O'REILLY®

2nd Edition

Test-Driven Infrastructure with Chef

BRING BEHAVIOR-DRIVEN DEVELOPMENT
TO INFRASTRUCTURE AS CODE

Stephen Nelson-Smith

O'REILLY®

Test-Driven Infrastructure with Chef

Since *Test-Driven Infrastructure with Chef* first appeared in mid-2011, infrastructure testing has begun to flourish in the web ops world. In this revised and expanded edition, author Stephen Nelson-Smith brings you up to date on this rapidly evolving discipline, including the philosophy driving it and a growing array of tools. You'll get a hands-on introduction to the Chef framework, and a recommended toolchain and workflow for developing your own test-driven production infrastructure.

Several exercises and examples throughout the book help you gain experience with Chef and the entire infrastructure-testing ecosystem. Learn how this test-first approach provides increased security, code quality, and peace of mind.

- Explore the underpinning philosophy that infrastructure can and should be treated as code
- Become familiar with the MASCOT approach to test-driven infrastructure
- Understand the basics of test-driven and behavior-driven development for managing change
- Dive into Chef fundamentals by building an infrastructure with real examples
- Discover how Chef works with tools such as VirtualBox and Vagrant
- Get a deeper understanding of Chef by learning Ruby language basics
- Learn the tools and workflow necessary to conduct unit, integration, and acceptance tests

Stephen Nelson-Smith, a foundational member of the emerging DevOps movement, is a principal consultant at Atalanta Systems, an agile infrastructure consultancy and Opscode training and solutions partner in Europe. He has implemented configuration management and automation systems for clients including Sony and the UK government.

Twitter: @oreillymedia
facebook.com/oreilly

US $34.99 CAN $36.99
ISBN: 978-1-449-37220-0

SECOND EDITION

Test-Driven Infrastructure with Chef

Stephen Nelson-Smith

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

Test-Driven Infrastructure with Chef, Second Edition

by Stephen Nelson-Smith

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: Mike Loukides and Meghan Blanchette
Production Editor: Melanie Yarbrough
Proofreader: Elise Morrison
Indexer: WordCo Indexing Services

Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2013: Second Edition

Revision History for the Second Edition:

2013-10-10: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449372200* for release details.

ISBN: 978-1-449-37220-0

[LSI]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

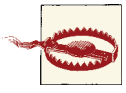
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/test-driven-infra-chef*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Writing the first edition of this book was an order of magnitude harder than I could ever have imagined. I think this is largely because alongside writing a book I was also writing software. Trying to do both things concurrently took up vast quantities of time, for which many people are owed a debt of gratitude for their patience and support.

Writing the second edition, however, made the first one look like a walk in the park. Since the first edition there's been a huge explosion in philosophies, technologies and enthusiastic participants in the field of TDI, all of which and whom are moving and developing fast. This has not only added massively to the amount there is to say on the subject but it has made it a real challenge to keep the book up to date.

So the gratitude is bigger than before too! Firstly, to my wonderful family, Helena, Corin, Wilfrid, Atalanta and Melisande (all of whom appear in the text)—you've been amazing, and I look forward to seeing you all a lot more. Helena, frankly, deserves to be credited as a co-author. She has proofed, edited, improved, and corrected for the best part of two years, and has devoted immeasurable hours to supporting me, both practically and emotionally. There is no way this book could have been written without her input—I cannot express how lucky I am to have her as my friend, colleague, and beloved.

The list of Opscoders to thank is also longer, and is testament to the success of both the company and its product. My understanding would be naught were it not for the early support of Joshua Timberman, Seth Chisamore and Dan DeLeo. However, the second edition owes also a debt of thanks to Seth Vargo, Charles Johnson, Nathen Harvey, and Sean O'Meara. Further thanks to Chris Brown, on whose team I worked as an engineer for six months, giving me a deeper insight into the workings of Chef, and the depths of brilliance in the engineering team.

Inspirational friends, critics, reviewers and sounding boards include Aaron Peterson, Bryan Berry, Ian Chilton, Matthias Lafeldt, Torben Knerr and John Arundel. Special mention must go firstly to Lindsay Holmwood, who first got me thinking about the subject, and has continued to offer advice and companionship, and secondly to Fletcher Nichol, who has been a constant friend and advisor, and has endured countless hours of being subjected to pairing with me in Emacs and Tmux, on Solaris! It must also not be forgotten that without the early support of Trang and Brian, formerly of Sony Computer Entertainment Europe—the earliest adopters and enthusiastic advocates of my whole way of doing Infrastructure as Code—I doubt I would have achieved what I have achieved.

The development and maintenance of Cucumber-Chef has been educational and fascinating—Jon Ramsey and especially Zachary Patten deserve particular thanks for this. The project has seen many enthusiastic adopters, and has evolved to do all sorts of things I would never have imagined. Its reincarnate future as *TestLab* is in safe hands.

I've been fortunate beyond measure to work with a team of intelligent and understanding people at Atalanta Systems—all of whom have put up with my book-obsessed scattiness for the best part of two years—Kostya, Sergey, Yaroslav, Mike, Herman, and Annie…you're all awesome!

Lastly, and perhaps most importantly—to my incredibly patient and supportive editor Meghan Blanchette—thank you a million times. I think you'll agree it was worth the wait.

I dedicate this book to my Grandfather, John Birkin, himself one of the earliest computer programmers in the UK. You taught me to program some thirty years ago, and it is the greatest blessing to me that you have been able to see the fruit of the seeds that you sowed.

CHAPTER 1

The Philosophy of Test-Driven Infrastructure

When the first edition of this book was published in late summer 2011, there was broad skepticism in response to the idea of testing infrastructure code and only a handful of pioneers and practitioners.

Less than a year later at the inaugural #ChefConf, the Chef user conference, two of the plenary sessions and a four-hour hack session were devoted to testing. Later that year at the Chef Developer Summit, where people meet to discuss the state and direction of the Chef open source project, code testing and lifecycle practices and techniques emerged as top themes that featured in many heavily attended sessions—including one with nearly 100 core community members.

Infrastructure testing is a hugely topical subject now, with many excellent contributors furthering the state of the art. The tools and approaches that make up the infrastructure testing ecosystem have evolved significantly. It's an area with a high rate of change and few established best practices, and it is easy to be overwhelmed at the amount to learn and bewildered at the range of tools available. This book is intended to be the companion for those new to the whole idea of infrastructure as code, as well as those who have been working within that paradigm and are now looking fully to embrace the need to prioritize testing.

This update is much expanded and provides a thorough introduction to the philosophy and basics of test-driven development and behavior-driven development in general, as well as the application of these techniques to the writing of infrastructure code using Chef. It includes an up-to-date introduction to the Chef framework and discusses the most widely used and popular tooling in use with Chef, before providing a recommended toolkit and workflow to guide adoption of test-driven infrastructure in practice.

Underpinning Philosophy

There are two fundamental philosophical points upon which this book is predicated:

1. Infrastructure can and should be treated as code.
2. Infrastructure developers should adhere to the same principles of professionalism as other software developers.

While there are a number of implications that follow from these assumptions, the primary one with which this book is concerned is that all infrastructure code must be thoroughly tested, and that the most effective way to develop infrastructure code is test-first, allowing the writing of the tests to drive and inform the development of the infrastructure code. However, before we get ahead of ourselves, let us consider our two axiomatic statements.

Infrastructure as Code

> "When deploying and administering large infrastructures, it is still common to think in terms of individual machines rather than view an entire infrastructure as a combined whole. This standard practice creates many problems, including labor-intensive administration, high cost of ownership, and limited generally available knowledge or code usable for administering large infrastructures."
>
> — Steve Traugott and Joel Huddleston

> "In today's computer industry, we still typically install and maintain computers the way the automotive industry built cars in the early 1900s. An individual craftsman manually manipulates a machine into being, and manually maintains it afterwards.
>
> The automotive industry discovered first mass production, then mass customization using standard tooling. The systems administration industry has a long way to go, but is getting there."
>
> — Steve Traugott and Joel Huddleston

These two statements came from the prophetic *www.infrastructures.org* at the very start of the last decade. More than 10 years later, a whole world of exciting developments have taken place: developments that have sparked a revolution, and given birth to a radical new approach to the process of designing, building, and maintaining the underlying IT systems that make web operations possible. At the heart of that revolution is a mentality and toolset that treats infrastructure as code.

We believe in this approach to the designing, building, and running of Internet infrastructures. Consequently, we'll spend a little time exploring its origin, rationale, and principles before outlining the risks of the approach—risks that this book sets out to mitigate.

The Origins of Infrastructure as Code

Infrastructure as code is an interesting phenomenon, particularly for anyone wanting to understand the evolution of ideas. It emerged over the last six or seven years in response to the juxtaposition of two pieces of disruptive technology—utility computing and second-generation web frameworks.

The ready availability of effectively infinite compute power at the touch of a button, combined with the emergence of a new generation of hugely productive web frameworks, brought into existence a new world of scaling problems that had previously only been witnessed by the largest systems integrators. The key year was 2006, which saw the launch of Amazon Web Services' Elastic Compute Cloud (EC2), just a few months after the release of version 1.0 of Ruby on Rails the previous Christmas. This convergence meant that anyone with an idea for a dynamic website—an idea that delivered functionality or simply amusement to a rapidly growing Internet community—could go from a scribble on the back of a beermat to a household name within weeks.

Suddenly, very small developer-led companies found themselves facing issues that were previously tackled almost exclusively by large organizations with huge budgets, big teams, enterprise-class configuration management tools, and lots of time. The people responsible for these websites that had become huge almost overnight now had to answer questions such as how to scale databases, how to add many identical machines of a given type, and how to monitor and back up critical systems. Radically small teams needed to be able to manage infrastructures at scale and to compete in the same space as big enterprises, but with none of the big enterprise systems.

It was out of this environment that a new breed of configuration management tools emerged. Building on the shoulders of existing open source tools like *CFEngine*, *Puppet* was created in part to facilitate tackling these new problems.

Given the significance of 2006 in terms of the disruptive technologies we describe, it's no coincidence that in early 2006 Luke Kanies published an article on "Next-Generation Configuration Management" in *;login:* (the USENIX magazine), describing his Ruby-based system management tool, Puppet. Puppet provided a high level domain specific language (DSL) with primitive programmability, but the development of *Chef* (a tool influenced by Puppet, and released in January 2009) brought the power of a third-generation programming language to system administration. Such tools equipped tiny teams and developers with the kind of automation and control that until then had only been available to the big players and expensive in-house or proprietary software. Furthermore, being built on open source tools and released early to developer communities, allowed these tools to rapidly evolve according to demand, and they swiftly became more powerful and less cumbersome than their commercial counterparts.

Thus a new paradigm was introduced—infrastructure as code. In it, we model our infrastructure with code, and then design, implement, and deploy our web application

infrastructure with software best practices. We work with this code using the same tools as we would with any other modern software project. The code that models, builds, and manages the infrastructure is committed into source code management alongside the application code. We can then start to think about our infrastructure as redeployable from a code base, in which we are using the same kinds of software development methodologies that have developed over the last 20 years as the business of writing and delivering software has matured.

This approach brings with it a series of benefits that help the small, developer-led company solve some of the scalability and management problems that accompany rapid and overwhelming commercial success:

Repeatability
: Because we're building systems in a high-level programming language and committing our code, we start to become more confident that our systems are ordered and repeatable. With the same input, the same code should produce the same output. This means we can now be confident (and ensure on a regular basis) that what we believe will recreate our environment really *will* do that.

Automation
: By utilizing mature tools for deploying applications, which are written in modern programming languages, the very act of abstracting out infrastructures brings us the benefits of automation.

Agility
: The discipline of source code management and version control means we have the ability to roll forward or backward to a known state. Because we can redeploy entire systems, we are able to drastically reconfigure or change topology with ease, responding to defects and business-driven changes. In the event of a problem, we can go to the commit logs and identify what changed and who changed it. This is made all the easier because our infrastructure code is just text, and as such can be examined and compared using standard file comparison tools, such as *diff*.

Scalability
: Repeatability and automation make it possible to grow our server fleet easily, especially when combined with the kind of rapid hardware provisioning that the cloud provides. Modular code design and reuse manages complexity as our applications grow in features, type, and quantity.

Reassurance
: While all the benefits bring reassurance in their way, in particular, the fact that the architecture and design of our infrastructure is modeled—and not merely implemented—in code means that we may reasonably use the source code as documentation and see at a glance how the systems work. This knowledge repository mitigates the risk of only a single sysadmin or architect having the full

understanding of how the system hangs together. That is risky—this person is now able to hold the organization ransom, and should they leave or become ill, the company is endangered.

Disaster recovery
: In the event of a catastrophic event that wipes out the production systems, if our entire infrastructure has been broken down into modular components and described as code, recovery is as simple as provisioning new compute power, restoring from backup, and redeploying the infrastructure and application code. What may have been a business-ending event in the old paradigm of custom-built, partially automated infrastructure becomes a manageable outage with procedures we can test in advance.

Infrastructure as code is a powerful concept and approach that promises to help repair the split-brain phenomenon witnessed so frequently in organizations where developers and system administrators view each other as enemies, to the detriment of the common good. Through co-design of the infrastructure code that runs an application, we give operational responsibilities to developers. By focusing on design and the software lifecycle, we liberate system administrators to think at higher levels of abstraction. These new aspects of our professions help us succeed in building robust, scaled architectures. We open up a new way of working—a new way of cooperating—that is fundamental to the emerging DevOps movement.

The Principles of Infrastructure as Code

Having explored the origins and rationale for managing infrastructure as code, we now turn to the core principles we should put into practice to make it happen.

Adam Jacob, co-founder of Opscode and creator of Chef, says that there are two high-level steps:

1. Break the infrastructure down into independent, reusable, network-accessible services.
2. Integrate these services in such a way as to produce the functionality our infrastructure requires.

Adam further identifies 10 principles that describe what the characteristics of the reusable primitive components look like. His essay—Chapter 5 of *Web Operations*, ed. John Allspaw & Jesse Robbins (O'Reilly)—is essential reading, but I will summarize his principles here:

Modularity
: Our services should be small and simple—think at the level of the simplest free-standing, useful component.

Cooperation
: Our design should discourage overlap of services and should encourage other people and services to use our service in a way that fosters continuous improvement of our design and implementation.

Composability
: Our services should be like building blocks—we should be able to build complete, complex systems by integrating them.

Extensibility
: Our services should be easy to modify, enhance, and improve in response to new demands.

Flexibility
: We should build our services using tools that provide unlimited power to ensure we have the (theoretical) ability to solve even the most complicated problems.

Repeatability
: With the same inputs, our services should produce the same results in the same way every time.

Declaration
: We should specify our services in terms of *what* we want to do, not *how* we want to do it.

Abstraction
: We should not worry about the details of the implementation, and think at the level of the component and its function.

Idempotence
: Our services should be configured only when required; action should be taken only once.

Convergence
: Our services should take responsibility for their own state being in line with policy; over time, the overall system will tend to correctness.

In practice, these principles should apply to every stage of the infrastructure development process—from low-level operations such as provisioning (cloud-based providers with a published API are a good example), backups, and DNS, up through high-level functions such as the process of writing the code that abstracts and implements the services we require.

This book concentrates on the task of writing infrastructure code that meets these principles in a predictable and reliable fashion. The key enabler in this context is a powerful, declarative configuration management system that enables engineers (I like the term *infrastructure developer*) to write executable code that both describes the shape,

behavior, and characteristics of the infrastructure that they are designing, and when actually executed, results in that infrastructure coming to life.

The Risks of Infrastructure as Code

Although the potential benefits of infrastructure as code are hard to overstate, it must be pointed out that this approach is not without its dangers. Production infrastructures that handle high-traffic websites are hugely complicated. Consider, for example, the mix of technologies involved in a large content management system installation. We might easily have multiple caching strategies, a full-text indexer, a sharded database, and a load-balanced set of web servers. That is a significant number of moving parts for the infrastructure developer to manage and understand.

It should come as no surprise that the attempt to codify complex infrastructures is a challenging task. As I visit clients embracing the approaches outlined in this chapter, I see similar problems emerging as they start to put these ideas into practice:

- Sprawling masses of infrastructure code
- Duplication, contradiction, and a lack of clear understanding of what it all does
- Fear of change; a sense that we dare not meddle with the manifests or recipes because we're not entirely certain how the system will behave
- Bespoke software that started off well-engineered and thoroughly tested, but is now littered with TODOs, FIXMEs, and quick hacks
- Despite the lofty goal of capturing the expertise required to understand an infrastructure in the code itself, a sense that the organization would be in trouble if one or two key people leave
- War stories of times when a seemingly trivial change in one corner of the system had catastrophic side effects elsewhere

These issues have their roots in the failure to acknowledge and respond to a simple but powerful side effect of treating our infrastructure as code: if our environments are effectively software projects, then they should be subject to the same meticulousness as our application code. It is incumbent upon us to make sure we apply the lessons learned by the software development world in the last 10 years as they have strived to produce high quality, maintainable, and reliable software. It's also incumbent upon us to think critically about some of the practices and principles that have been effective in that world and to begin introducing our own practices that embrace the same interests and objectives. Unfortunately, many who embrace infrastructure as code have had insufficient exposure to or experience with these ideas.

There are six areas where we need to focus our attention to ensure that our infrastructure code is developed with the same degree of thoroughness and professionalism as our application code:

Design
: Our infrastructure code should seek to be simple and iterative, and we should avoid feature creep.

Collective ownership
: All members of the team should be involved in the design and writing of infrastructure code and, wherever possible, code should be written in pairs.

Code review
: The team should be set up to pair frequently and to see regular notifications when changes are made.

Code standards
: Infrastructure code should follow the same community standards as the Ruby world; when standards and patterns have grown up around the configuration management framework, the standards and patterns should be adhered to.

Refactoring
: This should happen at the point of need as part of the iterative and collaborative process of developing infrastructure code; however, it's difficult to do this without a safety net in the form of thorough test coverage of one's code.

Testing
: Systems should be in place to ensure that one's code produces the environment needed and that any changes have not caused side effects that alter other aspects of the infrastructure.

