tools, so they've become relatively well known as a result.

Berkshelf
: Berkshelf was initially developed by the engineering team at Riot Games to solve some of the issues they were experiencing around managing cookbook dependencies. Berkshelf is essentially bundler for Chef - it allows you to quickly and simply install all the cookbook dependancies your own cookbook has, and stores them in a special "berkshelf" in a similar way to how Ruby installs gems. Berkshelf treats cookbook dependancies as libraries to be utilized and extended by your cookbooks,

rather than adding them into your main cookbook repository to be customized themselves.

Librarian-chef
: Librarian-chef is based on a similar idea to Berkshelf, that of managing cookbook dependencies. Rather than storing cookbook dependancies in a separate area however, Librarian-chef effectively takes control of the cookbooks directory in your Chef repository and uses a special "Cheffile" to specify what cookbooks should be installed and from where. Librarian-chef is designed to work with cookbooks which are each separate projects (like those on the Chef Inc. Community Cookbooks site) rather than cookbooks which are solely stored in your chef repository.

Knife-spork
: Knife-spork (Disclosure: written by me) was developed at Etsy to support the unusually large number of developers and operations staff who were regularly making Chef changes. Etsy found that having 30 or 40 chef users regularly changing and uploading the same environment files resulted in uncertainty over whether or not changes had gone "live", been overwritten by subsequent changes or just flat out been lost in the noise. Since its initial release, knife-spork has been extended to include plugins for various tasks such as automatically running foodcritic prior to uploading cookbooks and broadcasting Chef changes to a number of different systems including IRC, HipChat, Campfire and Graphite.

Knife Plugins

Some chef setups such as chef-solo do not use a centralized Chef server and will not work with some knife commands or plugins. Please see "Chef Installation Types and Limitations" on page 18 for more details

Knife is one of the most versatile tools provided with Chef, and the community has been quick to take advantage of this. Out of the box it supports a number of build in commands for uploading, editing and deleting a variety of Chef objects such as cookbooks, roles, users and clients. But knife's main strength is its flexibility and customizability. Knife provides you with an extremely powerful interface to Chef and its underlying functionality which can be leveraged in any number of ways. We'll look in much greater detail at how you can write your own knife plugins in Chapter 8.

One of the categories of knife plugin which has seen the most development activity is that surrounding working with Cloud providers. Plugins already exist for provisioning nodes on a number of different cloud providers such as Amazon EC2, Rackspace and Joyent to mention but a few. Knife ships with the "bootstrap" command which is used to install and configure Chef on nodes which are already running an OS and as more

and more people started to use cloud infrastructure it was a logical progression that knife plugins be written to handle the provisioning of cloud instances prior to setting up Chef.

Here's a quick sampling of some of the other knife plugins that have been created by the Chef community (the knife- prefix is a naming convention rather than functional requirement):

knife-elb
: knife-elb is a knife plugin to automate the adding and removing nodes from Elastic Load Balancers on Amazon's EC2 service

knife-kvm
: knife-kvm gives you the ability to provision, manage and bootstrap Chef on virtual machines using the open-source KVM virtualization platform

knife-rhn
: knife-rhn is a knife plugin for managing your nodes within the Red Hat Satellite systems management platform. The plugin lets you add and remove nodes from RHN, as well as manage system groups.

knife-block
: knife-block allows you to configure and manage multiple knife configuration files in the event you're using multiple Chef servers.

knife-crawl
: knife-crawl is a knife plugin to display role hierarchies. It will show you all roles which are included within the specified role, and optionally all roles which in turn include the specified role.

knife-community
: knife-community is a Knife plugin to assist with deploying Chef cookbooks to the Chef Inc. Community Cookbooks site. It aims to help users comply with a number of best practices for submitting community cookbooks such as correct cookbook version numbering, and git-tagging new cookbook releases.

Handlers

Handlers in Chef are extensions which can be triggered in response to specific situations to carry out a variety of tasks. Handler are typically integrated with the chef-client run process, and are called depending on whether or not errors occurred during the Chef run as shown here:

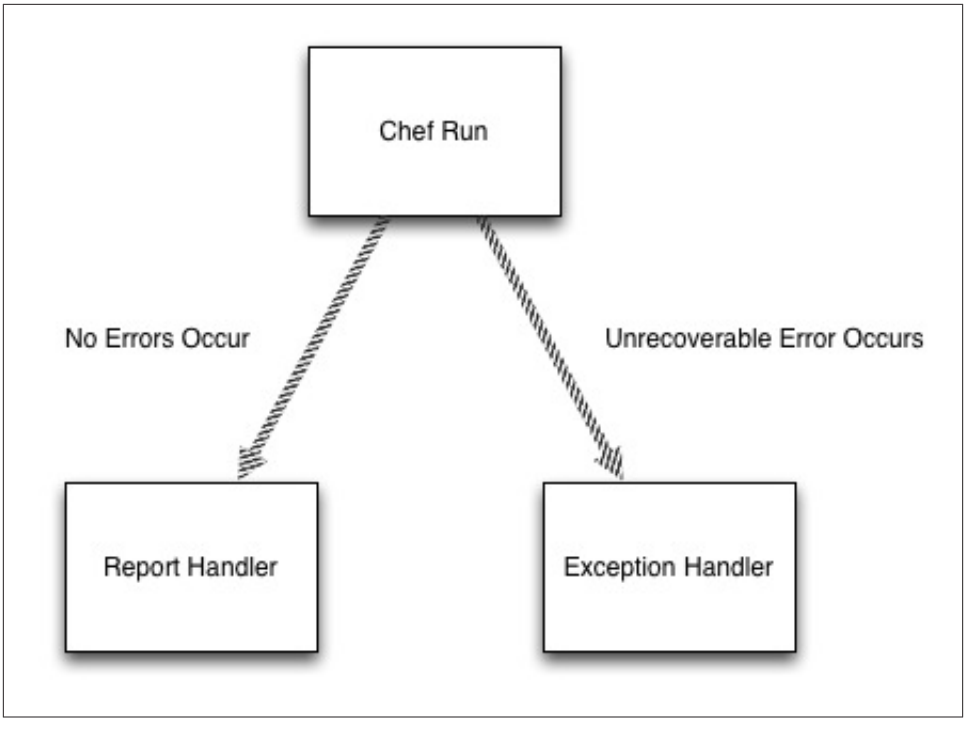

Figure 1-1. Chef Handler Flow

Report Handlers
: As shown in the above diagram, report handlers are used to carry out actions when the Chef Client run has succeeded without error.

Exception Handlers
: As shown in the above diagram, exception handlers are used to carry out actions when an error has occured during the Chef Client run which can not be recovered from.

Handlers allow you to capture all sorts of data about how Chef runs are behaving which we will look at further in Chapter4, but in the meantime let's look at some of the ways that the Chef community have made use of this information:

chef-graphite_handler
: chef-graphite_handler is a report handler (provided as a cookbook) for sending Chef run results to the open-source Graphite graphing system.

chef-handler-splunkstorm
: chef-handler-splunkstorm is a combination report and exception handler which reports Chef run statuses to Splunk. It will also log full stacktraces in the event of a run failure.

chef-irc-snitch
: chef-irc-snitch is an exception handler which sends notifications of Chef run failures to an IRC channel, complete with a Github gist containing node information, the exception message itself and the backtrace to assist in debugging.

Recipes and Resources

In addition to publishing cookbooks on the Chef Inc. Community Cookbooks site, a number of community members have open sourced libraries and providers that are more generally useful to the community than to users of a specific cookbook. We'll look at how to write providers in Chapter 5, but until then here are a selection of those currently available:

chef-whitelist
: chef-whitelist is a library developed at Etsy to allow host-based rollouts of changes. It allows you to define a whitelist (containing hostnames, roles, or wildcard pattern matches) in a databag and then add simple control statements to your cookbooks to apply different logic to roles which match an entry in the whitelist.

chef-deploy
: chef-deploy is a Rubygems which provides chef resources and providers to allow you to deploy ruby web applications without using Capistrano. It maintains forwards and backwards compatibility with Capistrano by using the same directory structure and deployment strategy.

Chef Installation Types and Limitations

Before we move on to the technical portion of the book, it's worth noting that not all of the material covered will work on all types of Chef setup. Throughout the book, when I list examples of techniques I'll note whether or not there are any compatibility issues with different types of Chef install and below I've listed each type of Chef install along with details on how they differ and which features are not supported.

Although both variants of Enterprise Chef listed below are commercial products, all of the material in this book is fully compatible with the Open Source version of Chef server. You do not need a paid-for version of Chef to work through this book.

Chef Solo

Chef Solo is the simplest chef setup you can run. It is a function-limited version of chef-client, and you have no central Chef server. Chef-solo requires a copy of all your cook-

books on each node (including all cookbook dependancies), and chef-solo then reads each cookbook from the local disk and applies it to your server.

You would typically use chef-solo if for some reason you don't want to use a central Chef server, say when initially evaluating chef, or when configuring appliance type systems which may not have network access.

As there is no Chef server in a chef-solo setup, the following features are not supported:

Node Data
: As there is no central Chef server in a Chef solo setup, nodes do not save state when they finish a run.

Persistent Attributes
: Again as there is no central Chef server, attributes will not persist across chef runs. Any attributes needed by a chef-solo run should either be set in cookbooks, or in a JSON configuration file passed to chef-solo.

Search
: Chef-solo does not allow you to use search in your recipes (or using knife) as there is no central server to store node state or attributes in search indexes.

Centralized Cookbooks
: Chef-solo doesn't support centralized cookbook storage and distribution. You must have a separate copy of your cookbooks, roles, environments etc on each node. This also means that you need to make sure you keep each copy of your cookbook in sync to ensure your changes are applied consistently across your infrastructure.

Centralized API
: The lack of a centralized Chef server in chef-solo means that you don't have access to the Chef Server API, which is the interface used to access the "system state" service that Chef provides.

Authentication
: Because chef-solo runs entirely self contained on the node, you don't have the authentication controls that Chef server provides to determine whether or not a client on a node is authorized to download and run cookbooks, roles and environments etc. Other than access to the node itself to initiate a Chef run, any node with a copy of your cookbook data can run those cookbooks.

Open Source Chef

Chef Inc. provides a free and open source version of Chef server which stores cookbooks, node run lists and attributes, and provides a Centralized API. Each node runs chef-client and queries the server to obtain information about the configuration to be applied during the run. The chef-client then performs the run on the node, using a local cached copy of the cookbooks which is synced with the copies stored on the server at the start

of the run. When the run concludes, chef-client saves the node state and updated attributes back to the central server.

The open source version of Chef server has to be configured, installed and updated locally by the user and will not automatically scale. It is entirely possible to scale out components of OS Chef, but this must be done manually by the user. Chef Inc. can provide optional paid support for Open Source chef servers, but the Chef community is the more usual source of technical advice. Open source Chef is recommended for advanced users, or at the very least those comfortable with manually maintaining the respective components.

All material covered in this book is fully compatible with the Open Source version of Chef Server.

Local Mode

Introduced in Chef version 11.8, *local mode* is an extension to chef-client designed to fill the gap between chef-solo and running with a full Chef server. Chef-solo is ideal for quickly testing cookbooks, or situations when chef needs to run in total isolation without access to a Chef server (embedded appliances for example). Sometimes, however, you may find yourself wishing to test out features usually provided by Chef server (like search) without the hassle of having to set up a full Chef server, or to test in a sandbox environment isolated from your production Chef server. This is where local mode comes in!

Local mode makes use of an open source tool created by Chef Inc. called chef-zero. Chef-zero provides an extremely simple memory resident Chef server which supports many of the command and features that can be run against a full Chef server. It's very lightweight and easy to use, but it does not perform any input validation or authorization checking, and it does not save any "state" data in the way a full Chef server would. Every time chef-zero starts up, it is completely empty and contains no cookbooks or saved node data. Provided that you bear these limitations in mind, it can be an extremely useful tool for testing more advanced cookbook features. It's important to remember that chef_zero is **not** intended to be used as a production replacement for the open source Chef server or any of the enterprise Chef servers.

To run chef-client under *local mode*, you simply run chef-client with the `-z` option. This will start up a chef-zero server, and your chef-run will communicate with chef-zero instead of any Chef server specified in the `client.rb` configuration file. We'll look more at how this functionality is implemented in "Tracing a Chef-client Run" on page 79, and you can find out more about how to use chef-zero in your cookbook testing workflow on the Chef Blog.

Chef-zero provides as a lightweight implementation of a Chef server, hence supports all of the same commands and operations as the open sourced Chef server.

Private Enterprise Chef

Enterprise Chef is the commercial version of the open source Chef server. The private version of Enterprise Chef is hosted inside your firewall on your own hardware and provides clustering support out of the box. Along with the features provided in the open source version, it provides a number of additional features, some of which are touched on elsewhere in this book and have been highlighted below:

After "Hosted Enterprise Chef" on page 21, Private chef is the next easiest option to get up and running with, and is particularly recommended for those new to Chef or looking to evaluate it in a setting with the budget to accommodate the hardware and licensing costs of running an private Chef appliance.

Role-based Access Control
: Enterprise Chef provides more fine grained access control than the open source version. Whereas open source chef will give access to all objects (cookbooks, nodes, users, roles etc) to any properly authenticated user, Enterprise chef allows you to refine this permissions model to grant different permission levels (for example create, read, update) to different users or groups of users.

Multi-tenancy
: Enterprise chef allows a user to maintain totally distinct chef setups for different "tenants". For example, if you were running a chef server which served a number of companies, Enterprise Chef would allow you to keep functionally separate chef "environments" for each client - cookbooks, node data, attributes, search indexes etc would all be separated per-client.

Push client runs
: Under the open source version of Chef server, chef-client runs on a "pull" model. This means that chef-client will typically run in daemonized mode, and will query the server for details on what it should run on the node at a configurable interval. When you make a change, unless you script a manual chef run on all of your nodes you have to wait until the next chef-run for the changes to go out. Enterprise chef allows you to push runs out to your nodes, which means that if you so desire you can kick off chef-client runs as soon as a change has gone out without the need for any manual scripting.

All material covered in this book is fully compatible with Private Enterprise Chef.

Hosted Enterprise Chef

The hosted version of Enterprise Chef provides all of the same features as the private version, with the addition that it is not hosted inside your organization but rather managed by Chef Inc.. This means that Chef Inc. host the systems which power hosted Chef and handle all updating, scaling and support.

Hosted Enterprise Chef is the easiest option to get up and running with, and is particularly recommended for those new to Chef or looking to evaluate it.

All material covered in this book is fully compatible with Hosted Enterprise Chef.

Prerequisites

The last thing I'd like to cover in this chapter is a quick note on the prerequisite tooling and knowledge I'm assuming in my readers. This book is aimed at people who are already comfortable using Chef and want to level up their skills, and assumes that if you're not using "Hosted Enterprise Chef" on page 21 you already have a Chef server installed and configured, or are using chef-solo.

Although advanced Chef knowledge is not a prerequisite for this material, you should ideally have read, or be familiar with the concepts covered in, "Learning Chef" (O'Reilly) and be familiar with most of the following areas:

Knife

Although you don't have to be familiar with every supported knife command, you should be comfortable with how to configure and install knife and run basic commands such as:

- Uploading cookbooks
- Creating clients
- Uploading and editing roles and environments

Nodes and Clients

You should understand how to register nodes with Chef, configure the run_list of a node and how to run chef-client. You should be at least roughly familiar with the anatomy of a chef run and be able to interpret basic errors when a run fails.

Cookbooks, Attributes, Roles, Environments and Databags

You should be comfortable writing recipes using the resources provided by out-of-the-box Chef, using `notifies` to initiate actions on other resources, and combining those recipes into a cookbook. You should have a basic understanding of the concept of attributes in Chef and how to set and read them.

You should at least roughly understand how to edit and use roles, environments and data bags. Advanced knowledge isn't necessary, but you should be aware of the concepts and how they integrate with the rest of Chef.

Chef Search

You should understand how to perform simple searches in Chef either in recipes or using knife, such as finding a list of all nodes containing "foo" in their names.

Ruby

You don't need to be a Ruby expert to work through the material in this book, but you should be comfortable with the level of Ruby covered in "Learning Chef" (O'Reilly), or Chef Inc.'s Just Enough Ruby for Chef page. We'll cover all of the additional Ruby concepts you'll need for this book in Chapter 2, but if you'd like a more comprehensive Ruby reference to help you along I'd recommend either of The Ruby Programming Language (Flanagan & Matsumoto, O'Reilly) or Learning Ruby (Fitzgerald, O'Reilly). Both are excellent and extremely comprehensive books, and although they contain far more Ruby than is necessary to follow the material in this book, will look a lot more about writing "correct" Ruby and how to stick to Ruby best practices than we do here.

Assumptions

As Chef is supported on a number of different platforms, throughout this book I've had to make several assumptions about the environment in which you're running Chef in order to keep examples as readable and simple as possible. I have assumed that:

- You're running on a Linux / Unix based operating system - this includes Mac OS X.
- You installed Chef via the omnibus installer documented on the Chef Inc. Install Chef page.
- You're running at least Chef version 11.10.0.
- You have the Git source code management system installed.
- You're familiar with the use of a programmer's text editor such as Vim, Emacs, or TextMate.

If any of these assumptions do not apply to you (for example if you're running Windows or installed chef from Rubygems), you may find that some of the directory paths and commands used throughout the book require a little tweaking to work in your environment. I've done my best to indicate when this is likely to be the case.

CHAPTER 2

Just Enough Ruby to Customize Chef

If you're already familiar with Object Oriented programming with Ruby, including inheritance, namespaces, exceptions and the Ruby scoping model then you may wish to skip this chapter.

As you'll most likely already have realized while writing cookbooks and recipes, the amount of actual Ruby knowledge needed to write Chef recipes is relatively light. The out-of-the-box resources provided with Chef do an excellent job of abstracting away many of the common tasks required, and a smattering of Ruby basics such as assigning values to variables and writing `if` statements fill in the gaps. But there's a lot more going on under the hood to provide those Chef resources, knife plugins and libraries and before we can customize them, it is necessary to understand them.

This chapter will teach you some fundamental Ruby concepts which are essential knowledge for customizing Chef. I've tried to make the material in this chapter as accessible as possible to those new to Ruby, but as I only have a limited space in which to present some sizable concepts don't worry if it feels like a little too much to take in in one go.

Needless to say this chapter will not cover **everything** there is to know about the Ruby programming language, and throughout the course of the book I'll be introducing some additional Ruby concepts or expanding on those covered here - the aim of this chapter is simply to give you a good foundation to build on as we work through the material in this book.

With that out of the way (hopefully I haven't scared you off yet!), let's get started by taking a look at what sort of programming language Ruby actually is. If you've worked with other languages in the past such as Java, C or Perl, this will help frame the material in this chapter. To give it its proper technical classification, Ruby is an object oriented,

dynamically typed language. But what does this actually mean? We'll come on to look at dynamic typing later on in this chapter, but first let's look at what it means for a language to be object oriented.

The material in this chapter is compatible with Ruby 1.9 and up. If you're using Ruby 1.8, some of the material and examples in this chapter will not work.

To run the example code yourself, copy and paste the program listing into a text file named as indicated in the example title, and execute it using:

```
$> ruby <name_of_example_file>
```

Ruby is Object Oriented

If you have previous experience of other programming languages you may already be used to the concept of **primitive** values but if you haven't encountered the term before, assigning a primitive value to `foo` in the below example would mean that we're assigning a simple string value of `"Chef Rocks!"` to the variable `foo`. We can't do anything fancy with it, it's just a value. In Ruby, all values are actually objects.

An Object is a special kind of structure which can posess both *attributes* which describe particular parts of it, and *methods* which can be used to control it and manupulate its attributes. In Ruby, objects are actually instances of a *class*, which can be thought of as the template for the Object. Don't worry too much about this terminology for now, we'll look at how classes are defined in more detail in the "Classes" on page 30 section of this chapter. For now, it's enough to remember that all values in Ruby are really objects under the hood.

```
foo = 'Chef Rocks!'
```

In the above example, what we're **actually** doing is creating a new object of class `String` with the value `"Chef Rocks!"` and assigning a **reference** to that object to the variable `foo`. Note that `foo` does not hold the actual object, just a reference to it. This may seem like a trivial distinction, but as we'll see later in this chapter, it is rather an important one. Don't worry if some of that terminology was unfamiliar to you, we'll examine each part in due course.

For certain object types such as `String`, Ruby performs this object creation under the hood, allowing us to use the simple code seen above. Let's try writing the full version of this code now to demonstrate this:

```
foo = String.new('Chef Rocks!')
```

As far as the Ruby interpreter is concerned, the longform example above is identical to the first simple example we looked at. Let's prove this with the following code, which we'll paste into a file called *example.rb*:

Example 2-1. example.rb

```
foo = 'Chef Rocks!'
bar = String.new('Chef Rocks!')
puts "foo class: #{foo.class}"
puts "bar class: #{bar.class}"
puts foo == bar
```

In the above code, when we say `foo.class` this means that we're calling the `class` method of the variable `foo`, which is a `String` object. In Ruby, a call to a method of an object is always indicated by the `.` character using the `<object>.<method>` syntax as we saw here.

When we run *example.rb*, we'll see the following output:

```
$> ruby example.rb
foo class: String
bar class: String
true
```

When this code is run, the Ruby interpreter will create two variables **foo** and **bar**, which hold references to `String` objects. It will `puts` a string containing some interpolated ruby which calls the `class` method of the `String` object - all Object types in Ruby have this method, and it does exactly what you might expect - return the object's class. It will then `puts` whether or not **foo** and **bar** have equal values which in this case this returns `true` as we'd expect.

`puts` is a Ruby function which prints a `String` to the console followed by a newline, unless the specified string already ends with a newline. If the object passed to `puts` is not a `String`, it will be converted to one. For example, in the above example, `puts "foo class: #{foo.class}"` will print `foo class: String` to the console.

The idea that variables only store references to objects holds true for the majority of Object types you'll encounter in Ruby - we'll touch on one exception to this rule further on in the chapter.

Ruby is Dynamically Typed

So now we know what is meant when we say that Ruby is an "Object Oriented" programming language. But what does "Dynamically Typed" mean? Consider again that first example we looked at:

```
foo = 'Chef Rocks!'
```

Note that here we didn't tell Ruby that the variable `foo` was going to contain a `String`, we just assigned a value to it and let the Ruby interpreter figure out what it was. Compare this with the equivalent code in Java, which is a statically typed language.

```
String foo = new String("Chef Rocks!");
```

In the Java example, we had to tell the Java compiler specifically that we wanted to create a `String` variable, and then assign an actual `String` value to it. Because Java is statically typed, once a variable has been declared as of a particular type, it can never be any other type unless you redeclare it all over again. If you attempt to assign an `Integer` to a variable previously declared as a `String`, you'll get an error.

In dynamically typed languages such as Ruby, you don't need to declare exactly what type of `Object` your variable is going to reference, you just assign values to it. You can even assign values of different types to the same variable, as seen here:

```
foo = 'Chef Rocks!' # Here we're assigning a String to foo

foo = 1 # Here we're assigning an Integer to foo
```

In Ruby, a # followed by text indicates a *comment* rather than actual Ruby code. We'll use this syntax extensively throughout this chapter as a way of documenting code examples

In the first two lines, we're assigning a `String` value to `foo`, which Ruby detects because the value is surrounded by quote marks. However on the last two lines, we're then assigning an `Integer` value to `foo`, which Ruby also detects since the `1` is not enclosed in quotes like a `String` would have been. This is a simple but effective demonstration of both the power and danger of dynamic typing. So where's the danger in dynamic typing? Doesn't it just make things simpler and your code more compact?

Well, dynamic typing does indeed do these things, but there are also some pitfalls to be aware of. Dynamic typing make our lives easier insofar as it means we don't have to worry about telling our code specifically what type every single object has every time we want to use it. But it also means that we can't always tell what type a variable is when we want to use it, without checking its `.class` method of course. When we start looking at creating classes and passing variables around this danger will become more apparent, but for now let's look at another simple example:

Example 2-2. example2.rb

```
foo = 'Chef Rocks!'
bar = 1
```

```
if foo = bar
  puts "Something's wrong, #{foo} shouldn't equal #{bar}"
else
  puts "All is well with the world, #{foo} does not equal #{bar}"
end
```

Simple enough code, right? We should always hit the `puts` statement that tells us `foo` does not equal `bar`. Let's put the above code into a file called *example2.rb*, run it, and see what happens:

```
$> ruby example2.rb
Something's wrong, 1 shouldn't equal 1
```

Huh, that's not what we expected at all, how did that happen? The more observant of you might have spotted the typo in example2.rb though - instead of my `if` statement testing for equality, which would have been `if foo == bar`, what I **actually** did was tested whether assigning `bar` to `foo` was successful with `if foo = bar`.

Not all Equality checks are created Equal

If you're used to the way checking for equality works in Java-like languages, where `foo == bar` tests whether or not the two objects are the same object and `foo.equals(bar)` tests whether the two objects have identical values, then right about now you might be a little confused. In Ruby, equality checking actually works the other way round. `foo == bar` tests whether or not the two objects have the same value, and `foo.equals(bar)` tests whether or not `foo` and `bar` are the same object.

Now in a statically typed language, the above example would have caused an error at compile time even before running the code. `foo` is clearly a `String`, and bar is clearly an `Integer`. You just can't run around assigning `Integers` to `String` variables in statically typed languages. In Ruby however, with its dynamic typing, the code did not produce an error, and we hit the part of the if statement that should be logically unreachable, all becuase of a misplaced equals sign.

Of course this was a contrived example designed to check specifically for this gotcha, but imagine if that typo had occured in a complex method containing tens of lines of code. All of a sudden, something you took for granted as being a `String` is all of a sudden an `Integer`. Tracking down this sort of issue in dynamically typed languages can be extremely tricky, and it's important to make you aware of this scenario so that you know to be careful. Dynamic typing **does** make things simpler than statically typed languages, but it also means we as programmers have to be extra careful that our variables refer to Objects of the type we think they do.

So now that we've got a good handle on what "Object Oriented" and "Dynamically Typed" actually mean, it's time to start learning how to create our own objects - after all, Ruby would be quite a limited language if you could only make use of the built-in Objects. As with Chef, Ruby is nearly infinitely customizable but it would take another book entirely to explore this topic in depth, so let's just focus on the bits that will be useful when customizing Chef.

Classes

All objects in Ruby are instances of a `Class`. As we've already seen earlier in this chapter, an `Object` in Ruby is a special structure which can contain both attributes to describe it and methods to control it. A `Class` definition is the blueprint for an object, which defines these attributes and methods. For example whenever you assign a `String` to a variable in Ruby, what you're actually doing is creating an instance (a `String Object) of the +String` class.

Classes are not always standalone templates either - classes can inherit methods from other classes and override methods defined in classes from which they inherit. In fact, all classes in Ruby inherit from a class called `BaseObject` which can be thought of as a "master blueprint" which defines the methods like `.class` that we've been using in our examples. But let's not get too far ahead of ourselves. Let's start at the very beginning, with how you define a class in the first place:

```
class Awesome
end
```

That's it, pretty simple huh! Now, let's use our new class definition and create an instance of it - we'll put the code in a file called *example3.rb*:

Example 2-3. example3.rb

```
class Awesome
end

awesome_sauce = Awesome.new
puts "awesome_sauce class: #{awesome_sauce.class}"
```

When we run *example3.rb*, we'll see the following output:

```
$> ruby example3.rb
awesome_sauce class: Awesome
```

When we create an *instance* of a Class, we're essentially creating a new copy of the template contained in the class definition. The methods and variables inside each instance of a Class are unique to that instance. This means that we could create two completely seperate instances of our `Awesome` class by replacing the code in `example3.rb` with the below:

Example 2-4. example3.rb

```
class Awesome
end

awesome_sauce = Awesome.new
awesome_sauce2 = Awesome.new
puts "awesome_sauce class: #{awesome_sauce.class}"
puts "awesome_sauce2 class: #{awesome_sauce2.class}"
```

Now when we run *example3.rb*, we'll see the following output:

```
$> ruby example3.rb
awesome_sauce class: Awesome
awesome_sauce2 class: Awesome
```

Unfortunately, our `Awesome` class isn't so awesome yet - we can create as many instances of our class as we want, but we can't really do anything with it because it has no methods. Let's define a method that lets us initialize our `Awesome` object with a parameter:

```
class Awesome
  def initialize(awesome_level)
    @awesome_level = awesome_level
  end
end
```

The `initialize` method in Ruby is a special method which is called when you instantiate your Object with `.new`. In this case, our `initialize` method takes one parameter, `awesome_level`. Inside the body of the method, we're assigning the values of this parameter to a **class instance** variable, named `@awesome_level` (that's what the `@` means). We'll go into more detail on the different sorts of variable scopes you can use later in "Variable Scoping" on page 34 but for now it's sufficient to say that the parameter is only accessible ("in scope", to use the correct terminology) inside the method which it was passed to, whereas the class instance variable is in scope to any methods in any instance of that class. So now, when we create our new `Awesome` method, we can do this:

```
awesome_sauce = Awesome.new(100) # We're now passing in the parameter our initialize method expect
```

We're getting there, but our class still isn't as awesome as its name implies. We've defined an initialize method to let us set an initial `awesome_level`, but how do we get that value out again later? What if we want to change the `awesome_level`?

Getter and Setter Methods

To retrieve the `awesome_level` from our new class, we could add a new method that returns the class instance variable like this:

```
class Awesome
  def initialize(awesome_level)
    @awesome_level = awesome_level
```

```
  end

  def awesome_level # getter method for awesome_level
    @awesome_level
  end
end
```

This new method is what's known as an **accessor** method - it lets you access the value of your class instance variable. Now that we've defined our accessor method, we can do this with our Object:

```
awesome_sauce = Awesome.new(99)
puts "Awesome level is #{awesome_sauce.awesome_level}"
```

Now that's more like it, we can create our object with an initial `awesome_value`, and we can get that value back out again. But what if we want to change the `awesome_level` after we've already created the object? Well, we could add a new method like this:

```
class Awesome
  def initialize(awesome_level)
    @awesome_level = awesome_level
  end

  def awesome_level # getter method for awesome_level
    @awesome_level
  end

  def awesome_level=(new_awesome_level) # setter method for awesome_level
    @awesome_level = new_awesome_level
  end
end
```

This new type of method is called a **setter** method. Note that we've used the same name as we did for the accessor method, but with an = sign after it. This is a special method naming convention used in Ruby to define **setter** methods - if your setter method is named `awesome_level=`, when you actually use it in code you are able to do the following:

```
awesome_sauce.awesome_level = 99
```

Tying this all together in *example4.rb*, we are now able to both set and get the value of Awesome.awesome_level:

Example 2-5. example4.rb

```
class Awesome
  def initialize(awesome_level)
    @awesome_level = awesome_level
  end

  def awesome_level # getter method for awesome_level
    @awesome_level
```

```
  end

  def awesome_level=(new_awesome_level) # setter method for awesome_level
    @awesome_level = new_awesome_level
  end
end

awesome_sauce = Awesome.new(100)
puts "awesome_sauce has an awesome_level of  #{awesome_sauce.awesome_level}"

awesome_sauce.awesome_level = 99
puts "awesome_sauce has an awesome_level of  #{awesome_sauce.awesome_level}"
```

When we run *example4.rb*, we'll see the following output:

```
$> ruby example4.rb
awesome_sauce has an awesome_level of  100
awesome_sauce has an awesome_level of  99
```

This combination of *getter* and *setter* methods in Ruby is extremely common. In the majority of cases, if you have declared a class instance variable inside your class, you will nearly always want to define accessor and setter methods to go along with it. Because this usage pattern is so common, Ruby provides a special method called `attr_accessor` which automates the creation of *getter* and *setter* methods. Using this technique, we can reduce our `Awesome` class down to the following:

```
class Awesome

  attr_accessor :awesome_level

  def initialize(awesome_level)
    @awesome_level = awesome_level
  end
end
```

This class is functionally identical to the longform version above - the `:` character before the name `awesome_level` indicates that `awesome_level` is a *symbol* being passed as a parameter to the `attr_accessor` method. Symbols in Ruby are the closest it has to the concept of *primitive values* that we looked at in "Ruby is Object Oriented" on page 26 - for the purposes of this example it's sufficient to treat them as *immutable* variables, ie their value cannot be changed once set. So in this case, we're passing the *immutable* value `awesome_level` to the method `attr_accessor` as we need to ensure the name of the method can't be changed after it's defined.

Try replacing the class definition in *example4.rb* with the above condensed definition and see for yourself!

Variable Scoping

In the examples in the "Classes" on page 30 section, we took a normal variable passed as a parameter to our `initialize` method and assigned it to a *class instance* variable. Why was that necessary? Why couldn't we just use the method parameter everywhere inside our class? The answer is something called variable scoping.

Although you will find far more complex definitions of the term elsewhere, the definition of variable scoping I'm going to use here is "the thing that determines how much of your program can see the variable in question". We're going to look at two different levels of variable scopes here: *local* variables and *class instance* variables.

Local Variables

Local variables are variables which are only visible to the method or block in which they were defined. Unless you add the prefixe necessary to indicate *class instance* variables, your variable will be locally scoped by default. Consider the following example:

Example 2-6. example5.rb

```
def mymethod(awesome_level)
  puts awesome_level # this will work
end

awesome_level = 50
puts awesome_level # This will print 50.
mymethod(100)
puts awesome_level # This will still print 50.
```

When we run *example5.rb*, we'll see the following output:

```
$> ruby example5.rb
50
100
50
```

In the above example, we're setting the value of `awesome_level` to 50, and then the following `puts` statement prints it out. Next, we make a call to the `mymethod` method, passing it a value of 100. Inside the body of `mymethod`, we also have a parameter called `awesome_level`, which we then print using another `puts` statment. It's important to note here that although we have two variables named `awesome_level_`, Ruby treats them as if they were totally seperate. The `awesome_level` variable used inside the body of `mymethod` is only visible (*in scope*) inside that method - outside of the method body, it's as if it doesn't exist, so our first *awesome_level* variable still has a value of 50. We demonstrate this by printing the value of `awesome_level` again using another `puts` statement.

Example 2-7. example6.rb

```
awesome_level = 99

5.times do
  puts "Awesome level is #{awesome_level}" # this will work
  how_much_awesome = "so much awesome!"
end

puts how_much_awesome # this won't work
```

When we run *example6.rb*, we'll see the following output:

```
$> ruby example6.rb
Awesome level is 99
Awesome level is 99
Awesome level is 99
Awesome level is 99
Awesome level is 99
example6.rb:8:in `<main>': undefined local variable or method
  `how_much_awesome' for main:Object (NameError)
```

The `puts` statement inside our `5.times` block will work perfectly, because `awesome_level` was declared outside of the block. However, the last line where we try to `puts how_much_awesome` will fail because the variable `how_much_awesome` is only in scope inside the block.

Class Instance Variables

Class instance variables are visible to everything inside the specific instance of the class in which they are defined. To indicate a variable as a class instance variable, we prefix it with the `@` symbol. We've already seen how *class instance* variables behave in the `Awesome` class we created earlier in the chapter:

```
module AwesomeInc
    class Awesome

    attr_accessor :awesome_level

    def initialize(awesome_level)
      @awesome_level = awesome_level
    end
  end
end
```

The `@awesome_level` variable is defined in our `initialize` method, and is then also visible inside our *getter* and *setter* methods as well. It's important to note though, `@awesome_level` is only visible **inside** the class and this is why we had to create our accessor method earlier to expose this value to code outside of our class definition.

Because *class instance* variables are only visible inside a specific instance of the class that defines them, we can quite happily have two `Awesome` objects created side by side without altering each others `awesome_level`:

Example 2-8. example7.rb

```
module AwesomeInc
    class Awesome

    attr_accessor :awesome_level

    def initialize(awesome_level)
      @awesome_level = awesome_level
    end
  end
end

foo = AwesomeInc::Awesome.new(10)
bar = AwesomeInc::Awesome.new(20)

puts foo.awesome_level
puts bar.awesome_level
```

When we run *example7.rb*, we'll see the following output:

```
$> ruby example7.rb
10
20
```

Inheritance

As we've talked about already in this chapter, classes don't just have to be standalone chunks of code. Ruby, in common with most other object oriented languages, allows you to base your class on already existing classes, modifying their behavior as you please. Before we look at how exactly to do this, there are a couple of definitions to get out of the way first.

Superclass
: When your class extends the behavior of another class, the class whose methods it inherits is known as the *Superclass*

Subclass
: Conversely, the class which is modifying the behavior of a class it inherits from is known as the *Subclass*.

Those definitions can sometimes be a little confusing, so let's look at an example using the `Awesome` class we defined before. Let's say we want to have another class, called `ReallyAwesome` which *inherits* the method we already defined in our `Awesome` class. This

would make `Awesome` the *superclass* and `ReallyAwesome` the *subclass* as shown in this diagram:

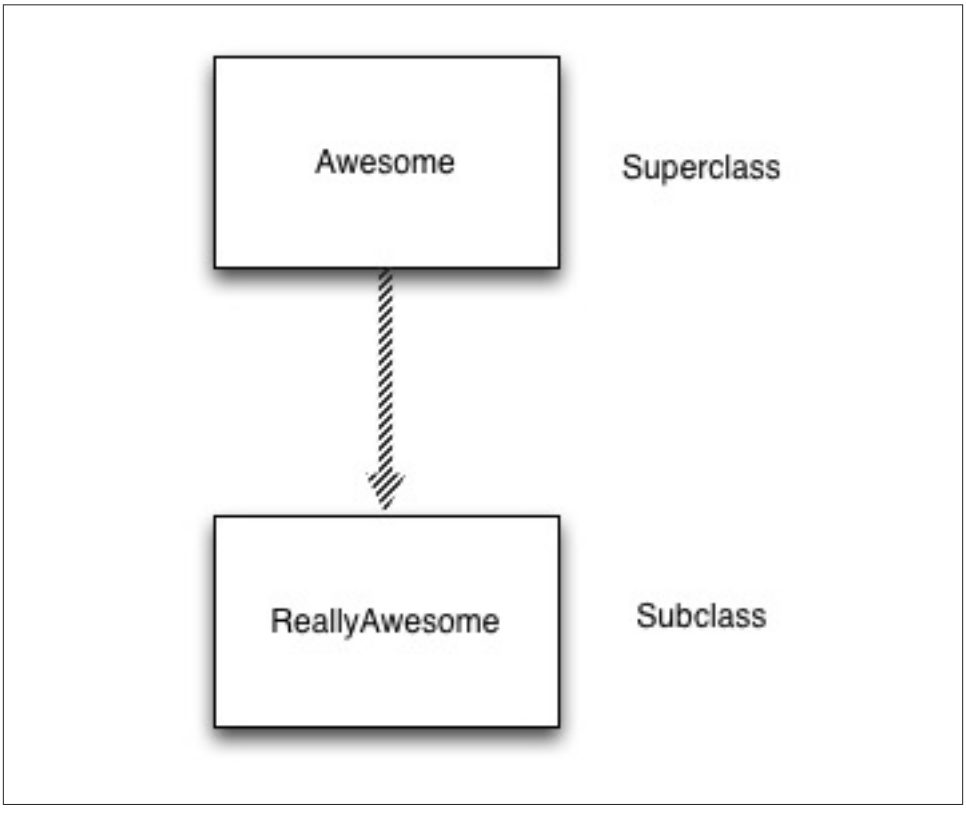

Figure 2-1. Inheritance

Here's how we define that inheritance in our code:

```
class ReallyAwesome < Awesome
end
```

In this example, we're declaring that the class `ReallyAwesome` *inherits* from `Awesome`. So in this case, `ReallyAwesome` is the *subclass* and `Awesome` is the *superclass* - it's worth noting that once defined, this superclass / subclass relationship cannot be changed while the program is running. We can demonstrate how method inheritance works by creating a new instance of the `ReallyAwesome` class:

```
really_awesome_sauce = ReallyAwesome.new(1000)
```

In this example, you can see that we called the `awesome_level` method of our `ReallyAwesome` object. But wait, in the `ReallyAwesome` class we didn't define any methods, did

we? That's where inheritance comes in. The `initialize` method and the `attr_accessor` method from the `Awesome` class were inherited by the `ReallyAwesome` class. When we come on to look at creating Knife plugins and handlers in later chapters, we'll see how inheritance allows us to extend the powerful object classes provided with Chef to avoid us having to re-implement them from scratch.

Modules

Much like a class, a *module* in Ruby is a collection of methods, class variables and constants. Unlike classes, however, you can't create *instances* of modules and you cannot declare classes to be a *subclass* of a module (this means that you cannot *inherit* module methods). Modules stand alone and have no concept of inheriting from a base object in the way that all classes inherit from `BaseObject`. So what **are** modules useful for? Modules in Ruby are typically used for two specific purposes - namespaces and mixins.

Modules as Namespaces

Namespaces in programming terminology are essentially "containers" which help you organize things like methods and classes. The usefulness of namespaces becomes more apparent as the programs you write become more and more complex and you start to make use of libraries and classes provided by other programmers - it becomes a useful tool to have objects and methods use descriptive and helpful naming schemes.

By way of an example, let's look at the `Awesome` class we created earlier:

```
class Awesome

  attr_accessor :awesome_level

  def initialize(awesome_level)
    @awesome_level = awesome_level
  end
end
```

Now that we've defined a class called `Awesome`, what happens we want to make use of another library in our code which **also** defines a class called `Awesome`. Do we change the name of our class? Do we ask them to change the name of their class? Fear not, brave programmer, this is where module namespaces come in! We can wrap our existing class in a `module` definition like this:

```
module AwesomeInc
    class Awesome

    attr_accessor :awesome_level

    def initialize(awesome_level)
      @awesome_level = awesome_level
```

```
    end
  end
end
```

This now means that we can easily keep **our** `Awesome` class separate from any others which happen to use that name. One important thing to note though - now that we've wrapped our class in a namespace, the code we used for creating new instances of our object before won't work:

```
awesome_sauce = Awesome.new # This will now produce an error
```

The reason for this is actually very simple. When we're asking Ruby to create a new instance of the `Awesome` class, what we're actually doing is asking ruby to look in the *global* namespace for a class called `Awesome`, and create an instance of that. The *global* namespace is where Ruby keeps classes that have not been given a specific namespace of their own - but we just added our class to its own namespace with a `module` definition, so how do we create instances of it now? Again, the answer is simple. We just have to tell Ruby which namespace to look in for the object, like this:

```
awesome_sauce = AwesomeInc::Awesome.new # Yay, it works again!
```

Our code now works again and our `Awesome` class is now nicely organized under the `AwesomeInc` namespace where it won't clash with any other classes. Although we've just used one module definition here, it's actually possible to have namespaces multiple levels deep. As we progress through the book, you might see objects being created with a statement like this:

```
query = Chef::Search::Query.new
```

All this means is that in the definition for that class, you'll find it declared like this:

```
module Chef
  module Search
    class Query
      ...
    end
  end
end
```

A useful feature of namespaces is that a class in a particular module is able to address other classes in that module without specifying the full namespace - Ruby will figure that part out for you by searching the inheritance chain. For example:

```
module Chef
    class AwesomeSearch
        # Here we're able to create a Chef::Search::Query object
        # without specifying the Chef part as our class is also
        # in that module
       query = Search::Query.new
    end
```

```
    end
  end
```

For large complex systems like Chef, module namespaces are an excellent way to keep things need and tidy, and also to keep programmers from going insane like they would if everything was declared in the *global* namespace. Imagine, hundreds or thousands of classes definitions, all having to be uniquely named - this is why module namespaces are an essential tool to keep in your Ruby toolbox!

Modules as Mixins

Modules in Ruby are able to hold **methods** as well as classes. Imagine for a moment that you're implementing several classes in Ruby which will make use of a shared set of methods to provide common functionality.

One approach to this problem might be to create a "Helper" class, which is instantiated inside each of your classes to provide access to these methods. Rather than creating objects that we don't strictly need, a more elegant approach is to store these common methods inside a Module, and include this module in each class which needs those methods. This technique of including methods from a module inside a class in this way is known as a *Mixin*, because we're literally "mixing in" the methods to our class.

But let's not get too far ahead of ourselves - let's look at an example of a mixin in action. We'll put this code into a file called `mixin_example1.rb`:

Example 2-9. mixin_example1.rb

```
# define our module called 'Awesome'
module AwesomeModule
  # define a method called add
  def add(a,b)❶
    # return parameter a added to parameter b
    a+b
  end
end

# Now define our class called 'AwesomeClass'
class AwesomeClass❷
  # Include methods defined in the module Awesome in this class
  include AwesomeModule❸

  attr_accessor :a, :b

  def initialize(a,b)
    @a = a
    @b = b
  end

  def add_numbers
    # Call the add method from the Awesome module
```

```
    add(@a,@b)❹
  end
end

awesome_class = AwesomeClass.new(1,2)
puts "Result is #{awesome_class.add_numbers}"
```

❶ Here we're creating a method called `add` inside a module called `AwesomeModule`. Note that this method doesn't live inside a class definition - if it did, we couldn't use it in a mixin.

❷ Here we're defining our `AwesomeClass` class. Note that this class is not defined inside the `AwesomeModule` namespace - it could be if we wished, but it's not necessary for mixins to function.

❸ Here we're using the `include` directive to tell Ruby that all methods defined in `AwesomeModule` should be available to this class

❹ Here we're calling the `add` method - this method isn't defined inside the `AwesomeClass` class, but rather in the `AwesomeModule`. The `include` statement allowed us to make use of this method.

When we run *mixin_example1.rb*, we'll see the following output:

```
$> ruby mixin_example.rb
Result is 3
```

As we see in the above code example, Ruby allows us to define methods in a Module, and then make use of those methods in our classes through the use of the `include` directive. This "mixin" technique is a powerful way of avoiding repetition of shared code throughout our classes, while avoiding the creation of hoardes of objects used just to share methods between classes.

By providing *namespacing* to help us organize our classes and methods and *mixins* to allow us to easily share code between class definitions, Ruby modules are an essential tool in the toolbox of any Ruby programmer.

Using Other Classes & Modules

So far we've looked at how to write our own classes and modules, but of course at some point we're probably going to want to incorporate classes and modules written by third parties - in the case of the material covered in this book, we're going to want to use the classes that Chef Inc. have provided for interfacing with your Chef setup. In this section we'll look at the different ways to incorporate third party modules and classes into our code and make use of them.

Including other classes and modules in your Ruby program is done using a special function called `require`. Adding `require` followed by a string will add the specified

classes to the list your program knows about. There are several different types of classes you can require:

Local Classes

The simplest use of the `require` function is to include a class defined in another file you want to require by name. Suppose we have the following file declared:

Example 2-10. awesomeclass.rb

```
module AwesomeInc
    class Awesome

    attr_accessor :awesome_level

    def initialize(awesome_level)
      @awesome_level = awesome_level
    end
  end
end
```

And we want to make use of this class in a new file - we can `require` and use it as follows:

Example 2-11. really_awesomeclass.rb

```
require '/path/to/awesomeclass'

module AwesomeInc
  class ReallyAwesome < Awesome # Awesome class is declared in the file we're requiring
  end
end
```

Note that here we've had to tell Ruby the full path to `awesomeclass.rb`, but did not include the `.rb` file extension. This is because by default, Ruby looks for included Ruby files in a special list of directories known as the `LOAD_PATH`. When you specify a full file path like we did in the above example, the `LOAD_PATH` is bypassed. This is somewhat inconvenient when you're only including files that are in the same directory as your new file or a subdirectory, so Ruby 1.9 introduced a function called `require_relative` which will look for files *relative* to the location of the current file. This would allow us to replace our full file path in the above example as follows:

Example 2-12. really_awesomeclass.rb

```
require_relative './awesomeclass'

module AwesomeInc
  class ReallyAwesome < Awesome
  end
end
```

Rubygems

When you want to make use of classes and modules written by third parties, you will most likely find that they've distributed them as a *gem*. *Gems* are the format in which Ruby's package management system *Rubygems* distributes classes and modules. Rubygems not only handles the installation of these third party modules and classes, but also handles installing any other classes and modules depended on by the gem you're installing. Most gems are distributed through the Rubygems website.