I would argue that good practice in all six of these areas is a natural by-product of bringing development best practices to infrastructure code—in particular by embracing the idea of test-first programming. Good leadership can lead to rapid progress in the first five areas with very little investment in new technology. However, it is indisputable that the final area—that of testing infrastructure automation—is a difficult endeavor. As such, it is the subject of this book: a manifesto for bravely rethinking how we develop infrastructure code.

Professionalism

The discipline of software development is a young one. It was not until the early 1990s that the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery began to recognize software engineering as a profession. The last 15 years alone have seen significant advances in tooling, methodology, and philosophy. The discipline of infrastructure development is younger still. It is imperative that those

embarking upon or moving into a career involving infrastructure development absorb the hard lessons learned by the rest of the software industry over the previous few decades, avoid repeating these mistakes, and hold themselves accountable to the same level of professionalism.

Robert C. Martin in, *Clean Code: A Handbook of Agile Software Craftsmanship* (Prentice Hall), draws upon the Hippocratic oath as a metaphor for the standards of professionalism demanded within the software development industry: *Primum non nocere*—first do no harm. This is the foundational ethical principal that all medical students learn. The essence is that the cost of action must be considered. It may be wiser to take no action or not to take a specified action in the interests of not harming the patient. The analogy holds as a software developer. Before intervening to add a feature or to fix a bug, be confident that you aren't making things worse. Robert C. Martin suggests that the kinds of harm a software developer can inflict can be classified as *functional* and *structural*.

By *functional* harm, we mean the introduction of bugs into the system. A software professional should strive to release bug-free software. This is a difficult goal for developer and medical practitioner alike; granted that software (and humans) are highly complicated systems, as professionals we must make it our mantra to "do no harm." We won't ever be able to eradicate mistakes, but we can accept responsibility for them, and we can ensure we learn from them and put mechanisms in place to avoid repeating them.

By *structural* harm we mean introducing inflexibility into our systems, making software harder to change. To put the concept positively, it must be possible to make changes without the cost of change being exorbitantly high.

I like this analogy. I think it can also be taken a little further. Of all medical professionals, the one I would most want to be certain was observing the Hippocratic oath would be a brain surgeon. The cost of error is almost infinitely higher when operating upon the brain than when, for example, operating on a minor organ, or performing orthopedic surgery. I think this applies to the subject of this book, too.

As infrastructure developers, the software we have written builds and runs the entire infrastructure on which our production systems, the applications, and ultimately the business, operate. The cost of a bug, or of introducing structural inflexibility to the underpinning infrastructure on which our business runs, is potentially even greater than that of a bug in the application code itself. An error in the infrastructure could lead to the entire system becoming compromised or could result in an outage rendering all dependent systems unavailable.

How, then, can we take responsibility for, and excel in, our oath-keeping? How can we introduce no bugs and maintain system flexibility? The answer lies in testing.

The only way we can be confident that our code works is to test it. Thoroughly. Test it under various conditions. Test the happy path, the sad path, and the bad path. The happy path represents the default scenario, in which there are no exceptional or error conditions. The sad path shows that things fail when they should. The bad path shows the system when fed absolute rubbish. In the case of infrastructure code, we want to verify that changes made for one platform don't cause unexpected side effects on other platforms. The more we test, the more confident we are.

When it comes to protecting and guaranteeing the flexibility of our code, there's one easy way to be confident of code flexibility. Flex it. We want our code to be easy to change. To be confident that it is easy to change, we need to make easy changes. If those easy changes prove to be difficult, we need to change the way the code works. We must be committed to regular refactoring and regular small improvements across the team. This might seem to be at odds with the principle of doing no harm. Surely the more changes we make, the more risk we are taking on. Paradoxically, this isn't actually the case. It is far, far riskier to leave the code to stagnate with little or no attention.

As infrastructure developers, if we're afraid to make changes to our code, that's a big red flag. The biggest reason people are afraid to make changes is that they aren't confident that the code won't break. That's because they don't have a test harness to protect them and catch the breaks. I like to think of refactoring as a little like walking along a curbstone. When you have six inches to fall, you won't have any fear at all. If you had to walk along a beam, four inches in width, stretching between two thirty story buildings, I bet you'd be scared. You might be so scared that you wouldn't even set out. The same is so with refactoring. When you have a fully tested code base, making changes is done with confidence and zeal. When you have no tests at all, making changes is avoided or undertaken with fear and dread.

The trouble is, testing takes time. Lots of testing takes lots of time. In the world of infrastructure code, testing takes even more time because sometimes the feedback loops are significantly longer than traditional test scenarios. This makes it imperative that we automate our testing. Testing, especially for complicated, disparate systems, is also difficult. Writing good tests for code is hard to do. That makes it imperative for us to write code that is easy to test. The best way to do that is to write the tests first. We'll discuss this in more depth later, but the essential and applicable takeaway is that consistent, automated, and quality testing of infrastructure code is mandatory for the DevOps professional.

At this stage it's important to acknowledge and address an obvious objection. As infrastructure developers we are asked to make a call with respect to a risk/time ratio. If it delays a release by three weeks, but delivers 100% test coverage, is this the right approach, given our maxim "do no harm"?

As is the case in many such trade-offs, there is an asymptotic curve describing a diminishing return after a certain amount of time and test coverage. It is a big step in the

right direction to be making the decision consciously. Consider what part of the "brain" we are about to cut in to, what functions it performs for the body corporeal or corporate, as it were, and where we draw our line will become clear.

I'll summarize by making a bold philosophical statement that underpins the rest of this book:

Testing our infrastructure code, thoroughly and repeatably, is non-negotiable, and is an essential component of the infrastructure developer's work.

This book sets out to provide encouragement for those learning to test their infrastructure code, and guidance for those already on the path. It is a call to arms for infrastructure developers, DevOps professionals, if you like, to maximize the quality, reliability, repeatability, and production-readiness of their work.

CHAPTER 2

An Introduction to Ruby

Before we go any further, I'm going to spend a little time giving you a quick overview of the basics of the Ruby programming language. If you're an expert, or even a novice Ruby developer, do feel free to skip this section. However, if you've never used Ruby, or rarely programmed at all, this should be a helpful introduction. The objective of this section is to make you feel comfortable looking at infrastructure code. The framework we're focusing our attention on in this book—Chef—is both written in Ruby, and fundamentally *is* Ruby. Don't let that scare you—you really only need to know a few things to get started. I'll also point you to some good resources to take your learning further. Later in the book, we'll be doing more Ruby, but I will explain pretty much anything that isn't explicitly covered in this section. Also, remember we were all once in the beginners' seat. One of the great things about the Chef community is the extent to which it's supporting and helpful. If you get stuck, hop onto IRC and ask for help.

What Is Ruby?

Let's start right at the very beginning. What is Ruby? To quote from the very first Ruby book I ever read, the delightfully eccentric *Why The Lucky Stiff's (poignant) Guide to Ruby*:

> My conscience won't let me call Ruby a computer language. That would imply that the language works primarily on the computer's terms. That the language is designed to accommodate the computer, first and foremost. That therefore, we, the coders, are foreigners, seeking citizenship in the computer's locale. It's the computer's language and we are translators for the world.
>
> But what do you call the language when your brain begins to think in that language? When you start to use the language's own words and colloquialisms to express yourself. Say, the computer can't do that. How can it be the computer's language? It is ours, we speak it natively!

> We can no longer truthfully call it a computer language. It is coderspeak. It is the language of our thoughts.
>
> — *http://bit.ly/1fieouZ*

So, Ruby is a very powerful, very friendly language. If you like comparisons, I like to think of Ruby as being a kind of hybrid between LISP, Smalltalk, and Perl. I'll explain why a bit later. You might already be familiar with a programming language—Perl, or Python, or perhaps C or Java. Maybe even BASIC or Pascal. As an important aside, if you consider yourself to be a system administrator, and don't know *any* programming languages, let me reassure you—you already know heaps of languages. Chances are you'll recognize this:

```
divert(-1)
divert(0)
VERSIONID(`@(#)sendmail.mc      8.7 (Linux) 3/5/96')
OSTYPE(`linux')
#
# Include support for the local and smtp mail transport protocols.
MAILER(`local')
MAILER(`smtp')
#
FEATURE(rbl)
FEATURE(access_db)
# end
```

Or possibly this:

```
Listen 80
<VirtualHost *:80>
    DocumentRoot /www/example1
    ServerName www.example.com

    # Other directives here
</VirtualHost>

<VirtualHost *:80>
    DocumentRoot /www/example2
    ServerName www.example.org

    # Other directives here
</VirtualHost>
```

What about this?

```
LOGGER=/usr/bin/logger
DUMP=/sbin/dump
# FSL="/dev/aacd0s1a /dev/aacd0s1g"
FSL="/usr /var"
NOW=$(date +"%a")
LOGFILE="/var/log/dumps/$NOW.dump.log"
TAPE="/dev/sa0"
```

```
mk_auto_dump(){
local fs=$1
local level=$2
local tape="$TAPE"
local opts=""

opts="-${level}uanL -f ${tape}"
# run backup
$DUMP ${opts} $fs
if [ "$?" != "0" ];then
$LOGGER "$DUMP $fs FAILED!"
echo "*** DUMP COMMAND FAILED - $DUMP ${opts} $fs. ***"
else
$LOGGER "$DUMP $fs DONE!"
fi
}
```

Or finally, this:

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
  $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
  $(CC) $(CFLAGS) $< -o $@
```

If you're anything like me, you'll know what all four of these are right away. You might know exactly what they do. They almost certainly don't scare you; you will recognize some of it, and you'd know where to go to find out more. My aim is to get you to the same point with Ruby. The thing is, a sysadmin knows a ton of languages; they just mostly suck quite badly. Thankfully, Ruby doesn't suck at all—Ruby is awesome—it's easy to use and highly capable.

Grammar and Vocabulary

All languages have grammar and vocabulary. Let's cover the basic vocabulary and grammar of Ruby. One of the best ways to learn a language is to have a play about in a *REPL*. REPL stands for "Read, Evaluate, Print, Loop." A REPL is an interactive environment in which the user writes code, and the shell interprets that code and returns the results immediately. They're ideal for rapid prototyping and language learning, because the feedback loop is so quick.

The idea of a REPL originated in the world of LISP. Its implementation simply required that three functions be created and enclosed in an infinite loop function. Permit me some hand-waving, as this hides much deep complexity, but at the simplest level the three functions are:

read
: Accept an expression from the user, parse it, and store it as a data structure in memory.

eval
: Ingest the data structure and evaluate it. This translates to calling the function from the initial expression on each of the arguments provided.

print
: Display the result of the evaluation.

We can actually write a Ruby REPL in one line of code:

```
$ ruby -e 'loop { p eval gets }'
1+1
2
puts "Hello"
Hello
nil
5.times { print 'Simple REPL' }
Simple REPLSimple REPLSimple REPLSimple REPLSimple REPL5
```

The first thing to note is every expression has a return value, without exception. The result of the expression *1+1* is `2`. The result of the expression `"puts "Hello""` is not `"hello"`. The result of the expression is nil, which is Ruby's way of expressing nothingness. I'm going to dive in right now, and set your expectations. Unlike languages such as Java or C, nil is not a special value or even a keyword. It's just the same as everything else. In Ruby terms, it's an object—more on this in a moment. For now, every expression has a return value, and in a REPL, we will always see this.

The functions in our basic REPL should be familiar—we have a `loop`, we have an `eval`, the `p` function prints the output of the `eva`, and `gets` reads from the keyboard. Obviously this is a ridiculously primitive REPL, and very brittle and unforgiving:

```
$ ruby -e 'loop { p eval gets }'
forgive me!
-e:1:in `eval': undefined method `me!' for main:Object (NoMethodError)
  from -e:1:in `eval'
  from -e:1:in `block in <main>'
  from -e:1:in `loop'
  from -e:1:in `<main>'
```

Thankfully Ruby ships with a REPL—Interactive Ruby, or *irb*. The Ruby REPL is launched by typing `irb` in a command shell. It also takes the handy command switch `--simple-prompt`, which declutters the display for our simple use cases.

```
$ irb
irb(main):001:0> exit
$ irb --simple-prompt
>>
```

Go ahead and try talking to irb:

```
>> hello
NameError: undefined local variable or method `hello' for main:Object
  from (irb):1
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

Methods and Objects

irb isn't very communicative. I'd like to draw your attention to two words in the preceding output—*method* and *Object*. This introduces one of the most important things to understand about Ruby. Ruby is a *pure object-oriented* language. Object-oriented languages encourage the idea of modeling the world by designing programs around classes, such as `Strings` or `Files`, together with classes we define ourselves. These classes and class hierarchies reflect important general properties of individual nails, horseshoes, horses, kingdoms, or whatever else comes up naturally in the application we're designing. We create instances of these classes, which we call *objects*, and work with them.

In object-oriented programming we think in terms of sending and receiving messages between objects. When these instances receive the messages, they need to know what to do with them. The definition of what to do with a message is called a *method*. I mentioned that Ruby is like Smalltalk; Smalltalk epitomizes this model. Smalltalk allows the programmer to send a message `sqrt` to an object `2` (called a *receiver* in Smalltalk), which is a member of the `Integer` class. To handle the message, Smalltalk finds the appropriate method to compute the required answer for receivers belonging to the `Integer` class. It produces the answer `1.41421`, which is an instance of the `Float` class. Smalltalk is a 100% pure object-oriented language—absolutely everything in Smalltalk is an object, and every object can send and receive messages. Ruby is almost identical.

We can call methods in Ruby using "dot" syntax—e.g., `some_object.my_method`. In Ruby everything (pretty much *everything*) is an object. As such everything (literally everything) has methods, even `nil`. In Java or C, `NULL` holds a value to which no valid pointer will ever refer. That means that if you want to check if an object is nil, you compare it with NULL. Not so in Ruby! Let's check in irb:

```
>> nil.nil?
=> true
```

So if everything is an object, what is `nil` an instance of?

```
>> nil.class
>> nil.class
=> NilClass
```

OK, and what about `NilClass`?

```
>> NilClass.class
=> Class
```

Go ahead and try a few others—a number or a string (strings are encased in single or double quotes):

```
>> 37.class
=> Fixnum
>> "Thirty Seven".class
=> String
```

In our case, `hello` isn't anything—we haven't assigned it to anything, and it isn't a keyword in the language, so Ruby doesn't know what to do. Let's start, then, with something that Ruby does know about—numbers. Have a go at using Ruby to establish the following:

1. What is 42 multiplied by 412?
2. How many hours are there in a week?
3. If I have 7 students, and they wrote 17,891 lines of code, how many did they write each, on average?

   ```
   >> 42 * 412
   => 17304
   >> 24 * 7
   => 168
   >> 17891/7
   => 2555
   ```

   Hang on, that last number doesn't look right! What's going on here? Let's look at the classes of our numbers:

   ```
   >> 17891.class
   => Fixnum
   >> 7.class
   => Fixnum
   ```

   Ruby only does integer division with `Fixnum` objects:

   ```
   >> 2/3
   => 0
   ```

   Thankfully `Fixnum` objects have methods to convert them to `Floats`, which means we can do floating point maths:

   ```
   >> 2.to_f/3
   => 0.6666666666666666
   ```

Let's try some algebra:

```
>> hours_per_day = 24
=> 24
>> days_per_week = 7
=> 7
>> hours_per_week = hours_per_day * days_per_week
=> 168
```

This introduces assignment and variables. Assignment is an operation that binds a local variable (on the left) to an object (on the right). We can see that now `hours_per_day` is an instance of class `Fixnum`:

```
>> hours_per_week.class
=> Fixnum
```

A variable is a placeholder. And it varies, hence the name:

```
>> puts "Stephen likes " + drink
Stephen likes Rooibos
=> nil
>> drink = "Beetroot Juice"
=> "Beetroot Juice"
>> puts "Stephen likes " + drink
Stephen likes Beetroot Juice
=> nil
```

Identifiers

A variable is an example of a Ruby identifier. Wikipedia describes an identifier as follows:

> An identifier is a name that identifies (that is, labels the identity of) either a unique object or a unique class of objects, where the "object" or class may be an idea, physical [countable] object (or class thereof), or physical [noncountable] substance (or class thereof).

There are four main kinds of identifiers in Ruby:

1. Variables
2. Constants
3. Keywords
4. Method names

Variables

Looking first at variables, there are actually four types of variables that you'll encounter in Ruby:

1. Local variables
2. Instance variables

3. Class variables
4. Global variables

You'll mostly interact with the first two. Local variables begin with a lowercase letter, or an underscore. They may contain only letters, underscores, and/or digits:

```
>> valid_variable = 9
=> 9
>> bogus-variable - 8
NameError: undefined local variable or method `bogus' for main:Object
  from (irb):34
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
>> number9 = "ok"
=> "ok"
>> 9numbers = "not ok"
SyntaxError: (irb):36: syntax error, unexpected tIDENTIFIER, expecting $end
9numbers = "not ok"
        ^
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

Instance variables store information for an individual instance. They always begin with the "@" sign, and then follow the same rules as local variables.

Class variables are more rarely seen—they store information at the level of the class—i.e., further up the hierarchy than an instance of an object. They begin with "@@".

Global variables begin with a "$"—these don't follow the same rules as local variables. You won't need to use these very often. These can have cryptic looking names such as:

```
$!  # The exception object passed to #raise.
$@  # The stack backtrace generated by the last exception raised.
$&  # Depends on $~. The string matched by the last successful match.
$`  # Depends on $~. The string to the left of the last successful match.
$'  # Depends on $~. The string to the right of the last successful match.
$+  # Depends on $~. The highest group matched by the last successful match.
$1  # Depends on $~. The Nth group of the last successful match. May be > 1.
$~  # The MatchData instance of the last match. Thread and scope local. MAGIC
```

The preceding global variables are taken from the excellent Ruby quick reference by Ryan Davis (creator and maintainer of Minitest)—I recommend you bookmark it, or print it out.

On the subject of cryptic symbols, I mentioned that Ruby is akin to Perl. Ruby's creator, Yukihiro Matsumoto (Matz), describes the history of Ruby in an interview with Bruce Stewart:

> Back in 1993, I was talking with a colleague about scripting languages. I was pretty impressed by their power and their possibilities. I felt scripting was the way to go.
>
> As a long time object-oriented programming fan, it seemed to me that OO programming was very suitable for scripting, too. Then I looked around the Net. I found that Perl 5,

> which had not released yet, was going to implement OO features, but it was not really what I wanted. I gave up on Perl as an object-oriented scripting language.
>
> Then I came across Python. It was an interpretive, object-oriented language. But I didn't feel like it was a "scripting" language. In addition, it was a hybrid language of procedural programming and object-oriented programming.
>
> I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language.
>
> — *http://bit.ly/18FHd3p*

He adds:

> Ruby's class library is an object-oriented reorganization of Perl functionality—plus some Smalltalk and Lisp stuff. I used too much I guess. I shouldn't have inherited $_, $&, and the other, ugly style variables.

If you're familiar with Perl, I commend to you: comparing Ruby and Perl.

Constants

Constants are like variables, only their value is supposed to remain unchanged. In actual fact, this isn't enforced by Ruby—it just complains if you waver in your constancy:

```
>> MY_LOVE = "infinite"
=> "infinite"
>> MY_LOVE = "actually, rather unreliable"
(irb):38: warning: already initialized constant MY_LOVE
=> "actually, rather unreliable"
```

Constants begin with an uppercase letter—conventionally they may simply be capitalized (Washington), be in all caps (SHOUTING), camelcase (StephenNelsonSmith), or capitalized snakecase (BOA_CONSTRICTOR).

Keywords

Keywords are built-in terms hardcoded into the language. You can find them listed at *http://ruby-doc.org/docs/keywords/1.9/*. Examples include `end`, `false`, `unless`, `super`, `break`. Trying to use these as variables will result in errors:

```
>> super = "dooper"
SyntaxError: (irb):1: syntax error, unexpected '='
super = "dooper"
      ^
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
>> false = "hope"
SyntaxError: (irb):2: Can't assign to false
false = "hope"
      ^
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
>> unless = 63
SyntaxError: (irb):3: syntax error, unexpected '='
unless = 63
```

```
        ^
from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

Method names

Method names are the fourth and final kind of identifier. We've already seen one of these at play:

```
>> 7.to_f
=> 7.0
```

Method names adhere to the same naming constraints as local variables, with a few exceptions. They can end in "?", "!", or "=", and it's possible to define methods such as "[]" or "<=>". This might sound like a recipe for confusion, but it's very much by design. Methods are just part of the furniture; Ruby without methods would be like ice cream… without ice. Or cream.

More About Methods

We discussed the idea of objects and methods at the very start of this section. However, it bears repeating, as the object is the most fundamentally important concept in Ruby. When we send a message to an object, using the dot operator, we're calling some code that the object has access to. Strings have some nice methods to illustrate this:

```
>> "STOP SHOUTING".downcase
=> "stop shouting"
>> "speak louder".upcase
=> "SPEAK LOUDER"
```

The pattern is: OBJECT dot METHOD. To the left of the dot we have the *receiver* and to the right, the *method* we're calling, or the message we're sending.

Methods can take arguments:

```
>> "Tennis,Elbow,Foot".split
=> ["Tennis,Elbow,Foot"]
>> "Tennis,Elbow,Foot".split(',')
=> ["Tennis", "Elbow", "Foot"]
```

The first attempted to split the string on white space but didn't find any. The second split the string on the comma. The result of each method is an *Array*—more on arrays shortly.

I mentioned that methods may end in signs such as "?" or "!". Here's an example:

```
>> [1,2,3,4].include? 3
=> true
```

Here we're asking Ruby if the array [1,2,3,4] includes the number 3. The answer—the result of evaluating the expression—was *true*. A method with "!" on the end means "Do this, and make the change permanent!" We looked at downcase. Here it is again:

```
>> curse = "BOTHERATION!"
=> "BOTHERATION!"
>> curse.downcase
=> "botheration!"
>> curse
=> "BOTHERATION!"
>> curse.downcase!
=> "botheration!"
>> curse
=> "botheration!"
```

One final important idea connected with methods is the idea of `method_missing`. It is possible for an object to have the special method `method_missing`. In this case, if the object receives a message for which there is no corresponding method, rather than just throwing away the message and raising an error, Ruby can take the message and redirect it or use it in many powerful ways. Chef uses this functionality extensively to implement the language used to build infrastructure. This is an advanced topic, and I refer you to some of the classic texts—particularly *Metaprogramming Ruby* (The Pragmatic Programmers) if you wish to learn more.

We create methods using the `def` keyword:

```
>> def shout(something)
>> puts something.upcase
>> end
=> nil
>> shout('i really like ruby')
I REALLY LIKE RUBY
=> nil
```

We created a method and specified that it take an argument called "something". We then called the `upcase` method on the something. This worked fine, because the argument we passed was a string. Look what happens if we give bogus input:

```
>> shout(42)
NoMethodError: undefined method `upcase' for 42:Fixnum
  from (irb):7:in `shout'
  from (irb):10
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
>> shout('more', 'than', 'one', 'thing')
ArgumentError: wrong number of arguments (4 for 1)
  from (irb):6:in `shout'
  from (irb):11
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

You might be wondering on what object the shout method is being called! The answer is that it's being called on the `self` object. The `self` object provides access to the current object—the object that is receiving the current message. If the receiver is explicitly stated, it's obvious which object is the receiver. If the receiver is not specified, it is implicitly the self object. The self object, when we run in irb, is:

```
>> self
=> main
>> self.class
=> Object
```

What's all this about? We're basically looking at the top level of Ruby. If we type methods, we see there are some methods available at the top level:

```
>> methods
=> [:to_s, :public, :private, :include, :context, :conf, :irb_quit, :exit,↵
 :quit, :irb_print_working_workspace, :irb_cwws, :irb_pwws, :cwws, :pwws, ↵
:irb_current_working_binding, :irb_print_working_binding, :irb_cwb, :irb_pwb, ↵
:irb_chws, :irb_cws, :chws, :cws, :irb_change_binding, :irb_cb, :cb, ↵
:workspaces, :irb_bindings, :bindings, :irb_pushws, :pushws, ↵
:irb_push_binding, :irb_pushb, :pushb, :irb_popws, :popws, ↵
:irb_pop_binding, :irb_popb, :popb, :source, :jobs, :fg, :kill, :help, ↵
:irb_exit, :irb_context, :install_alias_method, ↵
:irb_current_working_workspace, :irb_change_workspace, :irb_workspaces, ↵
:irb_push_workspace, :irb_pop_workspace, :irb_load, :irb_require, :irb_source, ↵
:irb, :irb_jobs, :irb_fg, :irb_kill, :irb_help, :nil?, :===, :=~, :!~, ↵
:eql?, :hash, :<=>, :class, :singleton_class, :clone, :dup, ↵
:initialize_dup, :initialize_clone, :taint, :tainted?, :untaint, :untrust, ↵
:untrusted?, :trust, :freeze, :frozen?, :inspect, :methods, ↵
 :singleton_methods, :protected_methods, :private_methods, :public_methods, ↵
:instance_variables, :instance_variable_get, :instance_variable_set, ↵
:instance_variable_defined?, :instance_of?, :kind_of?, :is_a?, ↵
:tap, :send, :public_send, :respond_to?, :respond_to_missing?, ↵
:extend, :display, :method, :public_method, :define_singleton_method, ↵
:object_id, :to_enum, :enum_for, :==, :equal?, :!, :!=, ↵
:instance_eval, :instance_exec, :__send__, :__id__]
```

These methods must be being called on something. We know that something is `self`. What is `self`? When we inspect it, we see:

```
>> self.inspect
=> "main"
>> self.class
=> Object
```

So `self` is an instance of `Object` that evaluates to the string `main`. By default it has no instance variables:

```
>> instance_variables
=> []
```

But we can add them:

```
>> @loathing = true
=> true
>> instance_variables
=> [:@loathing]
```

So `self` is just an instance of `Object`. But what's going on when we define a method?

```
>> def say_hello
>>   puts "hello"
>> end
=> nil
```

We can see that the method is now available to `self`.

```
>> self.methods.grep /hello/
=> [:say_hello]
```

Here we called the `methods` method, which we know returns an array, and then called the `grep` method on the array. The `grep` method takes a pattern to match—we specified `/hello/`, which matched only one method, so Ruby returned it. What we've effectively done is defined a top-level method on `Object`. We can see this by inspecting the method:

```
>> method(:say_hello).owner
=> Object
```

Normally we would define methods in the context of a class, so let's look at classes.

Classes

Classes describe the characteristics and behavior of objects—simply put, a class is just a collection of properties with methods attached. We're familiar with the idea of biological classification—a mechanism of grouping and categorizing organisms into genus or species. For example the walnut tree and the pecan tree are both instances of the family *Juglandaceae*. In Ruby every object is an instance of precisely one class. We tend not to deal with classes as much as instances of classes. One particularly powerful feature of Ruby is its ability to give instances of a class some attributes or methods belonging to a different type of object. This idea—the *Mixin*—is seen fairly frequently, and we'll cover it later.

Let's see an example of creating a class:

```
>> class Pet
>> end
=> nil
```

The class doesn't do anything interesting yet, but we can create an instance of it:

```
>> rupert = Pet.new
=> #<Pet:0x00000001d97b68>
>> rupert.class
=> Pet
```

Let's extend our class a little. It might be nice to be able to know the name of the pet. This allows us to introduce the idea of the *constructor*. The constructor is the method that is called when a new instance of a class is initialized. We can use it to set up state for the object. Let's switch to using a text editor for this example:

```
$ emacs pet.rb
class Pet

  def initialize(name)
    @name = name
  end

  def name
    @name
  end

end

corins_pet = Pet.new("Rupert")
puts "The pet is called " + corins_pet.name

$ ruby pet.rb
The pet is called Rupert
```

So the constructor has the method *initialize*. We've said that it takes an argument, and we're setting an instance variable to hold the state of the pet's name. Later we have a method, *name*, which returns the value of the instance variable. Simple.

The trouble is, children are fickle. What they thought was a great name turns out to be a dreadful name a few days later. Unless we were going to be draconian, and insist that pet names be immutable, it might be nice to allow the child to rename the pet. Let's add an instance method that will change the name:

```
$ emacs pet.rb
class Pet

  def initialize(name)
    @name = name
  end

  def name=(name)
   @name=name
  end

  def name
    @name
  end

end

pet = Pet.new("Rupert")
puts "The pet is called " + pet.name
puts "ALL CHANGE!"
pet.name = "Harry"
puts "The pet is now called " + pet.name

$ ruby pet.rb
```

```
The pet is called Rupert
ALL CHANGE!
The pet is now called Harry
```

Here's another example of a method name with some odd-looking punctuation at the end. But this is how we implement a method that allows assignment. This class is looking a bit lengthy (and frankly, ugly) for such a featureless class. Thankfully Ruby provides some syntactic sugar, which provides the ability to get and set instance variables. Here's how it works:

```
class Pet

  attr_accessor :name

  def initialize(name)
    @name = name
  end

end

pet = Pet.new("Rupert")
puts "The pet is called " + pet.name
puts "ALL CHANGE!"
pet.name = "Harry"
puts "The pet is now called " + pet.name
```

What's actually going on here is that when the `Class` block is evaluated, the `attr_accessor` method is run, which generates the methods we need. Ruby is particularly good at this—metaprogramming—code that writes code. In more advanced programming, it's possible to overwrite the default `attr_accessor` method and make it do what we want—great is the power of Ruby. But why all the fuss? Why can't we just peek into the class and see the instance variable? Remember, Ruby operates by sending and receiving messages, and methods are the way classes deal with the messages. The same is so for instance variables. We can't access them without calling a method—it's a design feature of Ruby.

Right, that's enough of messages and classes for the time being. Let's move on to look at some data structures.

Arrays

Arrays are indexed collections of objects, which keep this in a specific order:

```
>> children = ["Melisande", "Atalanta", "Wilfrid", "Corin"]
=> ["Melisande", "Atalanta", "Wilfrid", "Corin"]
>> children[0]
=> "Melisande"
>> children[2]
=> "Wilfrid"
```

The index starts at zero, and we can request the nth item by calling the "[]" method. This is very important to grasp. We're sending messages again! We're sending the [] message to the children array, with the argument "2". The array knows how to handle the message and replies with the child at position 2 in the array. Arrays have convenient aliases:

```
>> children.first
=> "Melisande"
>> children.last
=> "Corin"
```

We can append to an array using the "<<" method. Suppose we adopted orphan Annie:

```
>> children << "Annie"
=> ["Melisande", "Atalanta", "Wilfrid", "Corin", "Annie"]
>> children.count
=> 5
```

Collections of objects can be iterated over. For example:

```
>> children.each { |child| puts "This child is #{child}" }
This child is Melisande
This child is Atalanta
This child is Wilfrid
This child is Corin
This child is Annie
=> ["Melisande", "Atalanta", "Wilfrid", "Corin", "Annie"]
```

This introduces two new pieces of Ruby syntax—the *block* and *string interpolation*. String interpolation is an alternative to the rather clumsy looking use of the "+" operator. Ruby evaluates the expression between #{} and prints the result.

```
>> dinner = "curry"
=> "curry"
>> puts "Stephen is going to eat #{dinner} for dinner"
Stephen is going to eat curry for dinner
=> nil
```

Of course the expression could be much more complex:

```
>> foods = ["chips", "curry", "soup", "cat sick"]
=> ["chips", "curry", "soup", "cat sick"]
>> 10.times { puts "Stephen will eat #{foods.sample} for dinner this evening." }
Stephen will eat chips for dinner this evening.
Stephen will eat soup for dinner this evening.
Stephen will eat soup for dinner this evening.
Stephen will eat cat sick for dinner this evening.
Stephen will eat chips for dinner this evening.
Stephen will eat curry for dinner this evening.
Stephen will eat chips for dinner this evening.
Stephen will eat soup for dinner this evening.
Stephen will eat soup for dinner this evening.
```

```
Stephen will eat curry for dinner this evening.
=> 10
```

Here we see another example of a block! The integer "10" has a method `times`, which takes a block as an argument.

Blocks allow a set of instructions to be grouped together and associated with a method. In essence, they're a block of code that can be passed as an argument to a method. They're a particular speciality of Ruby and are incredibly powerful. However, they're also a bit tricky to understand at first.

> For programmers new to Ruby, code blocks are generally the first sign that they have definitely departed Kansas. Part syntax, part method, and part object, the code block is one of the key features that gives the Ruby programming language its unique feel.
>
> — Russ Olser
> *Eloquent Ruby*

Blocks are created by appending them to the end of a method. Ruby takes the content of the block and passes it to the method. Depending on the length of the block, Ruby convention is either:

- If one line, then place in curly braces {} (unless the code has a side effect, such as writing to a file, in which case the `do … end` form applies)
- If more than one line, then replace curly braces with `do … end`

The method definition itself has code to handle the contents of the block. For now it's sufficient to understand that blocks are a kind of anonymous function—that is a function that we defined and call, without ever binding it to an identifier. Ruby uses them a great deal to implement iterators.

Although present in Smalltalk, I think that it's when looking at blocks that we see most evidence of Lisp in Ruby. Lisp provides the lambda expression as a mechanism for creating a nameless or anonymous function, and passing it to another function. Lisp also has the concept of a *closure*—that is an anonymous function that can refer to variables visible at the time it was defined. Referring again to a Matz interview, the creator of Ruby says:

> …we can create a closure out of a block. A closure is a nameless function the way it is done in Lisp. You can pass around a nameless function object, the closure, to another method to customize the behavior of the method. As another example, if you have a sort method to sort an array or list, you can pass a block to define how to compare the elements. This is not iteration. This is not a loop. But it is using blocks … the first reason [for this implementation] is to respect the history of Lisp. Lisp provided real closures, and I wanted to follow that.
>
> — Bill Venners
> *http://www.artima.com/intv/closuresP.html*

Ruby features a wide range of iterators for various purposes. One commonly used one is *map*. The map method takes a block, and produces a new array with the results of the block being applied, without changing the initial array:

```
>> children.map do |child|
?> if child == "Annie"
>> child + " the Orphan"
>> else
?> child + " Nelson-Smith"
>> end
>> end
=> ["Melisande Nelson-Smith", "Atalanta Nelson-Smith", "Wilfrid Nelson-Smith",
"Corin Nelson-Smith", "Annie the Orphan"]
>> children
=> ["Melisande", "Atalanta", "Wilfrid", "Corin", "Annie"]
```

The block arguments lie between the two pipe symbols. I find Why The Lucky Stiff's description particularly apt:

> The curly braces give the appearance of crab pincers that have snatched the code and are holding it together. When you see these two pincers, remember that the code inside has been pressed into a single unit…. I like to think of the pipe characters representing a tunnel. They give the appearance of a chute that the variables are sliding down. Variables are passed through this chute (or tunnel) into the block.
>
> — WTLSPGTR

Conditional logic

Ruby supports various control structures to manage the flow of data through a program. The most commonly used are those that fork based on decisions:

```
>> 10.times do
?> grub = foods.sample
>> if grub == "cat sick"
>>   puts "Stephen is not very hungry, for some reason."
>> else
?>   puts "Stephen will eat #{grub} for dinner this evening."
>> end
>> end
Stephen will eat chips for dinner this evening.
Stephen will eat curry for dinner this evening.
Stephen will eat soup for dinner this evening.
Stephen is not very hungry, for some reason.
Stephen is not very hungry, for some reason.
Stephen will eat chips for dinner this evening.
Stephen will eat chips for dinner this evening.
Stephen will eat soup for dinner this evening.
Stephen will eat curry for dinner this evening.
Stephen will eat soup for dinner this evening.
=> 10
```

In addition to `if` and `else`, we also have `elsif`:

```
>> def editor_troll(editor)
>> if editor == "emacs"
>>   puts "Best editor in the world!"
>> elsif editor =~ /vi/
>>   puts "Be gone with you, you bearded weirdo!"
>> else
?>   puts "*yawn* - sorry - were you talking to me?"
>> end
>> end
=> nil
>> editor_troll("emacs")
Best editor in the world!
=> nil
>> editor_troll("elvis")
Be gone with you, you bearded weirdo!
=> nil
>> editor_troll("nano")
*yawn* - sorry - were you talking to me?
=> nil
>> editor_troll("vim")
Be gone with you, you bearded weirdo!
=> nil
>> editor_troll("textmate")
*yawn* - sorry - were you talking to me?
=> nil
```

A handy option is the `unless` keyword:

```
>> def mellow_opinion(editor)
>> unless editor.length == 0
>> puts "Cool, dude.  I hear #{editor} is really nice."
>> end
>> end
=> nil
>> mellow_opinion("emacs")
Cool, dude.  I hear emacs is really nice.
=> nil
>> mellow_opinion("notepad")
Cool, dude.  I hear notepad is really nice.
=> nil
>> mellow_opinion("")
=> nil
```

The final control structure you'll come across is the case statement:

```
>> def seasonal_garment(season)
>> case season
>> when "winter"
>> puts "Wooly jumper and hat!"
>> when "spring"
>> puts "Shirt and jacket!"
>> when "summer"
```

```
>> puts "Shorts and t-shirt!"
>> when "autumn"
>> puts "Hmm... English? Raincoat!"
>> when "fall"
>> puts "Bit like spring, really."
>> end
>> end
>> seasonal_garment("winter")
Wooly jumper and hat!
=> nil
>> seasonal_garment("fall")
Bit like spring, really.
=> nil
>> seasonal_garment("autumn")
Hmm... English? Raincoat!
=> nil
```

Typically, the case statement is used if there are more than three options, as multiple `elsif` statements look a bit ugly, but it's really just a matter of style.

Hashes

A hash is another sort of collection in Ruby. Variously called a dictionary or associative array in other languages, its defining feature is that the index can be something other than a static value. Hashes are commonly used in Chef for key/value pairs:

```
>> wines = {}
=> {}
>> wines['red'] = ["Rioja", "Barolo", "Zinfandel"]
=> ["Rioja", "Barolo", "Zinfandel"]
>> wines['white'] = ["Chablis", "Riesling", "Sauvignon Blanc"]
=> ["Chablis", "Riesling", "Sauvignon Blanc"]
>> wines
=> {"red"=>["Rioja", "Barolo", "Zinfandel"], "white"=>["Chablis", "Riesling",
"Sauvignon Blanc"]}
```

The great thing about hashes is they can be deeply nested. We can add, for example:

```
>> wines['sparkling'] = {"Cheap" => ["Asti Spumante", "Cava"], "Moderate" =>
["Veuve Cliquot", "Bollinger NV"], "Expensive" => ["Krug", "Cristal"]}
=> {"Cheap"=>["Asti Spumante", "Cava"], "Moderate"=>["Veuve Cliquot", "Bollin-
ger NV"], "Expensive"=>["Krug", "Cristal"]}
>> wines['sparkling']["Cheap"]
=> ["Asti Spumante", "Cava"]
>> wines['sparkling']["Expensive"]
=> ["Krug", "Cristal"]
```

Again, of great significance for a Chef developer is to understand the message sending aspects. We're calling the [] method on the wines hash, which gives us another hash, on which we're calling the [] method, to get the expensive sparkling wines array. This is a

common pattern in Chef and leads to perhaps the most common error message you'll see:

```
>> wines['sparklin']['Cheap']
NoMethodError: undefined method `[]' for nil:NilClass
  from (irb):88
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

What happened? Well, I've drilled you so hard, I am sure you can say right away. Everything in Ruby is an object. Everything is an instance of a class. Even nothing at all:

```
>> nil.class
=> NilClass
>> nil.class.methods
=> [:allocate, :superclass, :freeze, :===, :==, :<=>, :<, :<=, :>, :>=, ↵
:to_s, :included_modules, :include?, :name, :ancestors, :instance_methods, ↵
:public_instance_methods, :protected_instance_methods, ↵
:private_instance_methods, :constants, :const_get, :const_set, ↵
:const_defined?, :const_missing, :class_variables, :remove_class_variable, ↵
:class_variable_get, :class_variable_set, :class_variable_defined?, ↵
:public_constant, :private_constant, :module_exec, :class_exec, :module_eval, ↵
:class_eval, :method_defined?, :public_method_defined?, ↵
:private_method_defined?, :protected_method_defined?, :public_class_method, ↵
:private_class_method, :autoload, :autoload?, :instance_method, ↵
:public_instance_method, :editor_troll, :mellow_opinion, :seasonal_garment, ↵
:nil?, :=~, :!~, :eql?, :hash, :class, :singleton_class, :clone, ↵
:dup, :initialize_dup, :initialize_clone, :taint, :tainted?, :untaint, ↵
:untrust, :untrusted?, :trust, :frozen?, :inspect, :methods, ↵
:singleton_methods, :protected_methods, :private_methods, :public_methods, ↵
:instance_variables, :instance_variable_get, :instance_variable_set, ↵
:instance_variable_defined?, :instance_of?, :kind_of?, :is_a?, :tap, ↵
:send, :public_send, :respond_to?, :respond_to_missing?, :extend, :display, ↵
:method, :public_method, :define_singleton_method, :object_id, :to_enum, ↵
:enum_for, :equal?, :!, :!=, :instance_eval, :instance_exec, :__send__, :__id__]
```

So an instance of `NilClass` has some methods, but it doesn't have a [] method! Where did we get nil from? Well, because of our typo, we requested the value of a non-existent key. Let's be more explicit:

```
>> wines['bogus']
=> nil
>> wines['bogus'].class
=> NilClass
>> wines['bogus']['reasonable']
NoMethodError: undefined method `[]' for nil:NilClass
  from (irb):93
  from /opt/rubies/1.9.3-p429/bin/irb:12:in `<main>'
```

Hashes have a number of convenient methods that you'll see used:

```
>> wines.keys
=> ["red", "white", "sparkling"]
>> wines.values
```

```
=> [["Rioja", "Barolo", "Zinfandel"], ["Chablis", "Riesling", "Sauvignon
Blanc"], {"Cheap"=>["Asti Spumante", "Cava"], "Moderate"=>["Veuve Cliquot",
"Bollinger NV"], "Expensive"=>["Krug", "Cristal"]}]
```

And, by definition, hashes are enumerable—they take a block and can be iterated over. When iterating over a block, we typically pass two arguments into the block, representing the key and value:

```
>> wines.each do |color, types|
?> if types.respond_to?(:keys)
>>   puts "For #{color} wine, you have more options: #{types.keys}"
>> end
>> end
For sparkling wine, you have more options: ["Cheap", "Moderate", "Expensive"]
```

Truthiness

Let's quickly talk about the abstract notion of truth. Truth, for Hegel, and many who followed in his footsteps is not merely semantic. It is a much richer metaphysical concept. Oh, sorry, wrong book. In Ruby, everything except nil and false is considered true. This includes 0. Let me say that again. In Ruby, *absolutely everything* is considered true. If in doubt, assume true. Obviously, false is false.

```
>> if 0
>>   puts "0 is true"
>> else
?>   puts "0 is false"
>> end
0 is true
=> nil
```

Note that nil is also not explicitly evaluated to false. A value can be coerced to true or false with "!!". !! is just ! (the boolean negation operator) written twice. It will negate the argument, then negate the negation:

```
>> !nil
=> true
>> !!nil
=> false
>> !0
=> false
>> !!0
=> true
```

The first ! will convert the argument to a boolean (e.g., true if it's nil or false, and false otherwise). The second will negate that again so that you get the boolean value of the argument, false for nil or false, true for just about everything else. This is mostly useful in the REPL, but occasionally you will see it in idiomatic usage.

Here's a little table of truthiness for reference:

Ruby Truthiness

Expression	Value
true	true
false	false
nil	nil
1 == 1	true
1 == true	false
!true	false
!false	true
!nil	true
1 !=2	true
1 !=1	false
!!true	true
!!false	false
!!nil	false
!!0	true
!!1	true

Operators

This leads nicely into a quick tour of Ruby operators. An *operator* is a token that represents an operation—for example, a comparison or a subtraction. It takes action on an *operand* and combines to form an *expression*. Ruby has lots of operators, covering boolean logic, assignment, and arithmetic. The best condensed summary is to be found in David Flanagan and Yukihiro Matsumoto's *The Ruby Programming Language* (O'Reilly), specifically section 4.6. We'll cover only those most pertinent to Chef.

First, arithmetic. Ruby provides the `\+`, `-`, `*`, `/`, and `%` operators to perform addition, subtraction, and integer division. The module operator can be used to calculate the remainder. This can also be used on `String` and `Array`. In the case of a string, `\+` will concatenate and `*` will repeat. In the case of an array, `+` will concatenate and `-` will subtract:

```
>> 6*6
=> 36
>> 24/12
=> 2
>> 37-8
=> 29
>> 17+17
=> 34
>> 20/3
=> 6
>> 20 % 3
=> 2
```

```
>> "Classical" + "Dutch"
=> "ClassicalDutch"
>> "Ruby" * 4
=> "RubyRubyRubyRuby"
>> ["Gatting", "Gower"] + ["Embury", "Edmunds"]
=> ["Gatting", "Gower", "Embury", "Edmunds"]
>> ["Lineker", "Platt", "Waddle"] - ["Waddle"]
=> ["Lineker", "Platt"]
```

Secondly, comparison. The `<`, `<=`, `>`,and `>=` operators make comparisons between objects that have a natural order—such as numbers or strings:

```
>> "llama" > "frog"
=> true
>> "apple" > "elephant"
=> false
>> "zelda" < "ganon"
=> false
>> "art" < "life"
=> true
>> 7 > 4
=> true
>> 6 >= 6
=> true
```

Particularly useful is the so-called "spaceship" operator `<=>`—another import from Perl. This makes a relative comparison between two operands. If the one on the left is less than the one on the right, it returns `+1`. If they are equal, it returns `nil` or `0`, and if the one on the right is greater than the one on the left, it returns `-1`. This is hugely useful in sorting!

```
>> "Individuals and interactions" <=> "Processes and tools"
=> -1
>> "Working software" <=> "Comprehensive documentation"
=> 1
>> "Customer collaboration" <=> "Contract negotiation"
=> 1
>> "Responding to change" <=> "Following a plan"
=> 1
>> "SunBlade" <=> "SunBlade"
=> 0
```

Thirdly, equality. `==` is the equality operator—it tests whether the left hand operand is equal to the right hand operand. Its opposite is `!=`. Take care not to use the assignment operator `=` in place of the equality operator!

```
>> melisande = 4
=> 4
>> adam = 4
=> 4
>> melisande == adam
=> true
```

```
>> helena = 42
=> 42
>> stephen = 37
=> 37
>> helena = stephen
=> 37
>> helena
=> 37
>> stephen
=> 37
```

Also very handy is the pattern-matching operator, `=~`. I think the characters in this resemble someone saying "kinda like" while rocking their hand in a circumspect manner, which gives a useful aide memoire. This operator is most commonly used when comparing strings:

```
>> "The Nimzo-Larsen attack" =~ /ck$/
=> 21
>> "The Nimzo-Larsen attack" =~ /xy$/
=> nil
```

The operator takes a regular expression as its rightmost operand. If the match is found, the index position where the match began is returned. Otherwise, `nil` is returned. Even a basic discussion of regular expressions is likely to be beyond the scope of this chapter, but with a little practice they're very easy to pick up. One of the best recent introductions to Ruby regular expressions can be found at Bluebox. This is the first of a three-part article and is highly recommended. The standard textbooks referenced in the bibliography also give ample coverage. For a more general discussion on the subject, and its fascination and beauty, see *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly).

Bundler

The final subject is Bundler, which should be covered before moving beyond the fundamentals of the Ruby language. Bundler is so fundamental to how Ruby development is carried out, and how tools are created and shared, that a thorough understanding of it is required.

Bundler exists to solve two problems:

- How to ensure that the appropriate dependencies are installed for a given problem without encountering unpleasant ordering issues or cyclical dependencies.
- How to share a software project between other developers, or other machines or environments, and be confident the application and its dependencies will behave in the same way.

The first problem can be illustrated by three imaginary Rubygems:

1. Oxygen
2. Whale
3. Human

Let's imagine that we have Oxygen 3.1.1 and Oxygen 1.8.5 installed on our system. Now, imagine that we want to install Whale 1.0.0. Whale depends on Oxygen, and the dependency is specified as ">= 1.5.0". When we run `gem install whale`, Rubygems will solve the dependency for Whale, and identify that Oxygen 3.1.1 will satisfy the dependency, and so will start using this version of Oxygen. Suppose we then decide to install Human 2.1.0. Human is a bit more picky about the Oxygen upon which it depends than Whale, and the dependency has been specified as "= 1.8.5". If we type `gem install whale`, Rubygems will identify that to solve the dependency, it will need to install version 1.8.5. The trouble is, 3.1.1 is already in use. This will result in an error. The problem is that Whale had a broader acceptance range for Oxygen than Human. Rubygems lacks a holistic view, and simply tries to solve dependencies on a case-by-case basis, resulting in nasty traps like this. The problem is magnified greatly the more Gems there are on the system. When you consider that these Gems themselves also have dependencies, it becomes apparent that trying to solve the dependencies one at a time isn't going to work.

The second problem may be illustrated by imagining two scenarios where a Ruby developer wishes to move software that was being developed on her workstation and share it with another team member, or perhaps run the code on a staging machine somewhere. If our developer is running the latest patch level of Ruby 1.9.3 on a Macbook, and her colleague is running Ruby 2.0.0 on Windows, ensuring she has the same versions of the Rubygems that are needed to run the application is going to be a bit of a challenge. This is a classic recipe for "it works on my machine." How can we somehow freeze known-good versions of Rubygems and ensure that when we move to the staging server, or share with another developer, that the versions are the same?

Bundler solves this problem in two ways. First, it provides an algorithm that solves all the dependencies for a given application at once. This is a more holistic and reliable approach. Second, it provides a kind of manifest file called a *Gemfile*, in which the top level dependencies for an application are specified, together with constraints, and optionally sources for the code. Between these two mechanisms, both problems are resolved.

The Gemfile has a straightforward syntax:

```
source 'https://rubygems.org'
gem 'some_dependency'
gem 'some_other_dependency'
```

Bundler will use the information in this file recursively to solve dependencies for the dependencies specified in the Gemfile, and to build a graph that satisfies the dependencies of each dependency in the Gemfile. The Gemfile allows for a high degree of sophistication in specifying source and version constraints. We can specify from where to retrieve the Gems. We can specify a particular version. We can specify that for a certain Gem, we should obtain the software directly from a Git repository, down to the commit, branch, or tag. We can also specify to use files on the local machine.

Once these dependencies have been resolved, developers can guarantee that only specific versions of the dependencies are used when the application is running on another user's computer or when deployed to a server.

Bundler is itself a Rubygem. We actually ensured it was installed in our developer role, but if we were doing things manually, we'd simply run:

```
$ gem install bundler
```

Once bundler has been installed, we create a Gemfile for the project and start to specify dependencies. Bundler provides a convenient method for generating a Gemfile:

```
$ bundler init
$ bundle init
Writing new Gemfile to /home/tdi/example/Gemfile
```

Once we've specified the dependencies in the Gemfile, we run:

```
$ bundle install
```

Bundler solves the dependencies and installs the Gems required. If those Gems are already on the system, it uses them, otherwise it fetches them. Once the dependencies have been solved and the Gems installed, Bundler creates a file called *Gemfile.lock*, which represents a snapshot of the dependency graph it built, and the versions it installed. By checking this and the Gemfile into version control, we can create a sandboxed environment for other users and be confident that they have exactly the same versions as we have on our system.

By way of exploring the functionality of Bundler, we're going to write a silly task to say "hello" three times in color. There's a very handy gem that is ideal for helping craft tasks, and ultimately provide them as command-line applications—it's called *Thor*. To print output in color, there is another handy Gem called *Colorize*. The process of setting up this project with Bundler looks like this:

```
$ mkdir /tmp/colorsay
$ cd /tmp/colorsay
$ bundle init
Writing new Gemfile to /private/tmp/colorsay/Gemfile
$ emacs Gemfile
$ cat Gemfile
source "https://rubygems.org"
```

```
gem "thor"
gem "colorize"

$ bundle install
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
Using colorize (0.5.8)
Using thor (0.18.1)
Using bundler (1.3.5)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.

$ cat Gemfile.lock
GEM
  remote: https://rubygems.org/
  specs:
    colorize (0.5.8)
    thor (0.18.1)

PLATFORMS
  ruby

DEPENDENCIES
  colorize
  thor
```

Now we write our simple task:

```
$ cat colorsay.thor
require 'colorize'

class ColorSay < Thor
  desc "hello", "Say hello in color"
  def hello
    puts "Hello".colorize( :red )
    puts "Hello".colorize( :green )
    puts "Hello".colorize( :yellow )
  end
end
```

This is our first case of *requiring* an external library or extension. In this case, we're bringing in `colorize` to provide strings with a `.colorize` method, which takes a *symbol* as its argument.

Symbols get a bit of bad press in the Ruby world. Well, at least to newcomers. What's that colon thing for? Why do we need it? Why bother? You'll see symbols cropping up in Chef recipes from time to time, so it's worth understanding what they are and what they're for.

Symbols are really just strings, dressed up. That's not quite fair, but they really don't deserve their exotic and sinister reputation. It helps if we think a little about the purpose of strings. The common use of strings is to carry some meaning or data. This might

change over time, and is generally designed to carry or convey information. For example, we might set the value of the variable `soup` to be the string value `"Leek and Potato"`. However, we also use strings as references or tags. We do this especially in Chef, when we're locating attributes on the Node. In this case, the string isn't going to change, and the extent to which it carries meaning is restricted to its role as a placeholder, or indicator —a pointer to somewhere where we'll find information that probably *will* change. Within the constructs of a programming language, these two roles exhibit significantly different profiles in terms of the resources, algorithms, and functions necessary to handle them. Strings as highly mutable, information carrying devices may need much manipulation and examination. Strings as placeholders just hold a place. That's it. In Ruby, the `String` class is optimized for the former use case, and the `Symbol` class is optimized for holding a place—for *symbolizing* something. Bringing this directly into the context of Chef, within a recipe, we may access the attributes of a Node in three ways:

- `node.attribute`
- `node['attribute\']`
- `node[:attribute]`

These can be used interchangeably (a side effect of the Node object actually being an instance of `Mash`). There's been strong debate in the community around which should be used as standard. As a Rubyist, I would advise you to use symbols. As a pragmatist, strings are perhaps a little less intimidating. For more information on the difference, and why it matters, see this post by Robert Sosinski.