To install a Rubygem, you can use the `gem` executable bundled with Ruby, followed by the name of the gem to be installed, like this:

```
$> gem install diffy

Fetching: diffy-3.0.2.gem (100%)
Successfully installed diffy-3.0.2
Installing ri documentation for diffy-3.0.2
Done installing documentation for diffy after 0 seconds
```

Since Rubygems takes care of installing the classes and modules from gems under the `LOAD_PATH` expected by Ruby, we can include our newly installed gem in our code like this:

```
require 'diffy' # Our newly installed diffy gem
require_relative './awesomeclass'

module AwesomeInc
  class ReallyAwesome < Awesome
  end
end
```

It's important to note that what we're actually requiring here is not the name of the gem itself, but a file under the `LOAD_PATH` called `diffy.rb` which was installed as part of the *diffy* gem by Rubygems. You may also note that we referred to the file as `diffy` not `diffy.rb` - Ruby allows you to optionally skip the `.rb` extension when requiring files. We could equally have written `require "diffy.rb"`, but convention when requiring files from gems is to ommit the `.rb` extension. Another convention is to name the "main" file from the gem that you need to require after the name of the gem itself, but this is not always the case.

Because we're able to specify which files provided by a gem we wish to include, it's common to only require those classes installed by a gem that are actually needed. For example, further on in this book, you might see something like

```
require 'chef/knife'
```

All this means is that we're requiring the `knife.rb` class file under the `chef` directory installed by the `chef` gem - you'll see this technique throughout the material presented in this book, as it makes our programs more efficient if we avoid requiring classes that

we don't plan to use. The below table illustrates the difference in load time between requiring the entire `chef` gem versus just requiring the `chef/knife` class:

Table 2-1. Class Load Time Comparison

Load Statement	Load Time
require "chef"	1.955s
require "chef/knife"	0.982s

Built-in Classes

Ruby also ships with a number of classes which are installed by default but not explicitly included in your program by default. These classes collectively form what's known as the *Standard Library* or *stdlib* and are available under the `LOAD_PATH` just like classes installed by Rubygems. You can make use of these classed by adding a `require` statement, just as you would when including local files or Rubygems. Let's include the `FileUtils` class from the *Standard Library* in the example we looked at earlier:

```
require 'fileutils' # A standard library class
require 'diffy' # A class installed by the diffy gem
require_relative './awesomeclass.rb' # A local class file

module AwesomeInc
  class ReallyAwesome < Awesome
  end
end
```

In this example, you can see that we're now making use of classes from all three categories - a local class file, a *stdlib* class and a class installed by a *gem*.

When Things Go Wrong

Now that we've discussed several Ruby concepts that you may not have previously encountered when writing Chef code, let's take a moment to examine what happens when your code goes wrong and what you can do to manage this. Code errors can be caused by wide variety of factors, from attempting to call a non-existent method of an object through trying to divide a number by zero to more complex errors like a Chef run failing. To make its error mechanism as convenient and easy to use as possible, Ruby implements standard method of telling you about errors - it throws an `Exception`.

Exceptions

The `Exception` class is specially designed to provide an error handling mechanism in Ruby. Exceptions in your code can either be raised by Ruby (including by any gems or other classes you're including), or raised directly from your code like this:

Example 2-13. example12.rb

```
foo = "Hello"
raise "This is an error!"
bar = Goodbye
```

The "raise" Keyword

The `raise` keyword, which we used in the examples we've just seen, is a method provided by the `Kernel` module which is in turn included by all instances of the `Object` class. Essentially, this means that any object in Ruby has access to the `raise` method and can throw exceptions.

When we run *example12.rb*, we'll see the following output:

```
$> ruby example12.rb
example12.rb:2:in `<main>': This is an error! (RuntimeError)
```

What we see here is known as a *Stack Trace*. Along with the actual error thrown by the code (`RuntimeError: This is an error!`), Ruby highlights the file in which the error occured (`example12.rb`) and even the exact line number (`:2`) and then stops the code from executing further.

This behavior is the main difference between printing an error message and raising an exception. When you print an error message, your code can continue executing as normal but when you throw an exception, you're telling Ruby to stop executing further code immediately because something has gone wrong. Perhaps more importantly, you're also letting code which might be using instances of your `Object` know that an error has happened, like in this example:

Example 2-14. example13

```
class Awesome
  def break_stuff
    raise "Whoa, this is broken!"
  end
end

foo = Awesome.new
foo.break_stuff # This will throw a RuntimeError
```

When we run *example13.rb*, we'll see the following output:

```
$> ruby example13.rb
example13.rb:3:in `break_stuff': Whoa, this is broken! (RuntimeError)
        from example13.rb:8:in `<main>'
```

In the above output, the usefulness of the *stack trace* that Ruby gives us when exceptions are raised becomes even more obvious - it doesn't just tell us that the exception was

caused at `example13.rb:3` in the `break_stuff` method, but also that the `break_stuff` method was called by `example13.rb:8:in '<main>'`. As your code begins to make use of more libraries and method calls, the *depth* of the stack traces shown when exceptions are thrown can increase dramatically - this sometimes makes them a little challenging to interpret, but also gives an invaluable level of detail into exactly what went wrong with your program.

`RuntimeError`, the exception class that was thrown in the above examples, is the default type of `Exception` that Ruby will raise if you don't specify the exception type. It is actually a subclass of `StandardError`, which is in turn a subclass of `Exception`, and both of these superclasses have a number of other exception subclasses that you can use. It's largely left up to you to decide what exception type to use, depending on the type of error being responded to, but here are a few of the more commonly encountered `Exception` classes:

`RuntimeError`
: A generic error class which is raised when an invalid operation is attempted.

`ArgumentError`
: Raised eg when the number of arguments passed to a method are incorrect.

`IOError`
: Raised eg when an input / output operation fails.

`TypeError`
: Raised when an object is encountered which is not of the type expected by the code.

`ZeroDivisionError`
: Raised when attempting to divide an integer by 0

Once you've identified the exception class you want to use, you can throw an exception of a specific type in your code like this:

```
foo = AwesomeInc::Awesome.new(10)
raise IOError.new("This is an IO error!")
```

This example will output `IOError: This is an IO error!`.

Handling Exceptions

As we've already seen, when your Ruby code raises an `Exception`, code execution stops immediately. But what happens if, for example, you're using a third party Rubygem to access a web service which throws an exception when a user specifies incorrect login credentials? It doesn't seem very user friendly for the program to stop dead in its tracks at this point - let's look at the code we used in *example13.rb* above:

```
class Awesome
  def break_stuff
    raise "Whoa, this is broken!"
```

```
  end
end

foo = Awesome.new
foo.break_stuff # This will throw a RuntimeError
```

Wouldn't it be nice if the code which is calling the `break_stuff` method of our `Awesome` class was able to gracefully handle this error and carry on? Fortunately, Ruby gives us a way to do just this with a special keyword called `rescue`. The `rescue` keyword is used within a block defined by a `begin` statement to indicate the parts of our code for which we want to capture exceptions. Let's try replacing the last two lines of *example13.rb* with the following:

```
foo = Awesome.new
begin
  foo.break_stuff # This will throw an exception
rescue
  puts "Looks like there was an exception!" # But this will handle it!
end
```

Try running our revised *example13.rb* again and see what happens - we should see the following output:

```
$> ruby example13.rb
Looks like there was an exception!
```

This time when the code throws an exception we now see the friendly error message defined inside the `rescue` block rather than the `Exception` error we saw before. If any code executed inside the `begin` block throws an exception, the code inside the `rescue` section will be run, allowing the error to be gracefully handled and program execution to continue.

Ruby also allows us to access methods of the `Exception` object being thrown by naming it in the `rescue` statement like this:

```
foo = Awesome.new
begin
  foo.break_stuff # This will throw an exception
rescue => ex # Let's name our exception object
  puts "Exception of class #{ex.class} thrown with message #{ex.message}" # And use it
end
```

If we alter the rescue block in *example13.rb* to the above, when we run it we will get the following output:

```
$> ruby example13.rb
Exception of class RuntimeError thrown with message Whoa, this is broken!
```

Rescuing exceptions is a powerful method of ensuring that your code can continue executing when errors occur, but it would make our programs rather unwieldy if we had to include code for every possible exception type in the same rescue section. To

make handling different exception types easier to manage, Ruby also lets us specify exactly which classes of `Exception` we want our `rescue` statement to apply to - you can even specify multiple different `rescue` statements for different exception types like this:

```
foo = Awesome.new
begin
  foo.break_stuff # This will throw an exception
rescue RuntimeError => ex
    # Code to handle RuntimeErrors
rescue IOError => ex
    # Code to handle IOErrors
end
```

The judicious combination of `begin` blocks with `rescue` statements which capture specific types of `Exception` allow you to gracefully handle errors in your code while presenting the user with meaningful feedback. Of course, it's totally possible to capture all exceptions with a single rescue block and return a generic error message, but as with much of the material in this book something you **can** do is not necessarily something you **should** do. In general, you want to focus your error handling as tightly as possible and be sure to make your error messages as descriptive and helpful as you can.

Defining Custom Exception Types

Sometimes the relatively limited number of standard `Exception` types provided in Ruby is not sufficient to allow us to properly manage our exception handling and we find ourselves needing to declare a new class of exception. The object oriented nature of Ruby makes this extremely easy to do by allowing you to create a class which inherits from one of the built in Ruby `Exception` classes such as `StandardError` as shown below:

```
class SuperSeriousProblem < StandardError
end
```

Note that this `Exception` class is simply an empty class definition - although you can add extra methods and attributes if you want, the `Exception` class which is their eventual superclass already provides all of the methods like `.class` and `.message` that we used in our earlier examples.

Custom exception classes are an excellent way of giving additional context to error messages that might not be possible with the built-in exception types. Let's now combine all the techniques discussed in this section to augment our example class with a custom `Exception` class, and code to handle that specific exception type:

Example 2-15. example14.rb

```
class SuperSeriousProblem < Exception #Our new custom exception class
end

class Awesome
  def break_stuff
```

```
    raise SuperSeriousProblem.new("Whoa, this is broken!") #Raise our new exception type
  end
end

foo = Awesome.new
begin
  foo.break_stuff # This will throw a SuperSecretProblem exception...
rescue SuperSeriousProblem => ex # Which we're now handling.
    puts "SuperSeriousProblem: Something went really, really wrong."
end
```

When we run *example14.rb*, we'll see the following output:

```
$> ruby example14.rb
SuperSeriousProblem: Something went really, really wrong.
```

By defining meaningful and descriptive exception classes in your code and making sure that your error handling code deals with all the errors that it can, you both improve the readability of your code and the overall usability of your application. Most people don't like having to try and interpret a Ruby stack trace, it's much more useful to present the user with a helpful error message or even better have your code handle the error transparently, recover from it, and move on.

Tying It All Together

We've covered a number of Ruby concepts so far in this chapter that are essential to understand when working through the material in this book. Thus far I've demonstrated these ideas in small standalone chunks, so to help tie everything together in a more realistic context, we're going to look at some more realistic code examples that will demonstrate these concepts working in concert in addition to introducing you to some useful *stdlib* classes.

We'll look at two short example programs:

File Operations
: This example will show you how to perform various operations on local files in Ruby by creating and using a class for writing data to log files. It will demonstrate opening, writing to and clearing files and also introduce you to the Ruby *stdlib* class `File`

HTTP Requests
: This example will introduce you to a two of the *stdlib* classes Ruby provides for talking to HTTP services (`Net::HTTP`) and working with URLs (`URI`) and also how to work with the response object returned by the request.

All of the examples in this section are compatible with Ruby 1.9 and up, and only make use of *Standard Library* classes. No third party gems or classes are required.

To run the example code yourself, copy and paste the program listing into a text file named as indicated in the sample title, and execute it using:

```
$> ruby <name_of_example_file>
```

File Operations

This example defines a module called `Examples` containing a custom exception class called `FileCreationError`, and our main class called `FileLogger`. It will introduce you to several methods provides by the `File` *stdlib* class.

The `FileLogger` class contains an `attr_accessor` method for our single class instance variable and also defines the following additional methods:

initialize
: The initialize method takes a single parameter for the log file path

file_writeable?
: This method will return `true` or `false` depending on whether or not the user running the example has permissions to write to the file.

write_to_log
: This method will take a string as a parameter, and write it to the file.

clear_log
: This method will clear the contents of the log file

Example 2-16. file_operations.rb

```
# The module namespace our classes will live in
module Examples
  # Custom exception class
  class FileCreationError < StandardError
  end

  # Our main class definition
  class FileLogger
    # attr_accessor method for log_file class instance variable
    attr_accessor :log_file

    #initialize method
    def initialize(log_file)
      # set @logfile class instance variable to value of parameter
      @log_file = log_file
```

```
      # wrap initial file creation in a begin block
      begin
        # try creating the file in "write" mode
        File.new(@log_file, "w")
      # rescue Errno::EACCES exception which occurs
      # when file can't be created due to insufficient permissions
      rescue Errno::EACCES
        # raise a custom exception with a more friendly message
        # which is split over two lines below for formatting
        raise FileCreationError.new("#{@log_file} could not be created. "\
          "Please check the specified directory is writeable by your user.")
      end
    end

    # methods of our FileLogger object

    def file_writeable?
      # Return true if our created file is writable, false if not
      File.writable?(@log_file)
    end

    def write_to_log(message)
      # Open our file in "append" mode and write the message string to it
      File.open(@log_file, 'a') {|f| f.write(message) }
    end

    def clear_log
      # clear the contents of the file
      File.truncate(@log_file, 0)
    end
  end
end

# Try creating and writing to a file we should have permissions to

# Initialize our object - note we're specifying Module::Class
puts "Creating log file /tmp/testfile"
file_logger = Examples::FileLogger.new("/tmp/testfile")
puts "file writeable: #{file_logger.file_writeable?}"
puts "Writing to log file"
file_logger.write_to_log ("Test log message")
puts "Clearing log file"
file_logger.clear_log

# puts a blank line for spacing
puts ""

# Try creating and writing to a file we should *not* have permissions to

# Initialize our object - note we're specifying Module::Class
puts "Creating log file /usr/testfile"
file_logger = Examples::FileLogger.new("/usr/testfile")
```

```
puts "file writeable: #{file_logger.file_writeable?}"
puts "Writing to log file"
file_logger.write_to_log ("Test log message")
puts "Clearing log file"
file_logger.clear_log
```

When you run the example code, you should see the following output:

```
$> ruby file_operations.rb
Creating log file /tmp/testfile
file writeable: true
Writing to log file
Clearing log file

Creating log file /usr/testfile
file_operations.rb:25:in `rescue in initialize': /usr/testfile could not
    be created.Please check the specified directory is writeable
    by your user. (Examples::FileCreationError)
        from file_operations.rb:17:in `initialize'
        from file_operations.rb:65:in `new'
        from file_operations.rb:65:in `<main>'
```

As the above output shows, our first attempt to create, log to and clear `/tmp/testfile` succeeded because we have permissions to write to that file.

Our second attempt to create, log to and clear `/usr/testfile` failed, because we don't have permissions to write to that file. The initialize method of our `FileUtils` object caught the exception that was thrown when this happened because the initial file creation was wrapped in a `begin` block, and the `rescue` block in turn raises our custom `FileCreationError` exception with a more friendly error message.

HTTP Requests

This example defines a module called `Examples` containing a custom exception class called `InvalidURLError`, and our main class called `HTTPRequester`. It will introduce you to two *stdlib* classes called `net/http` (the `HTTP` class of the `Net` module), and `uri`. Note that in this example we have to explicitly require both classes because unlike the `File` class they are not included for us by default.

Unlike in the previous example, the `HTTPRequester` class doesn't use an `attr_accessor` method for the class instance variable `@url`, because we want to define custom behavior when setting a new value for `@url`. The `HTTPRequester` class defines the following methods:

initialize
: The initialize method takes a single parameter for the URL we want to work with

url, url=
: The getter and setter methods for the @url class instance variable. We're specifying getter and setter methods instead of an attr_accessor method here because we're doing more than just assigning a value to the @url.

get_request
: This method performs a GET request on the parsed URL in the @url variable, and returns a `Net::HTTP::Response` object.

Example 2-17. http_requests.rb

```
# Require the two stdlib classes we need
# which aren't included by default
require "net/http"
require "uri"

# The module namespace our classes will live in
module Examples
  # Custom exception class
  class InvalidURLError < StandardError
  end

  # Our main class definition
  class HTTPRequester

    # custom getter method for @url class instance variable
    def url
      # The to_s method returns the String representation
      # of the @url variable.
      @url.to_s
    end

    # custom setter method for @url class instance variable
    # because we're parsing the url parameter before assigning
    # to @url
    def url=(url)
      @url = URI.parse(url)
    end

    #initialize method
    def initialize(url)

      # wrap initial HTTP object creation in a begin block
      begin
        # set @url class instance variable to parsed value of url parameter
        @url = URI.parse(url)
      # rescue URI::InvalidURIError exception which occurs
      # when we try to parse an invalid URL
      rescue URI::InvalidURIError
        # raise a custom exception with a more friendly message
        raise InvalidURLError.new("#{url} was not a valid URL.")
```

```
      end

      # Once we're sure our URL is valid, create a Net::HTTP object
      # and assign it to the @http_object class instance variable
      @http_object = Net::HTTP.new(@url.host, @url.port)
    end

    # class method to make a get request
    def get_request
        # Use our @http_object object's request method to call the
        # Net::HTTP::Get class and return the resulting response object
        @http_object.request(Net::HTTP::Get.new(@url.request_uri))
    end
  end
end

#Let's try out our class with a valid URL

# Initialize our object - note we're specifying Module::Class
puts "Initializing Example::HTTPRequester for http://www.oreilly.com"
http_requestor = Examples::HTTPRequester.new("http://www.oreilly.com")
puts "Performing GET request"
# Here we're calling the .code method of the Net::HTTP::Request
# object which is returned by the get_request method.
puts "Response code was #{http_requestor.get_request.code}"

# puts a blank line for spacing
puts ""

#Let's try out our class with an *invalid* URL

# Initialize our object - note we're specifying Module::Class
puts "Initializing Examples::HTTPRequester for 123"
http_requestor = Examples::HTTPRequester.new(123)
puts "Performing GET request"
# Here we're calling the .code method of the Net::HTTP::Request
# object which is returned by the get_request method.
puts "Response code was #{http_requestor.get_request.code}"
```

When you run the example code, you should see the following output:

```
$> ruby http_requests.rb
Initializing HTTPRequester for http://www.oreilly.com
Performing GET request
Response code was 200

Initializing HTTPRequester for 123
http_requests.rb:36:in `rescue in initialize': 123 was not a valid URL.
    (Examples::InvalidURLError)
        from http_requests.rb:29:in `initialize'
        from http_requests.rb:68:in `new'
        from http_requests.rb:68:in `<main>'
```

As the above output shows, our first attempt to make a GET request to `http://www.oreilly.com` succeeded because we passed a valid URL to our object. Our second attempt to attempt to make a GET request to `123` failed because the URL we passed was not valid. The initialize method of our `HTTPRequester` object caught the exception that was thrown when this happened because the initial object creation was wrapped in a `begin` block, and the `rescue` block in turn raises our custom `InvalidURLError` exception with a more friendly error message.

Summary

As I mentioned at the start of this chapter, we've covered a goodly number of quite complex Ruby concepts in a relatively condensed format. If you'd like to dive a little deeper into the topics covered here before moving on, I recommend picking up one of Learning Ruby (Fitzgerald, O'Reilly) or The Ruby Programming Language (Flanagan & Matsumoto, O'Reilly). Both are excellent books, and explain the Ruby programming language in far more comprehensive detail than I've been able to in this single chapter.

CHAPTER 3
Chef Internals

In Chapter 1, we examined **why** we might want to customize our Chef setup. Before we can dive into the meat of the book and start actually creating our own Chef customizations, it's important to ensure that we are able to make an informed decision about **what** to customize - Chef is designed to be extensible in a wide variety of ways, each suited to different types of task.

One of the most powerful tools in your toolbox to help you make this sort of decision is a comprehensive understanding of how Chef works under the hood and how the different components interact. To use an analogy from AwesomeInc's line of work, before we can customize a particular component of a car we need to have an overall picture of how the car functions, and the effect that a change to one component might have on other components.

In this chapter we'll look at:

- An overview of the architecture behind a Chef Server - we won't be customizing Chef server itself in this book, but it's useful to be aware of its components and structure nonetheless.
- The anatomy of a Chef run
- How Chef's design methodology allows us to examine what a run would do to a node if executed
- A quick tour of Chef's source code to familiarize you with how the classes that make up Chef are organized.
- Taking the Ruby knowledge we gained in Chapter 2 to trace the execution path of a Chef Client run through the Chef codebase.

The material in this Chapter is based on (and compatible with) the latest stable release of Chef at the time of writing, version 11.8.2. If you're using older versions of Chef, some of the features or behavior mentioned in this chapter may differ slightly - I'll note when this is the case.

Chef Architecture

Under the hood, a number of different components interact to form a Chef server and client, along with the supporting tooling. These components are illustrated in the Figure 3-1 diagram below, in which I've split Chef into two sections:

Chef Client Tools
: This section describes the chef client programs which run on your nodes, and the other tooling which is used to interface with Chef.

Chef Server
: This section describes the components and interfaces which make up a Chef server, and applies to all forms of Chef which make use of a centralized Chef server - in the case of an open source Chef setup, all of the server components might be on the same physical or virtual machine and in Hosted Enterprise Chef each component is likely to be powered by multiple machines behind the scenes, but the basic architecture is the same.

After the diagram, I'll go on to explain each component in more detail.

If you're a chef-solo user, the Figure 3-1 diagram won't apply to you as chef-solo does not make use of a central server. If you're using hosted enterprise chef, the Figure 3-1 diagram represents a simplified form of the hosted chef platform. Hosted enterprise chef users cannot access this infrastructure other than via the Chef API.

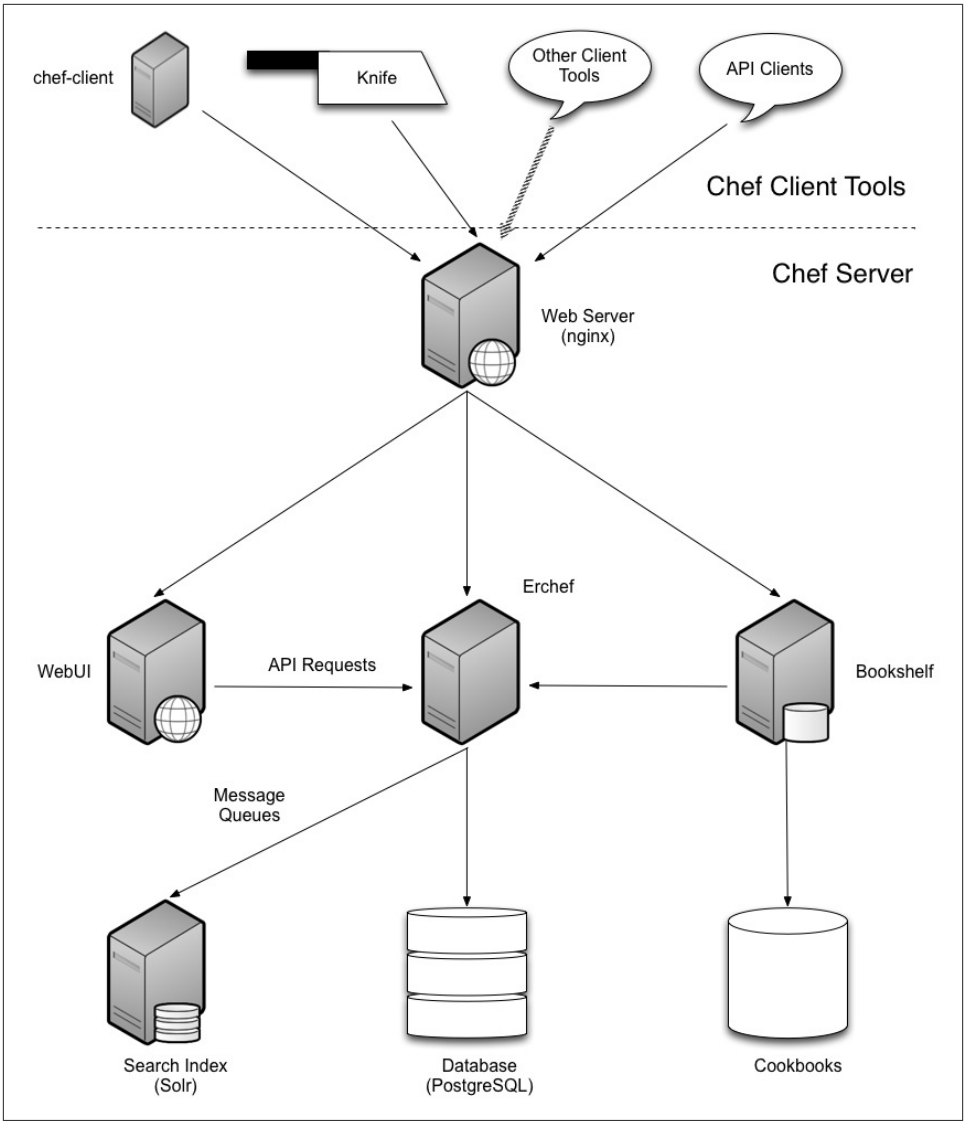

Figure 3-1. Chef Architecture

Now that we have a top-level view of how the components that make up Chef link together, let's examine each of them in a little more detail.

Chef Client Tools

Chef-client

Chef-client is the client program which runs on all of your nodes and actually executes cookbook code during Chef runs. As we looked at in "So Why Chef?" on page 4, Chef is designed to use a "thin server, thick client" design, and as much work as possible is done on the client side by chef-client. Chef client can be run as a *daemonized* process on Unix / Linux based systems, or it can run in a continuous loop controlled by the *interval* and *splay* configuration options. If run with the `--once` command line option, chef-client can also be made to perform a single run and terminate.

Knife

Knife is the primary command line tool for interfacing with Chef. It's used for a number of tasks from uploading cookbooks through running search queries to editing roles. Knife is also extremely extensible - we'll look at this in more detail in Chapter 6.

Other Client Tools

Chef also provides a number of other client tools out of the box such as `chef-shell` (A chef-specific version of ruby's `irb` interactive ruby shell), and `chef-apply` (a tool which allows you to run a single recipe from the command line). Some of these tools (such as chef-shell) can also communicate with the Chef server whereas some like chef-apply do not, and run solely on the client node - this is why the *Client Tools* bubble in the Figure 3-1 diagram is joined to the server with a dashed arrow.

API Clients

Chef's API is accessible to anybody who is authorized to use it, not just out-of-the box Chef tooling. This makes it possible to write scripts and programs which talk directly to the Chef API to perform any number of tasks around the data that the API exposes. We'll look more at interfacing with the Chef API in Chapter 7.

Chef Server

Web Server

The nginx web server is the first "port of call" when communicating with the Chef server. It allows everything which interfaces with Chef to communicate with it on a single port - Chef's API is entirely HTTP based, so this is port 443 as HTTPS is used by default. Depending on the route specified in the URL sent to the nginx server, it will proxy each request to the appropriate backend component. For example, requests to nginx containing `/bookshelf` will be passed straight through to the `Bookshelf` wheras other requests by default will be passed onto `Erchef`.

Web UI

The Chef Web UI gives you a graphical interface for working with Chef. The exact UI and features will vary depending on whether you're using open source Chef or Enterprise Chef, but all versions will allow you to perform tasks such as editing the run list of nodes, monitoring the status of your nodes and managing users. See "Chef Installation Types and Limitations" on page 18 for more details on the different features supported by each version.

Erchef

Erchef is the Core API component of Chef server. It got the name Erchef after the original Ruby version of Chef server was rewritten in Erlang in 2013 - the API interface remains unchanged between both versions however. Erchef is the component of Chef server which exposes the "system state" service we discussed in "So Why Chef?" on page 4 via it's HTTP based API. Whether you're creating a client or uploading a cookbook, your code is talking to Erchef.

Even though Erchef is written in Erlang, you do not need to use Erlang to interface with the Chef API. All of the examples and techniques in this book will be written in Ruby, talking to Chef servers which run Erlang behind the scenes.

Search Index

The Chef search index is powered by the open source Solr search platform. When you run chef search queries like `node_platform:centos`, the Chef API is actually sending a query to these search indexes. Under the hood, chef stores a number of different search indexes for nodes, environments, clients, roles and data bags. This is why we specify Chef searches like `knife search node +node_platform:centos`. What we're actually doing here is querying the `node` search index. Chef wraps Solr with a services called *chef-solr*, which provides a REST API for indexing and searching.

Message Queues

Chef uses *message queues* to send items to be added to the search indexes. Each item to be indexed is added to a *message queue* - these are powered by the open source RabbitMQ messaging queue. Queued items are pulled from the queue by a process called *chef-expander* which processes the items into the correct format for indexing. The processed items are then passed to chef-solr for indexing.

Database

The backend data storage repository behind Chef-server is powered by the open source PostgreSQL database server. This repository stores data such as saved node attributes, client configuration, roles, data_bags and environments.

Bookshelf
: The bookshelf is where Chef stores the files and content uploaded with a version of a specific cookbook. All content in the Bookshelf is stored alongside a checksum so that if different cookbooks (or two different versions of the same cookbook) upload the same file, it is only stored once in the bookshelf.

Cookbooks
: The cookbook content managed by the Bookshelf service is stored in *flat files* in a dedicated data repository. The file naming structure and format here isn't especially "human friendly", and is only used by the bookshelf service.

The "thin server, thick client" model of Chef that we looked at in "So Why Chef?" on page 4 mean that the vast majority of the time, your Chef customizations will be to one of the "Chef Client Tools" on page 60 simply because that is where most of Chef's logic lives. The Chef server can mainly be thought of as the "system state" service we learned about in "So Why Chef?" on page 4 - but just because we won't be directly customizing the server doesn't mean we won't be using it.

As we work through the material in this book, we'll need to communicate with the Chef server for a variety of different tasks - particularly when we come on to write knife plugins in Chapter 6 and interface with the Chef API in Chapter 7 - so a good understanding of the different components which make up a chef server will make working through the material in these chapters much easier.

Anatomy of a Chef Run

When we initiate a chef run on our nodes, we expect the eventual state of the node to be that which we described in our recipes. This concept is known as *convergence*.

Convergence
: To give the term its proper definition, in infrastructure automation *convergence* is the act of bringing a system closer to a *correct* state with each action taken. In Chef, *correctness* means bringing the node to the state defined by the recipes to be run on that node. In an ideal world, the initial Chef run performed on a node will result in full convergence with the node ending up in exactly the state defied by its recipes, and this is what you should aim for when writing cookbooks. In the event that this is not the case however, performing a second run will bring the node closer to the state defined in its recipes. Essentially, even if a node cannot become fully converged with one Chef run, subsequent Chef runs will always bring the node **closer** to the state defined in its recipes and not further from it.

Another key concept to remember when writing Chef cookbooks and performing Chef runs is that we also expect our recipes to be applied to the node with only required actions being performed - this is known as *idempotence*.

Idempotence

> *Idempotence* in the configuration management sense is the idea that no matter how many times you run a recipe or resource on a node, the outcome should not change from the first time the recipe or resource was executed. A good example of an idempotent recipe is one which uses the built in `package` resource to install a specific version of a package. Once the package has been installed, running the recipe over and over again will never result in the package being re-installed, as the resource will always detect that the specified version has already been installed. Although it is possible to write **non**-idempotent recipes in Chef, this is not recommended - idempotence is always something you should strive for, especially when creating your own resources, providers and definitions - we'll look at how to do this in Chapter 5.

Under the hood, a number of distinct steps take place behind the scenes to form a complete run and provide this convergent behavior. Much in the same way that modifying the engine of a car requires an understanding of how the different components function, when customizing Chef it's very important to understand the "anatomy" of chef runs and how the different stages interact.

The first stage of a Chef run is to execute *chef-client* or *chef-solo*, either manually or via a scheduled task. Once this has been done, the run will proceed through a number of different stages, illustrated in Figure 3-2 below:

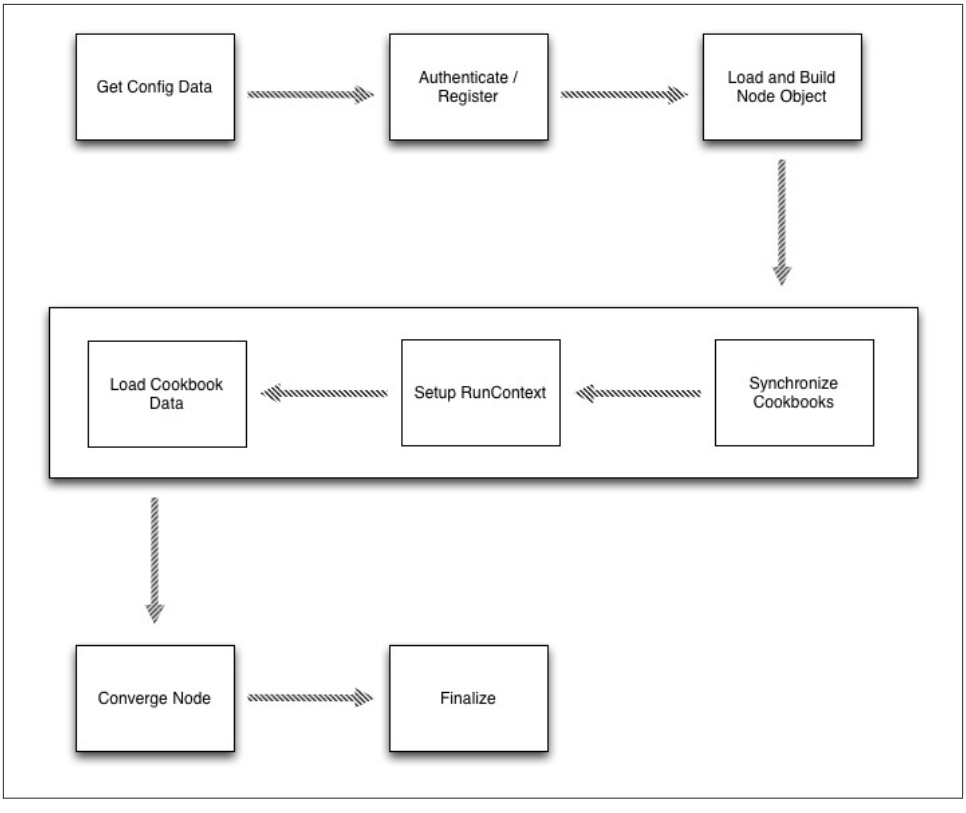

Figure 3-2. Chef Run Process

Some stages of the Chef run process differ depending on whether or not you're using chef-solo, or chef-client with a Chef server. These differences will be clearly indicated in tip boxes like this one.

Let's examine these stages in a little more detail:

Get Configuration Data

The first step performed once a Chef run has been initiated is for chef-client to load its configuration from the *client.rb* file. This file contains a number of configuration settings such as the URL of the chef server, the authentication credentials to use and where to store its local copy of cookbooks etc.

A full list of options supported by *client.rb* can be found on the Chef Documentation site.

Ohai

Once configuration has been loaded from the configuration file, the chef client runs an included tool called `ohai` to build up an initial collection of data about the node such as operating system, hardware platform etc. This data also includes the node's fully-qualified hostname, which will be used to ensure the correct data is loaded from the Chef server.

You can run `ohai` yourself from the command line to see the exact data it produces (in JSON format) and passes to chef-client - here's a sample of some of the output you might see:

```
$> ohai

{
  "languages": {
      "php": {
        "version": "5.4.20",
        "builddate": "Sep 27 2013"
      },
      "lua": {
        "version": "5.1.4"
      },
      "nodejs": {
        "version": "0.10.16"
      },
      "perl": {
        "version": "5.10.1",
        "archname": "x86_64-linux-thread-multi"
      }
    }

    <snip>
```

We'll look at how `ohai` collects attributes and how you can implement your own attributes in Chapter 6, and you can also find more `ohai` documentation on the Chef Documentation site.

When using chef-solo, the configuration file is called *solo.rb* and contains a number of solo-specific settings documented on the Chef Documents site.

Authenticate / Register

Once the node's hostname has been obtained, chef-client uses the authentication key and server URL specified in *client.rb* to attempt to authenticate to the chef server. If a node with that particular hostname doesn't exist on the server yet, the chef client attempts to register the node using a special *chef_validator* key also specified in *client.rb*. If the server does not verify that the server is authenticated or if registration fails, the run does not continue, and jumps straight to the "Finalize" on page 70 step.

When using chef-solo, the authentication / registration step is bypassed as there is no central Chef server to communicate with.

Load and Build Node Object

Chef-client will now download a *node* object representing the current node from the Chef server which contains the attributes saved during its last Chef run - if this is the first time the node has executed a run however, there will be no object to download. Chef combines this historical data with the `ohai` data obtained in the "Get Configuration Data" on page 64 step and any other attribute or run list changes to be applied on this run.

The node data downloaded from the server includes the run list that has been specified for this node. The chef-client then *expands* the run list to produce an ordered list of roles and recipes to be applied to the node. This is how Chef provides its guarantee that run lists will always be applied in the same order - each expanded copy of a run list will always be identically ordered, providing no run list changes have been made by the user.

When using chef-solo, the lack of a Chef server means that no saved node object data exists. During the *load* stage, chef-solo builds a new node object from the supplied node configuration data. Optionally, attributes can also be passed to chef-solo as a JSON file, using the `-j` option.

The next part of this step is the creation of a *run status* object. The *run status* object keeps track of the overall status of the Chef run, and contains a number of different attributes which are populated over the course of the chef-run depending on whether or not the run is successful. Amongst other things, it contains a reference to the *node* object created in this step, along with the *run context* object created in the "Setup Run-Context" on page 67 step, and if an exception occurs during the course of the chef-client

run this will also be stored. We'll look at the *run status* object in more detail in "Runstatus Object" on page 121

The final part of this step is to run any *start* handlers that have been defined. Handlers come in a variety of different forms we'll encounter throughout this chapter, and are designed to trigger certain actions in response to specific situations encountered by the Chef run. *Start* handlers run before the Chef run has fully begun and are typically used to initialize reporting systems etc which will be used throughout the rest of the run or to notify external systems about the start of a Chef run. We'll look at how to define and implement start handlers in detail in Chapter 4.

Synchronize Cookbooks

Once the expanded run list has been calculated by the "Load and Build Node Object" on page 66 step, chef-client asks the Chef server for a list of all cookbook files which will allow the client to complete all actions specified by the run list. Requesting only those files needed to complete the run allows chef-client to avoid maintaining a complete copy of all of your cookbooks on the each node. For example, templates and files not used within the requested recipes are not downloaded. Once the server provides the list, chef-client compares it against local copies of those files stored in its file cache. If any files are missing (as they would be on the first chef-client run) or different from those stored on the server, chef-client will download a copy of any new or changed files.

This *cookbook collection*, to use Chef's terminology, is eventually stored in the `RunCon text` object created in the next step, "Setup RunContext" on page 67.

When using chef-solo, this step is bypassed as a complete copy of all cookbooks is stored on each node - there is no central server to download files from.

Setup RunContext

The next stage in the run process is for Chef to create a *run context* object, which is used by Chef to track a variety of data about the state of the current Chef run. The data stored in the *run context* object is built up over the course of the remaining stages of the chef-client run, and by the time Chef actually executes recipe code on the node itself, the *run context* object will contain a complete list of cookbook files required for the run and an ordered list of all resources to be applied to the node.

Note that the *run context* object is distinct from the *run status* object created by in the "Load and Build Node Object" on page 66 - where the *run status* object contains data on the chef run as a whole, the *run context* object only stores data to be used by the "Converge Node" on page 70 step, as we'll see later on in this section.

We'll be interacting with the *run context* object more directly in Chapter 5, but I'll give a quick summary of the data it stores here. As previously mentioned, it's important to note that some of this data is populated during later stages of the Chef run - I've noted this where applicable.

Cookbook Collection
: The *cookbook_collection* contains the cookbook files needed to perform the Chef run. This is populated by the "Synchronize Cookbooks" on page 67 step above.

Definitions List
: The *definitions list* is a list of all resource *definitions* to be used during this run. Resource definitions are populated by the "Load Cookbook Data" on page 69 step, when the `/definitions` folder of each cookbook is loaded. We'll look at how to create definitions in Chapter 5.

Events
: Chef-client keeps track of a large number of events which occur during the course of the Chef run, such as when ohai has finished running, when a recipe file has been loaded or when the run has completed. These events are used in combination with an *event dispatcher* publish / subscribe system to provide the chef-client we're all used to seeing, but also to allow the customization the formatting and verbosity of chef-client's output or the implementation of own client-side data collection and reporting. We'll see some examples of how Chef adds events to this collection in "Tracing a Chef-client Run" on page 79. We'll look in detail at how to interact with the event dispatcher system in Chapter 4.

Delayed Notification Collection
: During the chef run, the *run context* object has to keep track of notifications triggered by resources, for example when a `file` resource notifies a `service` resource to restart. By default, notifications are processed at the end of the Chef run - these notifications are stored in the *delayed notifications* collection.

Immediate Notification Collection
: It is also possible to tell Chef that a notification must be processed immediately (by adding `:immediately` to your `notifies` statement). As the name suggests, these notifications are processed as soon as they are triggered, rather than at the end of the Chef run. The *immediate notification* collection is where the *run context* object keeps track of these.

Node
: This is the node object we created in the "Load and Build Node Object" on page 66 step, which represents the current state of the node being updated and contains all of the data fetched by ohai. Storing a reference to the *node* object in the *run context* object is useful for later stages of the run such as when we save new attributes to the Chef server.

Resource Collection
: The *resource collection* is where Chef stores its ordered list of all the resources to be applied to the node during the Chef run. This list is populated during the "Load Cookbook Data" on page 69 step.

Load Cookbook Data

Now that the chef-client has an up to date copy of all the cookbook files it needs, it loads all of the cookbook data into memory to build an up-to-date version of attributes to be used by the node, along with a *resource collection* containing all of the resources to be applied to the node. As with the previous step, the *resource collection* is stored inside the *run context* object for this run. Cookbook data is loaded in a very specific order during this step, shown in this diagram:

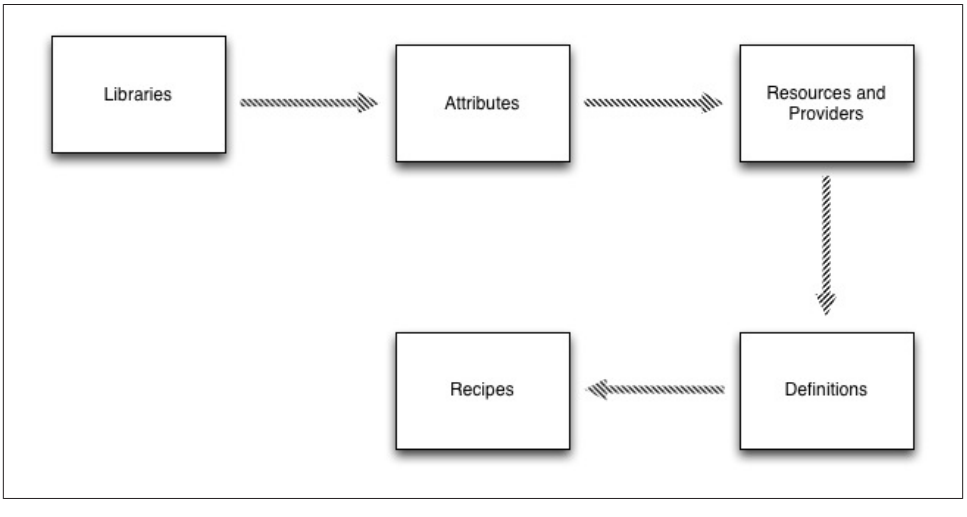

Figure 3-3. Cookbook Load Order

This strict load order exists to ensure that cookbook components are loaded prior to other components which might need to use them. Let's examine this ordering in a little more detail:

1. **Libraries**: Located in the `/libraries` directory of cookbooks, libraries are loaded before any other cookbook files as they typically extend Chef classes or define custom Ruby classes that you have created. These will usually be included in recipes, definitions etc so need to be loaded first. We'll look more at creating libraries in Chapter 5.

2. **Attributes**: Located in the `/attributes` directory of cookbooks, attributes files are loaded next. Attributes are loaded at this stage as they need to be accessible to recipes, definitions, resources and providers.
3. **Resources and Providers**: Located in the `/resources` and `/providers` directories of cookbooks, resources and providers are loaded next. These must be loaded prior to recipes as they declare new resource types which can be used in recipe code. We'll look more at creating resources and providers in Chapter 5.
4. **Definitions**: Located in the `/definitions` directory of cookbooks, definitions are loaded at this state as they declare new "pseudo-resources" which can be used within recipes, but may also in turn make use of resources and providers. We'll look more at creating definitions in Chapter 5.
5. **Recipes**: Located in the `/recipes` directory of cookbooks, recipe files are only loaded once all other cookbook components have been loaded. Recipes are loaded last as they can make use of all of the cookbook components listed in the previous load steps.

Converge Node

In this stage, the information collected throughout all of the previous stages is used to apply configurations to the node itself. Until now, the node itself has not actually been changed.

It's also during the converge step that the *run context* object created in "Setup RunContext" on page 67 really demonstrates it's usefulness. Chef-client (or chef-solo) now has access to a single object which contains all of the cookbook files it needs for the run, alongside an ordered list of all of the resources to be applied to the node. Chef-client (or chef-solo) steps through this resource collection, carrying out the actions requested by each resource such as installing software packages, writing template files, restarting services etc to bring the node to *convergence*.

If exceptions are raised by any resources during this stage for any reason, Chef will jump straight to the "Finalize" on page 70 step and execute any *exception handlers* that have been defined. If no exceptions are raised, it will carry on until all resources in the *resource collection* have been applied before continuing to the "Finalize" on page 70 step and executing any *report handlers* that have been defined

Finalize

During the finalize step, the behavior of chef-client will depend on the outcome of the "Converge Node" on page 70 step or whether an exception earlier in the run caused this stage to be started early, but both possible outcomes will result in the execution of *handlers*.

Put simply, handlers trigger certain actions in response to specific situations encountered by the Chef run - we'll cover creating and using handlers in more detail in Chapter 4, but for now let's look at the what happen when the "Converge Node" on page 70 step succeeds or fails and what type of handlers are executed:

Successful Run
: If the "Converge Node" on page 70 step completed without any errors, chef-client will save the new node object containing any run list changes and updated attributes back to the Chef server. It will also execute any *report handlers* which have been defined. Report handlers can be created to perform a wide variety of tasks such as sending metrics to a graphing system to produce a graph of successful chef runs and their durations.

Failed Run
: If an exception occurred during the "Converge Node" on page 70 step, chef-client will **not** save the updated node object back to the server, and will execute any *exception handlers* which have been defined. Exception handlers can be created to perform tasks such as broadcasting chef run failures to IRC channels or other chat systems.