The other characteristic of our little Thor task is that we see the class `ColorSay` is a *subclass* of class `Thor`. This is the meaning of the < symbol. This makes Thor the parent, the *superclass* of ColorSay. This is the idea of *inheritance* at work. Inheritance, like the word would suggest, is all about passing traits down from parent to child. Instances of the subclass pick up the methods of the superclass.

Now let's run our task:

```
$ thor list
color_say
---------
thor color_say:hello  # Say hello in color

$ thor color_say:hello
Hello
Hello
Hello
```

There's not really much more to it than that. For more comprehensive documentation and discussion, see the Bundler website.

The only consideration to bear in mind is that when using command-line tools provided by Rubygems, such as Thor, Rake, Cucumber, RSpec, Chef—in fact, pretty much all the tools we discuss in this chapter—there's the potential for confusion and bugs if you don't explicitly use the version installed in your bundle. Let me give you a trivial example. Suppose you had a need to use an earlier version of Thor because some functionality in your tasks relied upon some features that had been removed in the latest release. That's easy to achieve—we simply specify the version we need in the Gemfile:

```
$ cat Gemfile
source "https://rubygems.org"

gem "thor", "= 0.15.4"
gem "colorize"
$ bundle install
Fetching gem metadata from https://rubygems.org/.........
Resolving dependencies...
Using colorize (0.5.8)
Installing thor (0.15.4)
Using bundler (1.3.5)
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

If we run Thor now, we'll get version 0.15.4, right?

```
$ thor version
Thor 0.18.1
```

What? What's going on? The answer is, our shell gave us the path to the newest Thor. This could well have undesired effects. Bundler has a couple of ways around this. The first is `bundle exec`. This will run the version installed in your bundle:

```
$ bundle exec thor version
Thor 0.15.4
```

There are three problems with this approach. First, it's easy to forget and accidentally just run Thor. This can lead to annoying wastes of time as you try to figure out why your beeping code isn't working anymore. Second, it's a pain to have to type `bundle exec` every time. Third, there's a significant performance penalty:

```
$ time thor version
Thor 0.18.1

real    0m0.263s
user    0m0.185s
sys     0m0.054s

$ time bundle exec thor version
Thor 0.15.4

real    0m0.565s
user    0m0.453s
sys     0m0.082s
```

Running under `bundle exec` was nearly twice as slow. I can offer you three solutions to this problem. Pick one and work with it, for the sake of your sanity and your productivity. The first, and easiest, is just to create a shell alias:

```
$ alias b='bundle exec'
$ b thor version
Thor 0.15.4
```

Put this in your shell config and deal with the hassle of having to type two extra characters. Try not to forget, and deal with the performance.

The second is to use Bundler's own solution—*binstubs*. Bundle install supports the `--binstubs` option, which will create a local bin directory in your application root and a little wrapper script that calls the bundled command. Now you can just type `./bin/thor`. It's fewer keystrokes than `bundle exec`, and it's as fast as `thor` because bundler doesn't have to search for the binary. You can add the local bin directory to your shell path, and now you don't even need to remember `./bin`:

```
$ export PATH="./bin:$PATH"
$ thor --version
Thor 0.15.4
```

The disadvantage of this is that it's widely accepted as a security risk to have a local bin directory on your path, especially on, for example, a shared host. For more background, see the Unix FAQ. My preferred solution is one created by a friend and former colleague, Graham Ashton—*bundler-exec*. Using the power of shell aliases, it replaces a given list of commands with a shell function that checks for a Gemfile in your current directory, or one of its parents, and then prefixes the command with a `bundle exec` if needed. I think this is the best of all worlds. You'll never forget, it's no more typing, there's no security risk, and while there is a performance penalty, we're talking in the region of milliseconds. If you're a fan of `zsh`, take a look at Robby Russell's oh my zsh GitHub page, which provides this functionality as a plug-in. To install it on a bash shell, simply run the following:

```
$ curl -L https://github.com/Atalanta/bundler-exec/raw/master/bundler-exec.sh >
~/.bundler-exec.sh
$ echo "[ -f ~/.bundler-exec.sh ] && source ~/.bundler-exec.sh" >> ~/.bashrc
```

Now everything works beautifully:

```
$ cd /tmp/colorsay/
$ thor version
Thor 0.15.4
$ cd ..
$ thor version
$ thor version
Thor 0.18.1
```

Once you've implemented a mitigating strategy for the bundle exec annoyance, there aren't really any obvious disadvantages to using Bundler. It's the standard tool in the

Ruby community for managing and solving inter-gem dependencies, and for maintaining shareable sets of known-good Gems.

Bundler is an underpinning tool to pretty much everything we do in this book. Spend some time getting familiar with it, read the documentation, and take the time to set up your shell to take the pain out of the need to bundle exec.

CHAPTER 3
An Introduction to Chef

The best way to learn is to do. A lot of technical books, even ones aimed at beginners, take the form of a lengthy discursive preamble, followed by some abstract example for the reader to digest and understand. The trouble with this is it doesn't map well onto how we learn technical skills. Learning a technical skill is like teaching a child to ride a bicycle. You can't really teach someone the theory, and then show them a video of someone else cycling, and then expect them to just pick it up by themselves at some point in the future. A much better way is to go out there and then, with a bicycle, plonk them on, give them a push, and help them when they wobble.

Learning a technical skill or a programming language is very much about immersion. The learning process is reinforced by mistakes, by looking up documentation, by asking other more experienced people, and building up competence ourselves. So, to introduce the fundamental ideas of Chef, we'll build some real infrastructure, which we'll actually use later in the book. This chapter and the next are unashamedly influenced by the excellent series of books and courses by Zed Shaw found at Learn Code the Hard Way. An approach that focuses on diving in and using real examples, this has been proven to be an excellent method for building confidence and expertise in a technical subject.

The approach, as explained on the website:

> "…emphasizes precision, attention to detail, and persistence by requiring you to type each exercise (no copy-paste!) and make it run, as well as to read up on outside topics and to return to exercises and ideas that you don't understand, and understand them."

At the end of this chapter and the next, you'll understand the basics of Chef, have hands-on experience writing cookbooks and recipes, and use community resources to frame your infrastructure as code. Once we've covered these fundamentals, we'll go on to look at some of the tools we can use to start thinking about test-driven infrastructure development, and then look at a full example of using these tools in practice.

I'm making some broad assumptions about your ability and set-up. They are as follows:

- You can type instructions into a command prompt.
- You can edit text.
- You have a computer and have administrative power over it.
- Your computer was made some time in the last four or five years, and has about 2G of memory or more.
- You have a connection to the Internet.
- You are not behind a proxy server, or can easily disable it.[1]

Anything beyond this is a bonus. For example, if you have access to dedicated test hardware and several machines, that's excellent. However, that's not needed at all. If you don't have administrative control over your computer, or have a very old computer with not much memory, you probably want to fix that before we continue. In the first edition, I made the assumption that people would have access to a public-cloud infrastructure, or would be prepared to pay for their own (minimal) use. In this edition I've moved toward the view that people are more likely to have adequate hardware, and want to work with local virtual machines rather than machines hosted with a public cloud provider. Most Chef users these days make heavy use of local virtualization in addition to the cloud, and so I've decided to include setting up such a capability as a fundamental task. If this is truly impossible for you, simply skim the sections in Chapter 4 where we install VirtualBox, and once we get to installing Vagrant, set it up to use the Rackspace cloud or EC2.

The basic format is that I will set an objective, or set of objectives, that you will be asked to achieve. The objectives will be the equivalent of acceptance criteria; you'll know you're done when those objectives have been met. I'll then give you high-level directions on how to meet the objectives. They categorically are not instructions for you to follow, but rather an outline of the high-level steps you need to follow. My expectation is that you will be able to work out how to follow those directions by a combination of referring to other sections in the book, using your own knowledge and common sense, and using the main online resources for Chef:

- *http://docs.opscode.com*
- *http://wiki.opscode.com*
- #chef and #learnchef on *irc.freenode.net*

1. Proxy support is provided in Chef, and most auxiliary utilities, but it can be a bit fiddly. Improvements are ongoing, and there are frequent discussions on the Chef mailing lists, and IRC and GitHub issues that will be relevant. Basically, you will be able to get up and running if you do have a proxy server in your environment, but it would be better for me to direct you to the latest discussion and details rather than attempt to provide a guide here, which will almost certainly date, rapidly.

- The *chef-users* mailing list

I will follow the instructions with a worked example. I ask explicitly that, if you're reading this digitally, you don't simply copy and paste this into your own system—this contravenes the spirit of "the hard way." Additionally, your system may be subtly different from mine. I suggest you use my worked example as guidance for you as you achieve the objectives yourself. If you want to use the material in the worked example, I ask that you type it out yourself. Try to solve the exercises yourself, and only once you've tried, move on to look at the worked example.

Finally, we'll discuss the way we achieved the objectives, covering any interesting points that arose, and ensuring the way we achieved them is fully understood. Again, I would firmly ask that if you don't understand the discussion, don't carry on with the next set of objectives. Go back over the instructions and discussion, and if you're still stuck, seek help via the online resources previously mentioned. This is for your sake—master the fundamentals and build on them.

The infrastructure we're going to build over the next two chapters is a cookbook development and testing environment, including some useful tools, and setting up VirtualBox, Vagrant, and Test Kitchen. We're going to imagine we're in a position where we want to share this infrastructure with a few other users, and that we're going to host it on a physical machine somewhere on the public Internet, so we can collaborate with our friends and colleagues in different locations and timezones.

Exercise 1: Install Chef

Objectives

After completing this exercise, you will have done the following:

- Installed the latest version of the Chef client tools on your machine
- Identified how to find help on your machine
- Understood the purpose of each of the tools that ship with Chef

Directions

1. Search for the term "omnibus" on *http://docs.opscode.com* and read and understand how this helps us install Chef on our systems.
2. Install Chef on your computer using the *Omnibus* package for your platform.
3. Access the documentation installed on the computer for `chef-apply`, `chef-solo`, `chef-client`, `chef-shell`, and `knife`.

4. Search *http://docs.opscode.com* for each tool and read about what they do.

Worked Example

I set up two machines—one running Ubuntu 12.04, one running CentOS 6.4, both 64-bit. I then browsed to *http://docs.opscode.com/search.html*, and searched for the word "Omnibus". The top link provided an overview of how to install Chef on a workstation. It contained more information than I needed, but I identified that I should visit the *http://www.opscode.com/chef/install* page, and that for Linux and Unix machines, the installation process was broadly to run an install script, piped through a shell, with super-user privileges.

I browsed to the install page, filled out the form, and followed the instructions, which on each machine amounted to me running the following command:

```
curl -L https://www.opscode.com/chef/install.sh | sudo bash
```

On my CentOS machine, `sudo` was not configured, so I changed to the root user, and ran the command without `sudo`.

During the writing process, I also had 32-bit Ubuntu 13.04 machines. I mention this because the installation process was a bit trickier, as there weren't any 32-bit packages for 13.04. Instead, I selected 12.10, which did offer a 32-bit package, downloaded the package manually, and installed it with the following command:

```
$ sudo dpkg --install chef-11.4-4.2.ubuntu*.deb
```

I verified the installation on each machine by opening a terminal and running:

```
$ chef-client --version
```

To obtain help for each of the listed commands, I ran the command with the `--help` switch. I identified that `chef-apply` didn't require a configuration file, but the others did. `chef-solo` and `chef-shell` seemed simpler than `chef-client`, which had considerably more option flags. `Knife` seemed to have much more information available, including a `knife help` subcommand. I ran `knife help knife` and `knife help list`, and skimmed the pages.

I then browsed to *http://docs.opscode.com* and searched for each command. I found that I needed to quote the commands in order to get appropriate results. I read the documentation on `chef-solo` and `chef-client`. `chef-apply` only had a single line, and `chef-shell` yielded only a result telling me that this was once called "Shef". A search for "Shef" didn't bring results either, so I tried *http://wiki.opscode.com*, where I found a page about Shef, *http://wiki.opscode.com/display/chef/Shef*, which I skimmed.

Discussion

As you can see, installing Chef is a breeze! Opscode provides a fully supported package install for most platforms, including Windows and commercial Unix operating systems. These packages vendor everything needed to run Chef into an isolated location (typically */opt*)—this includes Ruby, OpenSSL, and other supporting tools and libraries.

When we ran the following code, it downloaded and executed a simple shell script that calculated the exact version of the native OS package required, downloaded the package, installed it, and added the vendored location of the Chef commands to your user's path:

```
curl -L https://www.opscode.com/chef/install.sh | bash
```

If you are worried about running arbitrary shell scripts on your machine, with root privileges you can always download the script, inspect it, and run it yourself. However, realistically, if you trust Opscode to develop an automation framework upon which you're going to base the running of your entire infrastructure, I think you can probably risk running the shell script that installs it.

Having installed Chef, we saw that we had five new commands available on our system:

- `chef-apply`
- `chef-shell`
- `chef-solo`
- `chef-client`
- `knife`

I asked you to make yourself familiar with the help available, both on your computer and on the Opscode documentation site. Naturally I don't expect much of this to make sense right now, but it's vital that you develop the impulse of using `--help`, `help`, and the Opscode documentation sites throughout the book. While I am "virtually" with you on this journey, in the real world, things won't work as expected, and knowing where to look for help from the start is a great foundation. I'll go on to explain what each of these tools is for, but first let's cover, at a high level, what Chef actually is.

Chef is an open source tool and framework that provides system administrators and developers with a foundation of APIs and libraries, which makes this kind of workflow possible.

Chef allows us to effectively write programs that generate configuration directly on the machines we need to manage. We then keep these programs in version control, and use them to gain control of the complex systems we need to manage.

Navigating the labyrinth of resources that we need to provide an application infrastructure becomes achievable because, through its libraries and APIs, Chef presents a

declarative interface to these resources. This allows us to define a policy and express the infrastructure requirements at a higher level—specifying *what* resources are required, but without specifying *how*.

Architecturally, machines managed by Chef *pull* configuration information rather than being passive receivers, which means that the infrastructure remains convergent—over time it will move into line with defined policy. A machine that was down for maintenance will pull its config as soon as it rejoins the network, rather than receiving a push, if the administrator remembers that that machine didn't get the last update.

Therefore, Chef furnishes us with the power to build tools to help us manage infrastructure at scale. At the heart of the Chef approach is the recognition that the person who knows best how to run their own infrastructure is the person who lives with it on a day-to-day basis. Encapsulated in that daily experience is a wealth of domain experience, which leads to a clear understanding of the business and technology problems that are most pressing. Chef aims to furnish such a person with the ability to solve these problems in a creative, scalable, repeatable, maintainable, and shareable manner.

Let's explore this a little further—Chef is a *framework*, a *tool*, and an *API*.

The Chef framework

As the discipline of software development has matured, frameworks have emerged with the aim of reducing development time by minimizing the overhead of having to implement or manage low-level details that support the development effort. This allows developers to concentrate on rapid delivery of software that meets customer requirements.

The common use of the word framework is to describe a supporting structure composed of parts fitted and joined together. The same is true in the software world. Frameworks tie together discrete components into a useful organic whole to provide structural support to the building of a software project. Frameworks also provide consistent and simple access to complex technologies by making wrappers available that simplify the interface between the programmer and underlying libraries.

Frameworks bring with them numerous benefits. In addition to increasing the speed of development, they can improve the quality of the software that is produced. Software frameworks provide conventions and design approaches that, if adhered to, encourage consistency across a team. Their modular design encourages code re-use and they frequently provide utilities to facilitate testing and debugging. By providing an extensive library of useful tools, frameworks reduce or eliminate the need for repetitive tasks and accord the developer a high degree of flexibility via abstraction.

Chef is a framework for infrastructure development—a supporting structure and package of associated benefits of direct relevance to framing one's infrastructure as code. Chef provides an extensive library of primitives for managing just about every conceivable resource that is used in the process of building up an infrastructure within

which we might deploy a software project. It also provides a powerful Ruby-based language for modeling infrastructure, and a consistent abstraction layer that allows developers and system administrators to design and build scalable environments without getting dragged into operating system and low-level implementation details. It also provides some design patterns and approaches for producing consistent, shareable, and reusable components.

The Chef tool

The use of tools is viewed by anthropologists as a hugely significant evolutionary milestone in the development of humans. Primitive tools enabled us to climb to the top of the food chain by allowing us to accomplish tasks that could not be carried out with our bodies alone. While tools have been available to system administrators and developers since the birth of computers, recent years have witnessed a further evolutionary leap, with the availability of network-enabled tools that can drive multiple services via a published API. These tools are frequently extensible, written in a modular fashion in powerful, flexible, high-level programming languages such as Python or Ruby.

Chef provides a number of such tools, built upon the framework:

`Ohai`
: A system profiling tool that gathers large quantities of data about the system, from network and user data to software and kernel versions. `Ohai` is extendable—plugins can be written (usually in Ruby) that will furnish data in addition to the defaults. The collected data is emitted in a machine-parseable and readable format (JSON), and is used to build up a database of facts about each system that is managed by Chef.

`chef-shell`
: An interactive debugging console that provides command-line access to the framework's libraries, the API, and the local system's data. This is an excellent tool for testing and exploring how Chef will behave under a variety of conditions. It allows the developer to run Chef within the Ruby interactive interpreter, IRB, and gives a `read-eval-print` loop ideal for debugging and exploring the data held on the Chef server.

`chef-solo`
: A fully featured standalone configuration management tool that allows access to a subset of Chef's features without using a Chef server; suitable for simple deployments.

`chef-client`
: An agent that runs on systems being managed by Chef, and the primary mechanism by which such systems communicate with the Chef server. `chef-client` uses the framework's library of primitives to configure resources on a system by talking to a central server API to retrieve data.

`chef-apply`
: A lightweight tool for configuring a machine to perform a function with a single command, needing no configuration or Chef server.

`knife`
: A multipurpose command-line tool that facilitates system automation, deployment, and integration. `Knife` provides command and control capabilities for managing physical, virtual, and cloud environments across a range of Linux, Unix, and Windows platforms. It is also the primary means by which the underlying model that makes up the Chef framework is managed. `Knife` is extensible and has a pluggable architecture, meaning that it is straightforward to create new functionality simply by writing custom Ruby scripts that include some of the Chef and `Knife` libraries. Used most frequently in conjunction with the client/server model, `Knife` assumes less significance if one's primary Chef implementation is Chef-solo.

The Chef API

In its most popular incarnation, Chef functions as a client/server web service.

The server component is written in Erlang and uses a JSON-oriented document datastore. The whole Chef framework is driven via a RESTful API, of which the `Knife` command-line tool is a client. We'll drill into this API shortly, but the critical thing to understand is that in most cases, day-to-day use of the Chef framework translates directly to interfacing with the Chef server via its RESTful API.

The server is open sourced, under the Apache 2.0 license, and is considered a reference implementation of the Chef Server API. The API is also implemented as a hosted software-as-a-service offering. The hosted version, called *Hosted Chef*, offers a fully resilient, highly available, multitenant environment. The platform is free to use for fewer than five nodes, so it's the ideal way to experiment with and gain experience with the framework, tool, and API. The pricing for the hosted platform is intended to be less than the cost of just the hardware resources to run a standalone server. For deployment in the enterprise, Opscode also provides a supported install on customer hardware, called *Private Chef*. This provides all the functionality of Hosted Chef, but behind the firewall with no multitenancy compromises.

The Chef server also provides an indexing service. All information gathered about the resources managed by Chef is indexed and searchable, meaning that Chef becomes a coordination point for dynamic, data-driven infrastructures. It is possible to issue queries for any combination of attributes—for example, VMware servers on VLAN 102 or MySQL slaves running CentOS 5. This opens up tremendously powerful capabilities—a simple example would be a dynamic load balancer configuration that automatically includes the web servers that match a given query to its pool of backend nodes.

The most important thing to understand is that the Chef server is fundamentally nothing more than a publishing platform with an API, an index, and a dependency solver. It does no heavy lifting. All interactions, without exception, are via the REST API.

The Chef community

Chef has a large and active community of users, with over 14,000 registered community members, over 700 individuals and companies as signed-up contributors, of which over 200 have committed code to the project. Opscode is a community-focused company. In the 55 releases that have been cut in the last four plus years, there have been 61 awards of *most valuable person* status (and another 24 for Ohai releases), for contributions to both the code and the community as a whole.

For a comparatively young product, uptake is very strong. Over a million known downloads of Chef have been recorded, with the real number being significantly larger. Adoption is on an exponential scale, from startups and small or medium enterprises (SMEs) through web operation poster-people such as Facebook, Etsy, 37signals, Rightscale, and Wikia to household names like Sony, Walt Disney, Turner, HP, and Adobe.

These companies all use Chef to automate the deployment of thousands of servers with a wide variety of applications and environments. Chef users can share their *"recipes"* for installing and configuring software with *"cookbooks"* on Opscode's community website. Cookbooks exist for a large number of packages, with over 800 cookbooks available on the Opscode community site alone.

The cookbooks aspect of the community site can be thought of as akin to RubyGems—although the source of most of the cookbooks can be obtained at any time from GitHub, stable releases are made in the form of versioned cookbooks. Both the Chef project itself and many of the cookbooks from the `opscode-cookbooks` Git organization are consistently in GitHub's list of the most popular watched repositories. In practice, these cookbooks are probably the most reusable IT artifacts I've encountered, partly due to the separation of data and behavior that the Chef framework encourages, and also due to the inherent power and flexibility accorded by the ability to configure and control complex systems with a mature 3GL programming language.

The community tends to gather around the mailing lists (one for users and one for developers), and the IRC channels on Freenode (again one for users, and one for developers). Chef users and developers tend to be highly experienced system administrators, developers, and architects, and are an outstanding source of advice and inspiration in general, as well as being friendly and approachable on the subject of Chef itself.

As the field of web operations has grown, the need to have a community of people who are solving hard problems, building tools, and sharing ideas has also expanded. Chef, as an expression of the concept of infrastructure as code is precisely that—a sharing of minds, ideas, awesome-sauce, and expertise, in reusable, testable, auditable, and versionable code.

Exercise 2: Install a User

Objectives

After completing this exercise, you will have achieved the following:

- Used Chef to create a user on your machine
- Understood the principles behind Chef's recipe DSL
- Understood how to use `chef-apply`, and what its limitations are

Directions

1. Create a file called *tdi.rb* using your text editor.
2. Read the documentation for the "user" resource at *http://docs.opscode.com/chef/resources.html#user*.
3. Declare a resource in *tdi.rb* to create a user called "tdi".
4. Create the user by running `chef-apply`.
5. Verify that the user has been created.
6. Add another resource of type `dotfile` to drop off a configuration file called *.tdi* with `content` parameter of "bogus".
7. Run `chef-apply` again.
8. Observe the failure characteristics.
9. Replace the resource type "dotfile" with "file" and run `chef-apply` again.
10. Replace the "file" resource with a "template" resource, and change the "content" parameter to "source".
11. Run `chef-apply` once more.

Worked Example

In my *tdi.rb* file I wrote the following:

```
user 'tdi' do
  action :create
  comment "Test Driven Infrastructure"
  home "/home/tdi"
  supports :manage_home => true
end
```

I saved the file and ran `chef-apply`. On my CentOS machine I was still using the `root` user, so I didn't need to use `sudo`. On my Ubuntu machine I was logged in as my `sns` user, so I used `sudo`:

```
$ sudo chef-apply tdi.rb
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * user[tdi] action create
    - create user user[tdi]
```

I then verified the user existed:

```
sns@ubuntu:~$  getent passwd | grep tdi
tdi:x:1001:1001:Test Driven Infrastructure:/home/tdi:/bin/sh
[root@centos ~]#  getent passwd | grep tdi
tdi:x:500:500:Test Driven Infrastructure:/home/tdi:/bin/bash
```

I noticed that on the Ubuntu machine, the user didn't set the default shell to Bash. Although this could be easily done by updating the recipe, I decided to fix it the quick and dirty way, with:

```
$ sudo chsh -s /bin/bash tdi
```

I added a bogus resource to *tdi.rb* as follows:

```
dotfile '/home/tdi/.tdi' do
  action :create
  content 'bogus'
end
```

When I ran Chef, I saw:

```
sns@ubuntu:~$ sudo chef-apply tdi.rb
[2013-06-26T20:09:10+01:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-06-26T20:09:10+01:00] FATAL: NameError: Cannot find a resource for dot-
file on ubuntu version 12.04
[root@centos ~]#  chef-apply tdi.rb
[2013-06-26T19:28:11+01:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-06-26T19:28:11+01:00] FATAL: NameError: Cannot find a resource for dot-
file on centos version 6.4
```

Changing the resource to a "file" yielded the following:

```
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * user[tdi] action create (up to date)
  * file[/home/tdi/.tdi] action create
    - create new file /home/tdi/.tdi with content checksum 81f7e3
               ---  /tmp/chef-tempfile20130528-13007-1cgpj8         2013-05-28
11:20:11.932272825 +0100
         +++ /tmp/chef-diff20130528-13007-ipe5ju 2013-05-28 11:20:11.932272825
+0100
        @@ -0,0 +1 @@
        +bogus
```

I altered my file resource as follows:

```
template '/home/tdi/.tdi' do
  action :create
  source 'tdi-bashfile'
end
```

When I ran `chef-apply`, this time I saw:

```
chef-apply tdi.rb
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * user[tdi] action create (up to date)
  * template[/home/tdi/.tdi] action create
================================================================================
Error executing action `create` on resource 'template[/home/tdi/.tdi]'
================================================================================

NoMethodError
-------------
undefined method `preferred_filename_on_disk_location' for nil:NilClass

Resource Declaration:
---------------------
# In tdi.rb

  6: template '/home/tdi/.tdi' do
  7:   action :create
  8:   source 'bogus'
  9: end

Compiled Resource:
------------------
# Declared in tdi.rb:6:in `run_chef_recipe'

template("/home/tdi/.tdi") do
  provider Chef::Provider::Template
  action [:create]
  retries 0
  retry_delay 2
  path "/home/tdi/.tdi"
  backup 5
  source "bogus"
  cookbook_name "(chef-apply cookbook)"
  recipe_name "(chef-apply recipe)"
end

[2013-05-28T11:24:48+01:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-05-28T11:24:48+01:00] FATAL: NoMethodError: template[/home/tdi/.tdi]
((chef-apply cookbook)::(chef-apply recipe) line 6) had an error: NoMethodEr-
ror: undefined method `preferred_filename_on_disk_location' for nil:NilClass
```

Discussion

To use Chef to manage infrastructure is to insert a very powerful and flexible abstraction layer between the engineer and the system. Instead of the developer logging onto three different types of machines and typing commands into a terminal, or navigating a sequence of menus, he types in a text editor, commits to a version control system, and effectively deploys what was written to a series of machines. We are practicing the discipline of infrastructure as code.

In practical terms, the way we do this is by thinking about the abstract system components that we need to configure our systems as we want. For example, if I want to ensure the clock on my Linux computer is regularly synchronized with an NTP server, I might need to install the package that provides NTP client functionality, alter the configuration file according to my requirements, and ensure the NTP daemon is running, or that the client is run as a scheduled task. In Chef we call these low-level components that we can reason about and discuss *"resources."*

Resources are the very essence of Chef—the atoms, if you like. When we talk about a complicated or even a simple infrastructure, that conversation takes place at a level of resources. For example, we might discuss a web server—what are the components of a web server? Well, we need to install Apache, we need to specify its configuration and perhaps some virtual hosts, and we need to ensure the Apache service is running. Immediately, we've identified some resources—a package, a file, and a service.

Managing infrastructure using Chef is a case of specifying what resources are needed and how they interact with one another. We call this *setting the policy*.

If resources are the fundamental configuration objects, *nodes* are the fundamental things that are configured. It's possible to get a bit confused when the word "node" is used. For most engineers, a "node" is synonymous with a physical (or virtual) machine on a network. To an extent this meaning is carried forward in Chef, as I just did: nodes are the things we're configuring. However, most of the time, in Chef, the term "node" refers to the Chef node, which is ultimately a Ruby object representing the machine we're configuring. This object behaves like a Hash: it has keys and values, getter and setter methods, and can be viewed, queried, and interacted with as JSON. With that caveat, a concise definition of what Chef does is this:

Chef manages resources on the node so they comply with policy.

It's important to understand that when we talk about resources in Chef, we're not talking about the *actual* resource that ends up on the box. Resources in Chef are an abstraction layer. If we were to write Chef code to install the korn shell package on a CentOS box, that would mean:

```
$  yum install ksh
```

This would be represented in Chef by:

```
package "ksh"
```

A resource in Chef can take action. Here again, note the difference—the user resource in Chef can *create* a user on a machine. It isn't *the* user on the machine. Resources take action through *providers*. A provider is some library code that understands two things: first, how to determine the state of a resource; and second, how to translate the abstract requirement (install Apache) into the concrete action (`run yum install httpd`). Additionally it understands that, depending upon the underlying operating system or distribution, the utilities or commands used to install a package will be different—for example, on a Debian system, the provider would use `dpkg` or `apt` rather than `yum` or `rpm`. Determining the state of the resource is important in configuration management; we only want to take action if it is necessary. If the user has already been created or the package has already been installed, we don't need to take action. This is the principle of *idempotence*. (See *http://bit.ly/15M3qwJ* for more on idempotency and its meaning in this context.) A provider knows how to check whether the user has already been created, and won't take action if it has. The mathematicians amongst you may complain about this appropriation of the term. Within the configuration management world, we understand that idempotence literally means that an operation will produce the same results if executed once or multiple times (i.e., multiple application of the same operation has no side effect). We take this principle, specifically the idea that all functions should be idempotent with the same data, and use this as a metaphor. Not taking action unless it's necessary is an implementation detail designed to ensure idempotence.

Resources have data. If we were to write code to create a user, in addition to a default action, which all resources have (in the case of a package it's to install the package; in the case of a user, it's to create the user), we'd also probably want to specify some additional configuration for a user. For example, we might want to set a shell, or a comment:

```
user "melisande" do
  comment "International Master Criminal"
  shell "/bin/ksh"
  home "/export/home/melisande"
  supports :manage_home => true
  action :create
end
```

Resources, then, have a *name* (in this case, `melisande`), a *type* (in this case, a `user`), data in the form of *parameter attributes* (in this case, `comment`, `shell`, `home` directory, and `supports`), and an *action* (in this case, we're going to `create` the user).

In our exercise we used the `user` resource to create a user called "tdi". I asked you to review the documentation on the `user` resource on the docs site. Again, there is far more information there than you need now, but as you go on to build more complex infrastructure, you will refer to the resource documentation time and again. The most confusing aspect of the documentation (at the time of writing) is the idea of "supported

features." The resource has the attribute `supports`, with key/value pairs representing whether a given feature is supported by the underlying provider (for example, `useradd` on Solaris versus Linux). One such feature is `manage_home`. This flag is used to make explicit whether a home directory will be created at the same time as the user is created. The `supports` syntax is a bit cumbersome, so there's a handy convenience method `manage_home` that can be set to true or false. It has the same effect, but looks a bit cleaner. I'll draw your attention to one particular wart that could catch you if you're a RHEL/CentOS user. The default behavior of the user resource is *not* to create the home directory. This is pretty much standard across Linux and Unix. However, an implementation detail of RHEL-family systems is that '`useradd` *does* create a */home/user* directory by default. The result is that you could get away with never declaring `home` or `manage_home` in your user resources on RHEL systems, but then get tripped up if you expected your code to work on other Linux systems. For this purpose, I recommend explicitly specifying both the home directory and `manage_home: true` in your user resource declarations.

You'll notice that we called the file we wrote *tdi.rb*. It's actually Ruby code (and if this is not familiar to you, don't worry—you'll learn all the Ruby you need to know in Chapter 2). We can prove this by adding some Ruby into the file, and running it again:

```
$ cat tdi.rb
10.times { puts "This is actually just Ruby" }

user 'tdi' do
  action :create
  comment "Test Driven Infrastructure"
  home "/home/tdi"
  supports :manage_home => true
end

template '/home/tdi/.tdi' do
  action :create
  source 'bogus'
end

# chef-apply tdi.rb
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
This is actually just Ruby
Recipe: (chef-apply cookbook)::(chef-apply recipe)
  * user[tdi] action create (up to date)
```

```
  * template[/home/tdi/.tdi] action create
================================================================================
Error executing action `create` on resource 'template[/home/tdi/.tdi]'
================================================================================

...
...
```

In Chef terms, the file that we wrote is called a recipe. It's a set of instructions, a set of resources that we need to configure the machine in the way we want it. When we say that an infrastructure developer is writing Chef code, we are typically talking about using Chef's "recipe" DSL. Let's quickly explore the idea of a DSL.

DSL, or domain specific language, in practice means a way of encapsulating shared knowledge relating to a specific task or series of tasks, in a small, clearly defined set of words, with a small and clearly defined set of rules.

The example I like to give when I'm training people to use Chef is the game of Blackjack. I used to take a ferry from the south of England to the north of France or Spain, every so often. Especially on the longer journeys, I used to sit in the ship's casino and play cards. A popular game was blackjack. The passengers were frequently French, Spanish, English, Dutch, or German. However, everyone was able to play blackjack because there was an established DSL in place. Everyone knew that "card" means "give me a card." Everyone knew that "stick" means "I don't want another card." Everyone knew that "split" means "separate my two cards into two piles of one, and deal one card to each pile." There were rules around the usage of the terms; you can't use the language when it's someone else's turn. You can't split if the cards aren't of the same value. This is the same of all DSLs—they have a few meaningful keywords, and a few grammatical and syntactical rules.

Whenever we speak about a DSL, it naturally follows that we explain the purpose of the DSL. Thus if we were to say, "Gherkin is a DSL," that doesn't really tell us much. If, however, we were to say, "Gherkin is a DSL for translating stakeholder requirements to executable Ruby acceptance tests," it makes much more sense. Similarly, as the old joke goes, Java is a DSL for producing stack traces.[2] It turns out that Chef has a DSL for several things: recipes, roles, environments, and the creation of custom resources and providers. We'll cover most of the Chef DSLs in this book, but at a high level Chef provides DSLs for programmatically declaring which resources should be configured on a machine, for grouping related resources together and applying them to machines of the same sort, for isolating systems of a certain class, ensuring they remain in a defined state, and several other powerful concepts. This allows us to bring into being services using code.

You'll notice that when we tried to use a bogus resource in our recipe, Chef complained that it couldn't find a resource of the type we declared:

2. The original line is from Scott Bellware.

```
[2013-05-28T11:11:59+01:00] FATAL: NameError: Cannot find a resource for dot-
file on centos version 6.4
```

Why then did we have a problem when we tried to use a template resource? Here we hit upon the limitations of `chef-apply`. `chef-apply` is really only useful for a quick job, or (as we've seen) for instructional purposes. It doesn't have any context outside the single Ruby file it is passed. Templates, by their very definition, have a source template that is populated with data. We don't have any way of providing a source template to Chef when using `chef-apply`, and so we get an error. In our next exercise, we'll graduate to using `chef-solo`, and explore some more resource types.

Exercise 3: Install an IRC Client

Objectives

After completing this exercise, you will:

- Be familiar with the `package`, `directory`, and `cookbook_file` resources
- Understand `chef-solo`, and how it is configured
- Understand the ideas of a recipe, a cookbook, and a run list

Directions

1. Ensure you don't still have the "This is actually just Ruby" code in your recipe.
2. Run `chef-solo` without any configuration options, and read the output.
3. Look at the `knife help` output for the cookbook subcommand, paying particular attention to `cookbook path`, and then create a cookbook called *irc*.
4. Verify that a skeleton cookbook has been created.
5. Read the package resource documentation at *http://docs.opscode.com/resource_package.html*
6. Read the cookbook_file resource documentation at *http://docs.opscode.com/resource_cookbook_file.html*
7. Read the directory resource documentation at *http://docs.opscode.com/resource_directory.html*
8. Open the *default.rb* recipe in your text editor, and copy the user resource into the file.
9. Add a resource to install the *irssi* package.
10. Add a resource to create a *.irssi* directory in the "tdi" user's home directory, owned by the "tdi" user.

11. Add a resource to drop off an irssi config file at *~/.irssi/config*, also owned by the "tdi" user. Use the irssi config at *https://gist.github.com/Atalanta/5676662*.
12. Create a *solo.rb* config file, and specify your cookbook path.
13. Search the docs site for "run list" to understand the high level concept.
14. Run `chef-solo`, telling it to converge the node with the default recipe from the irc cookbook.
15. Become the "tdi" user, and launch your IRC client, by typing `irssi` at the command prompt, and say "ohai!" in the `##tdi` chat room!

Worked Example

I ran `chef-solo` on one of the machines, and read the output, noting that it was unable to find a configuration file, but would take its configuration from the command line, and that it failed to compile any cookbooks, having looked in two locations. It suggested I make sure my *cookbook_path* was set correctly:

```
$ sudo chef-solo
[sudo] password for stephen:
[2013-05-28T18:05:01+01:00] WARN: ****************************************
[2013-05-28T18:05:01+01:00] WARN: Did not find config file: /etc/chef/solo.rb,
using command line options.
[2013-05-28T18:05:01+01:00] WARN: ****************************************
Starting Chef Client, version 11.4.4
Compiling Cookbooks...
[2013-05-28T18:05:03+01:00] FATAL: No cookbook found in ["/var/chef/cookbooks",
"/var/chef/site-cookbooks"], make sure cookbook_path is set correctly.
[2013-05-28T18:05:03+01:00] FATAL: No cookbook found in ["/var/chef/cookbooks",
"/var/chef/site-cookbooks"], make sure cookbook_path is set correctly.
[2013-05-28T18:05:03+01:00] ERROR: Running exception handlers
[2013-05-28T18:05:03+01:00] ERROR: Exception handlers complete
Chef Client failed. 0 resources updated
[2013-05-28T18:05:03+01:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-05-28T18:05:03+01:00] FATAL: Chef::Exceptions::CookbookNotFound: No cook-
book found in ["/var/chef/cookbooks", "/var/chef/site-cookbooks"], make sure
cookbook_path is set correctly.
```

I looked at `knife help cookbook` and was given a choice of two pages to read:

```
$ knife help cookbook
WARNING: No knife configuration file found
Multiple help topics match your query. Pick one:
1. knife-cookbook-site
2. knife-cookbook
```

I selected the second, and read the documentation, discovering that I could set the cookbook path with the `-o --cookbook-path` switch. I then created an irc cookbook as follows:

```
$ knife cookbook create irc -o .
WARNING: No knife configuration file found
** Creating cookbook irc
** Creating README for cookbook: irc
** Creating CHANGELOG for cookbook: irc
** Creating metadata for cookbook: irc
```

I verified the skeleton with the following:

```
$ ls -1F irc/
attributes/
CHANGELOG.md
definitions/
files/
libraries/
metadata.rb
providers/
README.md
recipes/
resources/
templates/
```

I read the documentation page for the `package` resource on the docs site, and concluded that I didn't need to specify any particular attributes, and that Chef would work out the right thing to do on my platform.

I edited the recipe at *irc/recipes/default.rb* and added the `user` resource (with the neater `manage_home` syntax) and the following to ensure the user was created, and to install the irssi package:

```
user 'tdi' do
  action :create
  comment "Test Driven Infrastructure"
  home "/home/tdi"
  manage_home true
end

package 'irssi' do
  action :install
end
```

I read the documentation for the `directory` resource, and added the following:

```
directory '/home/tdi/.irssi' do
  owner 'tdi'
  group 'tdi'
end
```

I read the documentation for the `cookbook_file` resource, and added the following resource:

```
cookbook_file '/home/tdi/.irssi/config' do
  source 'irssi-config'
  owner 'tdi'
  group 'tdi'
end
```

I then created a file at *files/default/irssi-config* with the following content:

```
servers = (
  {
    address = "irc.freenode.net";
    chatnet = "Freenode";
    port = "6667";
    autoconnect = "Yes";
  }
);

chatnets = { Freenode = { type = "IRC"; }; };

settings = {
  core = {
    real_name = "Sir Edward Elgar";
    nick = "elgar";
    user_name = "elgar";
  };
  "fe-text" = { actlist_sort = "refnum"; };
};

channels = (
  { name = "#learnchef"; chatnet = "Freenode"; autojoin = "Yes"; },
  { name = "#chef"; chatnet = "Freenode"; autojoin = "Yes"; },
  { name = "##tdi"; chatnet = "Freenode"; autojoin = "Yes"; }
);
```

I searched the docs page for "run list" and read the first two hits (*http://docs.opscode.com/essentials_node_object_run_lists.html* and *http://docs.opscode.com/essentials_cookbook_recipes_run_lists.html*), which helped me to understand the idea of a run list. Having run `chef-solo --help`, I determined that I could pass the configuration file as an option using the `-c, --config` option, and that I could specify a run list using `-o, --override-runlist`. Armed with this knowledge I created a *solo.rb* config file within a *.chef* directory, and then ran Chef (again as root on Centos, and with sudo, and as sns on Ubuntu):

```
$ mkdir ~/.chef
$ cat ~/.chef/solo.rb
cookbook_path ENV['HOME']
$ sudo chef-solo --config ~/.chef/solo.rb --override-runlist 'recipe[irc]'
Starting Chef Client, version 11.4.4
[2013-05-30T10:46:23+01:00] WARN: Run List override has been provided.
```

```
[2013-05-30T10:46:23+01:00] WARN: Original Run List: []
[2013-05-30T10:46:23+01:00] WARN: Overridden Run List: [recipe[irc]]
Compiling Cookbooks...
Converging 4 resources
Recipe: irc::default
  * user[tdi] action create (up to date)
  * package[irssi] action install
    - install version 0.8.15-5.el6 of package irssi

  * directory[/home/tdi/.irssi] action create
    - create new directory /home/tdi/.irssi
    - change owner from '' to 'tdi'
    - change group from '' to 'tdi'

  * cookbook_file[/home/tdi/.irssi/config] action create
    - create a new cookbook_file /home/tdi/.irssi/config
              ---  /tmp/chef-tempfile20130530-15376-1m5nhvp        2013-05-30
10:46:28.698288821 +0100
             +++  /root/irc/files/default/irssi-config              2013-05-30
10:40:50.313288775 +0100
        @@ -0,0 +1,25 @@
        +servers = (
        +  {
        +    address = "irc.freenode.net";
        +    chatnet = "Freenode";
        +    port = "6667";
        +    autoconnect = "Yes";
        +  }
        +);
        +
        +chatnets = { Freenode = { type = "IRC"; }; };
        +
        +settings = {
        +  core = {
        +    real_name = "Sir Edward Elgar";
        +    nick = "elgar";
        +    user_name = "elgar";
        +  };
        +  "fe-text" = { actlist_sort = "refnum"; };
        +};
        +
        +channels = (
        +  { name = "#learnchef"; chatnet = "Freenode"; autojoin = "Yes"; },
        +  { name = "#chef"; chatnet = "Freenode"; autojoin = "Yes"; },
        +  { name = "##tdi"; chatnet = "Freenode"; autojoin = "Yes"; }
        +);

Chef Client finished, 3 resources updated
```

I su'd to `tdi`, ran `irssi`, and found the `##tdi` room, and said "Ohai".

Discussion

When we ran `chef-solo` we learned three important things:

```
# chef-solo
[2013-03-12T16:41:53+00:00] WARN: *****************************************
[2013-03-12T16:41:53+00:00] WARN: Did not find config file: /etc/chef/solo.rb,
using command line options.
[2013-03-12T16:41:53+00:00] WARN: *****************************************
Starting Chef Client, version 11.4.0
Compiling Cookbooks...
[2013-03-12T16:41:54+00:00] FATAL: No cookbook found in ["/var/chef/cookbooks",
"/var/chef/site-cookbooks"], make sure cookbook_path is set correctly.
[2013-03-12T16:41:54+00:00] FATAL: No cookbook found in ["/var/chef/cookbooks",
"/var/chef/site-cookbooks"], make sure cookbook_path is set correctly.
[2013-03-12T16:41:54+00:00] ERROR: Running exception handlers
[2013-03-12T16:41:54+00:00] ERROR: Exception handlers complete
Chef Client failed. 0 resources updated
[2013-03-12T16:41:54+00:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-03-12T16:41:54+00:00] FATAL: Chef::Exceptions::CookbookNotFound: No cook-
book found in ["/var/chef/cookbooks", "/var/chef/site-cookbooks"], make sure
cookbook_path is set correctly.
```

1. Chef expects a configuration file, but will accept options on the command line, in lieu of a configuration file.
2. It expects the configuration file to reside in */etc/chef*.
3. It looked for cookbooks in the */var/chef/cookbooks* directories and the */var/chef/site-cookbooks* directory, but failed to find any.

What are these cookbooks of which Chef speaks? My Chambers English Dictionary (highly recommended for budding cruciverbalists) defines a cookbook as:

> A book of recipes for cooking dishes.

Well obviously we're not cooking dishes, but the rest of the metaphor makes sense—cookbooks contain recipes. So what's a recipe? Turning again to my trusty dictionary, I'm told that one definition of a recipe is:

> A method laid down for achieving a desired end.

This is perfect! That's exactly what a recipe is. It's a method of achieving a desired outcome—the desired state of our infrastructure. That method might be fairly complex because realistically speaking, our infrastructures are much more complicated than can be expressed in a single or even a collection of independent resources. As infrastructure developers, the bulk of the code we write will be in the form of these *recipes*.

Recipes in Chef are written in a domain-specific language (DSL), which allows us to declare the state in which a node should be. Remember, a domain-specific language is a computer language designed to address a very specific problem space. It has grammar

and syntax in the same way as any other language but is generally much simpler than a general purpose programming language. Ruby is a programming language particularly suited to the creation of DSLs. It's very powerful, flexible, and expressive. As we already mentioned, DSLs are used in a number of places throughout the framework. However, a particularly important thing to understand about Chef is that not only do we have DSLs to address particular problem spaces, we also always have direct access to the entire Ruby programming language. This means that if at any stage you need to extend the DSL—or perform some calculation, transformation, or other task—you are never restricted by the DSL. This is one of the great advantages of Chef.

In Chef, order is highly significant. Recipes are processed in the exact order in which they are written, every time. Recipes are processed in two stages—a compile stage and an execute stage. The compile stage consists of gathering all the resources that, when configured, will result in conformity with policy, and placing them in a kind of list called the *resource collection*. At the second stage, Chef processes this list in order, taking actions as specified. As you become more advanced in Chef recipe development, you will learn that there are ways to subvert this process, and when it is appropriate to do so. However, for the purposes of this book, it is sufficient to understand that recipes are processed in order, and actions taken.

Recipes by themselves are frequently not much use. Many resources require additional data as part of their action—for example, the template resource will require, in addition to the resource block in the recipe, an Erubis template file. As you advance in your understanding and expertise, you may find that you need to extend Chef and provide your own custom resources and providers. For example, you might decide you want to write a resource to provide configuration file snippets for a certain service. Chef provides another DSL for specifically this purpose.

If recipes require supporting files and code, we need a way to package this up into a usable component. This is the purpose of a cookbook. Cookbooks can be thought of as package management for Chef recipes and code. They may contain a large number of different recipes and other components. Cookbooks have metadata associated with them, including version numbers, dependencies, license information, and attributes.

Cookbooks can be published and shared. This is another of Chef's great strengths. Via the Opscode Chef community website, you can browse and download over 800 different cookbooks. The cookbooks are generally of very high quality, and a significant proportion of them are written by Opscode developers. Cookbooks can be rated and categorized on the community site, and users can elect to "follow" cookbooks to receive updates when new versions become available.

`Knife` provides a subcommand that will create a skeleton cookbook, ready to be used for modeling infrastructure. By default, `Knife` will attempt to create and populate a directory at */var/chef/cookbooks*; this is the cookbook path, the place `Knife` looks for cookbooks:

```
$ knife cookbook create silly
WARNING: No knife configuration file found
** Creating cookbook silly
ERROR: Errno::EACCES: Permission denied - /var/chef/cookbooks

$ ls -ld /var/chef/*
drwxr-xr-x 2 root root 4096 May 28 18:05 /var/chef/cache

$ sudo knife cookbook create silly
[sudo] password for stephen:
WARNING: No knife configuration file found
** Creating cookbook silly
** Creating README for cookbook: silly
** Creating CHANGELOG for cookbook: silly
** Creating metadata for cookbook: silly
$ ls -ld /var/chef/*
drwxr-xr-x 2 root root 4096 May 28 18:05 /var/chef/cache
drwxr-xr-x 3 root root 4096 May 30 08:59 /var/chef/cookbooks
$ ls -ld /var/chef/cookbooks/*/*
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/attributes
-rw-r--r-- 1 root root  409 May 30 08:59 /var/chef/cookbooks/silly/CHANGELOG.md
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/definitions
drwxr-xr-x 3 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/files
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/libraries
-rw-r--r-- 1 root root  274 May 30 08:59 /var/chef/cookbooks/silly/metadata.rb
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/providers
-rw-r--r-- 1 root root 1439 May 30 08:59 /var/chef/cookbooks/silly/README.md
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/recipes
drwxr-xr-x 2 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/resources
drwxr-xr-x 3 root root 4096 May 30 08:59 /var/chef/cookbooks/silly/templates
```

This configuration can be set in `Knife`'s own configuration file, which we'll come to later. However, it can also be set on the command line with the `-o, --cookbook-path` option.

The cookbook generator will create a default recipe in the recipes directory of the cookbook. It was this file we opened in our text editor, to declare the package resource to install the IRC client. You'll notice that at the top of the file, some boilerplate was generated:

```
#
# Cookbook Name:: irc
# Recipe:: default
#
# Copyright 2013, YOUR_COMPANY_NAME
#
# All rights reserved - Do Not Redistribute
#
```

The contents of this can be modified by making further changes in your `Knife` configuration file, which we'll come to in the next exercise.

A Word About Text Editors

The art of modeling infrastructure as code is a discipline that fits firmly within the software development world. We're writing software that generates configuration dynamically on machines, in order to allow us to deploy and run applications that deliver business value. Software developers use full-featured text editors that remain open on the desktop at all times. They support syntax highlighting, may have the concept of a project drawer, may provide powerful search features, may be programmable, allow for multiple files to be edited at once and viewed side-by-side, and offer integration with source code management systems. Professional software developers use professional tools.

As an infrastructure developer, you're now a professional software developer, and you should use the same quality of tools. If you already use an editor that provides these kinds of features, then this exhortation is not for you. However, if you use vanilla *vi*, *nano*, or *Notepad*: please stop. Different editors have their own fierce advocates. Personally, I'm a huge fan of *Emacs*. However, *TextMate*, *Sublime Text*, *Vim*, *Emacs*, or maybe even *Eclipse* would make a fine choice. If you've never used such a tool, I'd suggest starting with *Sublime Text 2*—it's an excellent, modern editor with plug-in support for Chef development, and it works on Linux, OSX, and Windows. If you're prepared to put in a few days on a somewhat steeper learning curve, I would wholeheartedly recommend *Emacs*. Whatever you do, pick an editor, make it part of your professional development to learn its features, and master it thoroughly.

So now that we have a recipe and a cookbook, how can we apply these to the machine we want to configure? We already know we can use `chef-apply`, but now that we have a config file in our recipe, we need something a bit more powerful. We placed the config file we wanted to drop off into the cookbook. Now we need to tell `chef-solo` where to find the cookbook. `chef-solo` takes a number of command-line options, but not one that tells it where to find the cookbooks. This gives us two options: either we put the cookbooks where `chef-solo` expects to find them, or we create a configuration file that tells `chef-solo` to find them where we want them to be. We already know that Chef looks for cookbooks in */var/chef/cookbooks*, so that's an option, but for my local machine, I prefer to keep them in my home directory and tell Chef how to find it. Hence, we set it in the *solo.rb* file.