When using Chef Solo, there is no central server to send updated node data to, but both types of handler are still executed.

At this stage, whether successful or failed, our Chef run is complete.

As we've seen in this section, although a Chef run is initiated with a single command and outputs a single unified view of run progress outputted to our screen, behind the scenes a Chef run is actually broken down into a number of discrete, modular steps. This modularity makes it much easier for us to identify possible customization types that might be of use to us than if we had to deal with a single monolithic step.

For example, if we want to customize the actions Chef performs on run success or failure, we now know that we need to focus on the behavior of the "Finalize" stage. Likewise if we want to customize the attributes collected by `ohai`, we know that we need to focus on the behavior of the "Get Config Data" stage. As we cover these different types of customizations in later chapters, this section will provide a handy reference point if you need a reminder of which exact stage of the Chef run we're working with, and how it might affect subsequent stages.

But wouldn't it be nice if we could verify exactly what actions a Chef run would perform on a node without actually having to perform a real Chef run?

Dry-run and Why-run

When writing infrastructure code (or for that matter any code) which involves as many complex steps and stages as Chef runs, it's extremely useful to be able to model what exactly will happen when the code is executed, without actually executing it. Many tools we use as Operations Engineers and Developers come with a "dry run" mode to provide this facility. For example, the *Rake* task management tool commonly used in the Ruby world allows you to specify `--dry-run` on the command line to print out the steps to be performed by the Rake task instead of executing them. The *Rsync* file transfer utility also provides `--dry-run` option which causes it to print the list of files to be transferred without actually transferring them.

The Problem with Dry-run

For tools such as *rake* and *rsync* which build up a list of commands and execute them, a dry-run mode is relatively easy to implement. With configuration management systems such as Chef however, things become a little more complex. At first glance, the reason for this might not be entirely obvious - after all, Chef builds up an ordered collection of resources and then executes them.

This is indeed the case, but each of those resources does not actually represent a single standalone task as, say, transferring a file with Rsync would. As we saw in the "Anatomy of a Chef Run" on page 62 section of this chapter, Chef resources are *convergent*, which means they bring the system closer to a *correct* state. For a resource to implement this convergent behavior, it actually requires two separate sets of instructions. The first set determine whether or not the object in question is already in the correct state, and the second defines the actions to take to *correct* the object if it is not already in a *converged* state.

Take the example of the `package` resource in Chef, used to install software packages on a wide variety of operating systems. This resource does not simply wrap `yum` or `apt` to run an "install" command every time Chef runs, but rather checks to see if the required package has already been installed at the right version, and then attempts to install the package if it is not found. Similarly, the "start" action of the `service` resource does not simply call the service management layer of the OS to start a service every time Chef runs, but rather checks to see if the required service is already running, only starting it if this is not the case.

To implement traditional dry-run behavior in Chef, it would be necessary for us to examine each convergent resource and have it describe the actions, if any, it would perform based on the current state of the system. However in configuration management systems such as Chef, the behavior of a resource can be affected by that of previously executed resources which potentially means the node state may change between each resource.

For example, let's use the `package` and `service` resources we looked at above. It might be the case that the package installed by the `package` resource adds the service subsequently used by the `service` resource. But without actually installing the package and observing the change to the node (which would defeat the point of dry-run mode entirely), this interdependency is impossible to reflect in dry-run mode. So how can we make dry-run functionality work in Chef?

Why-run mode

In order to mitigate these issues somewhat, Chef-client (and chef-solo) can be run in a mode called *why-run* (by using the `--why-run` option). Why-run mode provides the functionality we want from the *dry-run* model we looked at above, but makes a number of assumptions to allow it to work with the *convergent* model of Chef. When using *why-run* mode, it's extremely important to understand these assumptions as they can cause a run executed in *why-run* mode to behave differently than one executed in "normal" mode.

When running in why-run mode, chef-client (or chef-solo) makes the following assumptions:

Services
: If Chef in *why-run* mode is unable to find the command required to configure a service, for example if the service would normally be installed earlier by a `package` resource in the same run (like in the above example), it will **assume** that this service command has been installed by a previous resource, and that the service is not running.

  It's important to ensure that the required service management command (e.g. the init.d or upstart script) actually **is** installed by a previous resource - otherwise you may discover that a run in *why-run* mode will complete without error, while a "normal" run will fail when it tries to start a service which is not yet installed or defined.

Resource Conditionals
: When Chef in *why-run* mode encounters `not_if` or `only_if` conditionals declared in resources, it will **assume** that the conditional is a command or a ruby block which is safe to run in *why-run* mode - resource conditionals in Chef are intended to help make resources *idempotent*, and they should not normally alter the state of the underlying node.

  However, it is theoretically possible to define an `only_if` or `not_if` conditional which tests the output of running a `yum install` command, for example. In this case, *why-run* mode will assume that it was safe to run the conditional block which could potentially result in a run executed in *why-run* mode modifying the system. If you find yourself in this situation, I would strongly recommend rewriting the

relevant portion of your recipe - using conditionals in a non-idempotent fashion is very much an *anti-pattern* and something to be avoided.

Usefulness of Why-run

As we've seen, why-run mode makes some assumptions to allow it to approximate the functionality of *dry-run* in the convergent world of Chef runs, which can result in *why-run* potentially behaving differently than a "normal" chef run. So how useful is why-run, given we can't be absolutely sure that it's telling us exactly how our run will behave in reality?

Simply put, the usefulness of why-run increases the closer the node is to the *correct* state at the start of the Chef run. If you're running the first ever Chef run on a node in *why-run* mode, your chef run will probably find that a large number of resources are not in the "correct" state, hence will be making a large number of changes to the underlying system to *converge* it. The greater the number of resources requiring corrective action to bring them to *convergence*, the greater the chance for the assumptions made under *why-run* mode to show different results than a "normal" run. On the other hand, if your *why-run* run is checking what will happen when you execute your changes to a single recipe, it's much more likely that the results will be mirrored by those seen in a "normal" Chef run.

Helpfully, *why-run* mode specifically informs you of the assumptions it has made during a run to help you identify potential issues which may crop up when performing a "normal" run. For example, consider the following `service` resource:

```
service "chef-client" do
    action :enable
end
```

When this resource is reached by a Chef run in *why-run* mode, we will see the following output:

```
 * service[chef-client] action enable
    * Service status not available. Assuming a prior action would have
        installed the service.
    * Assuming status of not running. (up to date)
```

Providing you understand the assumptions being made, *why-run* mode can be an extremely helpful development tool when writing Chef recipes, particularly when writing your own providers and resources as we'll be doing in Chapter 5. As with many Operations and Development tools, why-run's usefulness depends on which functionality you expect from it versus what it was designed to provide.

If you expect why-run to provide you a 100% accurate list of every single step performed by a Chef run which will be mirrored exactly by a normal run, you may not find it especially useful - it is not designed to provide a replacement for properly validating your cookbooks in a test environment using the tools we looked at in "Development

Tooling" on page 13, and will not behave in the same way. On the other hand, if you treat why-run as an addition to your cookbook development toolbox which will help you check the probable behavior of your cookbooks, the combination of why-run's explicitly stated assumptions with your own intuition and skill as a cookbook developer can make it an extremely useful tool. When we come on to develop our own recipe resources, providers and libraries in Chapter 5, we'll be making extensive use of why-run to help us identify whether or not our recipe customizations are behaving correctly.

For a more detailed analysis of dry-run, why-run and the difficulties which arise when applying these concepts to configuration management systems such as Chef, I highly recommend taking a look at Sean O'Meara's excellent article on the subject over at his blog A Fistful of Servers.

Using the Source

When we want to dive even deeper into the internals of Chef, an extremely powerful tool that we have available to us is the Chef source code itself. Although the code for the **server** side of Chef varies between the open source and enterprise variants, all versions of Chef use the same open-sourced **client** code which is freely available on GitHub. This repository contains code for chef-client, chef-solo, knife and all of the other client tools we looked at in "Chef Client Tools" on page 60. Chef, Inc. actively encourage community contributions to this code, and any new Chef release will usually combine a mix of features and patches added by Chef Inc themselves, with features and patches added by the Chef community.

The Chef Client codebase is an invaluable reference guide when analyzing exactly how Chef implements particular functionality and also allows us to benefit from the experience and expertise of the developers who work on Chef day in, day out. Often when implementing customizations in Chef, a good starting point can be found in looking at how Chef Inc have implemented similar functionality.

In this section, we'll start with how to get a copy of the Chef source code and find your way around the repository, along with where to find the code which controls some of the Chef behavior we've seen already. We'll then combine the Ruby knowledge we gained in Chapter 2 with the Chef internals knowledge gained so far in this chapter to trace the execution of an actual Chef run through the Chef codebase.

We're taking a whistle-stop tour around the Chef codebase here, aimed to familiarize you with how Chef's code is organized and the classes that make up an actual Chef run. Don't worry if you aren't able to fully understand the code in some of Chef's class definitions - in parts of the book where a more comprehensive understanding is required as we extend or inherit from Chef classes, we'll cover the code in much greater detail.

Getting the Chef Source Code

You may find it useful to download your own copy of the Chef source code from Github when working through this chapter, and the rest of the material in the book. To download a copy of the Chef repository, run the following command (which requires `git` to be installed):

```
$> git clone https://github.com/opscode/chef.git
```

This will give us a copy of the Chef codebase in a directory called `chef` under the directory where we ran the command.

This command will give us a copy of the *master* branch of the Chef repository. It's important to note that the *master* branch may contain code not yet released to the public, so if you're writing code which inherits from Chef code, please check to ensure that code is deployed in the latest stable release.

If you want to switch your local copy of chef to the latest stable release (11.8.2 at time of writing) you can use the following command, run from the directory produced above:

```
$> git checkout --track -b 11-stable origin/11-stable
```

Chef Source Structure

Let's take a moment to look at how the Chef source code repository is structured, to give you the lay of the land so to speak. Please note, the directory structure shown here is somewhat abbreviated to remove directories which aren't of immediate interest to us here - it's worth exploring the full directory structure at your leisure, but for now we'll focus on the areas we'll be exploring throughout this book.

Where directories in the below repository directory listing have been removed for brevity, this is indicated by ...

```
.
├── bin ❶
```

```
├── ...
├── lib
│   └── chef ❷
│       ├── ...
│       ├── application ❸
│       ├── ...
│       ├── cookbook ❹
│       ├── ...
│       ├── event_dispatch ❺
│       ├── ...
│       ├── formatters ❻
│       │   └── error_inspectors
│       ├── ...
│       ├── knife ❼
│       │   ├── bootstrap
│       │   └── core
│       ├── ...
│       ├── provider ❽
│       │   ├── cookbook_file
│       │   ├── cron
│       │   ├── deploy
│       │   ├── env
│       │   ├── file
│       │   ├── group
│       │   ├── ifconfig
│       │   ├── mount
│       │   ├── package
│       │   ├── remote_file
│       │   ├── service
│       │   ├── template
│       │   └── user
│       ├── resource ❾
│       └── ...
└── ...
```

❶ The `bin` directory contains the executable scripts for Chef client tools such as chef-client, chef-solo and knife.

❷ The `lib/chef` directory is where the majority of the classes used by the Chef client tools are stored. This directory contains a number of sub-directories covered in more detail below, but also contains a number of class definitions for core Chef objects such as cookbooks, nodes and roles. You can typically find these definitions in files called `<object>`.rb, ie `node.rb` or `cookbook.rb`

❸ The `lib/chef/application` directory contains a number of class definitions for objects which are used by the Chef client tools. We'll look in more detail at some of these classes in "Tracing a Chef-client Run" on page 79.

❹ The `lib/chef/cookbook` directory contains a number of class definitions for objects associated with cookbooks. This directory contains classes such as `synchronizer.rb` (used by the "Synchronize Cookbooks" on page 67 step) and `cookbook_collection.rb` (the object which contains the Cookbook Collection produced by the "Synchronize Cookbooks" on page 67 step).

❺ The `lib/chef/event_dispatch` directory contains the classes used for the *event dispatcher* we touched on briefly in the "Setup RunContext" on page 67 step which handles events which occur during Chef runs - we'll look at the event dispatcher in much more detail, including how to extend and interact with it, in Chapter 4.

❻ The `lib/chef/formatters` directory contains the formatter classes used to display the output of chef-client and chef-solo to the user when run in an interactive terminal - these make use of the *event dispatcher* system we touched on in "Setup RunContext" on page 67. Inside this directory, `doc.rb` is the default formatter used by chef-client - if you have a peek inside this file you'll probably recognize a number of the output messages it contains. We'll look at formatters and the overall event dispatcher system in more detail in chapter 4.

❼ The `lib/chef/knife` directory contains class definitions for all of the default knife commands provided with Chef, along with classes used to power knife itself. Class files for knife commands are named similarly to the name of the knife command which makes use of them - the code for `knife node list` for example, is contained in `node_list.rb`. We'll be using and extending some of these classes in Chapter 6 when we look at creating our own knife plugins.

❽ The `lib/chef/provider` directory contains the class definitions for providers, which contain the code to carry out the actions defined in cookbook resources. Provider class definitions are named according to the resource they support, so for example you'll find the class for the `execute` provider inside `execute.rb`. Some more complex providers such as `package` declare several classes inside their own directory named `package`, as `package` providers have to be implemented for a variety of packaging systems. We'll explore providers in greater detail in Chapter 5.

❾ The `lib/chef/resource` directory contains class definitions for all of the cookbook resources that come with chef out of the box - you've likely already used many of them in your recipes. Resource class definitions are named according to the resource they define, so for example you'll find the class for the `execute` resource inside `execute.rb`. We'll be creating our own resources in Chapter 5.

Tracing a Chef-client Run

Now that we've leveled up our Ruby skills in Chapter 2 and explored the structure of Chef's source code above, let's put this knowledge to use and dive head first into a real example. When the `chef-client` command runs, what happens behind the scenes to actually drive the process we've looked at?

We're going to trace the execution of chef-client command down through the Chef codebase to help demonstrate that the concepts we learned in Chapter 2 such as inheritance, dynamic typing and modules work in practice as well as in theory - Chef contains some complex Ruby code, but the basic principles are no different to those we covered in Chapter 2.

A number of the methods utilized by the classes in this example contain fairly complex Ruby code. To avoid making this example too overwhelming, I've summarized the behavior of these methods in plain english or listed excerpts of the code. For the interested reader or more confident Ruby programmers, I'll link to the source code of the files involved if you'd like to look at the full code.

Execute the "chef-client" Command

The first step in tracing the execution of the chef-client command is, of course, to run it. When the command `chef-client` is run, your operating system scans a list of directories to search for an executable program by that name. This directory list is stored in the `PATH` environment variable, which is used by your operating system in much the same way that Ruby uses the `LOAD_PATH` variable to locate Ruby files.

By default, executable scripts used by Rubygems are not installed under directories listed under `PATH` so when Chef is installed, the omnibus installer (or Rubygems if you did a manual install) drops a `chef-client` script into a directory listed in `PATH`, typically `/usr/bin/` on Linux systems - I've used that path for the script in this section, but it will vary for users of Windows and other operating systems.

This `chef-client` script is just actually a wrapper file which locates and executes the **real** chef-client script under the `bin` directory of the `chef` gem installed under the Omnibus install directory. Inside this chef gem, code is structured exactly as we saw in "Chef Source Structure" on page 76.

The first line of the wrapper script specifies that the code should be run using the Ruby binary located at `/opt/chef/embedded/bin/ruby` - if you're running on a non Linux / Unix based system or installed chef from Rubygems, this line will contain a different directory path.

The first line of the wrapper script specifies that the code should be run using the Ruby binary located at `/opt/chef/embedded/bin/ruby` - if you're running on a non Linux / Unix based system or installed chef from Rubygems, you will need to alter this to the correct path to your ruby binary.

Inside this wrapper script, we find the following Ruby code:

Example 3-1. /usr/bin/chef-client

```
#!/opt/chef/embedded/bin/ruby

# comments added for clarity

# require the 'rubygems' gem, so we can use its classes and methods
require 'rubygems'

# Specify what version of the chef gem we require (>=0 means "any version")
version = ">= 0"

<snip> # Code to handle running a specific version of a command

 # Check the chef gem is installed at the right version using
# the "gem" method provided by the "rubygems" gem
gem 'chef', version ❶

# Find the "bin_path" of the installed 'chef' gem,
# locate the binary called 'chef-client' and load it
load Gem.bin_path('chef', 'chef-client', version) ❷
```

❶ This line verifies that the required version of the `chef` gem is actually installed. If a version of the *chef* gem is not installed which matches the version requirement *'>= 0'* (this means "any version is installed") then an exception will be thrown and execution will stop. There's no point in trying to continue when the gem we need isn't present!

❷ We then locate the "bin_path" of the Chef gem (where its executable scripts are located), find the exact path to the "chef-client" gem and then run it using the built-in `load` keyword. The `load` keyword (defined under the `Kernel` module we looked at in Chapter 2) simply takes a string containing the path to the file to be executed, and then loads and runs that file.

Run the Real "chef-client" Script

Now that the location of the real `chef-client` script has been located and executed by the `load` statement in the previous example, we need to take a look at that script. Although the above wrapper script did not print out the actual location of the real `chef-client` script, you can check the location of the script for yourself by pasting the following Ruby code into a file called `chef_client_path.rb` and running it:

Example 3-2. chef_client_path.rb

```
#!/opt/chef/embedded/bin/ruby
puts Gem.bin_path('chef', 'chef-client')
```

When you run `chef_client_path.rb`, you should output similar to the following, with the full path to the real `chef-client` script being printed:

```
$> ./chef_client_path.rb
/opt/chef/embedded/lib/ruby/gems/1.9.1/gems/chef-11.8.2/bin/chef-client
```

We've identified and located the script that powers the next stage in our Chef run, so let's have a look at what this script actually does. Like the wrapper script we saw above, the actual `chef-client` script is also extremely short - in fact it's only 4 lines of code. nonetheless, it demonstrates a number of the concepts we looked at in Chapter 2:

Example 3-3. /usr/lib64/ruby/gems/1.9.1/gems/chef-11.8.2/bin/chef-client

```
require 'rubygems'❶
$:.unshift(File.join(File.dirname(__FILE__), "..", "lib"))❷
require 'chef'❸
require 'chef/application/client'❹

Chef::Application::Client.new.run❺
```

❶ Here we're requiring the `rubygems` gem, like we looked at in "Rubygems" on page 43, so that the methods and classes it provides are available to our ruby program.

❷ The `$:` syntax used here is a shortcut to the `LOAD_PATH` variable we looked at in "Local Classes" on page 42. What we're actually doing here is adding the `lib` directory located one directory level up from the knife script to our `LOAD_PATH`. Since `LOAD_PATH` is an array, here we're doing that with the Array class's `.unshift` method which adds an element to the start of an array.

❸ ❹ Now that our `LOAD_PATH` is correct, we're requiring a gem called `chef` and a class called `chef/application/client`, just like we saw in "Rubygems" on page 43

❺ Finally we're creating a new instance of the `Chef::Application::Client` class (which we can now use thanks to the previous `require` line) and calling it's `run` method.

The Chef::Application::Client Class

The class definition for the `Chef::Application::Client` object created in the previous step can be found, as its name might suggest, in `lib/chef/application/client.rb` in the Chef repository (Full Code on Github). If we look inside this class, we'll see it's defined as follows:

Example 3-4. Excerpt of lib/chef/application/client.rb

```
class Chef::Application::Client < Chef::Application
 # Lots of code
end
```

Much in the same way as we examined in "Inheritance" on page 36, the `Chef::Application::Client` class inherits from the *superclass* `Chef::Application`, located at `lib/chef/application.rb` in the Chef repository (Full Code on Github). The `Chef::Application` class is shared between all of the Chef client tools including *chef-client* and *knife*, and declares a number of methods common to all these tools.

The `run` method we call above in the last line of the `chef-client` script is actually defined by the `Chef::Application` class, but because `Chef::Application::Client` is a *subclass* of this, it has that method available to it as well. Let's look at what's inside this `run` method in `Chef::Application`:

Example 3-5. Excerpt of lib/chef/application.rb

```
class Chef::Application
  ...
  def run
    reconfigure
    setup_application
    run_application
  end
  ...
end
```

The `run` method is also extremely short and straightforward, it calls three methods one after the other - but this is where some cleverness starts to creep in. Although there are method definitions for all three of these methods inside `Chef::Application`, they are *overriden* by `Chef::Application::Client` to provide logic specific to Chef client.

Overriding a method in Ruby means that a *subclass* can essentially change the implementation of a method which was already defined by the *superclass*. This demonstrates some of the power of object oriented programming - without the *subclass* and *superclass* concept, all of the chef client tools would have to each contain separate copies of common methods. With OO languages like Ruby, the *superclass* is able to define methods common to all *subclasses*, while still allowing the *subclasses* to *override* methods implementing specific behavior.

To demonstrate overriding in action, if we look at the definition of the `run_application` method in `Chef::Application`, we see:

```
class Chef::Application

  # other code removed for clarity, comments added

  def run_application
    # Raise an exception printing the name of the class which has called this method
    # and tell it to override.
    raise Chef::Exceptions::Application, "#{self.to_s}: you must
      override run_application"
  end

end
```

but if we look in `Chef::Application::Client`, we see

```
class Chef::Application::Client < Chef::Application

  # other code removed for clarity

  def run_application
    # Lots of code
  end

end
```

Note here that the method names are identical. When an object of the `Chef::Application::Client` class is created, it will contain all the methods defined by `Chef::Application` and **also** any methods it defines itself, including those which override methods from `Chef::Application`. This somewhat subtle behavior allows all subclasses of `Chef::Application` to share a common `run` method while implementing their own versions of the `setup_application`, `reconfigure` and `run_application` methods.

Now let's take a closer look at the three methods of `Chef::Application::Client` being run by the `run` method - to keep things simple, I'll summarize the behavior of each

method in plaintext rather than listing the full code, but interested readers can explore the actual code on Github or in their local copies of the repository.

reconfigure
: The `reconfigure` method carries out two steps. First, it validates and parses the command line options given to chef-client. Next, it loads the configuration contained in the *client.rb* configuration file and stores it in a `Chef::Config` object for use by later steps in the run process. As a side exercise for the interested reader, why not see if you can locate the class definition for *Chef::Config* in the Chef repository and explore its methods.

setup_application
: The `setup_application` method sets the *user* and *group* who own the chef-client process if one has been specified in the configuration loaded from *client.rb*. This is particularly important when chef-client is running in *daemonized* mode as typically in this case we want it to always run under a particular user or group.

run_application
: The `run_application` method is one we're most interested in here, as it's the method which carries out the next step in executing the Chef run. This method contains logic to either run chef-client in *daemonized* mode, or run it in a continuous loop if the `--once` option was not passed to chef-client. Incidentally, it's here that the *splay* and *interval* options that we can specify in `client.rb` come into play - when running in a loop, this method ensures that chef-client will only initiate a run every *interval* seconds, with an added delay of up to *splay* seconds at the start of the run. If we look at an excerpt of the `run_application` method, we see the following code (comments added for clarity):

Example 3-6. Excerpt of `run_application` method from Chef::Application::Client

```
# If a splay has been specified
if Chef::Config[:splay]
  # Pick a random number between 0 and 'splay'
  splay = rand Chef::Config[:splay]

  # Log how long 'splay' will be
  Chef::Log.debug("Splay sleep #{splay} seconds")

  # Sleep for 'splay' seconds
  sleep splay
end

# Run 'run_chef_client' method
run_chef_client
```

As we can see here, after program execution has paused (using the built-in Ruby `sleep` method) for *splay* seconds, we're now calling a method called `run_chef_client`.

Chef::Application::Client - run_chef_client Method

The `run_chef_client` method is defined in the *superclass* `Chef::Application`, as its implementation is common to any Chef client tool such as chef-client or chef-solo which is going to initiate a Chef client run. This method is the last "preparatory" step before the actual Chef run is intiated, and it contains the following code:

Example 3-7. `run_chef_client` method from Chef::Application

```
# Initializes Chef::Client instance and runs it
def run_chef_client
  Chef::Application.setup_server_connectivity❶

  @chef_client = Chef::Client.new(
    @chef_client_json,
    :override_runlist => config[:override_runlist]
  )❷
  @chef_client_json = nil ❸

  @chef_client.run❹
  @chef_client = nil❺

  Chef::Application.destroy_server_connectivity❻
end
```

❶ The `start_server_connectivity` method of the `Chef::Application` class starts up the chef-zero server we saw in "Local Mode" on page 20 if the `-z` option was passed to chef-client to run it in *local mode*. This step also updates configuration parameters such as *chef_server_url* to point to Chef-zero instead of the server specified in `client.rb`. If the `-z` parameter was not passed to chef-client, this step does nothing and chef-client will connect to the server specified by the `chef_server_url` configuration option as normal.

❷ Next we create a new instance of the `Chef::Client` class, passing as parameters the JSON representation of the configuration loaded from `config.rb`, and an optional overridden run list passed to chef-client with the `--override-runlist` option. We'll look at the `Chef::Client` class in more detail in "The Chef::Client Class" on page 86.

❸ Now that we've created our `Chef::Client` object, we can delete the JSON version of the loaded configuration since we don't need it any more.

❹ Next, we call the `run` method of the `Chef::Client` object we created above to actually carry out the Chef run. We'll look at this step in more detail in the next part of this section

❺ Once the chef run has completed, we delete the `Chef::Client` object we created.

❻ If we were using chef-zero to run chef-client in *local mode*, kill the chef-zero instance we created in step 1

The Chef::Client Class

The `Chef::Client` class is the final crucial component needed to carry out a Chef run, and is located in `/lib/chef/client.rb` in the Chef repository (Full Code on Github). This class handles all of the run stages shown in the Figure 3-2 diagram aside from the "Get Config Data" step which, as we saw in "The Chef::Application::Client Class" on page 82, was handled by the `reconfigure` method of the `Chef::Application::Client` class.

When this object is created by the `run_client` method we looked at above, the initialize method (shown below) creates a number of the objects that will be used during the Chef run, including placeholder objects for the node, run status and events collection.

The `run` method of the `Chef::Client` class which we called above is a "wrapper" method for the actual method which carries out a chef run. Before going ahead with the run, the `run` method contains code to handle forking the run itself off as a new process if the `--fork` option was passed to chef-client. The code also makes sure that if chef-client is running on Windows (which does not support process forking), we don't try to fork the process.

Let's have a look at the code:

Example 3-8. Excerpt of lib/chef/client.rb

```
class Chef
  class Client

    # other methods etc removed for simplicity

    # Creates a new Chef::Client.
    def initialize(json_attribs=nil, args={})

        # Create objects needed during chef run
        @json_attribs = json_attribs
        @node = nil
        @run_status = nil
        @runner = nil
        @ohai = Ohai::System.new

        # Load event handler and formatter classes
```

```
    event_handlers = configure_formatters
    event_handlers += Array(Chef::Config[:event_handlers])

    # Create new events collection
    @events = EventDispatch::Dispatcher.new(*event_handlers)

    # Load any overridden runlist passed in as a parameter
    @override_runlist = args.delete(:override_runlist)
    runlist_override_sanity_check!
end

def run

  # If the chef-client run should be forked...
  if(Chef::Config[:client_fork] && Process.respond_to?(:fork) &&
    !Chef::Platform.windows?)

    Chef::Log.info "Forking chef instance to converge..."
    # Start a "fork" block
    pid = fork do

      # Removed fork configuration code for simplicity

      # Code is wrapped in a begin / rescue block to trap errors
      begin
        # Call the 'do_run' method
        do_run
      rescue Exception => e
        Chef::Log.error(e.to_s)
        # Terminate the program with the status code "1"
        # which indicates that an error occurred
        exit 1
      else
        # Terminate the program with the status code "0"
        # which indicates that the program finished successfully
        exit 0
      end
    end

    # Removed code to handle cleaning up after forked process has finished
    # for simplicity

  # If forking is not needed, just call 'do_run' straight away
  else
  # Call the 'do_run' method
    do_run
  end
end
```

As we see in the above example, regardless of whether the chef run is forked or not, both branches of the code end up calling the `do_run` method, which is also defined in

`Chef::Application::Client`. It is this method which actually carries out the steps we looked at in "Anatomy of a Chef Run" on page 62.

Chef::Client class - do_run Method

After working our way through a number class definitions and methods in the Chef codebase, we've finally reached our destination - the method which actually carries out the majority of the steps of our chef-client run we looked at in "Anatomy of a Chef Run" on page 62.

Unlike many of the examples so far in this section, I've listed the code of the `do_run` method in full, but have not listed the code for the methods it calls. We'll look in detail at the methods called by `do_run` after the code listing and I've added descriptive comments throughout the method body, but for additional clarity I've also included an expanded version of the chef run stage diagram we saw earlier in the chapter with each method call aligned with the run stage it represents.

If you're interested in diving even deeper into the Chef source code, the full code for each method called by `do_run` can be found in `/lib/chef/client.rb` in the Chef repository (Full Code on Github).

Example 3-9. do_run method of Chef::Client class

```
def do_run

  # Create a lock so we can't perform multiple Chef
  # runs at once
  runlock = RunLock.new(Chef::Config.lockfile)
  runlock.acquire

  # don't add code that may fail before entering this section to be sure
  # to release lock

  begin

    #Save the process ID of the chef-client process into our run lock
    runlock.save_pid

    #The variable we will use to store our RunContext Object
    run_context = nil

    # Add an event to our Events collection to indicate run_start
    @events.run_start(Chef::VERSION)

    # Log that the run has started, and it's PID
    Chef::Log.info("*** Chef #{Chef::VERSION} ***")
```

```
Chef::Log.info "Chef-client pid: #{Process.pid}"

# Call enforce_path_sanity method which ensures
# PATH variable is set correctly for chef-client
enforce_path_sanity

# Call run_ohai method
run_ohai❶

# Add an event to our Events collection to indicate ohai_completed
@events.ohai_completed(node)

# Call register method unless running chef-solo
# Registers  / authenticates chef-client with the server
register unless Chef::Config[:solo]❷

# Call load_node method
# Loads existing node object from the server
load_node❸

# Calls build_node method
# Build node object from ohai data
build_node❹

# Call the start_clock method of the run_status object.
# This is a Chef::RunStatus object created by the build_node method
run_status.start_clock
Chef::Log.info("Starting Chef Run for #{node.name}")

# Runs any start handlers which have been defined
# and adds an event to the Events collection to indicate run started
run_started❺

# Call the do_windows_admin_check method
# If we're running on Windows, this checks chef is running as
# an administrative user
do_windows_admin_check

# Call the setup_run_context object and assign it to
# a variable called run_context
run_context = setup_run_context❻

# Call the converge method, passing run_context variable
# as a parameter
converge(run_context)❼

# Call save_updated_note method which
# saves node changes to the server
save_updated_node❽

# Call the stop_clock method of the run_status object
run_status.stop_clock
```

```
    Chef::Log.info("Chef Run complete in #{run_status.elapsed_time} seconds")
    # Call the run_completed method which
    # runs any report handlers which have been defined
    run_completed_successfully❾

    # Add an event to our Events collection to indicate run_completed
    @events.run_completed(node)

    # We're all done! Return "true"
    true

  # Code to handle any exceptions which occur during the run
  # Here we're catching *all* exceptions.
  rescue Exception => e
    # Log error message before doing anything else
    Chef::Log.debug("Re-raising exception: #{e.class} - \
      #{e.message}\n#{e.backtrace.join("\n  ")}")
    # Check if we failed after creating the run_status object
    # If we didn't, our error handlers won't be of much use.
    if run_status
      run_status.stop_clock
      # Store the exception in the run_status object
      run_status.exception = e
      # Call the run_failed method which
      # runs any exception handlers that have been defined
      run_failed❿
    end
    Chef::Application.debug_stacktrace(e)
    # Add an event to our Events collection to indicate run_failed
    @events.run_failed(e)
    # Now that we've run our error handlers, re-raise the exception
    # And let ruby stop program execution as normal.
    raise
  # Code to clean up after exception handling:
  # delete all of our objects, release the runlock and
  # start garbage collection to clean up memory
  ensure
    @run_status = nil
    run_context = nil
    runlock.release
    GC.start
  end
  true
end
```

Before diving in to what each method does, let's take a moment to remind ourselves of the different stages of a Chef run and match the above method calls with each step - Here's the same diagram we saw earlier in the chapter, expanded to show which stage of the run each method call relates to:

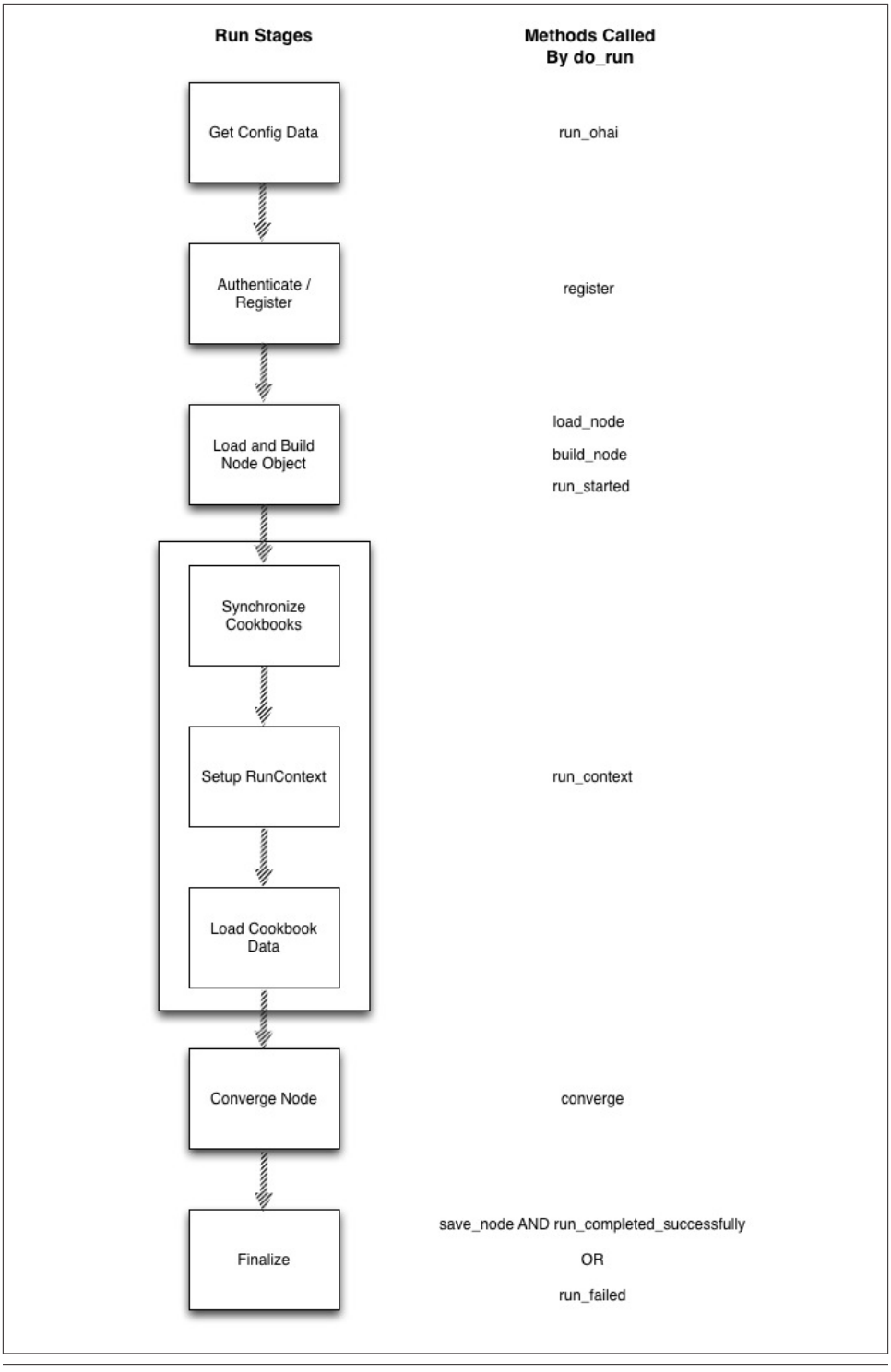

Figure 3-4. Chef Run Process With Method Calls

Now let's take a closer look at each method call and examine exactly what it does - I've included a reference to the explanation of each run stage we covered earlier in the chapter:

❶ **run_ohai** method - This runs the included `ohai` tool to gather updated information about the nodes for the "Get Configuration Data" on page 64 step.

❷ **register** method - This registers the node with the configured Chef server, or authenticates it if it already exists. This drives the "Authenticate / Register" on page 66 step.

❸ **load_node** method - This loads the existing `Chef::Node` object (defined in `/lib/chef/node.rb`) from the Chef server. If running in chef-solo mode, this method creates a new blank Node object. This method forms the first part of the "Load and Build Node Object" on page 66 step

❹ **build_node** method - This takes the `Chef::Node` object loaded in step **3** and adds in the ohai data gathered in step **1** together with any additional attributes or run list overrides before finally expanding the run list into a list of roles and recipes to apply to the node. It also also creates the `Chef::RunStatus` object (defined in `/lib/chef/run_status.rb`) which will be used to keep track of the run status throughout the rest of the run. This forms the second part of the "Load and Build Node Object" on page 66 step.

❺ **run_started** method - This executes any *start handlers* which have been defined and forms the final part of the "Load and Build Node Object" on page 66 step.

❻ **run_context** method - This performs the three steps from the Figure 3-2 diagram which are enclosed by a separate box. First, cookbook data is synchronized with the server (the "Synchronize Cookbooks" on page 67 step). Secondly, a `Chef::RunContext` object (defined in `/lib/chef/run_context.rb`) is created and initialized (the "Setup RunContext" on page 67 step). Lastly, the "Load Cookbook Data" on page 69 step is performed which loads the data from each cookbook in the *cookbook collection* into memory, and stores it in the run context object as the *resource collection* we looked at in "Setup RunContext" on page 67.

❼ **converge** method - This single method call triggers the entire "Converge Node" on page 70 step which applies the *resource collection* to the node. This is the first point during the entire run process which actually changes the underlying node to bring it to *convergence*.

❽ **save_node** method - Providing that the run was successful, this method saves the updated node object back to the server. This forms the first part of the "Successful Run" section under the "Finalize" on page 70 step.

❾ **run_completed_successfully** method - Providing that the run was successful, this method executes any *report handlers* that have been defined. This forms the last part of the "Successful Run" section under the "Finalize" on page 70 step.

❿ **run_failed** method - If the run failed, this method will execute any exception handlers that have been defined. This forms the last part of the "Failed Run" section under the "Finalize" on page 70 step.

In this section, we've taken our existing knowledge on the different stages of a chef run from "Anatomy of a Chef Run" on page 62 and augmented it by looking under the hood at the actual code that makes the entire process happen. The Chef source code is not "magic" in any way, nor is it something to be afraid of - although it contains some complex code in places, it is still fundamentally a collection of Modules, Classes and Methods like the examples we looked at in Chapter 2. Every single operation you carry out with Chef's client tools from knife commands to converging cookbooks can be traced through the code as we did here with Chef runs.

Throughout the rest of the book, we'll be building on this introduction to some of Chef's classes and methods to develop a wide variety of customizations - my hope is that the foundations I've laid in this chapter will enable you to gain a fuller understanding of where and how these customizations fit into your Chef setup.

Summary

Over the course of this chapter we've dived deep into the internals of Chef from an initial top-level overview of the components which make up Chef's sever and client tools, through the anatomy of how exactly a Chef run is carried out, down through looking at the Chef codebase itself to examine some of the actual classes and methods that power this behavior. I strongly believe that an in-depth understanding of how Chef functions under the hood is an invaluable tool when looking to customize and extend its behavior and it's my hope that this chapter has assisted you in that regard.

Before moving on to the rest of the chapters in this book, if a lot of the material we've covered so far was new to you I'd encourage you to skim through chapters 2 and 3 one more time. A good understanding of the Object-Oriented programming with Ruby and the internals of Chef is essential to make the most of Chef's incredible flexibility and extensibility, so it's worth taking a little extra time to let everything sink in.

With that out of the way, let's get customizing! Most of the remainder of the book is broken down into chapters focussing on customizing specific aspects of Chef and although it's my hope that you'll read through all of "Customizing Chef", Chapters 4 through 9 are written so that they may be treated as self contained units if you choose to skip ahead.

PART II
Customizing Chef Runs

Now that we've looked at an overview of the necessary Ruby concepts for customizing Chef and examined in detail how Chef runs works under the hood, we're ready to start looking at how to create our own customizations. The upcoming chapters will focus in detail on how to customize different aspects of Chef, and the tasks to which each type of customization is best suited.

In Part 2, we'll focus specifically on customizing various components of the Chef run process that we looked at in Chapter 3, and the Chef classes we'll be making use of along the way. We'll learn:

- In Chapter 4, we'll learn how to write plugins for `ohai` that let you store your own attributes in your Chef nodes.
- In Chapter 5, we'll learn how to implement our own customizations to leverage the *run status* information that Chef exposes to create our own start, report and exception handlers.
- In Chapter 6, we'll learn how to make use of the event collection that Chef builds up during the course of a run to create our own customized views into the *progress* of Chef runs through the use of the Chef event dispatcher.

We'll then finish off by using the concepts learned throughout these chapters to revisit one of the problems AwesomeInc has been experiencing that we explored in "Criteria for Customization" on page 9.

CHAPTER 4
Extending Ohai

As we saw in Chapter 3, during the "Get Configuration Data" on page 64 stage Chef runs the ohai tool to build up a collection of data about the node which is saved on the Chef server as part of the node object. Ohai is installed as part of the chef installation process and must be present on a node for chef-client or chef-solo to function correctly.

The core ohai tool is driven by its flexible and powerful plugin interface. Ohai ships with a number of useful plugins which collect data on various aspects of the underlying system such as hardware, operating system and networking configuration but it also provides us with the capability to define **new** plugins which capture information on the node from any source we might wish such as third party tools or API interfaces.

Much in the same way as the Chef recipe DSL (domain specific language) provides us with a number of useful resource blocks to abstract away the complexities of the underlying Ruby code, ohai defines its own ruby-based DSL to simplify the process of writing plugins. In this chapter, we'll explore this DSL by analyzing the "skeleton code" necessary to create a working plugin, and how to test your plugins and integrate them into Chef runs. We'll then add functionality to our plugin skeleton to explore some of the different features provided by the ohai DSL before looking at some of the actual plugins that ship with ohai.

This chapter assumes that you are using ohai v7.0 or greater, which was released in early 2014. Version 7 of ohai introduced a new DSL for defining plugins, which supersedes the DSL used in previous ohai versions.

Introduction

Before we start to look at Ohai's recipe DSL however, let's take a moment to examine the way in which Chef treats attributes gathered by ohai and how they are given precedence over attributes defined in recipes, roles and environments.

Ohai Attribute Collection

When ohai runs, it iterates through all of the plugins it knows about to build up a collection of attributes describing the underlying node which it then passes to the chef client process. Because these attributes represent information about the node itself rather than the recipes being executed, Chef needs to ensure that they do not get altered during the course of the Chef run - when we use attributes collected by ohai like `node[:fqdn]` and `node[:platform_version]` in our recipes, it wouldn't be particularly desirable to be able to accidentally switch our node's `platform` from `centos` to `debian` in the middle of a run. To provide this guarantee that ohai attributes will not be changed during the run, Chef assigns them *automatic* precedence.

Chef Attribute Precedence

In Chef, where an attribute is defined and what level it is defined at will determine whether or not it takes precedence over other attributes defined with the same name in recipes, roles, environments etc - this system of attribute precedence can often be somewhat confusing to new Chef users. Here's a handy list of the order in which attributes take precedence (lowest to highest):

1. A `default` attribute located in a cookbook attribute file **(lowest precedence level)**
2. A `default` attribute located in a recipe
3. A `default` attribute located in an environment
4. A `default` attribute located in role
5. A `force_default` attribute located in a cookbook attribute file
6. A `force_default` attribute located in a recipe
7. A `normal` attribute located in a cookbook attribute file
8. A `normal` attribute located in a recipe
9. An `override` attribute located in a cookbook attribute file
10. An `override` attribute located in a recipe
11. An `override` attribute located in a role
12. An `override` attribute located in an environment
13. A `force_override` attribute located in a cookbook attribute file

14. A `force_override` attribute located in a recipe
15. An `automatic` attribute identified by Ohai at the start of the chef-client run **(highest precedence level)**

As shown in the sidebar above, *automatic* attributes take the absolute highest level of precedence. This means that even if we were to try and override an attribute collected by ohai in our recipe, the ohai version would always "win" as it is at a higher precedence level. This enables ohai to guarantee that the attributes it gives us are immutable during the course of a chef run, and that (providing the chef run is successful) they will be saved to the node object on the Chef server exactly as collected by ohai at the start of the run.

The Ohai Source Code

Just as the Chef source code is an extremely useful tool when examining the behavior of Chef, the source code of ohai (particularly its plugins) is a useful but optional aid when working through the material in this chapter.

The ohai source code is located in a seperate repository from that of Chef, and is available on Github. To clone a copy of the ohai source code, run the following command (which requires `git` to be installed):

```
$> git clone https://github.com/opscode/ohai.git
```

We won't be examining the under-the-hood implementation of ohai in as much detail as we did chef-client in "Tracing a Chef-client Run" on page 79, although interested readers may wish to dive a little deeper into its inner workings. We will, however, be looking at the implementation of ohai plugins, so the source code of the plugins shipped with ohai can be a useful reference to augment the examples shown in this chapter. These are located in the `lib/ohai/plugins` directory in the repository.

Now, on to the code!

Ohai Example 1: Plugin Skeleton

Throughout this chapter we'll actually be running the example plugin code we create, so at this point I'd recommend creating a directory to place your example plugins in - I've used `/tmp/ohai_plugins` in all of the examples but if you're on Windows or want to use a different directory path, you'll need to amend the path used in examples as appropriate.

Let's start off by looking at an example of the absolute minimum code necessary to produce a functioning ohai plugin - it won't actually add any attributes to our node yet, but it will demonstrate the framework with which we'll create more functional plugins

as the chapter progresses. Paste the following code into `/tmp/ohai_plugins/example1.rb`:

Example 4-1. /tmp/ohai_plugins/example1.rb

```
Ohai.plugin(:Example1) do❶
  provides "awesome_level"❷

  collect_data do❸
    awesome_level 100❹
  end
end
```

❶ This first line tells ohai what the name of the plugin *class* will be - under the hood, all ohai plugins are actually treated as classes which live under the `Ohai::NamedPlugin` namespace, and ohai needs to know what the class will be called. The name passed to the `Ohai.plugin` method (a symbol, since it must be immutable) must be a valid Ruby class name and start with a capital letter. Here we've used "Example1".

❷ Next, we call the `provides` method with a comma-separated list of all the attributes which our plugin will provide. In our example here, we're just creating one attribute called `awesome_level`.

❸ The `collect_data` block is called by Ohai when it executes each plugin to ask it to collect its data. This block is mandatory - if a `collect_data` block is not defined in the plugin code, the plugin will not do anything.

❹ Inside the `collect data` block, we're stating that the value of our `awesome_level` attribute should be `100`. Note that the Ohai DSL does not require us to use the = sign when assigning a value to an attribute - in fact if you **do** use an equals sign, the plugin will not work correctly. This variance from normal Ruby behavior is worth remembering, as although it makes for shorter plugin code, it can sometimes be confusing when debugging.

This is a nice simple example of an ohai plugin, but examples are much more meaningful if they can be observed working in real life. Let's take a look at the ways in which we can test and run our own ohai plugins.

Testing & Running Ohai Plugins

There are two ways in which we can try out our custom Ohai plugins. The first is to use Ruby's *irb* interactive shell, and the second is actually to include the plugin during a chef-client run - this is typically used when you want to deploy the plugin for real. We'll look at both of these methods in this section, and when you might want to use them.

Testing Using IRB

While ohai plugins are being developed, running them in the *irb* ruby shell is by far the most convenient method. IRB allows us to paste lines of Ruby code into a shell prompt, and immediately see the results of each line as it is executed. This functionality allows us to repeatedly run our plugin without the need to perform full chef-runs or alter our node's chef configuration.

The *irb* shell is started by running the `irb` command in a terminal, and will give you a prompt looking something like this:

```
irb(main):001:0>
```

The exact text shown before the > symbol will vary depending on your operating system and ruby version, so in the examples shown in this section I've used >> to represent a standard irb prompt.

Let's look at how we can test our example plugin from "Ohai Example 1: Plugin Skeleton" on page 99 using irb. Note, you must have the example code saved in a file called `/tmp/ohai_plugins/example1.rb` - if you chose to store your example plugins are stored in a different location, you'll need to substitute `/tmp/ohai_plugins` below for the path you are using.

Once you've executed the *irb* command, each time you see the >> prompt below followed by text this indicates a line Ruby code that you should paste into your IRB shell (don't copy the >>). The results you should see are shown below each >> line on lines beginning with =>. Where the output of a command is particularly long, I've replaced it with `<snip>`.

Now, let's try out example1.rb:

```
$> irb

>> require 'ohai'❶
=> true

>> Ohai::Config[:plugin_path] << '/tmp/ohai_plugins'❷
=> ["/usr/lib64/ruby/gems/1.9.1/gems/ohai-7.0.0/lib/ohai/plugins", "/tmp/ohai_plugins"]

>> o = Ohai::System.new❸
=> <snip>

>> o.all_plugins❹
=> <snip>
```

```
>>  o.attributes_print("awesome_level")❺
 => "100"
```

❶ The first thing we need to do is `require` the `ohai` gem so we can make use of its objects and methods.

❷ Next, we're appending the path where we stored our example plugins (`/tmp/ohai_plugins`) to the `[:plugin_path]` array stored under the `Ohai::Config` class. Just as Ruby uses the `LOAD_PATH` variable to tell it where to locate class definition files, this line tells ohai to look for plugins in our examples directory in addition to the default location. The exact directory paths outputted by this command may vary depending on the versions of ruby and ohai you have installed, but the important part is that we see `/tmp/ohai_plugins` at the end of the array.

❸ Now we need to create a new instance of the `Ohai::System` object, and assign it to a variable called `o`.

❹ Next, we call the `all_plugins` method of `o` (our `Ohai::System` object). This method loads all plugins that it finds under the `plugin_path` that we set in step 2, and executes their `collect_data` block. I've abbreviated the output of this command above as it is extremely verbose, but the eventual result of this command is to populate a class instance variable called `@data` inside our `Ohai::System` object with the attributes collected and returned by all the plugins. You can run this step repeatedly (without repeating steps 1-3) to re-execute all ohai plugins, which is handy when making small tweaks to your plugin code.

❺ Finally, we're calling the `attributes_print` method of our `Ohai::System` object, and passing the name of the attribute we're interested in - in this case, it's the `awesome_level` attribute we created in our `example1.rb` plugin. This method searches the `@data` class instance variable mentioned above to find the method name we're looking for, and if everything worked correctly we should see the value `"100"` returned, just as we set it in our plugin. As with step 4, this step can be run repeatedly to verify correct output when tweaking your plugin code.

The code we pasted into *irb* here is part of the actual code used by ohai when it's running as part of a Chef client run. Executing this code in *irb* allows us to quickly and easily test our plugin code to verify that it works correctly, but how do we use it as part of a Chef run for real when we're happy that the plugin is finished?

Running Using Chef

Once you're actually ready to run your plugin for real, you need to tell chef-client (or chef-solo) where your new plugins are located so that ohai is able to locate them when it runs. This is done by adding the `Ohai::Config[:plugin_path] << /location/of/`

plugins line to your *client.rb* or *solo.rb* configuration file, the same variable we modified when testing our plugin example with `irb`.

When chef-client or chef-solo runs, this configuration will be passed to ohai and your plugin will be executed when ohai calls the `all_plugins` method we saw above, as ohai will automatically load any plugins it finds under `Ohai::Config[:plugin_path]`. You also need to also make sure that your ohai plugins are cheffed out to the correct location on all of your nodes - to make this process easier, Chef Inc provide the ohai cookbook, which will automatically chef out plugins from the configured locations to your nodes, and dynamically update the Chef client configuration to load and run these plugins.

We've looked at a very simple example ohai plugin and seen how to test it and run it on our nodes, but what happens if we want the attributes provides by our plugin to contain more complex data or behave differently on other operating systems? Let's look at a slightly more advanced example plugin.

Ohai Example 2: Adding More to Plugin

In "Ohai Example 1: Plugin Skeleton" on page 99, we looked at a very simple plugin which assigned a string value to an attribute. As with many examples, it served well to demonstrate the concepts involved, but in the real world you're likely to need to write plugins which work across multiple operating systems, and store more complex and structured attribute data. So how do we do that? Fortunately for us, the ohai plugin DSL provides convenient functionality to support both of these cases. Consider the following example, which we'll paste into `/tmp/ohai_plugins/example2.rb`:

Example 4-2. /tmp/ohai_plugins/example2.rb

```
Ohai.plugin(:Example2) do
  provides "awesome"❶

  # Method to initialize object for our attribute
  def create_objects
    # Create a new "Mash" object and assign it to "awesome"
    awesome Mash.new
  end

  # Collect data block with symbol :default
  collect_data(:default) do

    # Call the create_objects method to initialize our "awesome" attribute
    create_objects
    # Assign the value 100 to the :level key of awesome
    awesome[:level] =  100
    # Assign the value "Sriracha" to the :sauce key of awesome
    awesome[:sauce] = "Sriracha"

  end
```

```
  # Collect data block with symbol :windows
  collect_data(:windows) do

    # Call the create_objects method to initialize our "awesome" attribute
    create_objects
    # Assign the value 101 to the :level key of awesome
    awesome[:level] =  101
    # Assign the value "Cholula" to the :sauce key of awesome
    awesome[:sauce] = "Cholula"

  end

end
```

❶ This line states that this plugin provides the "awesome" attribute - this also means that when Ohai runs this plugin, the `awesome` object is in scope for all plugin methods - this is how the `collect_data` method is able to access an object created in the `create_objects` method.

I've introduced two key new concepts in this plugin example - the `Mash` object that we're now using to store our `awesome` attribute, and the fact that we now have two `collect_data` blocks, each with a different *symbol* as its parameter. Let's examine these one at a time:

The Mash Object

The `Mash` class, defined in `lib/ohai/mash.rb` in the repository (Github Link) is a subclass of Ruby's built in `Hash` class. In our example above, we're using our `Mash` object `awesome` to store two related sub-attributes, `level` and `sauce`. This multi-level organizational structure is what allows us to use nested attributes like `node[:languages][:ruby][:version]` in our recipes - under the hood, ohai collected those attributes into a `Mash` object.

`Mash` can mostly be used just as if it were a regular `Hash`, but there is one important distinction between these classes which is in fact the reason `Mash` is used in Chef in the first place. With a standard `Hash` in Ruby, `foo[:bar]` and `foo["bar"]` are actually treated as two different items in the `Hash` - try running the following code in IRB:

```
$>irb

>> foo = Hash.new
=> {}

>>foo[:bar] = "hello"
=> "hello"
```

```
>>foo["bar"] = "goodbye"
=> "goodbye"

>>foo
=> {:bar=>"hello", "bar"=>"goodbye"}
```

In this code sample, `foo[:bar]` has the *symbol* `:bar` as its key while `foo["bar"]` has the *string* `"bar"` as its key. To avoid this potentially confusing behavior, the `Mash` class will treat `foo["bar"]` and `foo[:bar]` as the *same* item by always converting the hash key to a string behind the scenes. Let's repeat the above example in IRB using a Mash:

```
$>irb

>> require 'ohai'

>> foo = Mash.new
=> {}

>>foo[:bar] = "hello"
=> "hello"

>>foo["bar"] = "goodbye"
=> "goodbye"

>>foo
=> {"bar"=>"goodbye"}
```

As we can see, this time `foo[:bar]` and `foo["bar"]` were treated as the same item. In practical terms, you can often treat `Mash` objects just as if they were standard `Hash` objects, but it's useful to be aware of the difference.

Now, let's move on to the other new concept introduced in "Ohai Example 2: Adding More to Plugin" on page 103, that of having multiple `collect_data` blocks in our plugin.

Multiple `collect_data` Methods

As I mentioned in the preamble to this example, in the real world we often want to create an ohai plugin which works on different operating systems. Many of the default ohai plugins also follow this pattern - for example, the plugin which provides the `hostname` attribute has to be able to retrieve the hostname from Windows systems as well as Linux systems, and it would not be ideal to have to implement separate plugins for every possible operating system which can run Chef.

Being clever folks, Chef Inc. have provided a neat piece of functionality in ohai to remove the need for this replication of code. Let's look again at the method definitions we used in our plugin code:

Example 4-3. Excerpt of /tmp/ohai_plugins/example2.rb

```
Ohai.plugin(:Example2) do

  ...

  collect_data(:default) do
    ...
  end

  collect_data(:windows) do
    ...
  end

end
```

We're actually defining *two* `collect_data` blocks here, each with a different *symbol* as its parameter. This is because ohai allows us to define a `collect_data` block for each operating system for which we require specific behavior.

The first block with the `:default` key will be used if no other `collect_data` block can be found with a more specific symbol for the operating system running the plugin. You may remember that in "Ohai Example 1: Plugin Skeleton" on page 99, the `collect_data` block we used had no `:default` symbol - this is because `collect_data` blocks are *implicitly* treated as `:default` if no other symbol has been specified. I've added the `:default` symbol in the this example for clarity, but the plugin will work just as well if the first `:default` block has no symbol at all.

The second block has the `:windows` symbol as its parameter. When ohai detects that it is being run on a Windows system, this `collect_data` method will be executed instead of the `:default` block. It's important to remember that within a plugin you can only define one `collect_data` block for each operating system - ohai will throw an error if you try to define two blocks with the `:windows` symbol for example.

Currently, ohai allows you to specify the following symbols (hence supported operating systems) as parameters to `collect_data` blocks:

```
:aix, :darwin, :hpux, :linux, :freebsd, :openbsd, :netbsd, :solaris2, :windows
```

Running Example 2

Now that we've examined the new concepts introduced in example 2, let's actually run our new plugin in IRB on both Linux and Windows to see them in action:

Example 4-4. Running example2.rb on Linux

```
$> irb

>> require 'ohai'
=> true

>> Ohai::Config[:plugin_path] << '/tmp/ohai_plugins'
=> ["/usr/lib64/ruby/gems/1.9.1/gems/ohai-7.0.0/lib/ohai/plugins", "/tmp/ohai_plugins"]

>> o = Ohai::System.new
=> <snip>

>> o.all_plugins
=> <snip>

>>  o.attributes_print("awesome")
=> "{\n  \"level\": 100,\n  \"sauce\": \"Sriracha\"\n}"

>>  o.attributes_print("awesome/sauce")❶
 => "[\n  \"Sriracha\"\n]"
```

❶ When calling the `attributes_print` method on a `Mash`, we use the `/` character to indicate when you want to "descend" through the Mash. For example, here we're asking for `awsome/sauce` which is the equivalent of `awesome[:sauce]` if we were referring to this attribute inside our plugin. Note that here, the `attributes_print` method includes `\n` newline characters as this method is typically used to output "pretty printed" formatted output to a terminal.

Those results were exactly what we expected - Since no `collect_data` block was defined with the `:linux` symbol, the `collect_data` block with the `:default` symbol was executed, giving us an `awesome` Mash containing two attributes - `level` with the value `100` and `sauce` with the value `"Sriracha"`. Now let's try running the same plugin in IRB on a Windows machine and see what happens:

Example 4-5. Running example2.rb on Windows

```
C:\> irb

>> require 'ohai'
=> true

>> Ohai::Config[:plugin_path] << '/tmp/ohai_plugins'
=> ["C:/ruby/gems/1.9.1/gems/ohai-7.0.0/lib/ohai/plugins", "C:/tmp/ohai_plugins"]

>> o = Ohai::System.new
=> <snip>

>> o.all_plugins
=> <snip>
```

```
>>  o.attributes_print("awesome")
=> "{\n  \"level\": 101,\n  \"sauce\": \"Cholula\"\n}"

>>  o.attributes_print("awesome/sauce")
 => "[\n  \"Cholula\"\n]"
```

As we see here, when the plugin is run on a Windows machine, the `collect_data` block with the `:windows` symbol is used instead of the `:default` block, and this time we get an `awesome` Mash containing our two attributes, but this time `level` has the value `101` and `sauce` has the value `"Cholula"`. If you have both Windows and another operating system available for testing, why not try running this plugin on both systems to replicate these results?

With the addition of `Mash` objects and multiple operating system specific `collect_data` methods to our toolbox, we're now getting to the stage where we can start to write more useful ohai plugins - but what if you don't just want to write a single plugin, but rather a family of plugins which provide different "branches" of the same attribute structure? Let's look at some real examples from the plugins which actually ship with ohai.

Ohai Example 3: Multi-level Plugins

As we worked through "Ohai Example 1: Plugin Skeleton" on page 99 and "Ohai Example 2: Adding More to Plugin" on page 103, we've progressed from a barebones ohai plugin with minimal functionality to a more advanced plugin which implements OS specific behavior and a nested attribute structure. But how far does that leave us from the functionality contained in real ohai plugins that it ships with? To finish off this chapter we're going to look at one final feature of ohai, its dependency system.

Rather than looking at another canned example to illustrate ohai plugin dependenecy, we're going to look at three plugins which ship with ohai and provide various attributes which live under the `languages` attribute - you can see an example of the attributes stored under `languages` in "Get Configuration Data" on page 64.

The first of these plugins is the `languages.rb` plugin, shown here:

Example 4-6. lib/ohai/plugins/languages.rb

```
Ohai.plugin(:Languages) do
  provides "languages"

  collect_data do
    languages Mash.new
  end
end
```

This plugin is actually fairly close to the code we saw in "Ohai Example 1: Plugin Skeleton" on page 99 - it defines a single `collect_data` method, and initializes the top level `languages` attribute with a new `Mash` object. Note that it does not itself set any attributes beyond initializing `languages` - this is a commonly used pattern in ohai when defining a plugin which will sit at the "top" of a dependency tree. But how does the `languages` Mash get populated with attributes which store information on a number of different programming languages? That's where ohai's dependency system comes in.

Let's now look at the code of the `php.rb` plugin, which provides attributes related to the PHP programming language:

Example 4-7. lib/ohai/plugins/php.rb

```
Ohai.plugin(:PHP) do
  provides "languages/php"❶

  depends "languages"❷

  collect_data do

    # Set the "output" variable to nil
    output = nil

    # Create a new Mash object called php
    php = Mash.new

    # Execute the shell command "php -v" and store the results in
    # the variable 'so'
    so = shell_out("php -v")

    # If the exit code of the command was 0, ie successful
    if so.exitstatus == 0

      # Parse the php version command's output to extract the data we care about...
      output = /PHP (\S+).+built: ([^)]+)/.match(so.stdout)

      # If the above step produced any results
      if output
        # Set the :version and :builddate keys of our php Mash
        php[:version] = output[1]
        php[:builddate] = output[2]
      end
      # If the php Mash has a :version key, assign the php Mash to the :php key
      # of the languages Mash.
      languages[:php] = php if php[:version] ❸
    end
  end
end
```

❶ This line tells ohai that this plugin is only providing the `php` key of the `languages` Mash. Just as we saw when requesting our nested ohai attribute when we ran "Ohai Example 2: Adding More to Plugin" on page 103, the `/` character here is used to indicate different levels of the languages Mash. Essentially, this plugin is stating that it does not provide the entire `languages` attribute, just a single element of it called `php`.

❷ This line states that this plugin depends on the `languages` plugin - when ohai evaluates this line, it looks for a plugin named `languages.rb` to satisfy the dependency. Any attributes defined in the `languages` plugin are available to all plugins which depend on it. In this case, this means that the `languages` mash we declared in `languages.rb` is available to `php.rb` - this allows `php.rb` to add php specific attributes to the `languages` Mash as we'll see later in the plugin.

❸ This line sets the value of the `:php` key of the `languages` Mash to the `php` Mash if there is a non-nil `php[:version]` key. Note that the `languages` Mash is actually defined in `languages.rb` rather than this plugin - this is where the ohai plugin dependency model comes in handy. We can declare separate plugins to provide different elements of the same top-level ohai attribute.

Now let's look at the code of the `perl.rb` plugin, which provides attributes related to the Perl programming language:

Example 4-8. lib/ohai/plugins/perl.rb

```
Ohai.plugin(:Perl) do
  provides "languages/perl"❶

  depends "languages"❷

  collect_data do

    # Set the "output" variable to nil
    output = nil

    # Create a new Mash object called perl
    perl = Mash.new

    # Parse the perl version command's output to extract the data we care about...
    so = shell_out("perl -V:version -V:archname")

    # If the exit code of the command was 0, ie successful
    if so.exitstatus == 0

      # Parse the perl version command's output to extract the data we care about...
      so.stdout.split(/\r?\n/).each do |line|
        case line
        when /^version=\'(.+)\';$/
          perl[:version] = $1
```

```
        when /^archname=\'(.+)\';$/
          perl[:archname] = $1
        end
      end
    end

    # If the exit code of the command was 0, ie successful
    if so.exitstatus == 0
      # Assign the perl Mash to the :perl key of the languages Mash.
      languages[:perl] = perl❸
    end
  end
end
```

❶ This line tells ohai that this plugin is only providing the `perl` key of the `languages` Mash.

❷ This line states that, just like `php.rb`, this plugin depends on the `languages` plugin.

❸ This line sets the value of the `:perl` key of the `languages` Mash to the `perl` Mash. Just as in `php.rb`, the `languages` Mash is actually defined in `languages.rb` rather than this plugin.

In this example, we see another plugin which also provides part of the `languages` attribute, but implements entirely different behavior to fetch the data needed to provide the `perl` key.

These three plugins combine to provide a useful demonstration of the power and flexibility behind ohai's plugin interface. We see a top-level `languages.rb` plugin which defines the parent attribute name `langages`, and two "dependant" plugins (`php.rb` and `perl.rb`) which implement totally separate behavior specific to the programming languages tools they describe, and set specific and separatem elements of the parent `languages` attribute.

When we actually run ohai and look at the results, this hierarchical plugin dependency is hidden from us - what we actually see is a single `languages` attribute containing all of the information set by the various plugins which depend on `languages`.

To actually see this behavior in action, let's run ohai in IRB again and look at the `languages` attribute in the output:

Note that because this time we're only running plugins that already ship with ohai by default, we don't need to add our example plugin path to `Ohai::Config[:plugin_path]`. It you do not have the same programming languages as I do installed on your machine, the output you see when running this example locally may vary.

Example 4-9. Exploring the languages attribute

```
$> irb

>> require 'ohai'
=> true

>> o = Ohai::System.new
=> <snip>

>> o.all_plugins
=> <snip>

>>  o.attributes_print("languages")
 => "{\n  \"lua\": {\n    \"version\": \"5.1.4\"\n  },\n
          \"nodejs\": {\n    \"version\": \"0.10.21\"\n  },\n
          \"perl\": {\n    \"version\": \"5.16.2\",\n
            \"archname\": \"darwin-thread-multi-2level\"\n  },\n
          \"php\": {\n    \"version\": \"5.4.17\",\n
            \"builddate\": \"Aug 25 2013\"\n  },\n  }\n}"
           <<snip>>

>>  o.attributes_print("languages/php")
 => "{\n  \"version\": \"5.4.17\",\n  \"builddate\": \"Aug 25 2013\"\n}"

>>  o.attributes_print("languages/perl")
 => "{\n  \"version\": \"5.16.2\",\n  \"archname\": \"darwin-thread-multi-2level\"\n}"
```

In the above IRB output, we see that the `languages` attribute contains data on a number of different programming languages - although the output of ohai doesn't show us this, as we explored above this output is provided by a number of different plugins. In our sample output for example, we see the specific attributes set by the PHP and Perl plugins we explored in this section in the same `languages` attribute as attributes describing the lua and nodejs programming languages.

Summary

Ohai's flexible and extensible plugin interface allows us to leverage any information source we choose to store information about our Chef nodes. If we want to store information about a new programming language for example, we can extend one of the existing ohai plugins to make use of its existing data structures, or we can define entirely new attributes of our own choosing.

In this chapter, we've worked through a number of examples starting with the minimal code necessary for a functioning ohai plugin, then moving onto a plugin which implemented operating system specific behaviour and nested data structures before finally looking at some of the plugins that actually ship with ohai to explore its dependency model. It's my hope that these examples have served to give you an idea of the flexibility

and power that ohai plugins give you to store information about your nodes in Chef, and given you some ideas about the information you might be able to store about your nodes to enhance your chef recipes.

In the next chapter, we'll examine how Chef allows us to specify custom behavior on the success or failure of our Chef runs through the use of *handlers*.

CHAPTER 5
Creating Handlers

As we learned when exploring the different stages of a Chef run in "Anatomy of a Chef Run" on page 62, Chef executes *handlers* as specific situations arise during the run to allow us to respond to these events. As we'll examine in this Chapter, handlers are special Ruby classes which we incorporate into our Chef run - three types of handler are supported:

Start Handlers
: *Start* handlers run before the Chef run has fully begun during the "Load and Build Node Object" on page 66 step and are typically used to notify monitoring systems etc about the start of a Chef run or initialize reporting systems which will be used throughout the rest of the run.

Report Handlers
: *Report* handlers run during the "Finalize" on page 70 stage of the run, and are triggered when the Chef run has been **successful**. Report handlers can be created for variety of tasks, and are typically used to report statistics and data on the successful Chef run to monitoring systems.

Exception Handlers
: *Exception* handlers run during the "Finalize" on page 70 stage of the run, and are triggered when the Chef run has **failed**. Exception handlers are typically used to generate alerts or notifications to inform people that Chef has failed, and data on the nature of the failure.

In this chapter, we'll create a test environment to allow us to safely run code examples before examining the three handler types and how to implement and run each one. We'll also look at the various Chef classes we'll make use of to power and enhance the usefulness of our handlers, and some best practices to keep in mind when implementing them.

Preparing a Test Environment

Before we move on to the other types of run customization that we'll examine in the remainder of the chapters in Part 2, we need to pause for a moment to prepare a test environment to allow us to run our example code - unlike ohai plugins, which we were able to test using the IRB interactive shell, the other customization types we'll cover in these chapters need to be tested as part of an actual Chef run as they make use of objects and methods which are created during the run process.

In this section we're going to create a test environment which enable us to safely and repeatedly simulate a full chef-client run to test each of our customizations.

Our test environment will run chef-client in *local mode* (which we learned about in "Local Mode" on page 20) with the *why-run* feature enabled (which we saw in "Why-run mode" on page 73). The use of *local mode* will allow us to simulate a full Chef run against a mock Chef server, while the use of *why-run* mode will ensure that the underlying node will not be changed at any point. We'll also create a simple test cookbook to use as our run list for these tests.

To run the examples in this section, you'll need to make sure you have Chef (at least version 11.8.0) installed on a machine suitable for running development code - please don't run example code on one of your production servers! As with other examples in this book, I've assumed that you're using a Linux / Unix based OS. If this is not the case, you'll need to adjust the directory paths used as appropriate.

To set up the test environment, you'll need to complete the following steps:

Create Test chef-client Configuration

The first step in creating our test environment is to create a blank `config.rb` file to use with chef-client - we don't want to pick up any settings from default configuration files like `/etc/chef/client.rb`.

The customizations we'll cover in this chapter are also fully compatible with chef-solo, however I've chosen to use chef-client for our test environment so that we can use *local mode* to replicate communicating with an actual Chef server. To use our customizations with chef-solo, the same configuration directives that we'll use here can also be used with `solo.rb`

Create a directory file called `/tmp/part2_examples`, and then create a blank file in that directory called `client.rb`:

```
$> mkdir /tmp/part2_examples
$> touch /tmp/part2_examples/client.rb
```

We'll flesh out this configuration file as we come to test our customization examples later in the chapter - since chef-client will be running in *local mode*, we don't need any other configuration details in our `client.rb` file at this stage as *local mode* will create the configuration options it needs to run.

If you're using a non Linux / Unix based operating system, you may need to alter the `/tmp/part2_examples` directory path we're using for our examples to one appropriate for your operating system.

Create Test Cookbook

Now that we have our blank `client.rb` file to use with our test chef run, our next step is to create a simple cookbook to use during our tests. First, we'll need to create a directory called `/tmp/part2_examples/cookbooks`:

```
$> mkdir /tmp/part2_examples/cookbooks
```

Next, we'll use knife's `cookbook create` command to create a cookbook skeleton in `/tmp/part2_examples/cookbooks` - when you run the below command, you should see output similar to that shown:

```
$> knife cookbook create testcookbook -o /tmp/part2_examples/cookbooks
WARNING: No knife configuration file found
** Creating cookbook testcookbook
** Creating README for cookbook: testcookbook
** Creating CHANGELOG for cookbook: testcookbook
** Creating metadata for cookbook: testcookbook
```

Next we're going to add two simple resources into the `default.rb` recipe created with our cookbook so that our test chef run will actually have some resources to run. Replace the contents of `/tmp/part2_examples/cookbooks/testcookbook/recipes/default.rb` with the following:

Example 5-1. /tmp/part2_examples/cookbooks/testcookbook/recipes/default.rb

```
# This directory already exists, so this resource
# should always do nothing
directory "/tmp/part2_examples"

# This file will not exist, so this resource
# should create it
file "/tmp/part2_examples/testfile"
```

As I mentioned above, we'll be running this recipe in *why-run* mode - this means that every time we kick off a Chef run the `directory` resource will of course not create `/tmp/`

`part2_examples` as this directory already exists. When executed under *why-run* mode the `file` resource should tell us that a normal Chef run would create the file `/tmp/part2_examples/testfile` as it does not yet exist. Note that *why-run* mode will not actually **create** the file, which means that we can reliably reproduce the behavior of this test run when running our example customization code.

When we want to test our customizations, this recipe will give us a base run list that we can run repeatedly with identical results - this will make observing the effect of our customizations much easier!

Verify Test Environment Works Correctly

With our blank `client.rb` file and our test cookbook created, we have all the components we require for our test environment - but now we need to make sure that it works correctly! To do this, we're going to perform a Chef run using the following command from `/tmp/part2_examples`:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default
```

In this command, I've passed the following options to chef-client:

--once
: This option tells chef-client to only run once instead of in a continuous loop, as it would by default.

--why-run
: This option tells chef-client to run in *why-run* mode as we saw in "Why-run mode" on page 73.

--local-mode
: This option tells chef-client to run in *local mode*, starting up a local chef-zero server as we saw in "Local Mode" on page 20.

list --config /tmp/part2_examples/client.rb
: This option tells chef-client to use the blank `client.rb` configuration file we created, instead of looking for the default configuration file at `/etc/chef/client.rb`

--override-runlist testcookbook::default
: This option tells chef-client to override the run_list to be executed on our node, and set it to `testcookbook::default`

Now let's actually run this command - if you followed all of the instructions in the previous steps, you should see similar output to that shown below:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default
```

```
Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
Running handlers complete
```

Note that here as we're running in *why-run* mode, Chef does not actually create the file `/tmp/part2_examples/testfile` but instead tells us that this is what **would** happen if Chef were not being run in *why-run* mode. Chef also reminds us that as we're running in *why-run* mode it will not save node attributes to the server.

Although we gave chef-client a blank `client.rb` configuration file, running in *local mode* means that chef-client started up chef-zero at the beginning of the run and set its configuration parameters to the chef-zero server (as opposed to a remote Chef server) just as we saw in "Chef::Application::Client - run_chef_client Method" on page 85.

Now that we've validated that our test environment works correctly, it's time to look at how to implement some more customizations.

Introduction to Handlers

Although triggered by very different situations during a Chef run, under the hood all handlers build on the same underlying framework. Understanding this common structure and how it integrates with Chef runs is crucial when implementing your own handlers, so we're going to examine these areas in detail before moving on to more specific implementation examples.

Regardless of their type, all handlers share the same skeleton code structure:

```
require "chef/handler"

class Chef
  class Handler
    class HandlerName < Chef::Handler ❶
      def report❷
```

```
      # Ruby code goes here
    def
  end
end
end
```

❶ The handler skeleton inherits from the *superclass* `Chef::Handler` which lives at `lib/chef/handler.rb` in the Chef repository (Github Link).

❷ Here we're overriding a single method from `Chef::Handler` called `report`. Whenever Chef needs to activate handlers of a particular type during the run, it calls their `report` method.

Given this common code structure and `report` method, what *distinguishes* each handler type? How does Chef know which handlers are *exception* handlers and which are *report* or *start* handlers so it can call the `report` method on the right handlers at the right times?

Part of the answer lies in the chef-client or chef-solo configuration file. For each supported handler type, Chef defines a list to which we can append our handler classes as in this example:

Example 5-2. Declaring handlers in client.rb

```
require "/var/chef/handlers/mystarthandler.rb"
require "/var/chef/handlers/myhandler.rb"

my_start_handler = MyStartHandler.new
my_handler = MyHandler.new

start_handlers << my_start_handler          # List of start handler classes
report_handlers << my_handler               # List of report handler classes
exception_handlers << my_handler            # List of exception handler classes
```

Don't worry too much about exactly what's going on here for now - we'll examine each handler type and how to run them in much more detail later on in this chapter. For now it's sufficient to understand that we create an instance of our handlers and add them to the appropriate list, which Chef then uses to decide which handlers to call at the appropriate time during the run.

For example, when Chef calls any *start* handlers during the "Load and Build Node Object" on page 66 step of the run, it does this by calling the *report* method of all handlers contained in the `start_handlers` list in the Chef configuration file. Likewise, when *exception* or *report* handlers are called during the "Finalize" on page 70 step, Chef calls the `report` method of any handlers contained in the `report_handlers` or `exception_handlers` list.

You may also have noticed that in this example we're defining a separate object to use as a *start* handler, but are using the **same** object as both a *report* and *exception* handler. This isn't a mistake - because Chef calls the *report* method on handler objects added to each list as we saw above, there's nothing to stop us defining a handler with a `report` method which can cater for both failed and successful runs.

But enough theory - how do our handler classes tell whether or not the Chef run they are responding to has succeeded or failed if the `report` method is called in both cases? Chef exposes this chef run information to handler classes through an object we already met in chapter 3 during the "Load and Build Node Object" on page 66 step - the *run status* object.

Runstatus Object

As we saw in Chapter 3, the *run status* object is created during the "Load and Build Node Object" on page 66 stage of the Chef run and contains a number of different attributes which are populated throughout the Chef run and hold information on different aspects of the Chef run's status. Chef exposes the *run status* object to all handlers to allow them to introspect into the run which triggered them and react accordingly.

The *run status* object is an instance of the `Chef::RunStatus` class which lives at `lib/chef/run_status.rb` in the Chef repository (Github Link) and provides us with a number of methods to access the attributes it stores on the status of the current Chef run. In this section we'll briefly examine some of these methods and look at how they might be of use when implementing the example handler code later in this chapter.

Run Status Methods

The *run status* object provides two methods for querying the status of the Chef run which triggered the handler. These methods are especially useful when implementing a *report* and *exception* handler in the same class, as it allows you to control the logic of the `report` method as appropriate for successful or failed runs.

success?
: The `success?` method returns `true` when the Chef run **succeeded** - *success* in this context means that no un-handled exceptions were thrown during the progress of the Chef run. When this method returns `true`, logic for a *report* handler should be executed.

failed?
: The `failed?` method returns `true` when the Chef run **failed** - *failure* in this context means that an un-handled exception was thrown during the progress of the Chef run. When this method returns `true`, logic for an *exception* handler should be executed.

As start handlers run at a very earliest stage of the run before much of the data in the *run status* object has been populated, there is no `started?` method to indicate that the run has just started - after all, the fact that the handler has been executed in the first place means that the run **must** have started. For this reason, start handlers are typically implemented as separate classes from *report* or *exception* handlers and controlled by making sure that they are only added to the `start_handlers` list in the Chef configuration file.

Run Information Methods

Once we've identified the status of the Chef run which has triggered our handler class, there are a number of method provided by the *run status* object which allow us to introspect deeper into the run to identify exactly what was happening at the time the handler was triggered.

exception
: If an unhandled exception occurred during the Chef run, the `exception` method returns the exception object that was thrown. It's worth noting that this method returns the actual `Exception` object, just as we would get in a `rescue` block if catching the exception ourselves. If no unhandled exceptions occured during the Chef run, this method will return `nil`.

backtrace
: If an unhandled exception occurred during the Chef run, the `backtrace` method returns an array containing the *backtrace* of the exception that was thrown. The *backtrace* is the text we see when an exception has been thrown which tells us in which lines of what class files the exception occurred. If no unhandled exceptions occured during the Chef run, this method will return `nil`.

start_time
: The `start_time` method returns a `Time` object containing the time that the Chef run started

end_time
: The `start_time` method returns a `Time` object containing the time that the Chef run ended

elapsed_time
: The `elapsed_time` method returns the difference between `start_time` and `end_time` as a `Float` object, if both methods have values. If either `start_time`, `end_time` or both have not been set, `elapsed_time` returns nil.

Run Object Methods

The remainder of the methods provided by the *run status* object provide access to a number of other objects used during the Chef run which can be used to add extra context or information to your handler code.

node
: The `node` method returns the `Chef::Node` object produced during the "Load and Build Node Object" on page 66 step. This *node* object contains the saved node data loaded from the Chef server (if applicable) which includes useful data like the run list to be applied to the node, attributes loaded by `ohai` etc. See "Chef::Client class - do_run Method" on page 88 for details of exactly where in the Chef code this object is created.

events
: The `events` method returns `EventDispatcher::Dispatcher` object which forms the *publisher* component of the event dispatcher. We'll look at the event dispatcher in more detail in Chapter 6

run_context
: The `run_context` method returns the `Chef::RunContext` object produced during the "Setup RunContext" on page 67 step. This object contains a variety of useful data about the overall Chef run such as the cookbook files needed to perform the Chef run, the list of all resources to be applied during the run and the list of all notifications triggered by resources during the run.

all_resources
: The `all_resources` method provides a list of all resources stored in the *resource collection* of the *run context* object - as we saw in "Setup RunContext" on page 67, this is the expanded list of resources which will be applied to the node during the Chef run.

updated_resources
: The `updated_resources` method provides a list of resources that were marked as *updated* during the Chef run, ie resources which were not already *converged*.

As start handlers are executed at the very beginning of the Chef run, the only *run status* attributes which will have been populated at the point when start handlers run are `start_time` and `node`. All other *run status* attributes are populated at later stages of the run and are will return `nil` if called in a *start* handler. Please see "Chef::Client class - do_run Method" on page 88 for a reminder of how the different Chef run stages fit together in code.

So far in this chapter, we've looked at the common code skeleton shared by all Chef handlers, how Chef keeps track of handlers of each type and the data exposed by the *run status* object to help us implement our handlers. Let's put that knowledge to use now and look at some examples of implementing each handler type.

Handler Example 1: Start Handler

Since *start* handlers are the first handlers to be executed during a Chef run, let's start there for our first handler example. As we touched on briefly in "Load and Build Node Object" on page 66, start handlers are most typically used for two specific purposes:

- Producing a notification that a Chef run has started on the node
- Initializing reporting systems used throughout the rest of the run

In this example, we'll look at the first of these use cases, generating a notification that a Chef run has started on the node. Usually, such a start handler would be used to send notifications to metrics or monitoring system such as Graphite and StatsD however as in this case as I want readers to be able to easily try these examples out for themselves, our example start handler will simply write to a text file.

I'll list handlers which interact with a number of common monitoring systems at the end of the chapter in "Handlers: Summary and Further Reading" on page 137, and we'll learn how to write a start handler to initialize reporting systems later on in this chapter in "Creating Custom Subscribers" on page 151.

Let's start by pasting the following code into `/tmp/part2_examples/awesome_start_handler.rb`:

/tmp/part2_examples/awesome_start_handler.rb.

```
require "chef/handler"

class Chef
  class Handler
    class AwesomeStartHandler < Chef::Handler ❶

      # Override report method from Chef::Handler
      def report

          # Print a log message when our handler is executed
          Chef::Log.warn("Handler #{self.class} executed") ❷

          # Grab data from the run_status object
          # and assign to local variables for clarity
          run_start_time = @run_status.start_time ❸
```

```
            node_name = @run_status.node.name ❹

            # Open output file in append mode and write handler output to it
            File.open("/tmp/handler_output", 'a') do |file|
                file.write("\n#{self.class}: Run started on #{node_name} at " +
                "#{run_start_time}\n")❺
            end
        end
      end
    end
  end
```

❶ Just as we saw in "Introduction to Handlers" on page 119, our *start* handler makes use of the common handler skeleton that forms the basis of all handlers - it inherits from the *superclass* `Chef::Handler` and overrides a method called `report`.

❷ Here we're logging a "warning" level message to show the handler being executed in the run output - by default, only *report* and *exception* handler executions are logged.

❸ Here we're making use of the *run status* object we learned about in "Runstatus Object" on page 121 to capture the `start_time` of the chef run on this node. Note that the *run status* object is accessed using the `@run_status` class instance variable - this is declared in the `Chef::Handler` *superclass*, hence is available to all handler *subclasses*.

❹ Here we're calling the `name` method of the `Chef::Node` object returned by the `@run_status.node` method - this gives us the name of the object representing our node, usually the node's FQDN.

❺ Finally we're using Ruby's `File` class to open the file `/tmp/handler_output` and append our handler output message to it - the `self.class` method call on this line simply adds the name of the current handler class to the start of the output string. I've also broken the `file.write` line here over two lines for formatting purposes, this is not a functional requirement.

Now that we've created our example *start* handler, before we can run it using the test environment we prepared in (to come) we need to tell our `client.rb` configuration file that we want our handler to be added to the `start_handlers` list. To do this, we need to add the following lines to `/tmp/part2_examples/client.rb`:

Example 5-3. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome_start_handler.rb" ❶

example_start_handler = Chef::Handler::AwesomeStartHandler.new❷

start_handlers << example_start_handler❸
```

❶ Here we're telling `client.rb` to *require* the `awesome_start_handler.rb` file we created earlier.

❷ Next we're creating an instance of our `Chef::Handler::AwesomeStartHandler` class. Note that because we specified in our class definition that our handler class lives in the `Chef::Handler` namespace, we need to refer to our handler by its full name here rather than just `AwesomeStartHandler`.

❸ Finally we're adding our `example_start_handler` object to the `start_handlers` list.

With our `client.rb` file configured to use our start handler, we can now perform a Chef run on our test environment using the same command we saw in (to come):

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default

Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Start Handler
  Chef::Handler::AwesomeStartHandler executed
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
Running handlers complete
```

If everything was correctly configured, you should see output from your chef run similar to the above - note the log message telling us when our start handler has been executed. If we take a look in the `/tmp/handler_output` file we specified in our handler class, we should now see output similar to the following:

```
$> cat /tmp/handler_output

Chef::Handler::AwesomeStartHandler: Run started on mynode.mydomain.com at
  2014-02-13 11:36:06 +0000
```

This tells us that when our test run reached the "Load and Build Node Object" on page 66 step, Chef executed our `AwesomeStartHandler` by calling its report method which resulted in the above output being printed to `/tmp/handler_output`. Now every time we kick off a Chef run, our start handler will output a notification line to `/tmp/handler_output` with the exact time that the run started - try running a few more example Chef runs and see for yourself!

As I mentioned before, this type of start handler would usually send this notification to a monitoring system such as Graphite or StatsD - the flexibility of *start* handlers means that you can send notifications to any source you desire, in any format you can implement in Ruby.

It's also important to remember that because the `start_handlers` list in `client.rb` is an array, you can add as many start handlers as you want and all will be executed at the beginning of the run - Chef doesn't limit the number of handlers we can run at once. Check out the "Handlers: Summary and Further Reading" on page 137 section for examples of handlers which interact with real-world monitoring systems.

Now that we've implemented a *start* handler to notify us that our run has started, it's time to move to the latter stages of the Chef run and examine how to implement a *report* handler to provide us with information on successful Chef runs.

Handler Example 2: Report Handler

As we saw in "Finalize" on page 70, *report* handlers run at the very end of a Chef run if the run *succeeded*. This means that unlike *start* handlers, *report* handlers are able to access the full range of data exposed through the *run status* object as by the end of the run it has been fully populated the attributes accessed by the methods we examined in "Runstatus Object" on page 121.

The information provided by *report* handlers - although technically relating to **successful** runs only - can often be extremely useful in identifying performance bottlenecks and inefficiencies in infrastructure automation code by feeding monitoring and metrics systems with information about our Chef runs such as:

- The number of resources updated during each run - useful for tracking how idempotent our recipes are.
- The average duration of chef runs across your infrastructure - useful for a global overview of how your Chef runs are performing.
- More detailed profiling of the performance of individual resources and recipes during Chef runs - useful for tracking down performance bottlenecks.

Consider for example a recipe which contains a `bash` resource which, as part of compiling a piece of software from source, uses the `wget` command to download a large file

from the internet on **every** Chef run instead of using a `remote_file` resource to download it once.

This recipe is not technically **broken** in the strictest sense as it does not cause the run to fail, but it **is** behaving in a decidedly sub-optimal manner which could seriously impact network performance if run on a large number of nodes. Report handlers are able to feed our metrics and monitoring systems with data on our Chef runs that make it much easier for us to capture this type of event, as we'll see in this example.

As with our *start* handler example, to keep our example implementation as simple and easy to test as possible we're going to have our handler write various details about the successful Chef run to the `/tmp/handler_output` file. Let's start by pasting the following code into `/tmp/part2_examples/awesome_report_handler.rb`:

/tmp/part2_examples/awesome_report_handler.rb.

```
require "chef/handler"

class Chef
  class Handler
    class AwesomeReportHandler < Chef::Handler ❶

      # Override report method from Chef::Handler
      def report

          # If the run was successful, run this code.
          if @run_status.success?❷

            # Grab data from the run_status object
            # and assign to local variables for clarity
            run_elapsed_time = @run_status.elapsed_time❸
            node_name = @run_status.node.name
            resource_count = @run_status.all_resources.length❹
            updated_resources = @run_status.updated_resources❺
            updated_resource_count = updated_resources.length❻

            # Open output file in append mode and write handler output to it
            File.open("/tmp/handler_output", 'a') do |file|❼
                file.write("\n#{self.class}: Run successfully completed on " +
                  "#{node_name} and took " +
                  "#{run_elapsed_time} seconds:\n")
                file.write("  #{resource_count} resources in total, " +
                  "#{updated_resource_count} updated:\n")

                # Write each resource in the updated_resources list to our
                # output file
                updated_resources.each do |resource|
                  m = "recipe[#{resource.cookbook_name}::#{resource.recipe_name}]" +
                    " ran '#{resource.action}' on #{resource.resource_name}" +
                    " '#{resource.name}'"
                  file.write("    #{m}\n")
```

```
                end
              end
            end
          end
        end
      end
    end
```

❶ As we saw in "Introduction to Handlers" on page 119, our *report* handler makes use of the common handler skeleton that forms the basis of all handlers - it inherits from the *superclass* `Chef::Handler` and overrides a method called `report`.

❷ We're making use of the `@run_success.success?` method here to only run our report handler code if the run succeeded. If it failed, our handler will do nothing.

❷ Here we're logging a "warning" level message to show the handler being executed in the run output

❸ Here we're using the `elapsed_time` method of the `@run_status` object to get the duration of the Chef run.

❹ Next we're fetching the total number of resources executed in this Chef run by getting the `length` of `@run_status.all_resources`. As we learnt in "Runstatus Object" on page 121, the `all_resources` method returns the *resource collection* array from the *run context* object.

❺ Next we're fetching the list of resources **updated** in this Chef run by calling the `updated_resources` method of the `@run_status` object.

❻ Next we're getting the total **number** of updated resources executed in this Chef run by getting the `length` of `@run_status.updated_resources`.

❼ As with our start handler example earlier in this chapter, we're opening the `/tmp/handler_output` file and appending our handler output to it.

Now that we've created our *report* handler class, we need to tell our `client.rb` configuration file that we want our handler to be added to the `report_handlers` list - this is done in a very similar fashion to the *start* handler we added in the last example. To add our report handler to the configuration, as we're going to leave our *start* handler example in place we need to add the lines ending with the comment `# ADD` to our `client.rb` file:

Example 5-4. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome_start_handler.rb"

require "/tmp/part2_examples/awesome_report_handler.rb" # ADD❶

example_start_handler = Chef::Handler::AwesomeStartHandler.new

example_report_handler = Chef::Handler::AwesomeReportHandler.new # ADD❷
```

```
start_handlers << example_start_handler

report_handlers << example_report_handler # ADD❸
```

❶ Here we're telling `client.rb` to *require* the `awesome_report_handler.rb` file we created earlier.

❷ Next we're creating an instance of our `Chef::Handler::AwesomeReportHandler` class.

❸ Finally we're adding our `awesome_report_handler` object to the `report_handlers` list.

With that done, we can now perform a Chef run on our test environment using the same command we saw in (to come) to see the output produced by our report handler:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default

Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 13:59:16 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 13:59:16 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 13:59:16 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
[Sat, 22 Feb 2014 13:59:16 +0000] WARN: Start Handler
  Chef::Handler::AwesomeStartHandler executed
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 13:59:16 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
  - Chef::Handler::AwesomeReportHandler
```

As we see above, at the end of the run Chef shows our `Chef::Handler::AwesomeReportHandler` class being executed. If we look in the `/tmp/handler_output` file we specified in our handler class, we should now also see output similar to the following:

```
$> cat /tmp/handler_output

Chef::Handler::AwesomeStartHandler: Run started on mynode.mydomain.com at
  2014-02-13 14:46:12 +0000
```

```
Chef::Handler::AwesomeReportHandler: Run successfully completed on
 mynode.mydomain.com and took 0.110945093 seconds:
   2 resources in total, 1 updated:
    recipe[testcookbook::default] ran 'create' on file '/tmp/part2_examples/testfile'
```

This tells us that when our test run reached the "Finalize" on page 70 step without any exceptions being thrown, Chef executed our `AwesomeReportHandler` by calling its report method which resulted in the above output being printed to `/tmp/handler_output`. Now every time we kick off a Chef run which does not throw any exceptions, our report handler will output a notification line to `/tmp/handler_output` with the output we saw here - try running a few more example Chef runs and see for yourself!

We've examined the implementation of two out of the three handler types supported by Chef now to notify us when our runs have *started* and *succeeded*. To finish off our examples, we need to look at *exception* handlers to allow us to respond to *failed* Chef runs.

Handler Example 3: Exception Handler

The final type of handlers supported by Chef are *exception* handlers, which are executed if an unhandled exception is thrown at any point during the Chef run. As we saw in "Handling Exceptions" on page 46, an *unhandled* exception is one which is not caught by a `rescue` block. In the context of our Chef run, this means that something has gone wrong during the run and it cannot continue.

Unsuccessful Chef runs generally indicate a situation which requires active remediation. This means that *exception* handlers - which alert us to those failures - are by far the most commonly used handler variety, as failing Chef runs are often of more immediate interest than successful ones. Correspondingly, a good number of the handlers open sourced by the Chef community are *exception* handlers to send data on failed runs to a wide variety of monitoring and notification systems.

As with the other handler examples we've looked at, to keep things simple our example exception handler will write its output to the `/tmp/handler_output` file. Rather than implementing an entirely separate class as we did for our *start* and *report* handler examples however, we're going to leverage the information exposed by the *run status* object to let us combine *exception* handling logic with the *report* handler logic from our previous example into a single multi-purpose handler class.

We **could** of course implement our exception handler as a separate class if we wished, but a combined report and exception handler provides a useful illustration of the flexibility and power of handlers in Chef. Let's add the *exception* and *report* handling code for our new combined handler below to `/tmp/part2_examples/awesome_handler.rb`:

/tmp/part2_examples/awesome_handler.rb.

```
require "chef/handler"

class Chef
  class Handler
    class AwesomeHandler < Chef::Handler❶
      def report

          # Existing logic from our report handler
          # If the run was successful, run this code.
          if @run_status.success?❷

            # Grab data from the run_status object
            # and assign to local variables for clarity
            run_elapsed_time = @run_status.elapsed_time
            node_name = @run_status.node.name
            resource_count = @run_status.all_resources.length
            updated_resources = @run_status.updated_resources
            updated_resource_count = updated_resources.length

            # Open output file in append mode and write handler output to it
            File.open("/tmp/handler_output", 'a') do |file|
                file.write("\n#{self.class}: Run successfully completed on " +
                  "#{node_name} and took " +
                  "#{run_elapsed_time} seconds:\n")
                file.write("  #{resource_count} resources in total, " +
                  "#{updated_resource_count} updated:\n")

                # Write each resource in the updated_resources list to our
                # output file
                updated_resources.each do |resource|
                  m = "recipe[#{resource.cookbook_name}::#{resource.recipe_name}]" +
                    " ran '#{resource.action}' on #{resource.resource_name}" +
                    " '#{resource.name}'"
                  file.write("    #{m}\n")
                end
            end

          # New logic to implement exception handler behavior
          # If the run failed, run this code.
          elsif @run_status.failed?❸

            # Grab data from the run_status object
            # and assign to local variables for clarity
            run_elapsed_time = @run_status.elapsed_time❹
            node_name = @run_status.node.name
            exception = @run_status.exception❺
            backtrace = @run_status.backtrace❻

            # Open output file in append mode and write handler output to it
            File.open("/tmp/handler_output", 'a') do |file|❼
                file.write("\n#{self.class}: Run failed on " +
                  "#{node_name} after #{run_elapsed_time} seconds:\n")
```

```
                file.write("  Exception: #{exception}\n")
                file.write("  Backtrace: \n")
                backtrace.each do |b|
                  file.write("    #{b} \n")
                end
              end
            end
          end
        end
      end
    end
```

❶ As with all of the previous handler examples, our new combined handler class makes use of the common handler skeleton that forms the basis of all handlers - it inherits from the *superclass* `Chef::Handler` and overrides a method called `report`.

❷ We're making use of the `@run_success.success?` method here to only run our **report** handler code if the run succeeded. The code in this part of the `if` statement is directly copied from the *report* handler we implemented in "Handler Example 2: Report Handler" on page 127

❸ We're making use of the `@run_success.failed?` method here to only run our **exception** handler code if the run failed.

❹ Here we're using the `elapsed_time` method of the `@run_status` object to get the duration of the Chef run.

❺ Next we're calling the `exception` method of the `@run_status` object to get the `Exception` object that was thrown during the run

❻ Here we're calling the `backtrace` method of the `@run_status` object to get the full backtrace which tells us exactly where our exception occurred.

❼ As with our other examples earlier in this chapter, we're opening the `/tmp/handler_output` file and appending our handler output to it.

Before we can run our new multi-purpose handler, we need to modify our `client.rb` file to use our separate start handler as before, but instead keeping the report handler we used in "Handler Example 2: Report Handler" on page 127 in the `report_handlers` list, we're going to add our new combined handler class to both the `exception_handlers` **and** `report_ handlers` lists. Let's replace the contents of `/tmp/part2_examples/client.rb` with the following:

Example 5-5. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome_start_handler.rb"
require "/tmp/part2_examples/awesome_handler.rb"

example_start_handler = Chef::Handler::AwesomeStartHandler.new
```

```
example_handler = Chef::Handler::AwesomeHandler.new

start_handlers << example_start_handler
report_handlers << example_handler❶
exception_handlers << example_handler❷
```

❶ ❷ Note that here, we're adding our `example_handler` object (which is an instance of `Chef::Handler::AwesomeHandler`) to both the `report_handlers` and `exception_handlers` lists.

Now we're ready to try out our combined *report* and *exception* handler and see how it behaves on successful and failed Chef runs. First, let's test the *report* handler portion of our `AwesomeHandler` class by performing the same Chef run as we used in "Handler Example 2: Report Handler" on page 127:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default

Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 14:22:49 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:22:49 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:22:49 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
[Sat, 22 Feb 2014 14:22:49 +0000] WARN: Start Handler
  Chef::Handler::AwesomeStartHandler executed
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 14:22:49 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
  - Chef::Handler::AwesomeReportHandler
```

As expected, the run completed successfully - let's check the contents of `/tmp/handler_output` to verify that the output of our handler was as we expected too:

```
$> cat /tmp/handler_output

Chef::Handler::AwesomeStartHandler: Run started on mynode.mydomain.com at
   2014-02-22 14:22:49 +0000

Chef::Handler::AwesomeHandler: Run successfully completed on
 mynode.mydomain.com and took 0.168382415 seconds:
```

```
  2 resources in total, 1 updated:
    recipe[testcookbook::default] ran 'create' on file
      '/tmp/part2_examples/testfile'
```

Just as in previous Chef runs, we see our `AwesomeStartHandler` being executed at the start of the run. When the run finishes we now see our combined `AwesomeHandler` class writing its *report* handler output, triggered by checking `@run_status.success?` and having it return `true`, indicating a successful Chef run. This output is exactly as we saw in our separate *report* handler class - exactly what we wanted to see.

Now that we've verified that the report handler portion of our `AwesomeHandler` class still works, we need to test the *exception* handler part. This means that we need to deliberately break our Chef run in such a way that it throws an exception. There are a number of ways that we could do this, but the simplest is to make use of Chef's build in safety mechanisms, and tell our test run to execute a run_list containing a recipe which does not exist. Let's change the value passed to the `-o` parameter of our run command from `testcookbook::default` parameter to `testcookbook::shrubbery`, which is a non existent recipe:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::shrubbery

Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 14:05:06 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:05:06 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:05:06 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::defaults]]
[Sat, 22 Feb 2014 14:05:06 +0000] WARN: Start Handler
  Chef::Handler::AwesomeStartHandler executed
resolving cookbooks for run list: ["testcookbook::defaults"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...

================================================================================
Recipe Compile Error
================================================================================

Chef::Exceptions::RecipeNotFound
================================
could not find recipe defaults for cookbook testcookbook

Running handlers:
[Sat, 22 Feb 2014 14:05:06 +0000] ERROR: Running exception handlers
  - Chef::Handler::AwesomeHandler
Running handlers complete

[Sat, 22 Feb 2014 14:05:06 +0000] ERROR: Exception handlers complete
[Sat, 22 Feb 2014 14:05:06 +0000] FATAL: Stacktrace dumped to
  /home/jcowie/.chef/local-mode-cache/cache/chef-stacktrace.out
```

```
Chef Client failed. 0 resources would have been updated
[Sat, 22 Feb 2014 14:05:06 +0000] ERROR: could not find recipe
  defaults for cookbook testcookbook
[Sat, 22 Feb 2014 14:05:06 +0000] FATAL: Chef::Exceptions::ChildConvergeError:
  Chef run process exited unsuccessfully (exit code 1)
```

As we see here, this time our Chef run failed, and showed that our `Chef::Handler::AwesomeHandler` was executed as an exception handler - the output of the run told us about the failure and wrote the exception's stacktrace to disk but of course if Chef had been running in daemonized mode here instead of in an interactive terminal, we'd never have known the run failed unless we were looking for it. Let's look at `/tmp/handler_output` now and see what output was written by our exception handler (I've abbreviated directory paths with ... for formatting purposes):

```
$> cat /tmp/handler_output

Chef::Handler::AwesomeStartHandler: Run started on mynode.mydomain.com
  at 2014-02-22 14:05:06 +0000

Chef::Handler::AwesomeHandler: Run failed on
  mynode.mydomain.com after 0.034007263 seconds:

  Exception: could not find recipe shrubbery for cookbook testcookbook
  Backtrace:
    /usr/.../lib/chef/cookbook_version.rb:226:in `load_recipe'
    /usr/.../lib/chef/run_context.rb:151:in `load_recipe'
    /usr/.../lib/chef/run_context/cookbook_compiler.rb:139:in `block in compile_recipes'
    /usr/.../lib/chef/run_context/cookbook_compiler.rb:137:in `each'
    /usr/.../lib/chef/run_context/cookbook_compiler.rb:137:in `compile_recipes'
    /usr/.../lib/chef/run_context/cookbook_compiler.rb:74:in `compile'
    /usr/.../lib/chef/run_context.rb:86:in `load'
    /usr/.../lib/chef/client.rb:250:in `setup_run_context'
    /usr/.../lib/chef/client.rb:498:in `do_run'
    /usr/.../lib/chef/client.rb:199:in `block in run'
    /usr/.../lib/chef/client.rb:193:in `fork'
    /usr/.../lib/chef/client.rb:193:in `run'
    /usr/.../lib/chef/application.rb:208:in `run_chef_client'
    /usr/.../lib/chef/application/client.rb:312:in `block in run_application'
    /usr/.../lib/chef/application/client.rb:304:in `loop'
    /usr/.../lib/chef/application/client.rb:304:in `run_application'
    /usr/.../lib/chef/application.rb:66:in `run'
    /usr/.../bin/chef-client:26:in `<top (required)>'
    /usr/bin/chef-client:19:in `load'
    /usr/bin/chef-client:19:in `<main>'
```

Here we see that as expected, the `AwesomeStartHandler` was executed at the start of the run. This time however, when the run **failed**, our combined `AwesomeHandler` class wrote its *exception* handler output, triggered by checking `@run_status.failed?` and having it return `true`, indicating a failed Chef run.

In this case, our exception handler outputs the string representation of the `Exception` object that was thrown during the run and then prints the backtrace to tell us exactly where in the codebase the exception was triggered, but it could equally have posted a notification to a chat system like IRC or Campfire - exception handlers make it extremely simple for us to capture and utilize detailed information on any errors which occur during our Chef runs.

Handlers: Summary and Further Reading

In this chapter we've looked at the underlying `Chef::Handler` *superclass shared by all handlers and the _run status* object it provides that can be used to extract incredible detailed information about what happened during our Chef run. We've also looked at example implementations of each type of handler, but the flexibility of handlers combined with the huge variety of available monitoring and notification systems means that we can't possibly cover all possible handler implementations in this limited space.

For readers interested in deploying or implementing their own Chef handlers, or those just interested in exploring more example code, here I've listed some of the open sourced Chef handlers implemented by the Chef community:

Sensu Handler
: The Sensu handler provides *start* and *report* handlers for silencing checks on a node in the Sensu monitoring system while Chef is running.

Datadog Handler
: The Datadog handler provides *exception* and *report* handlers for feeding detailed information on your Chef runs to the Datadog monitoring service.

Graphite Handler
: The Graphite handler provides *exception* and *report* handlers for feeding the Graphite monitoring system with a number of metrics on Chef runs including successful and failed runs, average elapsed run time and number of resources updated.

IRCSnitch Handler
: The IRCSnitch handler provides an *exception* handlers for sending an IRC notification when a Chef run fails. The plugin also creates a private Github Gist containing information on the node, the exception that was thrown and the backtrace of the exception.

Nagios NSCA Handler
: The Nagios NSCA handler provides *exception* and *report* handlers to send notifications to the Nagios monitoring system when Chef runs fail or take an unusually long time to run.

To make deploying handlers easier without having to modify the `client.rb` file, Chef Inc also developed the chef_handler cookbook. This cookbook provides a resource to allow handlers to be configured in recipe code instead of hard coding them in `client.rb`, which can be particularly useful for open sourcing handlers as it allows you to distribute your handler with a recipe to enable or disable it.

Hopefully this chapter has served to give you a good understanding of how handlers integrate with the Chef run process to allow us to mine the behavior of Chef runs for useful information - deployed effectively, handlers are able to give us detailed insight into exactly how our Chef runs are performing, and make sure that we find out when they fail (and why) without us having to visually observe the output of every Chef run.

In the next chapter, we'll examine how Chef tracks the individual events which occur during runs, and how we can tap into this information to give us even deeper insight into how our infrastructure automation code is behaving under the hood.

CHAPTER 6

Extending Chef Run Events

Introduction

As we touched on briefly in Chapter 3 in the "Setup RunContext" on page 67 step, Chef builds up a comprehensive collection of the events which occur during the course of a run that it exposes through a system called the *event dispatcher*. The event dispatcher works on a what's known as a *publish / subscribe* model. This means that classes that wish to be notified of events which occur during the Chef run can **subscribe** to the event dispatcher, and each time the event dispatcher receives an event it **publishes** it to all registered subscribers - in fact the output you see displayed on your terminal when running chef-client or chef-solo manually is generated by a type of subscriber called a *formatter* which we'll be looking at later in this chapter.

In this chapter we'll look at how exactly the Chef *event dispatcher* works and an overview of the different types of subscriber we can create to make use of the event notifications it publishes - later sections in this chapter will examine how to actually implement these subscribers to allow us to tap into this detailed information resource to introspect even deeper into our Chef runs.

Before running the example code contained in this chapter, please ensure that you've followed the steps covered in "Preparing a Test Environment" on page 116 to prepare the test environment we'll use to run our examples. If you've already completed these steps and worked through the examples in Chapter 5, you can reuse the same test environment for this chapter.

Event Dispatcher Initialization

As we saw in "The Chef::Client Class" on page 86, the event dispatcher is created in the *initialize* method of the `Chef::Client` class with this line:

```
@events = EventDispatch::Dispatcher.new(*event_handlers)
```

Here we're creating a new instance of the `EventDispatch::Dispatcher` class which lives at `lib/chef/event_dispatch/dispatch.rb` in the Chef repository (Github Link) and passing in `*event_handlers` as a parameter to this object. The `*` character used before `event_handlers` is known as the "splat operator" and turns the `event_handlers` array into a list of parameters to be passed to the method.

The `event_handlers` parameter contains a list of *subscribers* to automatically register with the event dispatcher - the only subscriber to be automatically registered at this stage of the run is usually Chef's default *formatter* which will be used to write the run output to the screen if chef is being run in an interactive terminal by the user - we'll look at formatters in more detail later on in the chapter in "Creating Formatters" on page 142.

Now that we've seen how the event dispatcher is initialized, let's lift the hood and look at how it actually works. We'll first examine how the **publish** side of the event dispatcher works, before going on to look at an overview of how classes can **subscribe** to the event dispatcher to receive event notifications.

Publish Overview

To implement the *publish* side of the publish / subscribe system, the `EventDispatch::Dispatcher` class we met above inherits from a class called `EventDispatch::Base` which lives at `lib/chef/event_dispatch/base.rb` in the Chef repository (Github Link). This class defines a number of empty methods representing all of the event types which can occur during the Chef run, for example:

Example 6-1. Excerpt of lib/chef/event_dispatch/base.rb

```
def run_start(version)
end

def run_started(run_status)
end

# Called at the end a successful Chef run.
def run_completed(node)
end

# Called at the end of a failed Chef run.
def run_failed(exception)
end
```

As we see above, these event method definitions also specify the number and expected content of parameters required for each event type. When Chef adds events to the event collection during a run throughout the "Chef::Client class - do_run Method" on page 88 step, it does so with lines like this:

```
@events.run_start(Chef::VERSION)
@events.ohai_completed(node)
@events.run_completed(node)
@events.run_failed(e)
```

What we're actually doing here is calling methods like `run_start` and `ohai_completed` on the `EventDispatch::Dispatcher` object that was created earlier in the run - each of these method calls tells the event dispatcher that an event has occurred, and should be published to its subscribers.

To publish these events to its subscribers, the event dispatcher does something rather clever. All **subscribers** to the event dispatcher must also inherit from `EventDispatch::Base`. This shared *superclass* means that both publisher and subscriber define methods for all possible events. If a subscriber wants to define specific behavior for a particular event it may override the method inherited from `EventDispatch::Base`, but even if it doesn't it will still inherit the empty methods defined in `EventDispatch::Base`.

This means that for example when the `run_start` method of the `EventDispatch::Dispatcher` object is called, it in turn is able to call the `run_start` method of any objects which have registered as subscribers. Interested readers may wish to explore the code of the `EventDispatch::Dispatcher` class to look at exactly how this behavior is implemented - the entire class definition is only 41 lines of Ruby code.

Subscribe Overview

The simple yet effective strategy of subscribers inheriting from `EventDispatch::Base` means that if a subscriber wants to implement specific behavior when notified about events, it simply needs to *override* the method for that event inherited from `EventDispatch::Base`. If no specific action needs to be taken, the empty method from `EventDispatch::Base` will be used instead. This approach gives us the flexibility to implement subscribers which are able to respond to any number of events, while also ensuring that the subscriber only has to implement code for those events it cares about.

Understanding the theory behind how subscribers to the event dispatcher work is one thing, but how do we actually implement them? I've divided the types of subscriber we'll be implementing in this chapter into two categories. Both behave in similar ways - they are after all both subclasses of `EventDispatch::Base` - but there are also enough differences to warrant each having its own section.

Formatters

Formatters format and display the output of Chef run events to the screen when running in an interactive terminal. Only one formatter can be active at a time and is specified by passing the `-F` option to chef-client - in the event that the `-F` option is omitted, Chef will automatically use its default formatter. As we saw in "Event Dispatcher Initialization" on page 139, the active formatter is automatically sub-

scribed to the event dispatcher. We'll look at how to create and use your own formatters in the "Creating Formatters" on page 142 section of this chapter.

Other Subscribers
: To create a subscriber to the event dispatcher system which is **not** a formatter, the subscribing class must be registered manually with the event dispatcher using a *start handler*. We'll look at how to create custom *subscriber* and a start handler to register it in "Creating Custom Subscribers" on page 151.

Creating Formatters

In this section, we'll explore the various classes Chef provides for implementing formatters, and explore how to create our own formatter classes through the following topics:

- The "skeleton code" necessary to create a working formatter using Chef's built-in formatter classes
- Adding functionality to our formatter skeleton to handle Chef run events
- Creating a more complex plugin to completely customize the output of a Chef run

Before running the example code contained in this section, please ensure that you've followed the steps covered in "Preparing a Test Environment" on page 116 to prepare the test environment we'll use to run our examples. If you've already completed these steps and worked through the examples in Chapter 5, you can reuse the same test environment for this chapter.

Formatter Example 1: Skeleton Formatter

Just as we did when looking at how to implement ohai plugins, let's start off by looking at the absolute minimum code needed to implement a functional formatter (we're not going to customize the output it produces yet) which we'll paste into `/tmp/part2_examples/awesome.rb`:

Example 6-2. /tmp/part2_examples/awesome.rb

```
require 'chef/formatters/base' ❶

class Chef
  module Formatters ❷
    class Awesome < Formatters::Base ❸
      cli_name(:awesome) ❹
    end
```

```
  end
end
```

❶ We're telling our formatter class to require the `Chef::Formatters::Base` class which lives at `lib/chef/formatters/base.rb` in the Chef repository (Github Link). This is the base class which all formatters must inherit from, and inherits from `EventDispatch::Base` as any *event dispatcher* subscriber must.

❷ Here we're stating that our formatter class will live under the `Formatters` module in the `Chef` class - this makes its full namespace `Chef::Formatters::Awesome`. This is purely for organizational reasons, there is no functional reason why our formatter class needs to live inside this namespace, but it does help make the purpose of our class clearer.

❸ We're stating that our Awesome class inherits `Formatters::Base` - because our class also lives under the `Chef` class, we're able to use `Formatters::Base` instead of the full `Chef::Formatters::Base` name.

❹ This line calls the `cli_name` method inherited from `Chef::Formatters::Base` - this method tells Chef what name we'll use to request our formatter on the command line with the `-F` option. Without this method call, our formatter will not work.

Before we can test our skeleton formatter, we need to add a line to our `client.rb` file to tell it to `require` our new formatter class. Add the following line to `/tmp/part2_examples/client.rb`:

Example 6-3. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome.rb"
```

Now that we've examined our skeleton formatter code and added a `require` statement to our `client.rb`, we can perform a Chef run on our test environment using the same command we saw in (to come), but this time adding the `-F awesome` option to specify that we want to use our new `awesome` formatter:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default -F awesome
[Sat, 22 Feb 2014 14:08:49 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:08:49 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:08:49 +0000] WARN: Overridden Run List:
  [recipe[testcookbook]]
[Sat, 22 Feb 2014 14:08:49 +0000] WARN: In whyrun mode, so NOT performing node
save.
```

As we can see from the output of the command shown above, our chef run worked perfectly - it just didn't show much output. This is because the `Chef::Formatters::Base` class doesn't itself define any methods to handle events generated by the

event dispatcher, so of course nothing is printed to the screen when each event is fired - that's left up to us, and we'll look at how to do that in "Formatter Example 3 - Custom Event Methods" on page 145.

Formatter Example 2: Slightly Less Skeletal

If you intend to implement a formatter which handles the formatting and output of **all** event produced during a Chef run then `Chef::Formatters::Base` is a good place to start, but for many use cases we might only want to customize output for selected events, for example when resources are updated or skipped.

For these cases a more useful formatter class to inherit from is `Chef::Formatters::Minimal` which lives at `lib/chef/formatters/minimal.rb` in the Chef repository (Github Link). `Chef::Formatters::Minimal` defines methods which will display very simplistic output for each event - if we only want to define custom output for a few event methods, this avoids us having to implement unnecessary methods to handle those events we aren't directly interested in.

Let's change our `awesome.rb` class to inherit from `Chef::Formatters::Minimal` instead:

Example 6-4. /tmp/part2_examples/awesome.rb

```
require 'chef/formatters/minimal'❶

class Chef
  module Formatters
    class Awesome < Formatters::Minimal❷
      cli_name(:awesome)
    end
  end
end
```

❶ Note that we're now requiring `chef/formatters/minimal` instead of `chef/formatters/base`

❷ Note that our class now inherits from `Chef::Formatters::Minimal` instead of `Chef::Formatters::Base`

The only thing that we've changed in our `awesome.rb` formatter class is that it now inherits from `Chef::Formatters::Minimal` instead of `Chef::Formatters::Base` - we're still not defining any of our own methods to generate custom output for events. Let's try running our test chef run again and see how the output of the run has changed:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default -F awesome
Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 14:24:55 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:24:55 +0000] WARN: Original Run List: []
```

```
[Sat, 22 Feb 2014 14:24:55 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing cookbooks
.done.
Compiling cookbooks
done.
Converging 2 resources
.U

System converged.


resources updated this run:
* file[/tmp/part2_examples/testfile]
  - create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 14:24:55 +0000] WARN: Skipping final node save
  because override_runlist was given
chef client finished, 1 resources updated
```

As we see from the above output, by simply changing the *superclass* of our `awesome.rb` formatter we now see much more output from our Chef run. `Chef::Formatters::Minimal` defines some useful methods to to output when cookbook synchronization and compilation has taken place, and also display a list at the end of the run of all resources updated during the run. During the *converge* stage of the run, it prints extremely simple output to tell us when resources have been skipped (`S`), are up to date (`.`) or updated (`U`).

Now we've looked at two possible *superclasses* for our custom formatters, it's time to actually start defining our own methods to format event output!

Formatter Example 3 - Custom Event Methods

In this example, we're going to extend our `Awesome` formatter class with our own methods to handle the output from three specific events:

synchronized_cookbook
: This event method is called when a cookbook has been synchronized by the "Synchronize Cookbooks" on page 67 step

resource_up_to_date
: This event method is called when a resource was already up to date, ie it was already fully converged

resource_updated
: This event method is called when a resource has been updated, ie it was not already converged

As we learned in "Subscribe Overview" on page 141, all formatters are subclasses of the `EventDispatch::Base` class which defines a specific method for each event that can occur during a Chef run. This means that for each of the events above, we need to identify the method names from `EventDispatch::Base` we'll need to override in our formatter - why not have a look in `lib/chef/event_dispatch/base.rb` in the Chef repository (Github Link) and see if you can identify the methods we'll need to define for the above events before I list them below?

We'll be overriding the following methods from `EventDispatch::Base` in our example:

Example 6-5. Excerpt of lib/chef/event_dispatcher/base.rb

```
class Chef
  module EventDispatch
    class Base

      # Other methods removed

      # Called when cookbook +cookbook_name+ has been sync'd
      def synchronized_cookbook(cookbook_name)
      end

      # Called when a resource has no converge actions, e.g., it was already correct.
      def resource_up_to_date(resource, action)
      end

      # Called after a resource has been completely converged, but only if
      # modifications were made.
      def resource_updated(resource, action)
      end
    end
  end
end
```

We can see here that each of the methods that we'll be overriding from `EventDispatch::Base` takes a number of parameters - we'll be able to make use of these in **our** methods to add context to the output of our formatter class.

As we saw in our first two formatter examples, the *superclass* we choose to build our formatter on will dictate on how much output is shown by our chef run **in addition** to that which we explicitly define in our formatter class. To keep the output of our test runs as easy to read as possible, I've used `Chef::Formatters::Base` as the superclass of our example formatter here to ensure that the only output printed is that defined by the methods we're overriding.

Let's modify `awesome.rb` with our new overridden methods:

Example 6-6. /tmp/part2_examples/awesome.rb

```
require 'chef/formatters/base'

class Chef
  module Formatters
    class Awesome < Formatters::Base
      cli_name(:awesome)

      def synchronized_cookbook(cookbook_name)❶
        puts "\nCookbook #{cookbook_name} synchronized.\n"
      end

      def resource_up_to_date(resource, action)❷
        puts "#{resource.cookbook_name}::#{resource.recipe_name}"
        puts "  #{resource}:\n    Up to date, skipped action #{action}\n"
      end

      def resource_updated(resource, action)❸
        puts "#{resource.cookbook_name}::#{resource.recipe_name}"
        puts "  #{resource}:\n    Updated, performed action #{action}\n"
      end
    end
  end
end
```

❶ In this method, we're simply using `puts` to display the name of the cookbook which has been synchronized - this is passed to the method via the `cookbook_name` parameter

❷ In this method, we're printing several attributes of `resource` which is passed to the method as a parameter, and also the value of the `action` parameter.

❸ In this method we're printing the same attributes of `resource` and `action` that we used in `resource_up_to_date`, but the explanatory text we're outputting has been changed to reflect that the resource has been updated this time.

EventDispatch::Base method parameters

Although in the above example we're using mainly using the parameters of our methods as strings injected into a `puts` statement, in the `resource_up_to_date` and `resource_skipped` methods we're also calling several methods of the `resource` parameter like `cookbook_name` and `recipe_name`.

Under the hood, the `resource` parameter is an instance of `Chef::Resource` object which lives in `lib/chef/resource.rb` in the Chef repository (Github Link). Whilst it can certainly be treated like a string, it also defines a number of additional methods which can be used to give more context to our output.

As a general rule, it's safe to assume that parameters passed to methods in `EventDispatch::Base` can be treated as strings (hence included in `puts` statements), however examining the code of existing formatters such as `Chef::Formatters::Minimal` can offer insight into when additional context can be extracted from parameters via method calls.

Interested readers may also wish to examine the object types passed as parameters to method calls on the `@events` object we met in "The Chef::Client Class" on page 86 - these will be passed through to all subscribers as the same type that they were sent to the publisher.

Let's try a Chef run with our expanded `awesome` formatter to see our new methods in action:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default -F awesome
[Sat, 22 Feb 2014 14:25:56 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:25:56 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:25:56 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]

Cookbook testcookbook synchronized.

testcookbook::default
  directory[/tmp/part2_examples]:
    Up to date, skipped action create

testcookbook::default
  file[/tmp/part2_examples/testfile]:
    Updated, performed action create

[Sat, 22 Feb 2014 14:25:57 +0000] WARN: Skipping final node
  save because override_runlist was given
```

As we can see in the above output, when the three events for which we've defined methods in our `Awesome` formatter are called by the *event dispatcher* we now see our custom text in the output of the Chef run.

Setting the Default Formatter

Out of the box, the default formatter used by Chef when no `-F` option is passed to chef-client is the `doc` formatter, which lives at `lib/chef/formatters/doc.rb` in the Chef repository (Github Link). It is this formatter which produces the on-screen output we've all seen when performing chef runs. In all of the examples we've seen so far, we've specified our `awesome` formatter by passing the `-F` option to chef-client - but what do we do if we want to use our new formatter as the default instead of `doc`?

Chef allows us to choose the default formatter by adding a single line to our `client.rb` file. Let's do that now:

Example 6-7. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome.rb"
formatters [:awesome] ❶
```

❶ In this line, we're overriding the list of formatters that chef-client knows about. The formatter name given here must match that we passed to the `cli_name` method in our formatter class definition. If you specify more than one formatter in this list, they will all be used - this can produce duplicate results for events, but can also be used to augment an existing formatter without having to re-implement it.

Now let's try one more Chef run, this time without passing the `-F` option to chef-client:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default
[Sat, 22 Feb 2014 14:27:56 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:27:56 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:27:56 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]

Cookbook testcookbook synchronized.

testcookbook::default
  directory[/tmp/part2_examples]:
    Up to date, skipped action create

testcookbook::default
  file[/tmp/part2_examples/testfile]:
    Updated, performed action create

[Sat, 22 Feb 2014 14:27:57 +0000] WARN: Skipping final node save
  because override_runlist was given
```

As we see above, our one line change to `client.rb` means that our `awesome` formatter is now being used as the default for chef-client runs instead of `doc` and we no longer need to specify it using the `-F` option.

Formatters: Summary & Further Reading

In this section, we've covered the foundations of creating your own formatters and the various formatter base classes you can inherit from, but there are simply too many event methods and implementation possibilities for us to comprehensively cover everything here.

Readers interested in diving deeper into implementing formatters may wish to take a look at the following:

EventDispatcher::Base class
: As we've seen throughout this chapter, the `EventDispatch::Base` class which lives at `lib/chef/event_dispatch/base.rb` in the Chef repository (Github Link) is the definitive source for the list of event methods supported by formatters and other *event dispatcher* subscribers.

Doc Formatter class
: The `doc` formatter, which lives at `lib/chef/formatters/doc.rb` in the Chef repository (Github Link) is the default formatter used by Chef when no `-F` option is passed to chef-client. The `doc` formatter implements methods for a sizable proportion of the events defined in `EventDispatch::Base`, and is also an excellent reference guide to extracting information from the parameters passed to event methods.

Nyan-formatter
: For a slightly more fun formatter example, have a look at Andrea Campi's Nyan Cat Formatter. Andrea's formatter returns *rspec* style output for Chef runs in the style of a *nyan cat* meme and implements some clever behavior to assemble and correctly color the "rainbow" while still correctly indicating updated and skipped resources etc.

Chef-Spec Formatter
: A more minimal formatter example is included with the Chefspec unit testing framework, which you can find on Github. As it forms part of a unit testing framework, this formatter suppresses the output of nearly all events from the output of a Chef run except for errors. This formatter is an excellent reference guide to best practices for implementing customized handling of events which indicate errors in your Chef run - it's also an easy class to read through as the non-error event methods are entirely empty.

Hopefully this section has given you a taste of the flexibility and power that formatters provide us to customize the output of our Chef runs - as they can be optionally specified on the command line and do not affect the Chef server or the recipes run on your node, formatters are one of the safest Chef customizations to experiment with. But formatters are only half of the story when it comes to *event dispatcher* subscribers.

As we saw in "Subscribe Overview" on page 141, Chef also allows us to register our own *event dispatcher subscribers* to provide functionality such as reporting or event streaming of our Chef runs. In the next section, we're going to take the knowledge we've gained in this section about working with Chef events and expand that to look at how to create our own subscriber classes and register them with the event dispatcher.

Creating Custom Subscribers

As we touched on briefly in "Subscribe Overview" on page 141, the Chef *event dispatcher* system also allows us to implement our own custom subscribers to receive event notifications. These subscriber classes work in a similar fashion to the formatters we examined in the last section, as they share the `EventDispatch::Base` superclass and define behavior for specific events by overriding methods from this class. These subscriber classes are typically used to send data on Chef run events to external monitoring or metrics systems, but can be used in any scenario where you want to receive a notification of specific events occurring during a run, and trigger specific behavior with that notification

Although functionally similar to formatters, custom event dispatcher classes have two key differences:

- They must be manually registered with the *event dispatcher*, unlike formatters which are registered automatically when loaded by Chef
- They inherit directly from `EventDispatch::Base` rather than a class under the `Formatters` namespace like `Formatters::Base` or `Formatters::Minimal` as formatters do

In this section, we'll look at how to create a start handler to register our *event dispatcher* subscriber with the publisher, and how to create and run our own class to handle Chef event notifications.

Before running the example code contained in this section, please ensure that you've followed the steps covered in "Preparing a Test Environment" on page 116 to prepare the test environment we'll use to run our examples. If you've already completed these steps and worked through the examples in Chapter 5, you can reuse the same test environment for this chapter.

Subscriber Example 1: Skeleton

Before we define any custom event behaviors, as we did with previous examples let's look at the absolute minimum code necessary to implement our own *event dispatcher* subscriber. As with our skeleton formatter example, our custom subscriber class won't actually do anything yet - we're just putting the framework in place to customize it further in later examples in this chapter. We'll put the code for this class into `/tmp/part2_examples/awesome_subscriber.rb`:

Example 6-8. /tmp/part2_examples/awesome_subscriber.rb

```
require 'chef/event_dispatch/base' ❶

class AwesomeSubscriber < Chef::EventDispatch::Base ❷
end
```

❶ We're telling our subscriber class to require the `Chef::EventDispatch::Base` class which lives at `lib/chef/event_dispatch/base.rb` in the Chef repository (Github Link).

❷ Next we declare that our subscriber class is a subclass of `Chef::EventDispatch::Base`. As we saw in "Publish Overview" on page 140, both the event dispatcher *publisher* and all *subscribers* must inherit from this *superclass*.

Although we have not yet declared any custom event methods, the `AwesomeSubscriber` class now contains all of the code necessary to function as an *event dispatcher* subscriber. As with the formatters we looked at earlier in the chapter, the empty method definitions inherited from `EventDispatcher::Base` means that our class has all of the method definitions that it needs to function - it won't actually **do** anything, but it also won't throw any errors.

Before we can kick off a test run to try out our custom subscriber class however, we need to register it with the event dispatcher. For this, we're going to create a special *start* handler.

Subscriber Example 2: Registration with Start Handlers

To register our custom subscriber class with the *event dispatcher*, we're going to create a *start* handler using exactly the same techniques we examined in "Handler Example 1: Start Handler" on page 124. The main difference between this handler and the examples we saw earlier is that it will only register our subscriber class rather than sending notifications itself. Let's paste the code for our start handler into `/tmp/part2_examples/awesome_subscriber_start_handler.rb`:

Example 6-9. /tmp/part2_examples/awesome_subscriber_start_handler.rb

```
require "chef/handler"
require "/tmp/part2_examples/awesome_subscriber.rb"❶

class Chef
  class Handler
    class AwesomeSubscriberStartHandler < Chef::Handler❷
      def report❸
          event_dispatcher_subscriber = AwesomeSubscriber.new❹
          @run_status.events.register(event_dispatcher_subscriber)❺
      end
    end
```

```
  end
end
```

❶ In addition to requiring the `chef/handler` class we'll inherit from, we also need to require the `/tmp/part2_examples/awesome_subscriber.rb` class we created above.

❷ Just as in the handler examples we looked at in Chapter 5, our start handler inherits from the `Chef::Handler` superclass.

❸ Next, again as in our previous handler examples, we're overriding the `report` method defined in the `Chef::Handler` class to define our own behavior.

❹ Next we need to create a new instance of our `AwesomeSubscriber` class so we can register it with the event dispatcher.

❺ As we saw in "Runstatus Object" on page 121, the `events` method of the *run status* object returns the `EventDispatcher::Dispatch` object which forms the *publish* side of the event dispatcher. Here we're calling the `register` method on this object and passing as a parameter the `AwesomeSubscriber` object we created on the previous line - our subscriber is now registered with the *event dispatcher*.

Now that we've created both our subscriber class and a start handler to register it with the *event dispatcher*, we're ready to test it out and make sure that the run still works as we expect - but first we need to add our *start* handler to the `client.rb` configuration file. Paste the following lines into `/tmp/part2_examples/client.rb`, replacing any lines contained in that file from previous examples:

Example 6-10. /tmp/part2_examples/client.rb

```
require "/tmp/part2_examples/awesome_subscriber_start_handler.rb"

awesome_subscriber_start_handler = Chef::Handler::AwesomeSubscriberStartHandler.new

start_handlers << awesome_subscriber_start_handler
```

Exactly as we did for the example handler we created in "Handler Example 1: Start Handler" on page 124, in our `client.rb` configuration file we first create an instance of our `Chef::Handler::AwesomeSubscriberStartHandler` class and then add it to the `start_handlers` array. Note that we're not requiring or instantiating our `AwesomeSubscriber` class here - that's handled by our *start* handler.

Although our `AwesomeSubscriber` does not define any event methods (hence will not do anything), let's kick off a test chef run to verify that everything still works as it should. We'll use the same command that we've been using for our other examples:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default
Starting Chef Client, version 11.10.4
```

```
[Sat, 22 Feb 2014 14:50:41 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:50:41 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:50:41 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 14:50:41 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
Running handlers complete

Chef Client finished, 1/2 resources would have been updated
```

As we see above, nothing in the above output visibly indicates that our event dispatcher subscriber has been initialized or received notifications, but that's what we'd expect at this stage as the `AwesomeSubscriber` class is still using the empty method bodies inherited from `EventDispatcher::Base`. At this stage, we just wanted to check that our subscriber didn't cause any errors during the run. Now that we've validated that our subscriber class works, let's add some event methods to it so that we can see it in action.

Subscriber Example 3: Custom Event Methods

To have our subscriber class respond to events published by the *event dispatcher*, as with the formatter example we looked at in "Formatter Example 3 - Custom Event Methods" on page 145 we need to override some of the methods defined in `EventDispatcher::Base`. A full listing of all supported event methods can be located in `lib/chef/event_dispatch/base.rb` in the Chef repository (Github Link), but for the this example we're going to override the following methods:

run_started
: This event method is called when the chef run has started, after the "Load and Build Node Object" on page 66 step has been completed.

resource_up_to_date
: This event method is called when a resource was already up to date, ie it was already fully converged

resource_updated
: This event method is called when a resource has been updated, ie it was not already converged

run_completed
: This event method is called when the chef run has successfully completed.

run_completed
: This event method is called when the chef run has failed

As with other examples in this chapter, I want to keep the behavior of our example subscriber as simple as possible so that readers can try it out for themselves without the need to set up third party monitoring systems etc. For this reason, our custom subscriber class will stick to the technique I've used throughout this chapter, and write its output to a local file - `/tmp/subscriber_output`. Let's paste the following expanded class definition into `/tmp/part2_examples/awesome_subscriber.rb` (some lines split over two lines for formatting purposes):

Example 6-11. /tmp/part2_examples/awesome_subscriber.rb

```
require "chef/event_dispatch/base"

class AwesomeSubscriber < Chef::EventDispatch::Base

  # Method to write a message string to our output file
  def write_to_file(message)❶
    File.open('/tmp/subscriber_output', 'a') do |f|
      f.write("#{self.class} #{Time.now}: #{message}\n")
    end
  end

  # Methods overridden from Chef::EventDispatch::Base

  def run_started(run_status)
     write_to_file("run_started: Run started")
  end

  def converge_start(run_context)
      write_to_file("converge_start: Coo")
  end

  def converge_complete
  end

  def resource_up_to_date(new_resource, action)
      write_to_file("resource_up_to_date: Resource #{new_resource} " +
      "action #{action} already up to date")
  end

  def resource_updated(new_resource, action)
      write_to_file("resource_updated: Resource #{new_resource} was " +
```

```
      "updated with action #{action}")
  end

  def run_completed(node)
    write_to_file("run_completed: Run completed")
  end

  def run_failed(exception)
    write_to_file("run_failed: Run failed with #{exception}")
  end

end
```

❶ The `write_to_file` method is the only method defined in our `AwesomeSubscriber` class which is not overriding an event method defined in `Chef::EventDispatch::Base`. It is used to provide a simple, reusable method for writing a string to the `/tmp/subscriber_output` file, prefixed with the name of the class and a timestamp.

The example code here is very similar to that used in our formatter examples earlier in the chapter - after all, both subscriber types override the same methods from `EventDispatcher::Base` and take the same parameters. Let's try one more test Chef run and see how our `AwesomeSubscriber` class behaves now that we've overridden some event methods:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part2_examples/client.rb \
  --override-runlist testcookbook::default
Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 14:51:34 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 14:51:34 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 14:51:34 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part2_examples] action create (up to date)
  * file[/tmp/part2_examples/testfile] action create
    - Would create new file /tmp/part2_examples/testfile

[Sat, 22 Feb 2014 14:51:34 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
Running handlers complete

Chef Client finished, 1/2 resources would have been updated
```

Just as we saw with our skeleton subscriber class, nothing in the above output visibly indicates that our event dispatcher subscriber has been initialized or received notifications - but let's take a look in the `/tmp/subscriber_output` file:

Example 6-12. /tmp/subscriber_output

```
AwesomeSubscriber 2014-02-22 14:51:34 +0000: run_started: Run started
AwesomeSubscriber 2014-02-22 14:51:34 +0000: resource_up_to_date:
  Resource directory[/tmp/part2_examples] action create already up to date
AwesomeSubscriber 2014-02-22 14:51:34 +0000: resource_updated:
  Resource file[/tmp/part2_examples/testfile] was updated with action create
AwesomeSubscriber 2014-02-22 14:51:34 +0000: run_completed: Run completed
```

As we see here, when each method is published by the *event dispatcher*, the corresponding method of our `AwesomeSubscriber` subscriber class is called, and a line is output to the `+/tmp/subscriber_output` file.

Custom Subscribers: Summary

As mentioned in previous sections, the custom subscriber examples we've looked at here have been kept intentionally simple, but the techniques used can easily be adapted to create more complex subscriber classes. The comprehensive list of event methods defined in `EventDispatch::Base` means that custom subscriber classes can be utilized to provide metrics on your chef runs that aren't exposed to more typically used methods such as handlers - these metrics can help you answer questions like:

- How long it takes your nodes to register with the Chef server at the start of the run?
- Exactly how long does it take Chef to get a list of cookbook versions from the server?
- Once the client has that list, how long does the cookbook synchronization process take?

Readers interested in looking at a real-world example of a custom subscriber class may wish to dive into the chef-reporting gem created by Chef Inc. to allow older Chef client versions to send reporting data to Enterprise Chef for use with it's reporting feature (this feature is not provided with open source Chef Server). The code is of course fairly specific to Chef's reporting system, but gives a good idea of how to implement a more complex event dispatcher subscriber that sends event data to an external system.

Revisiting AwesomeInc - Which Customization?

Throughout the course of Part 2, we've looked a number of different possibilities for customizing different aspects of the Chef run process. But when you're trying to solve a problem that could be solved by several of these customizations, which do you pick? To try and assist that decision making process, let's revisit one of the problems that AwesomeInc had identified in "Criteria for Customization" on page 9:

How do we find out when our Chef runs are failing and why?

As we've explored throughout these chapters, Chef exposes data on failing runs to nearly all of the customization types we've looked at here except for ohai plugins. So should AwesomeInc pick an exception handler? A formatter? A custom event dispatcher subscriber? At this stage, without knowing more about exactly what AwesomeInc wants to get from Chef and how they want to expose it the best answer we could give them is **it depends**.

If you cast your minds back to "Think Critically" on page 7, I said that a large part of effectively customizing your infrastructure code is choosing the best solution to add value to your business - this may be the same solution that everybody else uses, but it may also be a much more bespoke solution specific to your infrastructure and business needs. I **could** simply say "well, clearly AwesomeInc should use an exception handler", but that's not going to help you understand **why** that may or may not be the best solution for them. To do that, we need a little more information about Mike's team at AwesomeInc and how they operate:

- AwesomeInc mainly utilize Nagios and Graphite to monitor their systems - they have checks and graphs for most of the critical services and components of their infrastructure, but there are some known monitoring blindspots - they're not looking to implement anything too complex at the moment until they finish their ongoing project to eliminate these blindspots.
- As AwesomeInc are currently all located in the same office, they mainly communicate face to face or via email. They've started to experiment with Chat systems, as they expect to need to hire international ops engineers and developers as the business expands.
- The team at AwesomeInc are not looking for an extremely granular view of their Chef runs - at the moment all they want is to easily and quickly identify when runs fail and what caused the failure.

With this new information, we're now better able to look at making a recommendation to AwesomeInc. In theory, it would be possible for them to extract information on failing runs from three of the customization types we've looked at in this chapter - formatters, custom event dispatcher subscribers and handlers. Let's look at each of these possible solutions in turn and explore which might fit AwesomeInc's needs most effectively:

Formatters
: Although it would be possible to create a formatter to both print output to screen **and** create a separate alert when a run failed, this is not really the purpose for which formatters were intended - as we saw earlier in this chapter, formatters are designed to format the event notifications published by the event dispatcher and display them to the screen when Chef is being run in an interactive terminal.

Solving this problem with a formatter would definitely be **possible** but there are more suitable methods we could make use of - remember the SMVMS criteria we looked at in "Criteria for Customization" on page 9. To use a formatter to solve this problem, AwesomeInc would have to create an entirely new formatter class and add their custom alerting behavior alongside the existing formatter behavior for printing text to screen. This would mean that AwesomeInc could no longer use Chef's default formatter, and would have to maintain their own. When we consider this alongside the fact that AwesomeInc are only really interested in a single run event - that of a failed run - a formatter really doesn't seem like the best solution to this problem.

Custom Event Dispatcher Subscriber

A custom event dispatcher subscriber class could be a much more attractive option for AwesomeInc - as we saw earlier in the chapter, event dispatcher subscribers are intended for tasks such as sending information on Chef events to reporting and monitoring systems. Although subscriber classes and formatters share the same *superclass*, subscribers are intended specifically for the sort of thing AwesomeInc are looking to do, and can co-exist perfectly happily alongside formatters and any other subscribers to the event dispatcher.

Although certainly more suitable than formatters, an event dispatcher subscriber is still in my opinion not the best solution to AwesomeInc's problem. Remember as we saw above, AwesomeInc are not currently looking to analyze their Chef runs in fine detail, they just want to know when runs have failed and why - this mean using just one of the event methods that can be defined in a custom subscriber class. AwesomeInc would have to create an event dispatcher subscriber class which generated a notification when the single run_failed event they're interested in was published, and a start handler to initialize and register the subscriber class. Given that a simpler and more lightweight solution exists to capture the specific information that AwesomeInc are interested in, in this instance I would not recommend a custom subscriber class as the best solution either.

Handlers

Based on the information we know about AwesomeInc's infrastructure and requirements, I would recommend that they explore using an exception handler to solve the problem they're experiencing. An exception handler would provide AwesomeInc with a *modular* solution to the problem which integrates with Chef and does not affect the functioning of any other components.

Creating a handler is also the *simplest* option in this case as once the handler class has been created, it can be added to the `exception_handlers` list in the Chef configuration file and will work straight away. No start handlers are needed, and no existing Chef functionality is being reimplemented to provide notifications. Exception handlers are also a more lightweight solutions for this purpose than the

other options we're considered - the single function of exception handlers is after all to respond to run failures.

In addition to the above, an important consideration is the number of open sourced handler implementations which already exist as we saw in "Handlers: Summary and Further Reading" on page 137. AwesomeInc would be able to implement or modify community created handlers which support both Email and a number of common chat systems, increasing the likelihood AwesomeInc will not have to write a new handler class from scratch themselves. Although using community customizations is not always possible, it's a valuable time saving option worth considering.

In the above example scenario, creating a handler emerged as the customization which is best suited to both solving AwesomeInc's problem given their **specific** use case and situation. If AwesomeInc's requirements had been different however, our recommendation may well have been different too. If, for example, AwesomeInc had been looking to create a full suite of metrics analyzing the performance and behavior of the Chef runs rather than only capturing run failure, we may have recommended that they create an event dispatcher subscriber class.

The purpose of this exercise was not to advocate any of the customizations covered in these chapters over any other, but to demonstrate the sort of decision making process that goes into evaluating and choosing the best customization to implement. It's very rare that we look to perform a task with Chef and discover that there is only one possible solution - more often than not, we will need to evaluate a number of different possibilities as we did here and choose that which best meets our needs, and the needs of our businesses. As we progress through the material in this book, we'll carry out this exercise again to evaluate possible solutions to some more of AwesomeInc's problems.

Summary

In Part 2, we've explored a number of different ways to customize different aspects of our Chef runs. We've looked at how to inject additional node attributes with Ohai plugins, how to create handlers to respond to particular situations during the Chef run, and how to leverage the event dispatcher to create our own formatters and subscriber classes before revisiting AwesomeInc to evaluate solutions for how they might gain more visibility into the behavior of their Chef runs.

In the Part 3, we're going to look at the different ways that Chef provides for us to customize our cookbooks - as we've already seen, Chef is extremely extensible and provides a number of ways in which to define new resources and providers to use in our cookbooks, and libraries to support them.

PART III
Customizing Recipes

In part 3 of this book, we're going to learn about the different ways Chef provides for us to customize our cookbooks and recipes by creating our own resource definitions and the code to support them. We'll learn

- How to create definitions
- How to create lightweight resources and providers
- How to create heavyweight resources and providers
- How to create libraries to support our resources
- The pros and cons of each customization type.

Up to this point, the majority of the customization types we've looked at have been largely "monotasked", that is they have a clear function or use case which guides when they are likely to be used and dictates your likely customization choice.

Recipe customizations on the other hand are not so easily separated. The chances are good that for any scenario in which you wish to create a custom resource, you could implement more than one of the customization types covered in this part of the book. Choosing the best customization type to use is extremely important, especially when considering the criteria we looked at in "Criteria for Customization" on page 9, so we'll cover this in much more detail than we have for other customization types.

We'll close off this part of the book by revisiting our friends at AwesomeInc to look at how they might solve the second problem they identified in "Criteria for Customization" on page 9 by choosing the best variety of recipe customization for their purposes.

CHAPTER 7

Introduction & Test Environment

Before we dive in to creating the various different types of recipe customization we'll be looking at in this part of the book, we're going to revisit the structure of Chef cookbooks and how the different components we'll be learning about fit together along with preparing a test environment to allow us to try our our customizations safely. In this chapter we're going to look at:

- How Chef cookbooks are structured and the different components which comprise them
- How Chef implements the out of the box resources and providers it ships with and how we can tap into that functionality
- How to create a test environment for trying out our cookbook and recipe customizations

Cookbook Structure

Let's start off by taking a look at the typical structure of a Chef cookbook and its component parts. The below directory listing shows the default directories created when running the `knife cookbook create` command - I've removed README files etc for clarity.

Example 7-1. Example Cookbook Structure

```
.
├── attributes
├── definitions❶
├── files
│   └── default
├── libraries❷
├── providers❸
```

```
├── recipes
│   └── default.rb
├── resources❹
└── templates
    └── default
```

❶ The `definitions` folder contains the simplest type of resources Chef allows us to create, *definitions*. Typically used as wrappers around frequently used resource combinations, or when passing data from multiple recipes into the same resource. We'll look at definitions in more detail in (to come)

❷ The `libraries` folder contains supporting libraries that allow us to add new classes to be used in our recipes or other resources, or to extend the functionality of existing Chef classes. We'll learn about libraries in more detail in Chapter 11.

❸ The `providers` folder allows us to define our own provider classes to support custom resources. An example of the relationship between resources and providers is the `package` resource which ships with Chef. Behind the scenes, the package resource is supported by multiple providers to allow the same `package` resource to be used in our recipe code, but also support multiple packaging systems such as yum, apt, and macports. We'll learn about creating our own providers in Chapter 9 and Chapter 10.

❹ The `resources` folder allows us to define our own resource types to be used in our recipes to implement our own custom functionality, backed by supporting providers. We'll learn about creating our own resources in Chapter 9 and Chapter 10.

In each chapter, alongside with how to actually create these customizations we'll examine the typical use cases for each and how to decide which customization type might be best for a particular use case. Before we dive into our customizations however, we need to create a test environment so that we can safely try them out.

Creating a Test Environment

To enable us to test our cookbook customizations, we're going to make use of the test environment we created for Part 2 of the book in "Preparing a Test Environment" on page 116. If you haven't already created this test environment and would like to run the examples in this chapter, please follow the steps in that section before moving on.

Next, we're going to make a copy of the test environment from Part 2 so that we can test our cookbook changes in isolation from our earlier customization examples. Run the following command to make a new copy of the test environment:

```
$> cp -R /tmp/part2_examples /tmp/part3_examples
```

Let's double check that our fresh copy of the test environment still works correctly. Run the following command from `/tmp/part3_examples`:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part3_examples/client.rb \
  --override-runlist testcookbook::default

Starting Chef Client, version 11.10.4
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Run List override has been provided.
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Original Run List: []
[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::default]]
resolving cookbooks for run list: ["testcookbook::default"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::default
  * directory[/tmp/part3_examples] action create (up to date)
  * file[/tmp/part3_examples/testfile] action create
    - Would create new file /tmp/part3_examples/testfile

[Sat, 22 Feb 2014 13:48:32 +0000] WARN: Skipping final node save
  because override_runlist was given

Running handlers:
Running handlers complete
```

If everything's working correctly, you should see the above output which lets us know that our chef run successfully executed the `testcookbook::default` cookbook in *why run* mode.

As with the test environments we created in previous chapters, we're using *why run* mode here to ensure that we don't actually execute any experimental code on our node and *local mode* to ensure that we can safely run experimental code against a mock Chef server without touching your production Chef setup.

Part Outline

- Chapter 7
  — Intro and test env set up.
- Chapter 8
  — Definitions
- Other Chapters
- LWRP & HWRP
- Libraries
- When to use each

- AwesomeInc
- Case studies

CHAPTER 8
Definitions

In Chapter 7, we looked at an overview of how Chef cookbooks are structured and where the various cookbook customization types live, so now it's time to dive into how to actually create them. In this chapter, we're going to learn about the simplest type of recipe customization, definitions. We'll examine:

- What exactly definitions are
- The code structure shared by all definitions
- The strengths and weaknesses of definitions, and when you may or may not want to use them

Throughout the chapter, we'll also look at a number of examples demonstrating the functionality and features of definitions.

What is a Definition?

Before we learn how to create our own definitions, let's pause for a moment to examine in a little more detail exactly a definition is and how it differs from the default resource blocks we're already familiar with such as `package` and `cookbook_file`. In Chef, a recipe resource such as `package` comprises two components - a resource definition which describes the characteristics of that resource and one or more providers to provide its underlying functionality. These resources and providers can be implemented using a lightweight DSL, or as fully fledged Ruby code. These are the *lightweight* and *heavy-weight* resources and providers we will be learning about in later chapters.

A *definition* on the other hand is a wrapper around an existing resource or group of resources which allows us to treat those resources as a single whole. Imagine taking a chunk of recipe code which you find yourself repeating throughout your cookbooks

and assigning that code chunk a name which you could use to represent that code each time you wanted to use it. This is exactly the functionality provided by definitions.

In programing parlance, this is what we might think of as a recipe *macro*. Rather than defining an entirely new set of behaviors with a new resource type, we're simply creating a wrapper around already existing resources to group them together into a single *macro* for convenience and reuse.

Definitions in Chef are marked by several characteristics:

- Definitions are always stored in the `/definitions` folder of cookbooks.
- A definition will create a new resource which groups 1 or more other resources together.
- A definition can not receive notifications, but the resources it contains can notify other resources.

Let's take a look at these characteristics in a little more detail.

Definitions are always stored in the `/definitions` *folder of cookbooks*
: As we saw in "Load Cookbook Data" on page 69, definitions are the last part of cookbooks to be loaded into memory other than recipes themselves - they live in the `/definitions` folder of cookbooks, and are treated as single blocks of code when executed by Chef rather than Ruby classes like fully fledged resources and providers. Definitions are loaded at the end of the "Load Cookbook Data" on page 69 stage because they may wrap other resources created during the earlier steps of this stage.

A definition will create a new resource which groups 1 or more other resources together
: Definitions in Chef are designed to fulfill a very specific purpose, which is to allow us to create a wrapper around a resource or resources to allow us to treat them as a single reusable block of code. Definitions should only contain code that would work equally well if it lived in the actual recipe.

  During the Chef run, a definition will be shown in the output of the run in fully expanded form, that is to say that its component resources will be shown in the output of the run but the name of the definition itself will not. We'll see an example of this later on in the chapter in "Definition Example 3 - Adding Resources" on page 175

A definition can not receive notifications, but the resources it contains can notify other resources
: Because definitions are intended to be used as a recipe *macro* rather than as a fully fledged new resource, they incur one very specific limitation - they cannot be notified or subscribed to by other resources. If we consider a notification example like this:

```
notifies :restart, resources(:service => "awesomeator")
```

what is actually happening here is that the `service` resource called `awesomeator` is being notified that it should carry out its `restart` action. Since definitions simply serve as wrappers around other resources, they don't actually define any actions hence cannot themselves be notified.

These characteristics serve to give a particularly clear indication as to when creating a definition might or might not be appropriate. If you find yourself using a definition to define behavior that cannot be easily constructed using existing resources or which might later want other resources to *notify*, you should probably be looking at implementing Lightweight or Heavyweight resource providers, which will be covered in subsequent chapters.

If you're looking to construct a reusable wrapper around a frequently used collection of resources and recipe code, definitions may be a good option. We'll look at some more detailed case studies to demonstrate this sort of decision making process with the aid of our friends at AwesomeInc at the end of this part of the book once we've learned about each customization type.

Definition Example 1: Skeleton

Now that we've examined the characteristics of a definition in Chef, let's dive into the code. Throughout this chapter, we'll use a series of examples to construct a definition which will solve a fictional scenario. Our friends at AwesomeInc have created a tool called "awesomeator" and are creating a cookbook to install and configure it. Before installing the tool, it is necessary to create a configuration file and directory structure for the tool to store its data in - rather than having the tool use a default directory, AwesomeInc have decided to make this configurable at install time, and want to create a definition to allow them to treat these preparatory steps as a single resource.

Throughout this chapter, we're going to incrementally build up a definition to solve AwesomeInc's problem which we'll call `awesomeator_prepare`.

Before continuing with this example, you should ensure that you have followed the steps in "Creating a Test Environment" on page 164 to create a test environment in which to run our example code.

We'll start by looking at the minimum code needed to create our `awesomeator_prepare` definition. As we saw earlier in the chapter, all definitions need to live under the `definitions` folder in cookbooks. The name of the definition file itself doesn't matter, but it's good practice to name the file the same as the definition it contains so we're going to paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb`:

Example 8-1. /tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb

```
define :awesomeator_prepare do
end
```

As we see in the above code listing, the code needed to create a definition is actually extremely minimal - we create a `define` block, with the name of our definition as its symbol. We haven't actually made our definition **do** anything yet, but let's try using it in a recipe now to prove that the above code does in fact work.

We'll create a new recipe for running the examples in this chapter under our `testcookbook` cookbook called `awesomeator.rb` - paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/recipes/awesomeator.rb`

Example 8-2. /tmp/part4_examples/cookbooks/testcookbook/recipes/awesomeator.rb

```
awesomeator_prepare "foo"
```

Here we've simply added a single `awesomator_prepare` resource to our recipe with the name "foo". Now let's try performing a Chef run with our `testcookbook::awesomeator` recipe and see what happens:

```
$> $ chef-client --once --why-run --local-mode list --config /tmp/part3_examples/client.rb  \
  --override-runlist testcookbook::awesomeator
Starting Chef Client, version 11.10.4
[Thu, 24 Apr 2014 10:04:11 +0000] WARN: Run List override has been provided.
[Thu, 24 Apr 2014 10:04:11 +0000] WARN: Original Run List: []
[Thu, 24 Apr 2014 10:04:11 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::awesomeator]]
resolving cookbooks for run list: ["testcookbook::awesomeator"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 0 resources
[Thu, 24 Apr 2014 10:04:11 +0000] WARN: Skipping final node save because
  override_runlist was given

Running handlers:
Running handlers complete

Chef Client finished, 0/0 resources would have been updated
```

As we see in the above output, our Chef run completed successfully. No resources were updated because our definition does not itself contain any resources yet, but the run also didn't throw any errors - we've just created and executed our first custom definition! Next, let's look at how we actually make our definition do something a little more useful.

Since we want our `awesomeator_prepare` definition to configure a directory structure and configuration file for the `awesome` tool, it would be useful to be able to instruct our definition where we want these files and directories to live.

Adding Parameters

As with the default resource provided with Chef, definitions wouldn't be especially useful if we couldn't pass them necessary configuration information as parameters - the whole point of a reusable pattern, after all, is that we're able to use it more than once! In this section, we'll look at how to make use of parameters in our definitions to provide them with the information they need to execute the resources they contain.

Definitions in Chef support parameters through the use of a built-in `Hash` called `params`. Although we didn't explicitly do so, we've actually already used this hash - if we cast our minds back to "Definition Example 1: Skeleton" on page 169, when we used the `awesomeator_prepare` resource in our recipe, we gave it the name "production":

```
awesomeator_prepare "production"
```

The name we give to our definitions when we use them in recipes is automatically made available to the code for that definition as `params[:name]` - assigning a name to our definition is optional however, and can be omitted. We can also easily add any other parameters we wish to the definition when we use it in recipe code:

```
awesomeator_prepare "production" do
  working_dir "/tmp/awesomeator"
  config_file "/etc/awesomeator.conf"
end
```

These parameters will automatically be available to the code of our definition as `params[:working_dir]` and `params[:config_file]`. We don't need to explicitly define parameters we wish to use in our definition because Chef automatically adds any parameters it finds to the `params` hash when our definition is executed.

Chef also allows us to specify default values for specific parameters when we define our definitions which will be used if no value is given to that parameter in the recipe. Let's look at another example to allow our `awesomeator_prepare` definition to make use of these `working_dir` and `config_file` parameters.

Definition Example 2 - Using Parameters

As we saw above, definitions wouldn't be of much use when creating reusable and modular recipe macros if we weren't able to configure them with parameters. Let's extend the definition we created in "Definition Example 1: Skeleton" on page 169 to allow it to make use of two parameters - `working_dir` and `config_file`.

We'll also set default values for both parameters to make sure that our definition can behave sensibly if no values are given for those parameters. Paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb`:

Example 8-3. /tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb

```
define :awesomeator_prepare,
       :working_dir => '/tmp/awesomeator',
       :config_file => '/etc/awesomeator.conf' do ❶
 puts "working_dir = #{params[:working_dir]}" ❷
 puts "config_file = #{params[:config_file]}"
end
```

❶ In addition to giving `:awesomeator_prepare` as the name of our definition, here we're setting default values for the `:working_dir` and `:config_file` parameters. We can still set values for these parameters in our recipes, but if we don't then the defaults will be used. Note that if we're not setting a default value for a parameter we can ommit the parameter from our `define` line - Chef will automatically add it to the `params` hash.

❷ Here we're simply using `puts` to write the value of `params[:working_dir]` to the screen and doing the same on the following line with `params[:config_file]`.

Now that we've added some parameters with default values and output statements to our definition, let's try performing our Chef run again - note here we're not changing anything in the `testcookbook::awesomeator` recipe, only the definition code itself has been altered.

```
$ chef-client --once --why-run --local-mode list --config /tmp/part3_examples/client.rb \
  --override-runlist testcookbook::awesomeator
Starting Chef Client, version 11.10.4
[Thu, 24 Apr 2014 12:41:18 +0000] WARN: Run List override has been provided.
[Thu, 24 Apr 2014 12:41:18 +0000] WARN: Original Run List: []
[Thu, 24 Apr 2014 12:41:18 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::awesomeator]]
resolving cookbooks for run list: ["testcookbook::awesomeator"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
working_dir = /tmp/awesomeator
config_file = /etc/awesomeator.conf
Converging 0 resources
[Thu, 24 Apr 2014 12:41:18 +0000] WARN: Skipping final node save because
  override_runlist was given

Running handlers:
```

```
Running handlers complete

Chef Client finished, 0/0 resources would have been updated
```

In the above output immediately after the "Compiling Cookbooks" log line we see that the two `puts` statements we added to our definition have been executed, printing the default values for the `working_dir` and `config_file` parameters that we specified in our definition code.

Let's now set a different value for the `working_dir` parameter when we call the `awesomeator_prepare` resource in our `awesomeator` recipe to see how the output changes. Paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/recipes/awesomeator.rb`

Example 8-4. `/tmp/part4_examples/cookbooks/testcookbook/recipes/awesomeator.rb`

```
awesomeator_prepare "production" do
  working_dir "/var/awesomeator"
end
```

We're still using the `awesomeator_prepare` resource in the same way as we would any other built-in resource, but this time we're adding a value for the `working_dir` parameter. Let's perform the same Chef run again and see how the output changes this time:

```
$ chef-client --once --why-run --local-mode list --config /tmp/part3_examples/client.rb  \
  --override-runlist testcookbook::awesomeator
Starting Chef Client, version 11.10.4
[Thu, 24 Apr 2014 12:50:30 +0000] WARN: Run List override has been provided.
[Thu, 24 Apr 2014 12:50:30 +0000] WARN: Original Run List: []
[Thu, 24 Apr 2014 12:50:30 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::awesomeator]]
resolving cookbooks for run list: ["testcookbook::awesomeator"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
working_dir = /var/awesomeator
config_file = /etc/awesomeator.conf
Converging 0 resources
[Thu, 24 Apr 2014 12:50:30 +0000] WARN: Skipping final node save because
  override_runlist was given

Running handlers:
Running handlers complete

Chef Client finished, 0/0 resources would have been updated
```

This time we see that the value of `working_dir` has been taken from the parameter we passed in our recipe (`/var/awesomeator`) but the value of `config_file` is still the default

set in the definition code (`/etc/awesomeator.conf`) as we didn't pass in an explicit value for this parameter in the recipe code.

Our definition is starting to resemble a solution to AwesomeInc's problem now. We've learned how to define a definition and use it in our cookbooks, how to pass it parameters and make use of default values, but we haven't actually added any resources to our definition yet. Next we'll look at how to add resources into our definition to build up the recipe macro concept that we discussed earlier in this chapter.

Adding Resources

So far in this chapter we've looked at the skeleton code needed to create a definition and how to supply configuration parameters to it but before our definition will be of much practical use we need to actually add some resources to it. Because definitions are essentially recipe *macros* as we discussed earlier in the chapter, the code we add to our definitions will be nearly identical to the code we would use to carry out the same tasks in our recipes.

If, for example, we wanted to add a `directory` resource to our `awesomeator_prepare` definition, we could do so like this:

```
define :awesomeator_prepare,
       :working_dir => '/tmp/awesomeator',
       :config_file => '/etc/awesomeator.conf' do

  directory params[:working_dir] do ❶
    action :create
    recursive true
  end
end
```

❶ Here we're using the value of `params[:working_dir]` as the name of the directory to be created by our `directory` resource.

It's also possible to use arbitrary Ruby code and resources inside the body of a definition just as we would in a recipe, however definitions are intended to be wrappers around simple and reusable chunks of recipe code. If you find yourself adding significant amounts of native Ruby code to a definition or too many complex cookbook resources, this may be an indication that you should consider creating an LWRP or HWRP instead (we'll learn about these in Chapter 9 and Chapter 10).

It's worth noting once again at this point that definitions cannot be notified by other resources. Although the `awesomeator_prepare` definition we created here can not itself be notified however, resources defined inside it can of course notify **other** resources just

as they would in recipes. Let's say that we wanted the `directory` resource we added above to send a `restart` notification to a fictional `awesomeator` service. We could do something like this:

```
define :awesomeator_prepare,
       :working_dir => '/tmp/awesomeator',
       :config_file => '/etc/awesomeator.conf' do

  directory params[:working_dir] do
    action :create
    recursive true
    notifies :restart, resources(:service => "awesomeator")
  end
end
```

Now that we've learned how to add resources to our definitions and the restrictions imposed by the *recipe macro* model, it's time to look at a final example to extend our definition with the resources needed to carry out the preparatory steps for the installation of AwesomeInc's tool.

Definition Example 3 - Adding Resources

Before we can add resources to our definition, we need to define what exactly we want our definition to do. AwesomeInc have identified the following steps that should be completed to prepare the tool's working environment:

- The working directory needs to be created
- The specified configuration file should be created, and populated with the chosen working directory.

As both of these steps can be performed with the default resources supplied with Chef, let's paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb`:

Example 8-5. /tmp/part4_examples/cookbooks/testcookbook/definitions/awesomeator_prepare.rb

```
define :awesomeator_prepare,
       :working_dir => '/tmp/awesomeator',
       :config_file => '/etc/awesomeator.conf' do

  directory params[:working_dir] do
    action :create
    recursive true
  end

  template params[:config_file] do
    source 'awesomeator.conf.erb'
```

```
    variables({
      :working_dir => params[:working_dir]
    })
  end
end
```

As we see here, we're using a `directory` resource to create the working directory specified by the `params[:working_dir]` parameter, then using a template which takes the `params[:working_dir]` as a variable to write a configuration file to `params[:config_file]` which contains the working directory.

Before we can actually try out this new definition, we need to add the `awesomeator.conf.erb` template that our `template` resource will be using to our cookbook. Paste the following code into `/tmp/part4_examples/cookbooks/testcookbook/templates/default/awesomeator.conf.erb`

Example 8-6. /tmp/part4_examples/cookbooks/testcookbook/templates/default/awesomeator.conf.erb

```
working_directory = <%= @working_dir %>
```

The code we're using for our template here is very simple. We're simply taking the template variable `@working_dir`, which is passed into our template resource when we call it in a recipe, and printing the value into the desired place in our configuration file.

Now that we've added some resources to our definition code and created the supporting template file that we need, let's try running our expanded definition and see what happens:

```
$> chef-client --once --why-run --local-mode list --config /tmp/part3_examples/client.rb \
  --override-runlist testcookbook::awesomeator
[sudo] password for jcowie:
Starting Chef Client, version 11.10.4
[Thu, 24 Apr 2014 14:58:30 +0000] WARN: Run List override has been provided.
[Thu, 24 Apr 2014 14:58:30 +0000] WARN: Original Run List: []
[Thu, 24 Apr 2014 14:58:30 +0000] WARN: Overridden Run List:
  [recipe[testcookbook::awesomeator]]
resolving cookbooks for run list: ["testcookbook::awesomeator"]
Synchronizing Cookbooks:
  - testcookbook
Compiling Cookbooks...
Converging 2 resources
Recipe: testcookbook::awesomeator
  * directory[/var/awesomeator] action create
    - Would create new directory /var/awesomeator

  * template[/etc/awesomeator.conf] action create
    - Would create new file /etc/awesomeator.conf

[Thu, 24 Apr 2014 14:58:30 +0000] WARN: Skipping final node save because
```

```
  override_runlist was given

Running handlers:
Running handlers complete

Chef Client finished, 2/2 resources would have been updated
```

As we saw in "What is a Definition?" on page 167, in the above output when our definition is executed it is expanded into its constituent resources - this means that we see our `directory` and `template` resources show up directly in the run output rather than the `awesomeator_prepare` definition that we defined to wrap them. It's worth bearing in mind that as the definition name itself does not show up in the default run output, if you have to debug any run issues you may find the resource in question located in definition files in addition to recipes.

As we saw in "Setup RunContext" on page 67, the RunContext object stores a list of all recipe *definitions* separately to the list of resources in the resource collection. This is because, rather than defining new resource types, definitions will contain one or more already existing resource types.

Summary

In this chapter, we've seen that definitions provide us with a quick and easy way to create reusable parameterized *macros* in our recipes which can contain frequently used resource combinations. We've also explored when definitions may be useful, but also some of their limitations- namely that they are not able to be notified by other resources and cannot define multiple actions. So what do we do if we want to create a custom resource which behaves more like the default resource types such as `package` and `cookbook_file` which Chef ships with?

That's where resources and providers come in. In the next chapter, we're going to dive into what exactly resources and providers are and the methods and techniques needed to create and use them.

PART IV
Other Customizations

In part 3 of this book, we're going to leave Chef runs and recipes behind and look at the different types of customization we can create to extend other parts of our Chef infrastructure, We'll examine

- How to create and extend Knife plugins
- How to interact with the Chef API to write scripts and reports
- How to create more advanced customizations that alter Chef objects themselves
- How to contribute our customizations back to the community

CHAPTER 11
Customizing Knife

In previous chapters, we mainly focussed on customizations to the Chef run itself, and creating providers and resources for use in our recipes. In this chapter, we're going to move outside of the Chef run and look at how we can customize Knife. As we saw in "Chef Client Tools" on page 60, Knife is the primary command line tool for interacting with Chef servers and is installed by default as part of the Chef installation process. Out of the box, Knife ships with a number of default plugins which support a variety of common tasks such as:

- Uploading cookbooks to the Chef server
- Bootstrapping nodes to install and run Chef Client
- Creating, modifying and deleting user and client objects
- Setting run_lists on nodes

You can find a full listing of the commands supported by Knife out of the box on the Chef Documentation site.

As with many of the features and tools that Chef ships with however, Knife is not limited to only those commands provided by Chef Inc. Under the hood, every command supported by Knife is driven by its extensible and flexible plugin interface which we can also use to implement our own Knife commands.

In this chapter, we'll learn about:

- How to obtain and navigate around the Knife source code

- The common framework shared by all Knife plugins
- The anatomy of executing a knife command
- How to create and run our own Knife plugins
- How Knife plugins work under the hood
- The objects and functionality that Chef provides to support Knife plugins

We'll then finish off by using the concepts learned throughout this chapter to revisit the last of the three problems AwesomeInc has been experiencing that we explored in "Criteria for Customization" on page 9.

The Knife Source Code

The source code for Knife (particularly its plugins) is a useful aid when working through the material in this chapter. Knife and its default plugins are shipped as part of the main Chef repository so to grab your own copy, please follow the instructions in "Getting the Chef Source Code" on page 76.

Let's take a moment to look at where the different components of Knife live inside the Chef repository - we'll be examining a number of these classes throughout this chapter and it's useful to have a mental picture of the lay of the land. Source files related to knife can be found in several different directories under the main Chef repository as shown here:

```
.
├── bin
│   └── knife❶
├── ...
├── lib
│   └── chef
│       ├── ...
│       ├── application
│       │   └──knife.rb❷
│       ├── ...
│       ├── knife❸
│       │   ├── bootstrap❹
│       │   └── core❺
│       ├── ...
│       └── knife.rb❻
└── ...
```

❶ `/bin/knife` is the executable wrapper which is run when `knife foo` commands are executed in the terminal.

❷ `lib/application/knife.rb` is called by the `bin/knife` wrapper script and serves to validate and parse command line options, before initializing the `lib/chef/knife.rb` class we see in step 6

❸ The `/lib/chef/knife` directory contains a large number of class definition files which implement the out-of-the-box plugins shipped with Knife. Each default knife command has a corresponding class definition file in this directory - these default plugin class definitions can serve as an excellent reference guide when implementing custom plugin classes.

❹ The `/lib/chef/knife/bootstrap` directory contains ERB templates for a number of different operating systems to be used by the `knife bootstrap` command.

❺ The `/lib/chef/knife/core` directory contains a number of supporting classes used by many knife plugins. We'll look at these classes in more detail later in this chapter.

❻ `/lib/chef/knife.rb` is the class definition file for the `Chef::Knife` class which implements much of the logic involved in running knife commands, similarly to how the `Chef::Client` class at `/lib/chef/client.rb` we saw in "The Chef::Application::Client Class" on page 82 implements much of the logic involved in a chef-client run. This class is also the `superclass` from which all Knife plugins inherit.

Now that we've looked at how the Knife source code is structured, let's take a closer look at what a Knife plugin actually looks like.

Introduction to Knife Plugins

As with many of the customizations we've looked at in this book so far, all Knife plugins share the same common structure. Understanding the features that all knife plugins share and more importantly why these commonalities exist is crucially important when implementing your own plugin classes, so before we go on to look at how Knife commands are executed, we're going to examine this skeleton plugin framework.

Regardless of their complexity, all Knife plugins are based on a common framework which looks like this:

```
class Awesome < Chef::Knife ❶

  deps do❷
    # Dependencies
    require 'useful/class'
  end

  def run ❸
    puts "Ohai chefs!"
  end
end
```

❶ All knife plugins inherit from the `Chef::Knife` superclass which lives at `lib/chef/knife.rb` in the repository (Github Link). The name of the class is used to determine the corresponding knife command implemented by the plugin.

❷ The `deps` block is used by Knife to allow plugin classes to *lazily load* dependencies. We'll examine what the lazy loading of dependencies means and how it differs from simply using a `require` statement as we've done in previous examples in more detail in the "Anatomy of a Knife Command" on page 218 section later on in this chapter, as this concept is directly related to the way in which Knife loads and executes plugins.

❸ The `run` method is the main entry point into a plugin class and is called by Knife to tell a plugin to carry out its programmed behavior in the same way that ohai calls the `collect_data` method of its plugins as we saw in "Ohai Example 1: Plugin Skeleton" on page 99.

Knife Plugin Class Naming

Unlike the formatter classes we looked at in (to come) which use the `cli_name` attribute to tell Chef how to refer to them, Knife uses the name of each plugin class (provided that it inherits from the `Chef::Knife` superclass) to determine the knife command which the plugin implements. Each upper cased word in the class name corresponds to a word in the knife command which will execute this plugin.

For example, the `knife node list` command is implemented by a class named `NodeList` and the `knife data bag from file` command is implemented by a class called `DataBagFromFile`. For this reason, the name that we choose for our Knife plugin classes is extremely important.

So now we know what a bare bones knife plugin class actually looks like, but how does Knife get from the user running a command like `knife node list` to the `run` method of the `NodeList` class actually being called?

Anatomy of a Knife Command

As Knife is typically used to execute single discrete commands, it treats its plugins somewhat differently to other Chef tools such as ohai (which we learned about in Chapter 6). Whereas ohai uses its plugins to augment its collection of attributes and thus executes all of the plugins it knows about each time it is run, Knife only wants to execute the specific plugin which implements the command executed by the user.

This means that each time Knife is executed, it must identify the command being executed and then locate and load the plugin class which implements that command - and

Knife can't simply look at its default plugins, it also has to be aware of any customized plugins we've installed. An understanding of how this process works is invaluable when creating our own knife plugins, so before looking at how to create our own plugin classes we're going to take a minute to learn a little more about the underlying structure and behavior of Knife.

Let's examine what actually happens under the hood when we execute a knife command such as `knife node list`. Every time we execute a Knife command it performs a number of distinct steps to enable it to identify, load and executed the correct plugin. These steps are illustrated in the following diagram:

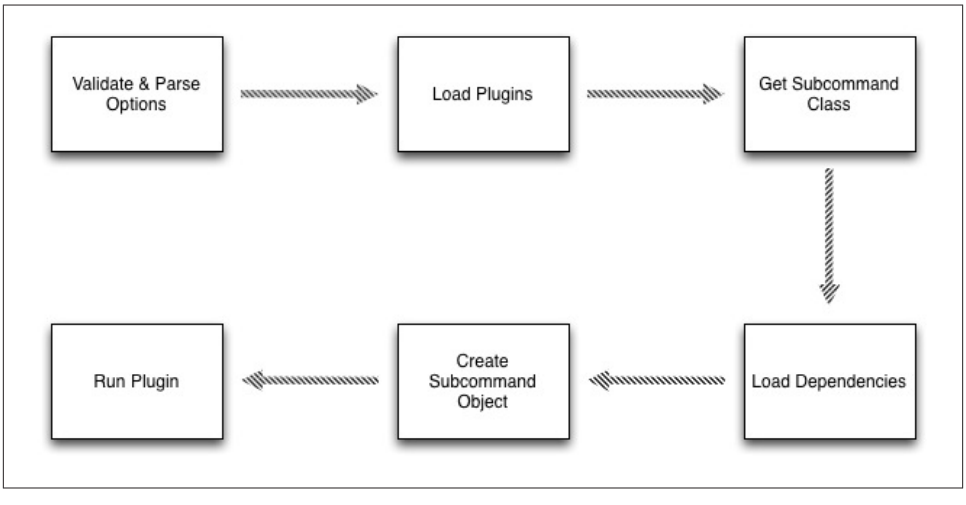

Figure 11-1. Knife Command Anatomy

Let's examine these steps in a little more detail:

Validate and Parse Options

The first step performed once a knife command has been run is for knife to validate and parse the subcommands (ie `cookbook upload`) and options (ie `--freeze`) that have been passed along with the command. If invalid subcommands or options have been passed, knife will display a *usage* guide showing a list of all valid commands and options and quit. If a partially invalid subcommand has been specified (ie `cookbook splat`), knife will display a *usage* guide for that subcommand only, and quit.

If all subcommands and options were valid, Knife proceeds onto the next step.

Load Plugins

After a list of subcommands and options has been successfully parsed, Knife next needs to searches for all of the available Knife plugins. It looks for plugin class definition files in the following locations:

- The `~/.chef/plugin_manifest.json` file - this is an optional JSON file containing the definitive list of all plugin files to be loaded
- If no `plugin_manifest.json` file is present, Knife will look in the `lib/chef/knife` directory (if it exists) of all gems installed via Rubygems.
- Knife will also explicitly look in the `lib/chef` directory of the Chef gem - this is where the default Knife plugin classes are located.
- Finally, Knife will look in the `~/.chef/plugins/knife/` directory (if it exists) under the home directory of the user running the knife command

Once the all available plugin class files have been identified, knife loads them into memory so that it can access the actual `Class` objects defined by these files - any non-lazily loaded dependencies are also required at this stage.

"Lazy" Dependency Loading

When Knife loads plugin classes into memory during this step, it does so using the `load` method of Ruby's build in `Kernel` module. This loading process results in any defined `require` statements being executed. Since Knife does not yet know which plugin class it will need, it can't make any decisions about which dependencies actually need to be loaded. As a result, **all** dependencies defined in **any** plugin class are loaded during this step regardless of which Knife command has actually been executed.

When complex Knife plugins are installed which `require` a number of dependencies, this global dependency loading can result in the execution time of even the simplest knife commands increasing significantly. To mitigate this problem, Chef allows plugins to declare dependencies using a strategy called *lazy loading*.

We'll examine lazy loading in more detail when we look at the implementation of our own plugin classes, but essentially the process involves a plugin class declaring its `require` statements inside a special `deps` block like this:

```
deps do
  require 'useful/thing'
end
```

As this block is **not** evaluated when the plugin class is loaded into memory during this step, this avoids dependencies being loaded into memory until later on in the process when Knife has identified the plugin class it actually needs - we'll meet this step shortly.

These `Class` objects are then assembled into a plugin list so that knife can locate and instantiate the correct class for the command being executed later on in the process. This list is stored as a `Hash`, where each *key* is a "snake cased" representation of the class name and each *value* is a `Class` object representing the class itself.

"Snake Casing" a string means changing all the words to use lower case letters and separating them with the underscore character instead of a space - effectively turning the string into a continual *snake* of characters. For example, the snake cased representation of `Customizing Chef is cool` would be `customizing_chef_is_cool`.

For example, the `Chef::Knife::NodeList` class (which lives at `lib/chef/knife/node_list.rb` in the Chef repository - Github Link) would be represented in the hash as follows:

```
"node_list" => Chef::Knife::NodeList
```

Get Subcommand Class

The next step of the process is for knife to evaluate the subcommands and options it was passed on the command line and construct the name of the class it needs to instantiate. During this step, knife will take all arguments passed to the knife command which do **not** contain the `-` character and joins them into a snake-cased string. As we saw above, "snake casing" means taking a string and lowercasing all the words, then separating them with the _ character.

For example, if Chef was given the command `knife node list -v`, Chef would separate out the `node list` component and turn it into the snake-cased string `node_list`.

Once this string has been constructed, knife looks up the `Hash` created during the "Load Plugins" on page 220 step to find the corresponding `Class` object representing the plugin to be instantiated.

To use our `knife node list -v` example again, during this stage of the process Knife will perform the following steps:

- Extract the `node list` component of the knife command and turn this into the snake cased string `node_list`
- Look up the `node_list` key of the plugins `Hash` assembled in the "Load Plugins" on page 220 step. As we saw above, the value of this key is the `Class` named `Chef::Knife::NodeList`.
- The `Chef::Knife::NodeList` class is then returned as the result of this step.

Load Dependencies

Having identified the exact `Class` which implements the command we specified, Knife now evaluates the optional `deps` block defined in the code of that plugin class to load any dependencies that the plugin may require - this loading of dependencies only when we are sure they will be needed is the second part of Knife's *lazy loading* strategy which we met in ""Lazy" Dependency Loading" on page 220.

Create Subcommand Object

Now that all of its dependancies have been loaded, knife instantiates our plugin class object using its `.new` method as we saw in "Classes" on page 30. In the case of our `knife node list` example, this would equate to running:

```
Chef::Knife::NodeList.new
```

As our plugin class has only just been instantiated, during this step knife also populates it with the settings loaded from the `knife.rb` configuration file. This ensures that the plugin uses the same configuration settings as the parent Knife process.

Run Plugin

The final step in the process is to actually run the plugin itself. Before doing this however, knife uses the settings loaded from the `knife.rb` configuration file to connect to the configured Chef server and authenticate using the client key specified in `knife.rb`

At this point, if we passed the `--local-mode` option to our knife command, the Chef server settings defined in `knife.rb` are overridden and a local chef-zero instance is started up. Knife then runs in *local mode*, sending all server commands to chef-zero instead of the Chef server configured in `knife.rb`.

Once server connectivity has been configured, knife calls the `run` method of the object we created in "Create Subcommand Object" on page 222, thereby handing the execution process over to the plugin object. Once the plugin has finished carrying out its tasks, it exits and the process of running a Knife command is complete.

Creating a Test Environment

The final thing we need to do before starting to implement our own knife plugins is to create a test environment in which to safely run our example code. As with the test environments we've created in previous chapters, our test environment will use `chef-zero` to run knife in *local mode*, which creates a safe sandbox for testing out customizations.

Although running the examples from this chapter in the test environment we'll create here is very much recommended, it is not strictly necessary - if you already have an existing knife configuration set up you may use this should you choose to do so although you will likely see different output from the example commands. As always, please do not run example code against your production Chef server.

Just as with the chef-client test environment we created in "Preparing a Test Environment" on page 116, knife supports the `--local-mode` option to start up a local chef-zero server which causes knife to run in *local mode*. Before we can do that however, we need to complete a couple of preparatory steps. Once these steps have been completed, our test environment will allow you to run all of the examples in this chapter on your local machine - no network access is needed.

Prerequisites and Preparation

The assumptions stated in "Assumptions" on page 23 still apply to this section with regards to the development environment I'm assuming that readers are working with

You will also need to have the `ssh-keygen` command installed on your development machine. For Redhat & CentOS systems this is provided by the `openssh` package, and on Debian / Ubuntu systems by the `openssh-client` package. On Mac OS X, this command is installed by default.

The first step in preparing our test environment is to create the `/tmp/part4_examples` directory for our test files to live in:

```
$> mkdir /tmp/part4_examples
```

Next, we need to create a dummy key for knife to use when authenticating to our `chef-zero` server. Unlike when communicating with a real Chef server, which requires the client key to be generated on the Chef server itself, Chef Zero will accept any key in a valid format. To create our dummy key, run the following command and hit "enter" for each input prompt shown:

```
$> ssh-keygen -f /tmp/part4_examples/customizing_chef.pem

Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /tmp/part4_examples/customizing_chef.pem.
Your public key has been saved in /tmp/part4_examples/customizing_chef.pem.pub.
The key fingerprint is:
```

```
d6:b2:09:f0:f8:2f:04:31:24:ef:58:19:8c:1b:8a:35 jcowie@mydomain.com
The key's randomart image is:
+--[ RSA 2048]----+
|  .+o            |
|  Eo+o           |
|.o +=o           |
|o .+.+   .       |
|  . o.o S .      |
|     ..o +       |
|     .. o        |
|      ..         |
|       ..        |
+-----------------+
```

You should then see output from this command similar to the above. Now that we have our dummy client key, we need to create a `knife.rb` configuration file to tell knife how to talk to our test environment. Paste the following lines into `/tmp/part4_examples/knife.rb`:

Example 11-1. /tmp/part4_examples/knife.rb

```
chef_server_url   'http://127.0.0.1:8889'
node_name         'cctest'
client_key        '/tmp/part4_examples/customizing_chef.pem'
cache_options( :path => '/tmp/part4_examples/checksums' )
chef_repo_path    '/tmp/part4_examples/chef-zero/playground'
```

Our final preparatory step is to grab a copy of the chef-zero code - this contains a ready-made "playground" directory containing cookbooks, node objects, roles and environments and is ideal for testing example code as we'll be doing here. To download the chef-zero code, run the following commands:

```
$> cd /tmp/part4_examples
$> git clone https://github.com/opscode/chef-zero.git
```

Verify Test Environment Works Correctly

The final step in the preparation of our test environment is verifying that it works correctly. To do this, we're going to run knife using the following command:

```
$> knife node list --config /tmp/part4_examples/knife.rb --local-mode
```

In this command, I've passed the following options to chef-client:

-c /tmp/part4_examples/knife.rb
: This option tells knife to use the configuration file `/tmp/part4_examples/knife.rb` which we prepared above

--local-mode
: This option tells knife to run in *local* mode, starting up a local chef-zero instance to communicate with instead of the server. chef-zero will also use the `chef_re`

`po_path` configuration parameter we set in `knife.rb` to use the `/tmp/part4_examples/chef-zero/playground` directory as its data store - this parameter determines the data chef-zero serves us when we request data such as nodes or cookbooks.

Now let's actually run this command:

```
$> knife node list --config /tmp/part4_examples/knife.rb --local-mode

desktop
dns
lb
ldap
www
```

If all of the instructions in the previous steps were completed successfully, you should see similar output to that shown above which shows us a list of all of the nodes stored in our chef-zero test environment.

Now that we have a working test environment to run our plugin examples under, let's finally start creating some knife plugins! Alongside each of the plugin examples we look at in this chapter, we'll examine the various helper classes both Knife and Chef provide us with and how they can be utilized in our own custom plugin classes.

Knife Example 1: Wrapping an Existing Plugin

One of the simplest and most common use cases for Knife plugins is augmenting the functionality provided by an existing plugin. In this example we're going to create a plugin which makes use of the built-in `Chef::Knife::Search` plugin, which provides the `knife search` command and lives at `lib/chef/knife/search.rb` in the Chef repository (Github Link).

The scenario here is that we have a commonly used search query to locate nodes within our infrastructure whose name match a particular pattern and are in the *_default* environment. Instead of repeatedly running the command `knife search node "name:*foo* AND chef_environment:_default"`, we're going to create a *wrapper* plugin which takes the name of the node we want to search for, then constructs the search query for us and runs the `Chef::Knife::Search` plugin to execute it.

Because Knife only loads plugins from the specific locations listed in "Load Plugins" on page 220, we need to store our example code under one of those locations - we'll use the `~/.chef/plugins/knife` directory. While logged in as the user you'll be running Knife commands as, run the following command to create this directory:

```
$> mkdir -p ~/.chef/plugins/knife
```

On Linux and other Unix based operating systems, the tilde symbol (~) is an abbreviation for the home directory of the currently logged in user. This abbreviation is **not** supported on Windows - if you're running these example knife plugins on Windows you'll need to replace the tilde symbol in directory paths with the full path to your user's home directory

Now that we've created the directory for our example plugins to live in, let's paste the following code into `~/.chef/plugins/knife/awesome_search.rb`:

Example 11-2. ~/.chef/plugins/knife/awesome_search.rb

```
module Awesome
  class AwesomeSearch < Chef::Knife❶

    deps do
      require 'chef/knife/search'❷
    end

    def run❸

      # Assign the first command line argument
      # after the plugin name to the variable "query"
      query = "name:*#{name_args.first}* AND chef_environment:_default"❹

      knife_search = Chef::Knife::Search.new❺
      knife_search.name_args = ['node', query]❻
      knife_search.run❼
    end
  end
end
```

❶ Our `AwesomeSearch` plugin inherits from the `Chef::Knife` superclass which lives at `lib/chef/knife.rb` in the repository (Github Link). This is the class from which all Knife plugins inherit.

❷ Here we're specifying a `require` statement inside the `deps` block, ensuring that our dependency `chef/knife/search` is *lazily* loaded as we saw in ""Lazy" Dependency Loading" on page 220.

❸ The `run` method of our class is the method which will be called by Knife to actually execute our plugin.

❹ The `name_args` method is a built-in method provided by Ruby which allows you to access the parameters passed to the program being executed as an array. In this case we want the first element after the actual knife command, which in this case is the name of the node we'll be searching for.

❺ Here we're creating a new instance of the `Chef::Knife::Search` class, just as Knife would if we executed the `knife search` command plugin directly.

❻ Next we call the `name_args` method of our `Chef::Knife::Search` object to pass it the parameters that it needs - if we were running the `knife search` command manually, these would be passed on the command line as the `name_args` object we used above but since we're calling this plugin ourselves, we need to pass these options through to the plugin class manually.

❼ Finally we're calling the `run` method of the `Chef::Knife::Search` object we created above. Note that our `AwesomeSearch` plugin class does not directly output any data itself - the output is produced by the `Chef::Knife::Search` plugin which we're wrapping, just as if we ran the `knife search` command manually from a terminal.

Now let's try running our plugin. As we called our plugin class `AwesomeSearch`, this means that the Knife knows that the command to execute it will be `knife awesome search` as we saw in "Knife Plugin Class Naming" on page 218. We'll also pass Knife the `-c` and `--local-mode` parameters to ensure that it loads the configuration we created in "Creating a Test Environment" on page 222 and runs in *local mode*.

```
$ knife awesome search ldap --config /tmp/part4_examples/knife.rb --local-mode
1 items found

Node Name:   ldap
Environment: _default
FQDN:
IP:
Run List:
Roles:
Recipes:
Platform:
Tags:
```

As we see in the above output, when our plugin is executed Knife parses the options passed and uses the `awesome search` portion of the command to identify that our `AwesomeSearch` plugin class should be executed, passing it the remaining word `ldap` as a parameter - Knife also knows that the `-c` and `--local-mode` options we specified are global rather than plugin-specific options. The result of the search query shown above was generated by the `Chef::Knife::Search` plugin once it was executed by our `AwesomeSearch` plugin.

The plugin example we looked at here is extremely simplified and does not contain any error handling or input validation, but it does serve to demonstrate how we can create our own custom Knife plugin classes and interact with other Knife plugins.

In our next example, we'll use the same scenario as we did here but this time instead of wrapping an existing plugin we're going to implement the search query and displaying of search results ourselves. As we'll be handling outputting the results of this plugin ourselves rather than leaving it to another plugin, before we can implement this plugin we're going to need to examine the classes Knife provides us with for formatting and displaying output. We're going to examine two class types:

Presenter Classes
: Presenter classes are used by Knife to format structured data such as `Chef::Node` objects so that it can be output to the user. We'll examine presenter classes in "Presenting Presenters!" on page 228

The UI Class
: Knife provides a special UI class to handle interacting with the user and displaying output, allowing us to ensure a consistent user experience across all Knife plugins. We'll meet the UI class in "The UI Class" on page 231

Presenting Presenters!

The first category of helper class provided by Knife that we'll examine here is *presenter* classes. Much of the data stored in Chef server about our Nodes, Roles, Cookbooks and other items is complex structured data with multiple levels of nested attributes - this sort of data is extremely easy for computers to work with, but not so easy for us to display nicely to the end user.

To help solve this problem, Knife ships with *presenter* classes to handle parsing this data and converting it into more convenient formats to output from our Knife plugin classes. In this section we'll meet the *presenter* classes shipped with Knife, and examine the methods and formats they support.

Chef::Knife::Core::GenericPresenter

The base presenter class provided by knife is `Chef::Knife::Core::GenericPresenter`, which lives at `lib/chef/knife/core/generic_presenter.rb` in the Chef repository (Github Link). Unlike more abstract base classes we've seen so far in this book the `Chef::Knife::Core::GenericPresenter` class does not merely define empty method bodies for its *subclasses* to inherit, but rather provides a comprehensive set of methods capable of formatting and outputting the majority of the structured data likely to be encountered when writing Knife plugins.

The `Chef::Knife::Core::GenericPresenter` class is used as the default presenter by Knife unless another presenter has been specified - we'll learn how to do this when we meet "The UI Class" on page 231 in the next section. This presenter class supports a number of different formats to display structured data which Knife users can request

by passing the `-F` option to Knife, which then passes it through to the plugin class being executed.

Supported Output Formats

The `Chef::Knife::Core::GenericPresenter` class supports outputting the formats listed below. Each format is named according to the value which is passed to Knife's `-F` option to request it, and shows a `Chef::Environment` object being output in that particular format.

json

The *json* format produces the JSON representation of the structured data, an example of which can be seen here:

```
{
  "name": "production",
  "description": "This is just production",
  "cookbook_versions": {
  },
  "json_class": "Chef::Environment",
  "chef_type": "environment",
  "default_attributes": {
  },
  "override_attributes": {
  }
}
```

yaml

The yaml format produces the YAML representation of the structured data, an example of which can be seen here:

```
--- !ruby/object:Chef::Environment
cookbook_versions: {}

default_attributes: {}

description: This is just production
name: production
override_attributes: {}
```

text

The *text* format produces a textual summary of the structured data, formatted to be easily readable when output to a terminal. An example of this format can be see here:

```
chef_type:          environment
cookbook_versions:
  apache: >= 1.0.0
default_attributes:
description:        This is just production
json_class:         Chef::Environment
```

```
name:               production
override_attributes:
```

pp
: The *pp* format is used to print a string representation of the underlying Ruby object representing the structured data. An example of this format can be seen here:

```
production
```

summary
: The *summary* format is used by default if no other format is specified - this currently generates the same output as the *text* format.

Although the `Chef::Knife::Core::GenericPresenter` class defines a number of methods, we're going to look at those methods most commonly used directly in knife plugins here. A full method listing can be found in the source code of this class at `lib/chef/knife/core/generic_presenter.rb` in the Chef repository (Github Link

format
: The `format` method is the main entry point into the `Chef::Knife::Core::GenericPresenter` class. This method takes structured data such as `Chef::Node` objects and returns it in the format requested by the user. If no format is specified, the default `summary` format we see in the above sidebar will be used.

format_for_display
: The `format_for_display` method performs preliminary formatting on structured data to prepare it for outputting to the user, such as extracting subsets of the data when only part of it is to be output. An example of when this method would be used is when the `-i` option is passed to the `knife search` command to return only the id of matching objects rather than the entire object.

Chef::Knife::Core::NodePresenter

Knife currently only ships with one additional presenter, the `Chef::Knife::Core::NodePresenter` class which lives at `lib/chef/knife/core/node_presenter.rb` in the Chef repository (Github Link).

This class is a *subclass* of `Chef::Knife::Core::GenericPresenter`, and overrides the `format` method from its superclass to nicely format `Chef::Node` objects when they are being output by Knife plugins - `Chef::Node` objects often contain a large number of deeply nested data structures to represent all their saved attributes and require some extra formatting prior to display.

You can see an example of this presenter in action in the output of "Knife Example 1: Wrapping an Existing Plugin" on page 225 - the `Chef::Knife::Search` plugin, which we wrap in that example, contains logic to use `Chef::Knife::Core::NodePresenter` if

the type of object being searched for is a node, and `Chef::Knife::Core::GenericPresenter` when we're searching for any other types of object.

Between `Chef::Knife::Core::GenericPresenter` and `Chef::Knife::Core::NodePresenter`, it is possible to format all of the structured data Chef produces that you might want to display in your Knife plugins - for this reason, although it would certainly be possible to define additional presenter classes, we're not going to do so here.

Now that we've examined the classes that Chef provides us for formatting data for output, let's meet the second of the helper classes we're going to need for our second example which allows us to interact with the user and terminal.

The UI Class

To standardize the user experience of interacting with Knife across all plugins, Chef ships with a special class which integrates with Knife and provides a number of methods for both interacting with the user and the terminal.

The `Chef::Knife::UI` class lives at `lib/chef/knife/core/ui.rb` in the repository (Github Link) and is available to all knife plugins as the `ui` object, which is defined in the `Chef::Knife` superclass from which all Knife plugins inherit. The `Chef::Knife::UI` class defines a number of methods, documented here broken down by category:

User Interaction Methods

The `Chef::Knife::UI` class provides two handy methods for asking questions to Knife users and collecting the responses:

ask_question
: The `ask_question` method allows you to ask the user a question and collect a free-text response. This method also accepts a default answer to use if none is given - this will be shown to the user when the question is asked.

confirm
: The `confirm` method is specifically designed for asking Knife users a question which expects a Yes or No response. If the user responds with No, Knife will immediately exit with status code *3*.

Message Output Methods

The `Chef::Knife::UI` class also provides a number of handy methods for displaying output messages on screen at different levels of severity to indicate informational events, error messages and so on. Where the users terminal supports colored output, some severity levels will use a colored prefix such as WARNING in yellow, ERROR in red etc to provide a further visual indication of the severity of the message.

msg
: The `msg` method prints output to the terminal at a standard *informational* severity level. This level essentially produces the same output that you would see if `puts` were used.

info
: The `info` method prints output to the terminal at a standard *informational* severity level, but prefixed with a white (if color is supported) `INFO` prefix.

warn
: The `warning` method prints output to the terminal at a *warning* severity level, prefixed with an yellow (if color is supported) `WARNING` prefix.

error
: The `error` method prints output to the terminal at a *error* severity level, prefixed with an red (if color is supported) `ERROR` prefix.

fatal
: The `fatal` method prints output to the terminal at a *fatal* severity level, prefixed with an red (if color is supported) `CRITICAL` prefix.

color
: The `color` method allows a message to be printed in a specific color, if the output terminal supports colored output.

There is no *functional* difference between these output methods, the only difference is the prefix attached to the output message. It is left the discretion of plugin creators as to which severity level should be used when displaying messages to the user.

Other Output Methods

The `Chef::Knife::UI` class provides several useful methods to allow plugin authors to output structured data (such as `Chef::Node` objects) to the user, and to query Knife about the format in which structured data should be presented and what features are supported by the current output environment.

color?
: The `color?` method allows a plugin author to query whether or not colored output is supported by the user's environment. This is typically used to avoid calling methods such as `color` unless colored output would actually be visible to the user. Currently, colored output is supported by Knife when the output device is a `tty` terminal, and the Knife command is not being run on a Windows machine.

output
: The `output` method is used to display structured data to the user in their desired format - although a default format can be used, it is also possible to specify a par-

ticular format by passing the `-F` option to Knife. Under the hood, this method calls the `format` method of the configured presenter to format the data, before outputting it. A list of output formats and method supported by Knife's presenter classes can be seen in "Presenting Presenters!" on page 228

interchange?
: The `interchange` method is used to query whether or not the user has requested output in a data interchange format such as JSON or YAML. This method is typically used to control whether or not output should be printed with explanatory text etc, or simply output in the requested format. An example of this technique can be seen in the `Chef::Knife::Search` plugin source code.

use_presenter
: The `use_presenter` method allows a Knife plugin author to specify that a specific *presenter* class should be used to handle output from the plugin, for example using the `Chef::Knife::Core::NodePresenter` class when node objects are being displayed.

Misc Methods

edit_data
: The `edit_data` method allows the user to use the editor configured via the system's `$EDITOR` environment variable to edit Chef objects via temporary files. This method is primarily used by the `edit_object` method documented below, but can also be used to edit object that have previous been loaded elsewhere in the plugin. We'll look at examples of editing objects using this method in "Editing and Saving Objects Interactively" on page 244

edit_object
: The `edit_object` method allows the user to load and edit objects on the Chef server such as Roles, Nodes and Environments using their text editor of choice. This method is used by various knife commands such as `knife role edit` and `knife node edit` to allow the JSON representation of objects stored on the Chef server to be edited by users - under the hood, the object is downloaded to the users machine, output to a temporary file, edited using the `edit_data` method above and then re-uploaded to the server. We'll examine how to edit objects in this way in "Editing and Saving Objects Interactively" on page 244

Highline Methods

Knife makes use of the third-party Highline library to provide some of its user interaction functionality. The `Chef::Knife::UI` class defines several methods which allow the methods of the same name from the Highline library to be called without the need to instante a new Highline object.

highline
: The `highline` method returns the `Highline` object used by many of the methods of `Chef::Knife::UI`, allowing it to be accessed directly if desired.

ask
: The `ask` method calls the corresponding `ask` method of the `Highline` library, which is documented in the Highline Documentation

ui.list
: The `list` method calls the corresponding `list` method of the `Highline` library, which is documented in the Highline Documentation

Examples of using the various methods of the `Chef::Knife::UI` class can be seen in subsequent examples in this chapter, and on the Chef Documentation site.

Knife Example 2: Search Plugin

In this example, we'll use the same scenario as we did in "Knife Example 1: Wrapping an Existing Plugin" on page 225 - that we want to locate nodes within our infrastructure whose name match a particular pattern and are in the *_default* environment. This time however, we're going to display a **warning** to the user that the node we're searching for has been found in this environment - in our hypothetical scenario, we don't want nodes to be in this environment.

To do this, we're going to have to implement the search query and format the results ourselves as the output will be different from that displayed by the `Chef::Knife::Search` plugin. Let's paste the following code into the same file as we used for example 1, `~/.chef/plugins/knife/awesome_search.rb`

Example 11-3. ~/.chef/plugins/knife/awesome_search.rb

```
module AwesomeInc
  class AwesomeSearch < Chef::Knife

    deps do
      require 'chef/search/query'
      require 'chef/knife/core/node_presenter' ❶
    end

    banner "knife awesome search QUERY" ❷

    def run
        ui.use_presenter Chef::Knife::Core::NodePresenter ❸
        search_object = Chef::Search::Query.new❹
        query = "name:*#{name_args.first}*"

        result_items = []
        result_count = 0
```

```
        # Call the "search" method of our Chef::Search::Query
        # object and add each item to our results array - also
        # increment our results counter
        search_object.search('node', query) do |item|❺
          result_items << item
          result_count += 1
        end

        # Display the number of results we have as a "Warning"
        ui.warn "#{result_count} Nodes found in the _default environment:"❻
        ui.warn("\n")

        # Loop over our results
        # and output them
        result_items.each do |item|
          output(item)❼
          unless config[:id_only]
            ui.msg("\n")
          end
        end
    end
  end
end
```

❶ Here we're requiring as a dependency the `Chef::Knife::Core::NodePresenter` class we met in "Presenting Presenters!" on page 228.

❷ The `banner` method we're calling here determines the output Knife shows when we run this command with incorrect or missing parameters. It's entirely optional, but can make it easier to use your plugin classes as it displays a more user friendly guide to the options and parameters to use.

❸ Here we're telling the `ui` object to use `Chef::Knife::Core::NodePresenter` as its presenter - since we know our search query should only return Node objects, we're able to benefit from the additional formatting this class provides us over `Chef::Knife::Core::GenericPresenter`

❹ Here we're creating a new `Chef::Search::Query` object, which lives at `lib/chef/search/query.rb` in the repository (Github Link). This class allows us to run search queries against the Chef API and is also used by the `Chef::Knife::Search` class we wrapped in "Knife Example 1: Wrapping an Existing Plugin" on page 225. The query on the next line is in exactly the same format we used in "Knife Example 1: Wrapping an Existing Plugin" on page 225.

❺ Next we call the `search` method of our `Chef::Search::Query` class, passing it the index that we want to search in (`node`) and the query we want to use. Remember, as we saw in (to come), Chef defines separate indexes for the various types of objects it stores so we need to specify which index our query will be run against.

❻ We're using the `warn` method of the `ui` object here to display the number of search results we found, formatted as a `WARNING`.

❼ Finally we're using the `output` method of the `ui` object to print our search results to screen.

Now let's try running our new example plugin:

```
$> knife awesome search ldap --config /tmp/part4_examples/knife.rb --local-mode
WARNING: 1 Nodes found in the _default environment:
WARNING:
Node Name:   ldap
Environment: _default
FQDN:
IP:
Run List:
Roles:
Recipes:
Platform:
Tags:
```

As we see above, the output displayed by our plugin here is similar to that shown when we wrapped the `Chef::Knife::Search` plugin class in "Knife Example 1: Wrapping an Existing Plugin" on page 225 - but this time, we're displaying a WARNING before our search results.

The decision to wrap an existing plugin or implement the desired behavior yourself as we did here is one you will likely have to make yourself when implementing custom Knife plugins which perform similar functions to one or more of the default Knife plugins - my advise would be to wrap existing plugins as we did in "Knife Example 1: Wrapping an Existing Plugin" on page 225 where possible, unless you're implementing functionality not provided by the existing plugin, or which can not be easily added by wrapping.

So far in this chapter, we've looked at how to wrap existing knife plugins, how to run search queries and the classes that Knife provides us with to interact with the user and display output from our plugins. More often than not when creating Knife plugins however, we're going to want to work with the actual data stored on the Chef server - before we dive into our next example, let's take a look at the helper classes which Knife provides us with to implement this behavior in our plugins.

Working with Chef Objects

One of the most common tasks performed by Knife plugins is the manipulation of data stored on the Chef server. For example, if you're creating a plugin to create nodes on a particular cloud system such as EC2, you're probably going to want to create a client on the Chef server for that node to authenticate against. If you're creating a workflow plugin to automate common Knife commands, you might want to create or update existing Node, Cookbook or Environment objects on the Chef server.

In this section, we're going to meet the class definitions which represent Chef's core objects types such as Nodes, Roles and Environments before learning about the helper classes that Chef provides to allow us to load and manipulate objects then saving them back to the server.

Chef define a number of useful classes in the Chef repository - which of course Knife also lives under - which represent the different types of Objects stored on the Chef server. These class definitions allow us to interact with the Chef server's API without having to worry about authenticating, constructing API requests and so on. Chef defines one class for each of the core object types stored on the server:

Clients
: *Client* objects are represented by the `Chef::ApiClient` class, which lives at `lib/chef/api_client.rb` in the repository (Github Link)

Cookbooks
: *Cookbook* objects are represented by the `Chef::CookbookVersion` class, which lives at `lib/chef/api_client.rb` in the repository (Github Link). As the Chef server treats each version of each cookbook separately, there is no single object for, say, the `apache` cookbook but rather an individual `Chef::CookbookVersion` object for each version of the `apache` cookbook that was uploaded.

Data Bags
: *Data bag* objects are represented by the `Chef::DataBag` class, which lives at `lib/chef/data_bag.rb` in the repository (Github Link)

Data Bag Items
: *Data bag item* objects are represented by the `Chef::DataBagItem` class, which lives at `lib/chef/data_bag_item.rb` in the repository (Github Link). Note that Chef defines separate objects for both the data bag itself and the items that it holds - this allows us to choose whether to access an entire databag, or specific items that it contains.

Encrypted Data Bag Items
: *Encrypted data bag item* objects are represented by the `Chef::EncryptedDataBagItem` class, which lives at `lib/chef/encrypted_data_bag_item.rb` in the repos-

itory (Github Link). As with standard data bag items, encrypted data bag items are defined by a separate object than the data bag which holds them. This class implements additional functionality on top of the `Chef::DataBagItem` class to provide the encryption behavior necessary to support encrypted data bag items

Environments
: *Environment* objects are represented by the `Chef::Environment` class, which lives at `lib/chef/environment.rb` in the repository (Github Link)

Nodes
: *Node* objects are represented by the `Chef::Node` class, which lives at `lib/chef/node.rb` in the repository (Github Link)

Roles
: *Role* objects are represented by the `Chef::Role` class, which lives at `lib/chef/role.rb` in the repository (Github Link)

Users
: *User* objects are represented by the `Chef::User` class, which lives at `lib/chef/user.rb` in the repository (Github Link). *Users* in Chef are distinct from *clients* - a user object would typically be created to allow access to your Chef server's Web UI if using Open Source Chef server or to grant certain permissions to if using Hosted Enterprise Chef

All of these classes, although they do not share a common *superclass*, allow us to use the same techniques and method names to access, manipulate and save them back to the server. We'll first examine the two principle ways of loading objects from the server, before looking at how to manipulate them and save them.

Loading Objects: Searching

The first way that Knife provides us with to access objects from the server is via the results of search queries - we've actually already used this method in "Knife Example 2: Search Plugin" on page 234. Consider the following snippet of the code we used in that example:

Example 11-4. Excerpt of ~/.chef/plugins/knife/awesome_search.rb

```
search_object = Chef::Search::Query.new
query = "name:*#{name_args.first}*"

search_object.search('node', query) do |item|
  # Do stuff with results
end
```

When we run a search query and iterate over the results like this, each result in the collection will be an instance of one of the classes we defined in "Working with Chef

Objects" on page 237. In the above snippet for instance, since we specified that we want to search in the *'node'* index, all results returned by the search query will be `Chef::Node` objects. Let's observe this in action with a quick demonstration which we'll paste into `~/.chef/plugins/knife/awesome_object_demo.rb`:

Example 11-5. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
  class AwesomeObjectDemo < Chef::Knife

    deps do
      require 'chef/search/query'
    end

    def run
        search_object = Chef::Search::Query.new
        query = "name:*#{name_args.first}*"
        search_object.search('node', query) do |item|
          ui.msg "#{item.name}: #{item.class}"❶
        end
    end
  end
end
```

❶ For each result of the search query, instead of formatting it for output as we did in "Knife Example 2: Search Plugin" on page 234 we're simply outputting a `ui.msg` message containing the `name` and `class` of each result.

When we run our demo plugin using our Knife test environment, we see the following output:

```
$> knife awesome object demo ldap --config /tmp/part4_examples/knife.rb --local-mode
ldap: Chef::Node
```

As we see here, Chef is outputting the `name` and `class` of each result returned by the Search query - our method call to `item.class` shows us that the object returned in the search results on this occasion is indeed a `Chef::Node` object.

For cases where we want to fetch all objects matching a specified criteria (such as `name:ldap` in our example here), using a Chef search is the recommended technique. However Chef search queries do not allow us to access **all** of the data types stored on the Chef server. As we learnt in (to come), Chef provides the following search indexes which we can specify in calls to the `Chef::Search::Query` object we used above:

- client
- DATA_BAG_NAME (Chef defines a separate search index for each databag)
- environment

- node
- role

But what do we do if we want to access a single specific object without the overhead of searching an entire index, or we want to access data such as `Chef::CookbookVersion` objects for which Chef does not define a search index? Well, luckily for us, there's a way to do that too.

Loading Objects: Direct Loading

To allow us to directly load Chef objects instead of having to run a search query to find them, each of the class definitions listed in "Working with Chef Objects" on page 237 defines a `load` method. This method will take either one or several parameters which give Chef the name of the object to be loaded, and any other information that might be needed such as (in the case of a data bag item) the name of the data bag which contains it. I've listed a quick summary of the parameters taken by the `load` method of each object type below (I've removed the body of each `load` method for brevity):

Clients - Chef::ApiClient

```
def self.load(name)
  # Takes the name of the client as its parameter
end
```

Cookbooks - Chef::CookbookVersion

```
def self.load(name, version="_latest")
  # Takes the name of the cookbook and the
  # desired version as parameters. version
  # defaults to "_latest" if not supplied
end
```

Data Bags - Chef::DataBag

```
def self.load(name)
  # Takes the name of the data bag as its parameter
end
```

Data Bag Items - Chef::DataBagItem

```
def self.load(data_bag, name)
  # Takes the name of the data bag as its first parameter
  # and the name of the item in that data bag as its second
end
```

Encrypted Data Bag Items - Chef::EncryptedDataBagItem

```
def self.load(data_bag, name, secret = nil)
  # Takes the name of the data bag as its first parameter,
  # the name of the item in that data bag as its second,
  # and the decryption secret as its third - this defaults to nil
end
```

Environments - Chef::Environment

```
    def self.load(name)
      # Takes the name of the environment as its parameter
    end
```

Nodes - Chef::Node

```
    def self.load(name)
      # Takes the name of the node as its parameter
    end
```

Roles - Chef::Role

```
    def self.load(name)
      # Takes the name of the role as its parameter
    end
```

Users - Chef::User

```
    def self.load(name)
      # Takes the name of the user as its parameter
    end
```

To view the full class definitions that define these methods, please see the source files listed against each object type in "Working with Chef Objects" on page 237.

Now that we've examined the `load` methods defined by each class, we're ready to try using them in another example. This time, we'll load a node object directly using the `load` method of the `Chef::Node` class. Let's past the following code into the same file that we used for the earlier example in this section, `~/.chef/plugins/knife/awesome_object_demo.rb`:

Example 11-6. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
  class AwesomeObjectDemo < Chef::Knife

    deps do
      require 'chef/node'❶
    end

    def run
        node_object = Chef::Node.load(name_args.first)❷
        ui.msg "#{node_object.name}: #{node_object.class}"❸
    end
  end
end
```

❶ As we're going to be calling a method of the `Chef::Node` class, we need to require it - note that the `require` statement is in the deps block to ensure that the dependency is *lazily loaded* as we examined in ""Lazy" Dependency Loading" on page 220.

❷ Here we're calling the `load` method of the `Chef::Node` class with the node name we passed to the plugin as its parameter, and assigning it to the `node_object` variable.

❸ Finally we're using a `ui.msg` method call to output the `name` and `class` of `node_object`

Let's run our example plugin code, this time using `www` as the node we want to load instead of `ldap`:

```
$> knife awesome object demo www --config /tmp/part4_examples/knife.rb --local-mode
www: Chef::Node
```

As we see here, when we run our plugin example we again see that we've got a `Chef::Node` object - but this time we loaded it directly from the Chef server instead of getting it from the results of a search query.

We've now explored the two different ways in which Knife plugins are able to load objects from the Chef server, but what happens when we want to then *modify* our object and save it back to the server? Chef provides us with two ways to do this too, depending on how we want to edit the object - we can edit objects interactively, where we allow the user to specify the desired changes in an editor or non-interactively where we simply change the object behind the scenes.

Editing and Saving Objects Non-Interactively

The first and most straightforward method of editing and saving objects that we'll examine here is the non-interactive method - this means that the plugin edits the object behind the scenes and saves it back to the Chef server with no input from the user other than parameters passed to the Knife plugin.

Essentially, this method involves utilizing "setter" methods provided by Chef's core object classes to allow us to alter the attributes of an object. It's important to note that this method does not allow totally freeform editing of the objects, we're only able to alter attributes of our objects for which they define a corresponding setter method.

A comprehensive list of methods supported by the objects we've looked at can be found in the class files listed in "Working with Chef Objects" on page 237, but as a general rule of thumb most of these objects define methods for each of their "top level" attributes.

For example, the `Chef::Node` object defines `chef_environment`, `run_list`, `normal_attrs`, `override_attrs` etc.

As with the other classes we've looked at in this book, I recommend having a browse through the class files listed in "Working with Chef Objects" on page 237 to explore the various methods and attributes each class provides.

Let's look at an example of amending the `chef_environment` attribute of a `Chef::Node` object. As before, we'll paste this code in to `~/.chef/plugins/knife/awesome_object_demo.rb`

Example 11-7. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
  class AwesomeObjectDemo < Chef::Knife

    deps do
      require 'chef/node'❶
    end

    def run
        node_object = Chef::Node.load(name_args.first)❷
        # Print our node's chef_environment before we change it
        ui.msg "chef_environment is currently #{node_object.chef_environment}"
        node_object.chef_environment = name_args.last❸
        # Print our node's chef_environment again after we change it
        ui.msg "chef_environment is currently #{node_object.chef_environment}"
        node_object.save❹
    end
  end
end
```

❶ As in our previous example, the only dependency our plugin has is the `Chef::Node` object since we'll only be loading node objects here.

❷ Here we're loading our node object just as we did in our previous example, by calling the `load` method of `Chef::Node` and passing in the first parameter we gave on the command line.

❸ Next we're calling the `chef_environment` method of our node object and setting it to the last parameter we gave on the command line. This changes the value in our local copy of the `Chef::Node` object - it has not yet been saved back to the server.

❹ Finally, we're calling the `save` method on our `Chef::Node` object. This method saves our local modified copy of the node back to the Chef server. It's important to note that only during this final step is the data stored on the Chef server altered.

Let's try running our new example code:

```
$> knife awesome object demo www production --config /tmp/part4_examples/knife.rb --local-mode
chef_environment is currently _default
chef_environment is currently production
```

The above output shows that the `chef_environment` attribute on the `www` node has now been changed from `_default` to `production`. To verify that our change has indeed been saved to the Chef server, let's use the `knife node show` to output the `www` node object saved on our test Chef server.

```
$ knife node show www --config /tmp/part4_examples/knife.rb --local-mode
Node Name:   www
Environment: production
FQDN:
IP:
Run List:
Roles:
Recipes:
Platform:
Tags:
```

As we can see from the output above, the copy of our `www` node object saved on the Chef server reflects the change we made locally.

Editing and Saving Objects Interactively

The second method of editing and saving objects is that used by Knife plugins such as `knife node edit` and `knife data bag edit`, which allow the user to edit an object stored on the Chef server via their configured text editor.

Under the hood, when we edit objects in this way the object is first loaded, then downloaded from the server as JSON to a temporary file, at which point the user edits the JSON file directly via their configured text editor. Once editing is complete, the object is saved back to the server. In this section we're going to look at some different methods of combining the object loading and saving techniques we've already learned in this chapter with the methods provided by the `ui` object to implement this behavior.

The first and most simple pattern for editing objects we're going to look at is the `Chef::UI` class's `edit_object` method. This method provides a convenient wrapper to download an object from the server, edit the object, then save it back to the server if any changes have been made. The `edit_object` method takes two parameters - the class of object being edited, and the name of the object to edit. Let's making use of this method

ourselves by adding the following code to `~/.chef/plugins/knife/awesome_object_demo.rb`:

Example 11-8. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
 class AwesomeObjectDemo < Chef::Knife
   def run
       ui.edit_object(Chef::Node, name_args.first)
   end
 end
end
```

Our plugin class is extremely simple this time around and our run method consists of a single call to the `edit_object` method of the `ui` object. We're passing `Chef::Node` as the first parameter to tell it the type of object we want to edit, and `name_args.first` as the second parameter to let it know the name of the object. When we try running our plugin and pass in the `www` node as our parameter, we should see a JSON representation of this node opened up in the text editor specified by the EDITOR environment variable (simulated below):

```
$> knife awesome object demo www --config /tmp/part4_examples/knife.rb --local-mode

Opens in editor:

{
  "name": "www",
  "chef_environment": "production",
  "json_class": "Chef::Node",
  "automatic": {
  },
  "normal": {
  },
  "chef_type": "node",
  "default": {
  },
  "override": {
  },
  "run_list": [

  ]
}
```

While still in the editor, let's try adding an attribute to the `normal` attribute Hash of our node object, so that this section of the file now looks like this:

```
  "normal": {
    "awesome_level" : 100
  },
```

Next, save the file and exit the editor. At this point, we should then see the following output:

```
Saved node[www]
```

This output lets us know that our changes to the `www` node have now been saved back to the server. Let's use the `knife node show` command to show us the JSON representation of our `www` node and check this for ourselves:

```
$ knife node show www --config /tmp/part4_examples/knife.rb --local-mode -f json
{
  "name": "www",
  "chef_environment": "production",
  "run_list": [

  ],
  "normal": {
    "awesome_level": 100
  }
}
```

As we see in the above output, when the `www` node is loaded from the server and output to the terminal we see that the `awesome_level` attribute we added above is still present in the object's JSON. For many of the circumstances in which you might want to interactively edit an object in your Knife plugins, the `edit_object` method of the `ui` object will provide the functionality you need while keeping your code as simple as possible.

However, this method does restrict what we can do with the object editing workflow - we can only load, edit and save the object and are not able to modify the process in any way. So what do we do if we want to introduce some slightly more advanced behavior? What if, for example, we want to make a non-interactive change to our object and then allow the user to interactively edit the object to confirm our change and add further changes if they wish to?

If we look at the source code of the `ui` class, we see that the `edit_object` method is really just a wrapper around the loading and saving techniques we've already seen in previous example along with a call to the `edit_data` method of the `ui` object:

Example 11-9. Excerpt of lib/chef/knife/core/ui.rb

```
class Chef
  class Knife
    class UI

      # Other methods and comments removed

      def edit_object(klass, name)
          object = klass.load(name)❶

          output = edit_data(object)❷
```

```
          # Generate the JSON representation of original and edited objects
          # then check if the object has actually been modified, and only
          # save if it has.
          object_parsed_again = Chef::JSONCompat.from_json(
            Chef::JSONCompat.to_json(object),:create_additions => false)
          output_parsed_again = Chef::JSONCompat.from_json(
            Chef::JSONCompat.to_json(output), :create_additions => false)
          if object_parsed_again != output_parsed_again
            output.save❸
            self.msg("Saved #{output}")
          else
            self.msg("Object unchanged, not saving")
          end
          output(format_for_display(object)) if config[:print_after]❹
        end
      end
    end
  end
```

❶ Here we're calling the `load` method of the class passed in as the `klass` parameter (this will be a `Class` like `Chef::Node` or `Chef::Role`) to load the object just as we did the earlier examples in this chapter

❷ Next we're calling the `edit_data` method to allow us to edit the object we just loaded. As we examined earlier in this section, behind the scenes this opens a temporary file containing the JSON representation of the loaded object, which we can then edit in our configured text editor. Note that because the `edit_object` method is in the `Chef::UI` class just as the `edit_data` method is, we don't need to addd any prefix to our method call.

❸ Finally, if the object has changed (ie the JSON representation of the object returned from `edit_data` is **not** identical to the JSON of the object passed in) then we call the `save` method of our edited object to save the object back to the Chef server.

❹ We can then optionally display the modified object to the user if the `:print_after` configuration option has been set

As the `edit_object` method we see above uses relatively simple method calls, if we want to customize this process we can easily replicate the behavior we see in this method and modify it as required. Let's test this out in our example plugin by implementing the example we talked about above - we're going to combine non-interactive and interactive Object editing by adding the following code to `.~/.chef/plugins/knife/awesome_object_demo.rb`:

Example 11-10. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
 class AwesomeObjectDemo < Chef::Knife

   deps do
     require 'chef/node' ❶
   end

   def run
       node_object = Chef::Node.load(name_args.first) ❷
       ui.msg "Setting chef_environment of #{name_args.first}" +
          " to #{name_args.last}"
       node_object.chef_environment(name_args.last) ❸
       output = ui.edit_data(node_object) ❹
       ui.msg "Saving #{output}"
       output.save ❺
   end
 end
end
```

❶ Since once again we're going to be editing a `Chef::Node` object, we need to require the `chef/node` class. As in previous examples, this is done in the `deps` block to ensure that we lazily load our plugins dependencies

❷ Just as we did in previous examples, we call the `load` method of the `Chef::Object` class, passing in `name_args.first`, to load our node object from the server.

❸ Here we're calling the `chef_environment` method of our loaded node object and passing in `name_args.last` to set the node's `chef_environment` attribute just as we did in "Editing and Saving Objects Non-Interactively" on page 242

❹ Next we're calling the `edit_data` method of the `ui` object and passing in the object we loaded and edited in steps 2 and 3 to allow the user to interactively edit our object. We're assigning the result of this method call to the `output` variable.

❺ Finally we're saving the `output` object back to the server by calling its `save` method. Note that to keep things simple, we're calling the `save` method without checking whether or not the object has been modified.

Now let's try running our example plugin and compare its results to those we saw when using the `edit_object` method above. We'll pass the parameters `www` and `_default` to our plugin (editor section simulated):

```
 $ knife awesome object demo www _default --config /tmp/part4_examples/knife.rb --local-mode -f js
Setting chef_environment of www to _default

Opens in editor:
```

```
{
  "name": "www",
  "chef_environment": "_default",
  "json_class": "Chef::Node",
  "automatic": {
  },
  "normal": {
  },
  "chef_type": "node",
  "default": {
  },
  "override": {
  },
  "run_list": [

  ]
}
```

Next, save the file and exit the editor. At this point, we should then see the following output:

```
Saved node[www]
```

As we see above, this time when we run our plugin once the node object has been loaded the `chef_environment` attribute is changed to `_default` non-interactively before the object is opened in our configured text editor, allowing us to confirm the change that has been made and add any other changes we might wish. Note that because we are not explicitly checking whether or not the object has been altered before saving it, when we exit out of our text editor the object will be saved regardless of whether or not we make any further changes.

Advanced Node Editing

Readers interested in looking at slightly more advanced object editing behavior may wish to take a look at the `Chef::Knife::NodeEditor` class which lives at `lib/chef/knife/core/node_editor.rb` in the repository (Github Link).

This class augments the techniques and methods we've examined throughout this section with error handling and checks geared specifically to editing Node objects such as:

- Ensuring that no invalid attributes are added to Node objects
- Warning about the creation of new node objects when a node is renamed while being edited
- Ensuring that a properly configured editor has been specified

We won't be using the `Chef::Knife::NodeEditor` class in this chapter, but you can see it in action in the `Chef::Knife::NodeEdit` class which lives at `lib/chef/knife/`

`node_edit.rb` in the repository (Github Link) and implements the `knife node edit` command.

As we saw in our examples in this chapter, Nodes can be edited just the same as any of Chef's other core object types by using methods of the `ui` object like `edit_data` and `edit_object`, but it is nonetheless worth being aware of the `Chef::Knife::NodeEditor` class and the additional safeguards it gives when node objects are being edited.

So far in this section, we've looked at how to load core objects from the server, edit them both interactively and non-interactively, before finally saving them back to the server. The final aspect of working with objects in our Knife plugins we're going to examine is how to create and update objects from JSON or Ruby files.

Creating and Updating Objects from Files

The final tool we're going to add to our toolbox for working with Chef Objects in our Knife plugin classes is the ability to create and update objects from local JSON or Ruby files. A number of default Knife plugins implement this behavior such as `knife node from file` and `knife role from file`, and the underlying techniques are extremely useful for those occasions when we need to interact with files contained in our Chef repository and upload them to the Chef server.

To implement this behavior, we're going to use the final Knife helper class we'll meet in this chapter, the `Chef::Knife::Core::ObjectLoader` class which lives at `lib/chef/knife/core/object_loader.rb` in the repository (Github Link). Unlike the other helper classes we've met so far in this chapter such as the `Chef::Knife::UI` class, the object loader is relatively simple.

The method of the `Chef::Knife::Core::ObjectLoader` class that we will primarily be using here is the `load_from` method. Just as the `load` method defined by Chef's core object classes lets us load an object from the Chef server, the `load_from` method of the `Chef::Knife::Core::ObjectLoader` class lets us do the same thing from a local file.

Other Chef::Knife::Core::ObjectLoader Methods

The `Chef::Knife::Core::ObjectLoader` class defines a number of other methods to assist with locating and verifying files to upload to the Chef server. These can be found in the class definition file at `lib/chef/knife/core/object_loader.rb` in the repository (Github Link) and include:

find_file
: This method searches for the specified file **either** in the directory from which the command was executed, or under the `cookbook_path` defined in knife.rb.

find_all_objects: This method locates all JSON and Ruby files under the specified path - essentially a list of valid file types to potentially load objects from.

find_all_object_dirs
: This method returns a list of all directories under the specified path to search for objects in.

file_exists_and_is_readable?
: As the name suggest, this method verifies that the specified file actually exists, and can be read by Knife.

Let's dive straight into another example and look at how to use this method in practice by writing a plugin to upload a new environment to the server from a JSON file. We'll add the following code to our `~/.chef/plugins/knife/awesome_object_demo.rb` file:

Example 11-11. ~/.chef/plugins/knife/awesome_object_demo.rb

```
module AwesomeInc
 class AwesomeObjectDemo < Chef::Knife

   deps do
    require 'chef/knife/core/object_loader'❶
   end

   def run
      loader = Chef::Knife::Core::ObjectLoader.new(Chef::Environment, ui)❷
      ui.msg "Loading #{name_args.first}"
      environment_object = loader.load_from('environments', @name_args.first)❸
      ui.msg "Saving environment #{name_args.first}"
      environment_object.save❹
   end
 end
end
```

❶ First we're lazily loading our dependency, the `Chef::Knife::Core::Object Loader` class.

❷ Next we need to instantiate the `Chef::Knife::Core::ObjectLoader` class by calling its `new` method. This method takes two parameters - the first is the type of object that will be loaded (`Chef::Environment` in this case) and the second is the `Chef::Knife::UI` object to use (`ui` in this case).

❸ Now we call the `load_from` method of our `Chef::Knife::Core::ObjectLoad er` object. This method takes two parameters - the directory of the `cook book_path` configured in Knife.rb to look for files in if they cannot be found in the current directory, and the name of the file to load.

❹ Finally we're calling the `save` method of our `environment_object` to save it to the server. If the object being uploaded is new, as in this case, it will be created. If it already exists on the server, it will be updated.

Before we can run our example plugin, we're going to need to create a file to upload. As we saw above, the `load_from` method of the `Chef::Knife::Core::ObjectLoader` class will look for files to load in the directory from which the knife command is being run, as well as the configured `cookbook_path` directory so let's paste the following JSON into `/tmp/part4_examples/awesome.json`:

/tmp/part4_examples/awesome.json.

```
{
  "name": "awesome",
  "description": "",
  "json_class": "Chef::Environment",
  "chef_type": "environment",
  "default_attributes": {
    "awesome_level": 100
  },
  "override_attributes": {}
}
```

Now that we have our plugin class and the file for it to load, we can go ahead and run it:

```
$> cd /tmp/part4_examples
$>  knife awesome object demo awesome.json --config /tmp/part4_examples/knife.rb --local-mode
Loading awesome.json
Saving environment awesome.json
```

As we see in the above output, the awesome.json file that we created above is first loaded, then saved. Let's list the environments on our test Chef server now with the `knife environment list` command to verify that our new environment has in fact been saved on the server:

```
$> knife environment list --local-mode --config /tmp/part4_examples/knife.rb
awesome
production
staging
```

As we see here, our `awesome` environment which we described in `awesome.json` has now been uploaded to the server, and can be used alongside the existing `production` and `staging` environments.

```
loader = Knife::Core::ObjectLoader.new(Chef::Node, ui)
updated = loader.load_from('nodes', @name_args[0])
updated.save
```

In this section, we've examined a variety of techniques and classes for working with Chef objects in our Knife plugins. We've looked at how to load objects both from Knife searches and directly, how to edit objects interactively and non-interactively, how to save them back to the server and how to create and update objects from local files. To tie these techniques together with the topics covered throughout the rest of the chapter, we're going to look at a final slightly more advanced example.

Knife Example 3: Tying it all Together

To round off the technical portion of the chapter, we're going to look at one final Knife plugin example. We're going to tie together the topics we've looked at so far in this chapter with a couple of final new plugin code techniques to create a plugin which will allow us to load a node object and set its environment to the specified value and *optionally* allowing the node object to be interactively edited before it is saved back to the server. Unlike in our previous plugin examples, in this example we'll include full input validation and error handling to more accurately represent the sort of plugin we might create in real life.

Let's paste the following code into `~/.chef/plugins/knife/awesome_setup.rb`:

Example 11-12. ~/.chef/plugins/knife/awesome_setup.rb

```
module AwesomeInc
  class AwesomeSetup < Chef::Knife

    # Lazily load our dependencies
    deps do
      require 'chef/node'
      require 'chef/knife/core/object_loader'
    end

    # Specify the banner to show when options are not specified
    banner "knife awesome setup NODE ENVIRONMENT (options)"

    option :interactive_edit, ❶
      :short => "-i",
      :long => "--interactive-edit",
      :description => "Allows node to be edited interactively"

    def run

      # Assign our parameters to variables for convenience
      node_name = name_args.first
      environment_name = name_args.last

      # Verify that our required options have been specified
      if node_name.nil? or environment_name.nil?
          show_usage ❷
          ui.fatal("You must specify a node name and an environment")
```

```
        exit 1
      end

      # Load the node object
      ui.msg "Loading node #{node_name}"
      node_object = Chef::Node.load(node_name)

      # Set the chef_environment of the node object to the
      # specified environment
      ui.msg "Setting environment of #{node_object} to #{environment_name}"
      node_object.chef_environment = environment_name

      # If the -i option was specified
      if config[:interactive_edit]❸
        # Then let the user edit the object interactively
        ui.info "Interactive edit requested, opening #{node_name} in configured editor:"
        edited_object = ui.edit_data(node_object)
      end

      # Finally save the object
      final_object = edited_object || node_object❹
      ui.info "Saving #{final_object}"
      final_object.save
    end
  end
end
```

❶ The `option` method we're calling here makes use of Chef, Inc's mixlib-cli Gem which provides methods for parsing command line options in applications such as Knife. Mixlib-cli is included in the Knife source code as a mixin and the options it supports are documented on the Chef Documentation site. Options specified in this way are added to the `config` object so that we can access them elsewhere in our plugin code.

❷ The `show_usage` method is inherited from our plugin's *superclass* `Chef::Knife`, and prints out a nicely formatted list of the options supported by our plugin combined with global options supported by all Knife plugins. The `show_usage` method is typically called when our plugin has not been passed the correct options, or an invalid option is used.

❸ Here we're checking if the `:interactive_edit` key of the `config` Hash has been set, ie the `-i` option was passed to our plugin. As we saw in step 2, options specified with calls to the `option` method are added to the `config` object so that we can access them in this way.

❹ The "double pipe" operator we're using here is the built-in Ruby "OR" operator which lets us say in this case "assign the value of `edited_object` to `final_object` if it is not nil OR, if `final_object` is nil, the value of `node_object` should be used instead".

Now let's try running our plugin class with some different combinations of options to see how the logic we've incorporated into our plugin class works. First, we're going to run the plugin with none of the required options specified:

```
$> knife awesome setup --config /tmp/part4_examples/knife.rb --local-mode
USAGE: knife awesome setup NODE ENVIRONMENT (options)
    -s, --server-url URL             Chef Server URL
        --chef-zero-port PORT        Port to start chef-zero on
    -k, --key KEY                    API Client Key
        --[no-]color                 Use colored output, defaults to false on
                                       Windows, true otherwise
   -c, --config CONFIG              The configuration file to use
        --defaults                   Accept default values for all questions
    -d, --disable-editing            Do not open EDITOR, just accept the data
                                       as is
    -e, --editor EDITOR              Set the editor to use for interactive
                                       commands
    -E, --environment ENVIRONMENT    Set the Chef environment
    -F, --format FORMAT              Which format to use for output
    -i, --interactive-edit           Allows node to be edited interactively
    -z, --local-mode                 Point knife commands at local repository
                                       instead of server
    -u, --user USER                  API Client Username
        --print-after                Show the data after a destructive operation
    -V, --verbose                    More verbose output. Use twice for max
                                      verbosity
    -v, --version                    Show chef version
    -y, --yes                        Say yes to all prompts for confirmation
    -h, --help                       Show this message
FATAL: You must specify a node name and an envronment
```

In the above output, we see our `banner` being combined with the output of the `show_usage` method to display a nicely formatted list of options supported by our plugin, and instructions for the format in which it expects its parameters. Next, let's try passing our plugin the correct options and having it edit our node object non-interactively:

```
$> knife awesome setup www awesome --config /tmp/part4_examples/knife.rb --local-mode
Loading node www
Setting environment of node[www] to awesome
Saving node[www]
```

This time, we see our plugin class loading the `www` node, setting its environment to `awesome` and then saving it back to the server. Finally, let's try passing the `-i` option to our plugin to specify that we want to interactively edit the node object after the environment is set (editor section simulated):

```
Loading node www
Setting environment of node[www] to production
Interactive edit requested, opening www in configured editor:

Opens in editor:
```

```
{
  "name": "www",
  "chef_environment": "production",
  "json_class": "Chef::Node",
  "automatic": {
  },
  "normal": {
  },
  "chef_type": "node",
  "default": {
  },
  "override": {
  },
  "run_list": [

  ]
}
```

Next, save the file and exit the editor. At this point, we should then see the following output:

```
Saved node[www]
```

This time, we see that passing the `-i` option to our plugin class lets us interactively edit the `www` node object after the plugin has non-interactively set its environment.

We've looked at a number of different techniques and examples for writing Knife plugins in this chapter, but most of the plugins we've looked at thus far have been intentionally relatively trivial. So in the real world, what sort of Knife plugins might we want to create, and how useful really are the techniques we've covered in this chapter? We'll see a number of real-world example of Knife plugins in "Summary & Further Reading" on page 258, but before that we're going to revisit our friends at AwesomeInc and see how Knife plugins might help them solve the final problem we learned about in "Criteria for Customization" on page 9.

Revisiting AwesomeInc - Plugin Best Practices

As we saw in "Criteria for Customization" on page 9, the folks at AwesomeInc are already using Knife's built in commands to drive their Chef workflow, but have been finding that as both the team and the infrastructure they manage grows, the corresponding increase in the frequency and complexity of Chef changes has made it increasingly hard to keep track of what has been changed by who and when. The team have also been finding that they are increasingly overwriting changes made by other team members when simultaneously working on the same cookbook, role or environment.

So given what we've learned in this chapter about writing Knife plugins, what advice can we offer AwesomeInc to answer the question

> How do we stop our developers and Ops staff treading all over each others changes?

Wrap existing plugins where possible

As we've seen already in this chapter, Knife ships with a number of default plugins for accomplishing common tasks - where possible, I would recommend that AwesomeInc create wrapper classes around existing plugins to add functionality allowing them to send notifications of Chef changes to their internal monitoring and chat systems. Just as in general software development, it usually doesn't pay to reinvent the wheel - AwesomeInc should make use of the functionality and classes supplied with Chef to minimize the amount of functionality they have to code yourself in their customizations.

Make use of helper classes

Where wrapping an existing Chef plugin is not possible, for example if they wanted to alter the workflow of a particular Chef task as we did in "Editing and Saving Objects Interactively" on page 244, AwesomeInc should make use of the Chef helper classes we've learned about in this chapter - much of the functionality provided by Knife's default plugins is underpinned by these classes, and using them helps to ensure a unified user experience across all Knife plugins - classes such as `Chef::Knife::ObjectLoader` and the `UI` class also provide methods which can be utilized to implement more complex Knife plugins while still minimizing the amount of "reinventing the wheel" that has to be done.

Make plugins configurable

To make their plugin classes as flexible as possible, I would recommend that AwesomeInc make the most of Knife's built in capability to handle command line options and parameters to add configurable behavior to their plugin classes as we did in "Knife Example 3: Tying it all Together" on page 253. Creating plugins which implement a core subset of functionality while also allowing users to tweak and control the specific options used means that a wide number of use cases and user preferences can be catered for in a much smaller number of plugin classes than if all options were mandatory.

Let the community help

When looking at creating Knife plugins to help solve problems, I would recommend that AwesomeInc investigate community resources such as the Community Knife Plugin Page provided by Chef, Inc. to see if anyone else in the Chef community has already solved that specific problem first. It is often the case that other Chef users will have experienced the same issues and may have open-sourced their solutions. Even if community solutions don't exactly fit AwesomeInc's needs, they can often provide a good starting point.

In the case of the specific problem AwesomeInc are trying to solve here, the desire to solve issues uncovered when scaling Chef and the teams that work with it were the motivation for the development of many of the Chef workflow tools we looked at in "Workflow Tooling" on page 14. Knife-spork in particular was originally de-

veloped at Etsy to solve exactly the sorts of problems that AwesomeInc have been experiencing - as our infrastructure and the number of users we had making Chef changes grew, it became harder to keep track of changes and avoid overwriting other people's changes - the flexibility and power of Knife plugins helped us solve these problems.

Summary & Further Reading

In this chapter we've learned about a number of different aspect of Knife plugin development. We learned about the common skeleton shared by all Knife plugins, before looking at the anatomy of a Knife command's execution. We then looked at how to wrap existing Chef plugins and run Search queries against Chef, creating a test environment to run our example plugins under along the way. We met the various helper classes that are provided with the Chef source code to help us interact with users, and format and display the output of our plugins. Finally, we learned about how to work with the various object types that are stored on the Chef server, before revisiting AwesomeInc to look at how Knife plugins might be utilized to solve one of the problems they're experiencing.

Hopefully the material we've covered in this chapter will help you in the implementation of your own Knife plugins and give you some ideas for plugins to create, but as with many of the customization types we've looked at in this book, there are simply too many possibilities for us to cover everything that could possibly be implemented in a Knife plugin.

Readers interested in diving deeper into Knife plugin development may wish to check out the source code of the Knife plugins we looked at in Chapter 1 in "Workflow Tooling" on page 14 and "Knife Plugins" on page 15. Chef, Inc also maintain a list of community contributed Knife plugins on the Chef Documentation site. You'll find a mix of plugins which add extra functionality to existing Knife commands, plugins which make use of Chef core objects to perform a variety of tasks and plugins which integrate with third party systems such as Cloud provider APIs.

In the next Chapter, we're going to examine the Chef API in much more extensive detail, and look at the methods and data it exposes and how we can use this to create reports and visualizations of our Chef infrastructure.

CHAPTER 12

The Chef API

In Chapter 8, we met several core Chef objects such as `Chef::Node` and `Chef::Environment` which provide us with an abstraction layer around the underlying API used to communicate with Chef Server. In this Chapter, we're going to look at the API itself and how and why you might want to use it. We'll learn:

- What the Chef API is and why it's provided
- The requirements for communicating with Chef's API
- How to authenticate to the API
- How to call API methods
- The endpoints supported by the API and the data they return
- The helper classes Chef provides for communicating with the API.

We'll also look at a number of examples of using the Chef API as we work through the chapter.

Introduction to the Chef API

If we cast our minds back to the Chef architecture diagram we saw in "Chef Architecture" on page 58, we see the *Erchef* component which provides the core API that enables both Chef's client tools and the outside world to communicate with Chef server. To use the technical definition, this is an HTTP-based RESTful API which accepts and returns JSON data to allow us to read from and write to a Chef server.

Entire books have been written around exactly what the term REST means, but for the purposes of this book we're going to use the definition "structured to make it easy for our code to talk to the API efficiently while still allowing us mere mortals to understand the information that is being exchanged"

What this means in practical terms is that to communicate with the Chef API we send HTTP requests to specific Chef server URLs - known as endpoints - and provided the server identifies us an an authorized client, it will then process our request and send us back JSON data with the results. This request could involve reading data **from** the server, writing data **to** the server, or even asking the server to **delete** data.

The fact that Chef's API is HTTP based actually helps implement this functionality, because the HTTP protocol supports different *verbs* such as GET, PUT and DELETE that are designed to tell the server we're communicating what action should be performed with the request data we send it. For example if we send the Chef Server API a DELETE request, the chances are good that we want the server to delete the object we specify in the URL.

Correspondingly, the Chef API will carry out different actions depending on the *verb* sent to its endpoints. For example, a properly authenticated GET request to `/roles/foo` will return that role whereas a `DELETE` request to `/roles/foo` would cause the server to delete it.

The full list of endpoints and verbs supported by the Chef API is documented in Appendix A

So Why Use the Chef API?

Many of the examples we've looked at so far in this book have communicated with the Chef API under the hood, but this has been abstracted away from us by the various helper classes and methods provided with Chef. After all, to use the roles scenario we discussed in the last section, Knife already provides us with a plugin for deleting roles, and the `Chef::Role` class already provides us with the `.load` method to fetch a role from the server. So why on earth would we want to talk to the Chef Server API manually if all of these handy helper classes are already provided for us? Let's look at some of the reasons:

Not everybody uses Ruby
: Although it's the only programming language we've looked at in this book thus far, Ruby is far from the only programming language in use today. As Chef Inc currently

only provides Ruby helper classes, users of other programming languages who want to communicate with the Chef API must talk to it directly, or implement their own helper classes which do so in that language. We'll look at some of these helper classes provided for other languages later on in the chapter.

Some helper classes & methods don't do everything
: The helper classes and methods that we've seen so far in this book which use the API such as the `load` and `save` methods of the core object classes are primarily used within Chef itself to fulfill specific tasks and as a result usually only utilize specific portions of the API. If you find yourself trying to perform tasks not directly supported by an existing class or method, or find yourself combining tasks in a way that would require using multiple helper objects at once, it may prove more convenient to access the Chef API directly.

Lightweight Code
: Although Chef provides a number of useful helper classes as we've seen, the Chef code is often somewhat heavyweight as many helper classes implement API wrapper methods alongside a variety of other functionality - if **all** we want our code to do is talk to the Chef API, it can make our code significantly more lightweight and performant to implement API communication directly, rather than including multiple Chef classes and using a method or two from each.

Because Chef Inc. want to make our lives as easy as possible, the Chef repository actually includes helper classes specifically designed solely for communicating with the Chef API when using Ruby - as we're going to be using Ruby for our code examples we'll meet these helper classes later on in this chapter, but before we do we're going to dive deeper into exactly how the Chef API fits together.

As with many of the other areas of Chef we've looked at in this book, a good understanding of what exactly a helper class is abstracting away from you not only helps you debug issues, but helps you gain a clearer understanding of how that class fits into the wider Chef ecosystem - so let's dive in!

Authenticating to the Chef API

As you might imagine, authenticating to the Chef API is the first step making an API request to Chef Server - we've actually already looked at a number of code examples which authenticate against the Chef API behind the scenes in "Authenticate / Register" on page 66 when we examined the anatomy of Chef runs, and in Figure 11-1 when we examined how Knife plugins authenticate against the Chef server before running.

As we saw in these sections however, the details of this authentication process were abstracted away so that we didn't have to worry about anything other than specifying the correct client key to use. When communicating directly with the API however, we need to properly authenticate our API requests ourselves.

Because the Chef API is HTTP-based, we pass our authentication information as *headers* in the request we make to the server. Let's remind ourselves how headers fit into the structure of HTTP requests with a quick look at what happens when we request the Google UK homepage:

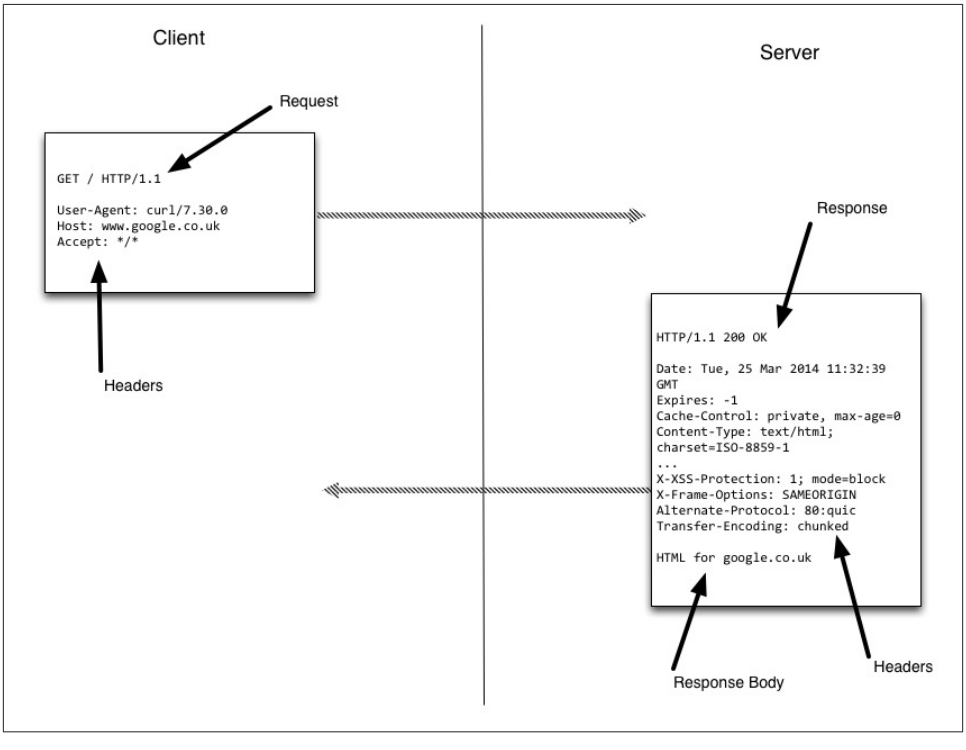

Figure 12-1. HTTP Request / Response

As we see in this diagram, each request to the server comprises the request itself, and an arbitrary number of headers. In this case, the headers tell the server the `User-Agent` string of the software making the request, the `Host` the request is being made against and finally the `Accept` header is used to indicate to the server what response types the client can accept.

The server then returns its own response and headers for various details such as how the response should be cached and whether or not it is encoded or compressed, followed by the actual response body - in this case, the source code for the Google UK homepage.

As with all HTTP based services, the Chef API follows this same pattern of communication. We submit HTTP requests to the server with the correct authentication headers, then the server carries out the requested action if we were authorized to make that request, or returns an error if we were not authorized. We're going to examine the

authentication headers required by Chef server by looking at an example API request - we're going to simulate asking the Chef API for a list of environments:

Example 12-1. Example Chef API Request with Headers

```
GET /environments HTTP/1.1
  Host: chef.mycompany.com:443
  Accept: application/json
  X-Chef-Version: 11.10.0
  X-Ops-Sign: algorithm=sha1;version=1.0;
  X-Ops-Userid: myuser
  X-Ops-Timestamp: 2014-03-26T13:37:00Z
  X-Ops-Content-Hash: 2jmj7l5rfasfgSw0ygaVb/vlWAghYkK/YBwk=
  X-Ops-Authorization-1: BE3NnHeh5yFTiT3ifuwLSPCCYasdfXaRN5oZb4c6hbW0aefI
  X-Ops-Authorization-2: sL4j1qtEZzi/2WeF67Uuytdsdfgbdc5CjgECQwqrym9gCUON
  X-Ops-Authorization-3: yf0p7PrLRCNasdfaHhQ2LWoE/+kTcu0dkasdfvaTghfCDC57
  X-Ops-Authorization-4: 155i+ZlthfasfhbfrtukusbIUGBKUYFjhbvcds3k0i0gqs+V
  X-Ops-Authorization-5: /sLcR7JjQky7sdafIHNEEBQrISktNCDGfFI9o6hbFIayFBx3
  X-Ops-Authorization-6: nodilAGMb166@haC/fttwlWQ2N1LasdqqGomRedtyhSqXA==
```

That's quite a lot of headers! Let's break them down and examine each one in detail:

Host
: The `Host` header is a standard header used in all HTTP requests, and is used to tell the destination server which hostname is being requested. As most HTTP servers are capable of serving multiple hostnames, this allows the server to properly route the request. In this case as we're communicating with the server over HTTPS, we specify `:443` to indicate the SSL port should be used.

Accept
: As we saw in our earlier example, the `Accept` header is used to tell the HTTP server what types of response our HTTP client can handle. Including this header here is technically optional here as Chef will only ever return JSON, but we're including it here for clarity.

X-Chef-Version
: The `X-Chef-Version` header is used to tell the Chef server which Chef-client version has been used to make the request. This ensures that responses are returned in the correct format for the client, since multiple chef-client versions may be communicating with the server. When we're talking to the API manually as we will be in this chapter, this header should be set to the installed chef-client version for the sake of simplicity.

X-Ops-Sign
: The `X-Ops-Sign` header is used to let the Chef server know what algorithm has been used to hash the data included with our request. At the time of writing, this header should always be set to `algorithm=sha1;version=1.0;` as shown above.

X-Ops-Userid
: The `X-Ops-Userid` header tells the server which client is making the request - this allows the server to validate that the headers have been signed using the correct client key.

X-Ops-Timestamp
: The `X-Ops-Timestamp` header contains a timestamp of when the request to the server was made. This timestamp should always be in the format `YYYY-MM-DDTHH:MM:SSZ`. Note that the time portion of this timestamp is prefixed with a `T` and suffixed with a `Z`. For the interested reader, this timestamp is in ISO-8601 compliant format.

X-Ops-Content-Hash
: The `X-Ops-Content-Hash` header contains the body of the request (if we're sending data to the server) hashed using the SHA1 algorithm, and then encoded as Base64. If you're not familiar with these terms, don't worry - we'll look at this in more detail in our code examples later in the chapter.

X-Ops-Authorization-n
: Finally we come to the `X-Ops-Authorization-n` headers. These headers (6 of them in the above example) contain a Base64 encoded specially formatted string which has been signed with the private key of the client making the request. This string, which is composed of a number of items of data about the request is then broken into 60 character long lines, which each resulting line being added to the next `X-Ops-Authorization-n` header. This header is used by the Chef server to ensure that the request has been correctly authenticated, and that the authentication headers from a previous request have not been copied and reused.

To see these headers in action, we're going to dive into a code example to look at how we can construct these headers ourselves, and send properly authenticated requests to the Chef API. But first, as in previous chapters, we need to prepare a test environment for us to safely run our example code under.

Creating a Test Environment

The final thing we need to do before starting to implement our own code to sent requests to the Chef API is to create a test environment in which to safely run our examples. As in previous chapters, we ideally don't want to send requests to a production Chef server while we're learning about how the Chef API works, so we're going to need to use a "mock" Chef server. Luckily for us, Chef Zero (which we've been using to power the running of Chef Client and Knife in *local mode* in previous chapters) is in fact a light-weight Chef Server which supports an identical API to full Chef Servers and should be ideal for our purposes.

We're going to be reusing the test environment we created in "Creating a Test Environment" on page 222 here, so if you haven't already done so you should complete the steps listed in that section. In that test environment, we created a client key to authenticate to the server, and downloaded the Chef Zero source code so that we could make use of the "playground" directory it provides to give us test data to use with our Chef Zero server.

As Chef Zero provides a mock Chef server, it will accept any authentication key it is given. Open source and Hosted Chef servers require valid authentication keys tied to a client before API requests will be permitted.

When we used this test environment to run Knife commands however, we were using the `--local-mode` option of the Knife command to start up Chef Zero for us. To allow us to run our API examples, we're going to need to start up Chef Zero manually ourselves (as we won't be running it as part of a knife or chef-client commanf) and load the "playground" data into it so that we have some data to work with.

I've created a quick bash script which will perform these tasks for us, so let's paste the following code into `/tmp/part4_examples/test_environment`:

Example 12-2. /tmp/part4_examples/test_environment

```
#!/usr/bin/env bash

if [ $# -ne 1 ]
then
  echo "Usage: 'test_environment start|stop'"
  exit $E_BADARGS
fi


case "$1" in

start)  echo "Starting Chef-Zero"
    chef-zero &
    echo "Loading playground data"
    cd /tmp/part4_examples/chef-zero/playground
    knife upload . -c /tmp/part4_examples/knife.rb
    echo "Test environment ready to rock!"
    ;;
stop)  echo  "Stopping Chef-Zero"
    ps -ef | grep /usr/bin/chef-zero | grep -v grep | awk '{print $2}' | xargs kill -9
    ;;
esac
```

Next we need to make the script executable, so you'll need to run the following command:

```
$> chmod 755 /tmp/part4_examples/test_environment
```

Once that's done, you can run the script from inside the `/tmp/part4_examples` with the `start` or `stop` parameters to respectively start and stop our Chef-Zero test environment. Below is an example of the output you should see from each command:

```
$> ./test_environment start

Starting Chef-Zero
Loading playground data
>> Starting Chef Zero (v1.7.2)...
>> Puma (v1.6.3) is listening at http://127.0.0.1:8889
>> Press CTRL+C to stop
Created environments/awesome.json
Created nodes/desktop.json
Created nodes/dns.json
Created nodes/www.json
Created nodes/ldap.json
Created nodes/lb.json
Created data_bags/dns
Created data_bags/passwords
Created environments/production.json
Created environments/staging.json
Created data_bags/users
Created cookbooks/apache2
Created data_bags/passwords/github.json
Created data_bags/passwords/twitter.json
Created data_bags/dns/services.json
Created data_bags/users/sethvargo.json
Created data_bags/users/jkeiser.json
Created data_bags/users/schisamo.json
Created cookbooks/php
Test environment ready to rock!

$> ./test_environment stop

Stopping Chef-Zero
```

We now have a working test server to make API requests against! Before moving on, you should make sure that your test environment is running (execute `./test_environment start` from inside `/tmp/part4_examples`). Now let's dive in to some code!

API Example 1: Authenticating & Making a GET Request

Now that we've examined the authentication headers required by the Chef API and created a test environment to run our example code in, let's look at a code example which constructs the required authentication headers and then makes a call to the API to

request a list of environments from the server. Paste the following code into `/tmp/part4_examples/api_manual_get.rb`:

Example 12-3. /tmp/part4_examples/api_manual_get.rb

```
#!/usr/bin/env ruby

# Require our dependencies (all in Ruby standard library)
require "base64"
require 'digest/sha1'
require "net/http"
require "uri"

# Our client key, client name and Chef server URL
client_key = "/tmp/part4_examples/customizing_chef.pem"
client_name="cctest"
chef_server = "http://127.0.0.1:8889"
# The path to call will be our first command line param
path = ARGV[0]
# We're going to do a GET request, so the request body
# will be empty
body = ""

# Base64 encode the path and body of our request
hashed_path = Base64.encode64(Digest::SHA1.digest(path)).chomp
hashed_body = Base64.encode64(Digest::SHA1.digest(body)).chomp

# Generate Timestamp
timestamp = Time.now.strftime("%Y-%m-%dT%H:%M:%SZ").to_s

# Construct our first set of headers, some of which will be used
# to construct the X-Ops-Authorization headers next
#'X-Ops-Timestamp' => '2014-03-27T13:15:16Z',
headers = {'X-Ops-Timestamp' => timestamp,
           'X-Ops-Userid' => client_name,
           'X-Chef-Version' => '11.10.0',
           'Accept' => 'application/json',
           'X-Ops-Content-Hash' => hashed_body,
           'X-Ops-Sign' => 'version=1.0'
          }

# Construct the specially formatted string needed to generate
# X-Ops-Authorization headers. This string must always be in the format
# and order shown here and contain the headers shown.
canonical_request="Method:GET\\n" +
                  "Hashed Path:#{hashed_path}\\n" +
                  "X-Ops-Content-Hash:#{hashed_body}\\n" +
                  "X-Ops-Timestamp:#{timestamp}\\n" +
                  "X-Ops-UserId:#{client_name}"

# Sign our canonical_request string with the specified client key
signed_hash = `printf \"#{canonical_request}\" | openssl rsautl -sign -inkey #{client_key}`
```

```
# Then encode it as Base 64Then split the encoded hash into 60 char long lines
encoded_hash = Base64.encode64(signed_hash).gsub("\n","").scan(/.{1,60}/m)

# Split up our newly constructed X-Ops-Authorization headers
# and add them to the headers hash
encoded_hash.each_with_index do |text,index|
  headers["X-Ops-Authorization-#{index+1}"] = text
end

# Then output our final headers to screen
puts "Request Headers:"
headers.each{|key,value|puts "#{key}:#{value}"}

# Construct the full path we're going to request from the API
full_path = "#{chef_server}#{path}"
puts ""
puts "Making GET request to #{full_path}"

# Parse the full_path into a URI object
uri = URI.parse(full_path)

# Initialize our Net::HTTP Object
http = Net::HTTP.new(uri.host, uri.port)
# Use SSL if our Chef server URL contained https://
if uri.scheme == "https"
  http.use_ssl = true
  # Ignore self signed SSL Cert Errors
  # Do NOT do this in production code.
  http.verify_mode = OpenSSL::SSL::VERIFY_NONE
end

# Create a new Net::HTTP::Get object, which we populate with
# our headers
request = Net::HTTP::Get.new(uri.request_uri,headers)

# Finally, send the request to the API.
resp = http.request(request)

# Print the response code and body of our request
puts "Response Code: #{resp.code}"
puts "Response Body: #{resp.body}"
```

Now let's try running our example code - we're going to request a list of environments, so we pass the `/environments` endpoint as our parameter:

```
$> ruby api_manual_get.rb /environments
Request Headers:
X-Ops-Timestamp:2014-03-27T14:07:18Z
X-Ops-Userid:cctest
X-Chef-Version:11.10.0
Accept:application/json
```

```
X-Ops-Content-Hash:2jmj7l5rSw0yVb/vlWAYkK/YBwk=
X-Ops-Sign:version=1.0
X-Ops-Authorization-1:THKX0SwaJdTsiQVa4gHPXQk5ZpeAe9O2qeWBggYfwC5C5EoUuKoetcJTGR/D
X-Ops-Authorization-2:/Lbn7TNqwrZJdqxQKeQRXA2wKrc+6XVgOC5iOhtacub1UsgKwBxQNo3mISd9
X-Ops-Authorization-3:2cV1baxL5vr4vVqC5RzcNVBBgua5rxmwh0ku1x7ZA20HyvowrQG+Eav/oyA8
X-Ops-Authorization-4:6kYdopiRHHqTojayvbk+lNALkS6XDKHmqQ8mOZFWnEP8hLtc4TWP4qQeIcOj
X-Ops-Authorization-5:7kYSG4tWXrmK50SqfoCRsS1DCf3J/wCvv5QrkIQlPqGDv8Pwg76BXFpuJ+5P
X-Ops-Authorization-6:hLlc+lVSGJi2moOFXFWpuh8Tvp9MU2t6CCeaO6wSjg==

Making GET request to http://127.0.0.1:8889/environments
Response Code: 200
Response Body: {
  "_default": "http://127.0.0.1:8889/environments/_default",
  "awesome": "http://127.0.0.1:8889/environments/awesome",
  "production": "http://127.0.0.1:8889/environments/production",
  "staging": "http://127.0.0.1:8889/environments/staging"
}
```

In the above output, we see that our code first constructs the headers needed to authenticate to the Chef API before going on to make a GET request to the `/environments` API endpoint. The Chef Zero server then authenticates our request and, once it has determined that we're authorized to issue that request, returns the list of environments to us as JSON.

A detailed understanding of how the Chef API works under the hood is extremely useful and can be invaluable when trying to debug issues or communicate with Chef from programming languages other than Ruby but having to create all that header generation code and run it every time we want to make an API request is somewhat tiresome.

I'm a firm believer in the importance of understanding the fine details of how Chef works under the hood, but I'm also a believer in using helper classes to abstract away complex code where possible. Fortunately, to allow us to make API requests from our Ruby code without having write authentication code ourselves or pick and choose through all of Chef's core object classes, Chef Inc provide us with another helper class designed specifically for making API requests to Chef server, which we're going to meet next.

Chef API libraries have also been created for a number of other popular programming languages including Python, Go and Node.js. If you want to implement the Chef API in a language without an existing client library, hopefully the above code example will give you a good starting point!

The Chef::Rest Class

To abstract away the need to manually construct authentication headers to include with our API requests to Chef Server, Chef provides the `Chef::Rest` class which lives

at `/lib/chef/rest.rb` (Github Link) and defines a number of methods to allow us to use different HTTP *verbs* to make requests to the Chef API. First, let's take a look at how to instantiate the class.

The `initialize` method of the `Chef::Rest` class takes three parameters - the URL of the Chef Server, the name of the client to authenticate API requests against, and lastly the path to the private key to use when generating the authentication headers as we see in the following code:

```
chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)
```

The `Chef::Rest` class also defines methods to allow you to make API calls to the Chef API using specific HTTP verbs. To avoid the need for users of this class to manually parse the JSON they send to and from the API, Chef processes the data fed into these methods to convert it into a JSON object, and converts the data returned from API calls into native Chef objects such as `Chef::Environment` and `Chef::Node` wherever possible. The following *verb* methods are defined:

get_rest
: The `get_rest` method allows you to make a GET request to the API. It takes one mandatory parameter: the API endpoint to send the request to. It also allows you to optionally specify whether you want to return raw JSON data rather than a Chef object, and an optional list of extra headers to send to the server.

delete_rest
: The `delete_rest` method allows you to send a DELETE request to the API. It takes one mandatory parameter: - the API endpoint to send the request to. It also allows you to optionally specify a list of extra headers to send to the server.

post_rest
: The `post_rest` method allows you to send a POST request to the API. It takes two mandatory parameters - the API endpoint to send the request to and the JSON data to be sent to the server. It also allows you to optionally specify a list of extra headers to send to the server.

put_rest
: The `put_rest` method allows you to send a PUT request to the API. It takes two mandatory parameters - the API endpoint to send the request to and the JSON data to be sent to the server. It also allows you to optionally specify a list of extra headers to send to the server.

POST vs PUT?

As we see in the above method list, there are actually two HTTP verbs that can be used to send data to the server, POST and PUT. So which should you use? Put simply, the HTTP specification recommends that POST should be used when an object is to be **created**, and PUT be used when an object is to be **updated**.

We can see this distinction in action in the `save` method of the `Chef::Role` class (my comments added) where the code first tries to update a role using the `put_rest` method, then if a 404 error is encountered (ie the role did not exist on the server), it is created using the `post_rest` method instead:

Example 12-4. Excerpt of lib/chef/role.rb+

```
# Save this role via the REST API
    def save
      begin
        # Try updating the role using the put_rest method
        chef_server_rest.put_rest("roles/#{@name}", self)
      rescue Net::HTTPServerException => e
        raise e unless e.response.code == "404"
        # If we got a 404 error, try creating it using the post_rest method instead
        chef_server_rest.post_rest("roles", self)
      end
      self
    end
```

Now that we've examined the methods provided by the `Chef::Rest` class, let's replicate the code example we created in "API Example 1: Authenticating & Making a GET Request" on page 266, but this time use the `Chef::Rest` class to demonstrate exactly how much code is abstracted away inside this helper class. Paste the following code into `/tmp/part4_examples/api_helper_get.rb`

Example 12-5. /tmp/part4_examples/api_helper_get.rb

```
#!/usr/bin/env ruby

require 'chef'

chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"
path = ARGV[0]

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)

returned_data = rest.get_rest(path)
```

```
puts returned_data
```

Now let's try running our example code, passing it the same `/environments` endpoint as we did in our manual example:

```
$> ruby api_helper_get.rb /environments
{"_default"=>"http://127.0.0.1:8889/environments/_default",
 "awesome"=>"http://127.0.0.1:8889/environments/awesome",
 "production"=>"http://127.0.0.1:8889/environments/production",
 "staging"=>"http://127.0.0.1:8889/environments/staging"}
```

As we see in the above output, we've just authenticated to the server and made exactly the same request as in the example code in "API Example 1: Authenticating & Making a GET Request" on page 266, with the server once again returning us a list of environments. This time however, we did it in much fewer lines of code. When writing code to talk to the Chef API in Ruby, the `Chef::Rest` class will nearly always be the best option and we'll be using this class for the remainder of the examples in this chapter.

In our next example, we're going to look at how to make use of the input and output conversion that the `Chef::Rest` class applies to our API calls to combine multiple API calls to several endpoints - this pattern is often used to retrieve a list of objects names and URLs from the server before then making a GET request to each object.

API Example 2: Combining Multiple API Requests

As we saw in the last section, the `Chef::Rest` helper class makes it extremely easy for us to make API requests directly against the Chef API - so far however, we've only looked at making a single GET request to the `/environments` endpoint. In this example, we're going to look at a slightly more complex example involving making multiple requests to the Chef API, which will also demonstrate Chef's input / output conversion in action.

In this example, we're going to GET a list of environments from the server, GET each environment in that initial list, update the `description` field of those in which that field is currently empty, then PUT the modified environment back to the server. Paste the following code into `/tmp/part4_examples/api_example_2.rb`

Example 12-6. /tmp/part4_examples/api_example_2.rb

```
#!/usr/bin/env ruby

require 'chef'

chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)
```

```
# GET a list of environments by calling the /environments endpoint
environment_list = rest.get_rest("/environments")

# Iterate over our environments list
environment_list.each do |env|
  env_name = env.first❶
  puts "Checking environment #{env_name}"

  # For each environment we got back, GET that environment
  # by calling the /environments/NAME endpoint
  env_object = rest.get_rest(env.last)❷

  # Check if the environment has an empty description attribute
  if env_object.description.empty?❸
    puts "Description empty, updating..."
    env_object.description("This is the #{env_object.name} environment")
    # Save the environment back to the server by making a
    # PUT request to the /environments/name endpoint
    puts "Saving #{env_object.name} environment to the server"
    rest.put_rest("/environments/#{env_object}",env_object)
  else
    puts "Description OK"
  end
end
```

❶ As we saw in "API Example 1: Authenticating & Making a GET Request" on page 266, the `/environments` endpoint returns a name and a URL for each environment. Since we only care about the name of the environment here, we're using the first element of that array only.

❷ Here we're making use of the `last` element of the array for this environment to give us the URL to get that environment directly - since the API already gives us this, there's no need for us to manually construct the URL!

❸ The `Chef::Rest` class has converted the environment downloaded from the server from JSON into a Chef::Environment object automatically - this means we can now access the methods and attributes of the environment without having to parse the returned JSON ourselves.

The results shown below for this command depend on you having created the `awesome.json file used in so that your test environment loads the +awesome`"Creating and Updating Objects from Files" on page 250 environment. The example code will work just fine without this environment, but if you haven't worked through that section yet you will see different output than that shown here.

Let's try running our example now:

```
$> ruby api_example_2.rb
Checking environment _default
Description OK
Checking environment awesome
Description empty, updating...
Saving awesome environment to the server
Checking environment production
Description OK
Checking environment staging
Description OK
```

As we see above, our code iterates over the list of environments retrieved from the server, getting and checking the description of each one. The only environment found to have an empty description was the `awesome` environment, so its description was updated and then the modified environment object was saved back to the Chef server. We can verify that this worked correctly by using Knife to inspect the `awesome` environment with this command:

```
 $ knife environment show awesome -c /tmp/part4_examples/knife.rb
chef_type:           environment
cookbook_versions:
default_attributes:
  awesome_level: 100
description:         This is the awesome environment
json_class:          Chef::Environment
name:                awesome
override_attributes:
```

Now that we've created a slightly more complex example, let's level our Chef API skills up a little further and look at the sorts of errors returned by API calls and how to handle them in our code.

Error Handling

When working with an API like that provided by Chef Server, in addition to knowing how to send requests to the server it's also extremely important to be aware of the possible responses that might be sent back by the server - especially when an error has occurred. Helper classes like `Chef::Rest` make things easier for us of course, but mistakes ranging from using the incorrect Knife key to mis-typing an API endpoint name can results in a range of errors being returned instead of the data we expected.

Being an HTTP based API, all responses (whether errors or not) sent by the server have a *response code* in addition to any data which is returned. The full HTTP specification defines a large number of possible codes for both successful and failed responses, but the number of these you're likely to encounter when talking to the Chef API is much smaller - I've listed them here with a short explanation of each:

Table 12-1. Chef API Response Codes

Response Code	Description
200	OK. The request was successful.
201	Created. The object was created.
401	Unauthorized. The user could not authenticate to the Chef Server.
403	Forbidden. The user which made the request authenticated, but is not authorized to perform the requested action.
404	Not found. The requested object does not exist.
409	Conflict. The object already exists.
412	Precondition Failed. Usually means that a required cookbook, cookbook_version or cookbook dependency is missing.
413	Request entity too large. A request may not be larger than 1000000 bytes.

Interested readers can find a detailed list of the response codes that can be returned by each of the Chef API endpoints on the Chef Documentation site. The full list of response codes defined by the HTTP 1.1 specification at the W3C site.

When `Chef::Rest` encounters an error - that is, the response code is **not** `200` or `201` - it raises an exception to alerts us to that fact. As `Chef::Client` can't predict exactly what we did to trigger the error, it can't decide by itself how to fix the error or if we expected it, so it alerts us by throwing an exception and lets us take things from there.

All exceptions thrown by the `Chef::Rest` object are of the type `Net::HTTPServerException` - this same exception type is also thrown by Ruby's built in `Net::HTTP` libraries when errors occur. The simple fact that we know any HTTP errors returned by the Chef API are always of this particular class means that we can `rescue` only exceptions of that type just as we saw in "Handling Exceptions" on page 46 while ensuring that any other types of exceptions are not accidentally caught.

Let's look at a simple example program which will GET the environment passed in on the command line, and rescues `404` (not found) responses so that we can nicely handle our users accidentally asking for non existent environments. Paste the following code into `/tmp/part4_examples/api_error_handling.rb`:

Example 12-7. /tmp/part4_examples/api_error_handling.rb

```
#!/usr/bin/env ruby

require 'chef'

chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
```

```
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"
env = ARGV[0]

# Our API calls will be inside a begin..rescue block this time
begin
  rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)

  # Make a GET call to /environments/name with the environment
  # name passed in on the command line
  returned_data = rest.get_rest("/environments/#{env}")

  puts returned_data

# Only catch exceptions of type Net::HTTPServerException
rescue Net::HTTPServerException => e
  # If the response code was 404...
  if e.response.code == "404"
    # Print our nice error message and exit with exit code 1
    puts "The environment #{env} does not exist."
    exit 1
  else
    #Re-raise the exception
    raise e
  end
end
```

If we run this command with an environment we know exists passed as the parameter, then we should see the following output:

```
$> ruby api_error_handling.rb production
production
```

If however we then try running our command with a non existent environment passed as the parameter, we should see our nicely formatted error message:

```
 $ ruby api_error_handling.rb foo
The environment foo does not exist.
```

Of course we've only captured a single category of error here in a very simple example, but the same techniques are used to capture other categories of error such as trying to authenticate to Chef server with an invalid client key, or trying to add invalid cookbooks to an environment. If you wanted to write a `rescue` block to cater for several different error response codes at once, you might do something like this:

```
rescue Net::HTTPServerException => e
  case e.response.code
  when "404"
    ...
  when "403"
  ...
  when "401"
    ...
```

```
    else
      #Re-raise the exception
      raise e
    end
  end
```

In the above snippet, we're still only rescuing the single exception class `Net::HTTPServerException`, but since we know all Chef API errors will be raised as this class we're able to respond to multiple error conditions with a tightly-scoped `rescue` block which won't inadvertently trap other exception types in its net.

So far in this chapter, we've looked at how to authenticate to the Chef API, make requests and handle errors - however much of the functionality we've implemented thus far is also implemented by other Chef helper classes. We explored some of the reasons why you might wish to talk to the Chef API directly instead of picking and choosing between multiple helper classes in "So Why Use the Chef API?" on page 260, but is it always the case that you can find a helper class to remove the need to directly talk to the API?

In the next section, we're going to look at some endpoints exposed by the Chef API which are not wrapped by any Chef helper classes and can only be accessed by making direct requests to the API itself.

Secrets of the Chef API

As we saw earlier in the chapter in "So Why Use the Chef API?" on page 260, although much of the Chef API is abstracted by Chef's wrapper classes, there are often considerations such as performance or language choice which mean that you might want to talk to the Chef API directly. In this section of the chapter, we're going to meet two functions of the Chef API which are **only** accessible by communicating directly with the Chef API - at the time of writing, there are currently no Chef classes which wrap the functionality of these API calls.

The _status Endpoint

If we cast our minds back to the Chef architecture diagram we looked at in Figure 3-1, we see that Erchef - the component of Chef server which provides the API we've been communicating to - communicates with several backend components, namely SOLR for search indexes, and PostgreSQL for the backing database. When debugging issues with Chef, or simply when checking the status of components in our monitoring systems, it's desirable to be able to ensure that Erchef is communicating properly with those other components.

To allow us to do this, the Chef API exposes the `_status` endpoint. This endpoint simply takes a GET request, and returns the status of Erchef's communication with Chef server's backend services. Chef-zero, which we've been using for our test environment, does not

provide this endpoint as it is a single process running entirely in memory to mock a Chef server. This means that we're unable to test the output of the `_status` endpoint in our test environment, but I've included the necessary code to run it against your production Chef server should you choose to do so:

Example 12-8. Code to Query the Chef API's _status Endpoint

```
#!/usr/bin/env ruby

require 'chef'

chef_server_url = "" # Your Chef server
client_name = "" # Your client name
signing_key_filename="" # Your client key

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)

returned_data = rest.get_rest("/_status")

puts returned_data
```

For those of you unable (or unwilling) to query the `_status` endpoint of a production Chef server, I've included the output returned by this endpoint for both functional and broken Erchef communication.

If everything is working correctly and Erchef can talk to all of its backend services, a GET request to the `_status` endpoint will return the following output:

```
{ "status":"pong",
  "upstreams":{
    "chef_solr":"pong",
    "chef_sql":"pong"
  }
}
```

If however there is a fault and Erchef cannot talk to all of its backend services, a GET request to the `_status` endpoint will return this output instead, indicating where the problem is:

```
{ "status":"fail",
  "upstreams":{
    "chef_solr":"fail",
    "chef_sql":"pong"
  }
}
```

In the above output for example, we can see that chef_solr is experiencing an issue - this also changes the top level `status` attribute to `failed`.

The `_status` point can be useful for a quick status check on your Chef servers, but it can also be extremely useful when hooking your Chef server up to monitoring systems.

In addition to checking that the required processes for Chef Solr, PostgreSQL and Erchef are running, this endpoint also allows you to verify the state of affairs from Erchef's point of view - that is, if the processes are running but communication isn't working for some reason, you're able to identify and remediate the issue.

Partial Search

The second API feature that we're going to look at in this section is known as "partial search" - before diving in to that however, let's quickly refresh our memories on how "standard" Chef search works. Most search queries are sent to the Chef server from one of the following sources:

- A recipe using the `search` method
- Knife plugins or other classes using the `Chef::Search::Query` object
- A script communicating with the API directly.

In all three cases, behind the scenes a GET request will be sent to the Chef API's `/search` API endpoint. When this happens, Chef will return the entirety of every object which matches the search query. In most cases (unless an API call is made manually without using the `Chef::Rest` object), each object returned in the results will then be converted automatically by Chef into the relevant native object type such as `Chef::Node` or `Chef::Role`.

Let's see this in action with a code example, which will search for nodes with a name matching the parameter we pass in on the command line. Paste the following code into `/tmp/part4_examples/api_search.rb`:

Example 12-9. /tmp/part4_examples/api_search.rb

```
#!/usr/bin/env ruby

require 'chef'
require 'uri'

chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"

index = "node"❶
name = ARGV[0]

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)

returned_data = rest.get_rest("search/#{index}?q=#{URI.escape("name:#{name}")}")❷

puts returned_data
```

❶ Here we're specifying that we want to search the `node` index. As we saw in "Chef Server" on page 60, Chef defines separate indexes for the various object types we can search for.

❷ Next we're sending a GET request to the `/search/INDEX_NAME` endpoint, with our search query passed as a query string parameter as documented on the Chef Documentation site. Note we're also wrapping our query string with `URI.escape` to ensure we don't have any invalid characters in the URL we sent to the Chef API.

Now let's try running our example code, passing the node name `ldap` as the parameter:

```
$? ruby api_search.rb ldap
{"rows"=>[node[ldap]], "start"=>0, "total"=>1}
```

As we see in this output, the results of our search query are contained in the `rows` key of the returned JSON. Inside this key, we see an array containing a single `Chef::Node` object - `node[ldap]` is the string representation of this object, as we're using `puts` to output our results to the screen.

Typically then, the results of a Chef search query will usually be a collection of Nodes, Roles, Environments etc. If you were really only interested in a couple of fields from each result, you then have to iterate over the collection and extract the relevant attributes. This isn't especially difficult - we did this very thing in "Loading Objects: Searching" on page 238 - but it does mean that just downloaded an entire collection of objects, only to throw most of the data away except for the attributes we're interested in.

Searches which might return hundreds or even thousands of results can cause significant bandwidth and memory usage as the data is downloaded onto the node and then processed - this especially applies to node searches, where returned node objects can contain a large number of attributes. Wouldn't it be nice if we were able to tell the Chef server to only return the specific fields of matching objects that we're actually interested in? Well, it turns out that the Chef API actually supports this very behavior, although you'll only find it documented on the Chef Documentation site - none of the existing Chef classes which implement search currently implement this feature which, as you may have already guessed is known as "partial search".

Provided that you're happy communicating directly with the Chef API, which hopefully at this stage in the chapter we now are, making use of partial search is actually very easy. We simply change our GET request to the `search` endpoint into a POST request, and supply it a list of keys defining the attributes of the results that we want to return as the request body.

Let's modify our earlier code example to still search the `node` index, but this time only return the `name` and `chef_environment` attributes of the matching `Chef::Node` objects.

Paste the following code into the file we used before, `/tmp/part4_examples/api_search.rb`:

Example 12-10. /tmp/part4_examples/api_search.rb

```
#!/usr/bin/env ruby

require 'chef'
require 'uri'

chef_server_url = "http://127.0.0.1:8889"
client_name = "cctest"
signing_key_filename="/tmp/part4_examples/customizing_chef.pem"

type = "node"
name = ARGV[0]
keys = {❶
          name: [ 'name' ],
          environment: [ 'chef_environment' ]
        }

rest = Chef::REST.new(chef_server_url, client_name, signing_key_filename)

returned_data = rest.post_rest("search/#{type}?q=#{URI.escape("name:#{name}")}",keys)❷

puts returned_data
```

❶ Here we're defining a Hash containing the list of attributes we want to be returned from our search query. Each element in this hash consists of a name and an array of the object attributes to be grouped under that name in the results. For example here, we've defined our keys as `name: [ name ]` to indicate that the "name" element of the results will contain just the `name` attribute of each result.

❷ Next we're calling the `post_rest` method of our `Chef::Rest` object instead of the `get_rest` method we used earlier, and passing it the `keys` Hash we defined in step 1 as a parameter. The input / output processing carried out by the `Chef::Rest` class will automatically convert this Hash into valid JSON data to sent to the Chef API.

Now let's try running our partial search example and see how the results of the search query change:

```
$ ruby api_search.rb ldap
{"rows"=>[{"url"=>"http://127.0.0.1:8889/nodes/ldap", "data"=>{"name"=>"ldap", "environment"=>"_de
```

This time, in our output we see that instead of a `Chef::Node` object, each row consists of an array of Hashes for each node which contains the URL to the node object, and a `data` hash which contains each of the keys we requested.

It's important to remember that using partial search is not a free optimization however - it moves the processing cost of extracting fields from search results to the Chef server from the individual node. It does however save network bandwidth as each object is processed locally on the server.

Partial search is not currently supported by any native Chef classes, however if you want to make use of it in cookbooks, Chef, Inc. provide the Partial Search Cookbook which defines the `partial_search` method that can be used in recipes in the same way as the existing `search` method. At the time of writing, Chef, Inc. currently plan to include support for partial search in native classes in Chef 11.14.0.

Summary and Further Reading

In this chapter, we've learned a number of different aspects of the Chef API. We started off by looking at why we might want to communicate directly with the Chef API, and created a test environment to allow us to send API requests to a mock Chef Zero server. We then examined the structure of HTTP requests, before looking at how to construct and format the specific authentication headers needed to sign our API requests.

We then looked at a code example to see how we can construct and send GET requests to the Chef API, before meeting the `Chef::Rest` helper class provided to abstract away a lot of the heavy lifting involved in this process. We then learned how to utilize the input / output formatting provided by `Chef::Rest` to combine multiple API requests, using the objects returned by each request to construct the next. We also examined how to respond to errors returned by our API requests, before diving into some API endpoints not implemented by other Chef classes.

Readers interested in learning more about the Chef API may find the following resources useful:

- Appendix A provides a handy quick-reference of the various endpoints supported by the Chef API, and the HTTP verbs supported by each endpoint.
- The Chef Documentation site provides much more comprehensive documentation on each endpoint, along with the specific error codes and data each endpoint might return.
- Readers interested in interfacing with the Chef API from other programming languages may find the Chef API clients written for Python, Go and Node.js useful.

In the next chapter, now that we've covered the various customizations supported by Chef we're going to look at some of the more advanced workflows, tooling and customizations in use today.

APPENDIX A

Chef API Endpoints and Verbs

Full documentation for the Chef Server API can be found at the Chef Documentation site, but I've included a list of supported endpoints and verbs here for reference:

Table A-1. Chef Server API Quick Reference

Endpoint	GET	PUT	POST	DELETE
/clients	√	X	√	X
/clients/NAME	√	√	X	√
/cookbooks	√	X	X	X
/cookbooks/NAME	√	X	X	X
/cookbooks/NAME/version	√	√	X	√
/data	√	X	√	X
/data/NAME	√	X	√	X
/data/NAME/I√EM	√	√	X	√
/environments	√	X	√	√
/environments/NAME	√	√	X	√
/environments/NAME/cookbooks/NAME	√	X	X	X
/environments/NAME/cookbook_versions	X	X	√	X
/environments/NAME/cookbooks	√	X	X	X
/environments/NAME/nodes	√	X	X	X
/environments/NAME/recipes	√	X	X	X
/environments/NAME/roles/NAME	√	X	X	X
/nodes	√	X	√	√
/nodes/NAME	√	√	X	√
/principals/NAME	√	X	X	X
/roles	√	X	√	X

Endpoint	GET	PUT	POST	DELETE
/roles/NAME	√	√	X	√
/roles/NAME/environments	√	X	X	X
/roles/NAME/environments/NAME	√	X	X	X
/sandboxes	√	X	X	X
/sandboxes/ID	X	√	X	X
/search	√	X	X	X
/search/INDEX	√	X	√	X
/_status	√	X	X	X
/users	√	√	√	√