```
cookbook_path ENV['HOME']
```

This introduces a common pattern in Chef: configuration files are Ruby files so we can use whatever Ruby constructs we need.

Now we can tell Chef where to find its configuration file and consequently the cookbooks, but Chef doesn't know which recipe to run. When we used `chef-apply` it was simple: we just told Chef exactly which recipe to run. Obviously this doesn't scale beyond

exceptionally simple cases. Chef, therefore, has the concept of a *Run List*—a list of recipes to run on the node. The simplest way to do this is to pass it as a command line to `chef-solo`. Recipes on the run list take the following form: `recipe[cookbook_name::recipe_name]`. The convention is that if the "default" recipe is run, there's no need to specify it, and so the run list item will be `recipe[cookbook_name]`.

When Chef runs, the resources in the recipes on the run list are evaluated and action is taken to bring the system into desired state.

Exercise 4: Install Git

Objectives

Upon completing this exercise, you should have:

- Used community cookbooks to build infrastructure
- Understood how Chef differentiates between platforms, taking appropriate action
- A Git repository containing Chef code and supporting files, and be able to interact with it
- Understood the basics of Chef node attributes
- Understood the concept of dependencies in cookbooks
- Understood the mechanism for including recipes from other cookbooks inside another recipe

Directions

1. Read the documentation for `knife cookbook site`.
2. Download the *git* recipe from the Opscode community site, and extract it within your cookbook path.
3. Examine the *metadata.rb* file for the Git cookbook, and download the cookbooks upon which the Git cookbook depends.
4. Recurse through each downloaded cookbook, downloading each cook dependency.
5. Ensure all the cookbooks are on the cookbook path.
6. Search the documentation site for *dna.json*, and create a *dna.json* file containing a run list containing the default recipe from both the *irc* and the *git* cookbooks.
7. Run `chef-solo` with the appropriate arguments.
8. As the `TDI` user, find or locate a convenient position in your filesystem, and clone the *https://github.com/opscode/chef-repo.git* repository.

9. Configure Git with your name and email address.
10. Sign up for a GitHub account (if you don't have one already).
11. Create an ssh key pair, and upload the public portion to GitHub.
12. Create a repository called *chef-repo*, and set the remote origin of the cloned repository to this new repository.
13. Copy your cookbooks into the cookbooks directory of the *chef-repo*, add them, and push to GitHub.

Worked Example

I ran `knife help cookbook site`, and read the manual page. I noted an `install` option, which seemed to do some magic with Git. Being skeptical of magic, I read on, and found the section on downloading a cookbook. Having digested this, I ran the following:

```
$ cd
$ knife cookbook site download git
WARNING: No knife configuration file found
Downloading git from the cookbooks site at version 2.5.2 to /home/stephen/
git-2.5.2.tar.gz
Cookbook saved: /home/stephen/git-2.5.2.tar.gz
```

I decided that if I were going to have multiple cookbooks, I might as well have a cookbooks directory, so I made one, and updated my *solo.rb*:

```
$ mkdir ~/cookbooks
$ cat ~/.chef/solo.rb
cookbook_path "#{ENV['HOME']}/cookbooks"
```

I moved my `irc` cookbook into the *cookbooks* directory, and then extracted the `git` cookbook:

```
$ mv ~/irc ~/cookbooks
$ tar xzvf git-2.5.2.tar.gz -C cookbooks/
git/
git/.gitignore
git/.kitchen.yml
git/attributes/
git/Berksfile
git/CHANGELOG.md
git/CONTRIBUTING
git/Gemfile
git/LICENSE
git/metadata.json
git/metadata.rb
git/README.md
git/recipes/
git/templates/
git/TESTING.md
```

```
git/templates/default/
git/templates/default/git-xinetd.d.erb
git/templates/default/sv-git-daemon-log-run.erb
git/templates/default/sv-git-daemon-run.erb
git/recipes/default.rb
git/recipes/server.rb
git/recipes/source.rb
git/recipes/windows.rb
git/attributes/default.rb
```

I looked at the *metadata.rb* of the cookbook and discovered this:

```
%w{ dmg build-essential yum windows }.each do |cookbook|
  depends cookbook
end

depends "runit", ">= 1.0"
```

I downloaded these dependencies with the following:

```
$ for dep in dmg build-essential yum windows runit; do knife cookbook site down-
load $dep; tar xzvf $dep*gz -C cookbooks; done
```

I then recursed into these cookbooks as follows:

```
$ cd cookbooks
$ grep depends */metadata.rb
git/metadata.rb:  depends cookbook
git/metadata.rb:depends "runit", ">= 1.0"
runit/metadata.rb:depends "build-essential"
runit/metadata.rb:depends "yum"
windows/metadata.rb:depends          "chef_handler"
```

And downloaded the missing dependency:

```
$ knife cookbook site download chef_handler && tar xzvf chef_handler*gz -C cook-
books
```

I verified that this in turn didn't have a dependency, by checking its *metadata.rb* file.

Having read about *dna.json* (*http://bit.ly/1fQWLQE*), I created a *dna.json* file in my *.chef* directory, with the following content:

```
{
  "run_list": ["recipe[irc]", "recipe[git]"]
}
```

Upon running Chef, the Git package was successfully installed:

```
$ sudo chef-solo --config ~/.chef/solo.rb --json-attributes ~/.chef/dna.json
Starting Chef Client, version 11.4.4
Compiling Cookbooks...
Converging 5 resources
Recipe: irc::default
  * user[tdi] action create (up to date)
  * package[irssi] action install (up to date)
```

```
  * directory[/home/tdi/.irssi] action create (up to date)
  * cookbook_file[/home/tdi/.irssi/config] action create (up to date)
Recipe: git::default
  * package[git] action install
    - install version 1:1.8.1.2-1 of package git

Chef Client finished, 1 resources updated
```

I switched to the `tdi` user (with sudo in the case of Ubuntu), and felt that the root of the `tdi` home directory would be an admirable place to clone the Opscode Git repository.

```
$ sudo su - tdi
$ git clone git://github.com/opscode/chef-repo.git
Cloning into 'chef-repo'...
remote: Counting objects: 209, done.
remote: Compressing objects: 100% (128/128), done.
remote: Total 209 (delta 74), reused 170 (delta 47)
Receiving objects: 100% (209/209), 36.40 KiB, done.
Resolving deltas: 100% (74/74), done.
```

I set up Git to use my name and address, as shown:

```
$ git config --global color.ui "auto"
$ git config --global user.email "stephen@atalanta-systems.com"
$ git config --global user.name "Stephen Nelson-Smith"
```

I already have a GitHub account, so I simply created a key pair:

```
$ ssh-keygen -t dsa -f tdi-example
Generating public/private dsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in tdi-example.
Your public key has been saved in tdi-example.pub.
The key fingerprint is:
98:fb:2d:c6:ff:66:76:ac:b0:da:1d:37:1e:92:ae:64        stephen@Stephens-MacBook-
Air.local
The key's randomart image is:
+--[ DSA 1024]----+
|                 |
|                 |
|                 |
|       o         |
|      o S        |
|       .     .   |
|      ..  E +.+  |
|       .+= =+=oo |
|       .o+**=o.  |
+-----------------+
```

To add my key, I simply logged into GitHub, and navigated to *https://github.com/settings/ssh*. There I clicked "Add SSH Key," gave a title, pasted the public key—which was created in my working directory—and clicked "Add Key".

I then clicked the "Create a new repo" button, just to the right of my username, and created a repo called *tdi-example*. I gave it a description, and clicked the button to create the repository.

I then changed the remote URL for the repository I cloned to match the one I created:

```
$ cd ~/chef-repo
$ git remote set-url origin git@github.com:atalanta-cookbooks/tdi-example
```

I changed back to a user with appropriate privileges (root or sns with sudo) and returned to the original directory where I had created my cookbooks directory and rsync'd them into the *chef-repo/cookbooks* directory:

```
$ cd
$ sudo rsync -Pvar cookbooks/ /home/tdi/chef-repo/cookbooks/
$ sudo chown -R tdi: ~tdi/chef-repo
```

To push the repo, I changed back to the `tdi` user, cached my ssh key, and then ran `git add` and `git push`:

```
$ whoami
tdi
$ ssh-agent bash
$ ssh-add tdi-example
Identity added: tdi-example (tdi-example)
$ cd chef-repo/
$ git add cookbooks
$ git commit -m "Adding TDI cookbooks"
$ git push -u origin master
The authenticity of host 'github.com (204.232.175.90)' can't be established.
RSA key fingerprint is 16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,204.232.175.90' (RSA) to the list of
known hosts.
Counting objects: 209, done.
Compressing objects: 100% (101/101), done.
Writing objects: 100% (209/209), 36.40 KiB, done.
Total 209 (delta 74), reused 209 (delta 74)
To git@github.com:atalanta-cookbooks/tdi-example
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.(((range="endo-
frange",       startref="ix_2-Introduction-to-Chef-asciidoc14")))(((range="endo-
frange", startref="ix_2-Introduction-to-Chef-asciidoc13")))
```

Discussion

Within the Chef world, pretty much everything is addressable via an API. This extends to the community cookbook site. There are hundreds of cookbooks—perhaps now even

more than a thousand—available on the community cookbook site. `Knife` provides an interface to the site, allowing the searching, downloading, and sharing of cookbooks. Although for the purpose of this series of exercises, our concern is to learn the fundamentals of Chef so we're writing recipes ourselves, a fairly standard workflow would be to query the cookbooks site for a key word, and then inspect or use an open source cookbook. To pick a random example, suppose I'd been discussing setting up some form of LDAP service. With one command, I immediately have a set of candidate cookbooks written using a framework I understand, in version control, rated and used by other infrastructure developers. Even if I decide, having reviewed the candidates, to write (and maybe share) my own cookbook, I have the work of other people to inspire, guide, and inform me.

```
$ knife cookbook site search ldap
ca_openldap:
   cookbook:               http://cookbooks.opscode.com/api/v1/cookbooks/ca_open-
ldap
  cookbook_description: Configures a node to be an OpenLDAP server or client.
  cookbook_maintainer:  carguel
  cookbook_name:        ca_openldap
ldap:
  cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/ldap
  cookbook_description: Installs/Configures ldap
  cookbook_maintainer:  someara
  cookbook_name:        ldap
ldapknife:
  cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/ldapknife
  cookbook_description: Installs ldapknife.pl to /usr/local/bin
  cookbook_maintainer:  jackl0phty
  cookbook_name:        ldapknife
opendj:
  cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/opendj
  cookbook_description: Installs OpenDJ LDAP server
  cookbook_maintainer:  elliotkendall
  cookbook_name:        opendj
opendj-openam:
   cookbook:                http://cookbooks.opscode.com/api/v1/cookbooks/opendj-
openam
  cookbook_description: Installs/Configures opendj
  cookbook_maintainer:  thomasalrin
  cookbook_name:        opendj-openam
openldap:
  cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/openldap
   cookbook_description: Configures a server to be an OpenLDAP master, replica-
tion slave or client for auth
  cookbook_maintainer:  opscode
  cookbook_name:        openldap
sssd_ldap:
  cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/sssd_ldap
  cookbook_description: Installs/Configures LDAP on RHEL using SSSD
  cookbook_maintainer:  tas50
```

```
    cookbook_name:        sssd_ldap
  zone2ldif:
    cookbook:             http://cookbooks.opscode.com/api/v1/cookbooks/zone2ldif
    cookbook_description: Installs/Configures zone2ldif
    cookbook_maintainer:  jackl0phty
    cookbook_name:        zone2ldif
```

Once a cookbook has been identified as worthy of further investigation, it can be downloaded. As I alluded to in the worked example, Chef does provide a somewhat magical `install` subcommand, which will install upstream community cookbooks to a local Git repository. The steps it takes are as follows:

1. Create a fresh *pristine copy* branch for tracking upstream.
2. Remove any existing cookbook versions from the branch.
3. Download the cookbook tarball.
4. Extract the tarball and commits the contents to Git.
5. Merge *pristine* copy into *master*.

The idea is that upstream changes can be maintained as a patch and merged with local changes when needed. This pattern was pretty common in CVS and is called "vendor branching" (see *http://bit.ly/19AU9os*). I tend not to recommend this approach. First, I don't much like magic. Git is complex. Blindly allowing branching and merging to happen without clearly understanding what is going on is a recipe for future pain. I also don't tend to recommend keeping all your cookbooks in one repository for anything beyond learning the basics, and the site install mechanism expects you to be already within a Git repository when running the command. We'll discuss this in more depth later, but for now I'd caution against using `knife cookbook site install`. The simplest approach is to use the `download` subcommand, which simply pulls down a tarball of the specified cookbook and version. You can then extract and work with it in your own way. Later we'll discuss more powerful ways of approaching this issue, focused particularly on treating upstream cookbooks as dependencies and artifacts, rather than a grab bag of modifiable source code. However, for now, use `knife cookbook site download`.

Cookbooks are effectively the packaging system for infrastructure code. If you've ever worked with a packaging system before—*RPM*, *dpkg*, *SVR4*, *pkgsrc*, *Rubygems*—you will be aware that there is a known set of problems that need to be solved. These problems include how to express dependencies upon other cookbooks, how to handle versioning, how to handle potential conflicts, license information, discoverability, and so forth. The common component of all approaches to the solution is to maintain package metadata within each package.

Cookbook dependencies are a challenge you meet very quickly when you start to build on and use community cookbooks. Unsurprisingly, with a library of 1,000+ high-quality cookbooks, cookbook writers tend to use one another's cookbooks to make life easy.

For example, one cookbook may contain functionality for service management (*runit* or *daemontools*), and another for managing third-party upstream package repositories (*apt* or *yum*). A cookbook delivering a service that needs process management across platforms, and needs to configure upstream repositories, might well make use of the yum, apt, and runit cookbooks as a result. It gets slightly more involved when Windows and OSX are included. Additional primitives for managing Windows and OSX are provided in cookbooks, and not yet in core Chef. The Windows cookbook itself depends on functionality provided by the *Chef Handler* cookbook. All these dependencies must be expressed in the cookbook metadata. The metadata doesn't support conditional logic so we can't say, "If this machine is running on Linux, we don't need the Windows or OSX-specific dependencies." This means that in the case of a cross-platform community cookbook, you'll find yourself depending on cookbooks for a platform you'll never use. It's a bit of a bore, but it's not a solved problem; these challenges exist in all packaging solutions.

The cookbook metadata file is another Chef DSL. It's a DSL for generating JSON that Chef uses for dependency solving and package management. An example metadata file would be:

```
name             "windows"
maintainer       "Opscode, Inc."
maintainer_email "cookbooks@opscode.com"
license          "Apache 2.0"
description      "Provides a set of useful Windows-specific primitives."
long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
version          "1.8.10"
supports         "windows"
depends          "chef_handler"
```

As a cookbook author, if your cookbook makes use of any recipes or library code from another cookbook, you *must* include this as a dependency in your metadata.

Obviously, this manual dependency solving is a bit of a pain. We'll introduce tooling that removes this pain later, but this example serves to demonstrate how the dependencies work and makes their existence and importance explicit.

The *dna.json* file introduces an important new concept—*node attributes*. An attribute is that which inherently belongs to and can be predicated of anything. The sky has the attribute `color: blue`. A web server has an attribute: `listen_port: 80`. A server has the attribute `disks: 8`. *Attributes*, therefore, are data associated with the node.

You'll remember I defined a node as a Ruby object representing the machine we're configuring. This object behaves like a Hash: it has keys and values, getter and setter methods, and can be viewed, queried, and interacted with as JSON. The keys and values are referred to as *node attribute data*.

Some of this data is collected automatically by Ohai, such as the hostname, IP address, and a large amount of other pieces of information. However, arbitrary data can be

associated with the node as well. Here we see a significant implication of using `chef-solo`. With `chef-solo`, there is no server; there is no persistent state that records the attributes of the node. That state must be handed to Chef, in the form of a JSON file. In our simple case, the only attribute that we're setting is the `run_list` attribute. However, we could provide any number of keys and values.

Attributes allow sane defaults to be set for a cookbook. Rather than hardcoding implementation detail in a recipe, we can use an attribute like a variable. If you look at the Git cookbook we download, you'll see that a number of attributes are set in the *attributes/default.rb* file:

```
case node['platform_family']
when 'windows'
  default['git']['version'] = "1.8.1.2-preview20130201"
  default['git']['url'] = "https://msysgit.googlecode.com/files/Git-#↵
{node['git']['version']}.exe"
  default['git']['checksum'] = ↵
"796ac91f0c7456b53f2717a81f475075cc581af2f447573131013cac5b63bb2a"
  default['git']['display_name'] = "Git version #{ node['git']['version'] }"
when "mac_os_x"
   default['git']['osx_dmg']['app_name']     =    "git-1.8.2-intel-universal-snow-
leopard"
   default['git']['osx_dmg']['volumes_dir'] =    "Git 1.8.2 Snow Leopard Intel
Universal"
  default['git']['osx_dmg']['package_id']  =   "GitOSX.Installer.git182.git.pkg"
  default['git']['osx_dmg']['url']  = "https://git-osx-installer↵
.googlecode.com/files/git-1.8.2-intel-universal-snow-leopard.dmg"
  default['git']['osx_dmg']['checksum']    =   ↵
"e1d0ec7a9d9d03b9e61f93652b63505137f31217908635cdf2f350d07cb33e15"
else
  default['git']['prefix'] = "/usr/local"
  default['git']['version'] = "1.8.2.1"
  default['git']['url'] =   "https://nodeload.github.com/git/git/tar.gz/v#↵
{node['git']['version']}"
  default['git']['checksum'] =   ↵
"bdc1768f70ce3d8f3e4edcdcd99b2f85a7f8733fb684398aebe58dde3e6bcca2"
end

default['git']['server']['base_path'] = "/srv/git"
default['git']['server']['export_all'] = "true"
```

We'll cover how these attributes function in more detail shortly, but for now, I'd draw your attention to the conditional logic. The node has an attribute `platform_family`. This comes from Ohai. Ohai is able to determine if a machine is, for example, of Debian flavor or Windows flavor. Based on that, we can make decisions in our cookbooks and recipes. In this case, we're specifying which versions of Git to obtain from an upstream provider, and which checksums should be used to verify that we obtained the correct file. Returning briefly to the metadata, you'll also note that the metadata specifies which platforms the cookbook supports:

```
$ grep -C1 supports chef-repo/cookbooks/git/metadata.rb
%w{ amazon arch centos debian fedora redhat scientific oracle amazon ubuntu win-
dows }.each do |os|
  supports os
end

supports "mac_os_x", ">= 10.6.0"
```

Again, as a cookbook author, you *should* specify which platforms you support. If you don't, you're implicitly stating your cookbook supports all platforms, which is almost certainly not true.

So we set the run list to be an array of recipes in order. First we said that the default irc recipe should be applied, and then the default git recipe. The result was that Git was installed on our machine.

The example Opscode *chef-repo* contains all the directories you will need and work with as part of your regular workflow as an infrastructure developer. It also contains a *Rakefile*, which provides some useful tasks, such as creating self-signed SSL certificates. In practice, you're unlikely to use `rake`, as `knife` will do more than 99% of the tasks you'll find yourself needing to do. The *chef-repo* pattern is somewhat out of favor, and might even be considered an anti-pattern. The reason for this is that by putting absolutely everything in a single repository, we're mixing temporal data—things that might change—with versioned artifacts. It also runs counter to the Git philosophy: have a repository for each software project and keep them light and mobile. Cookbooks are very much software projects, with independent versions, tags, development teams, and purposes. It just doesn't make much sense to stick them all in one place. The emerging recommendation is to put temporal data in a *chef-data* repository and maintain a repository per cookbook. This makes it easy to track upstream by adding a remote, and pulling and merging when required. Note that this is without the magic of the `knife cookbook site install` command, and is a much more explicit procedure.

However, as a starting point, the monolithic Chef repository has its place. It gathers everything we need in one place. For users new to the idea of using version control at all, let alone something with the Byzantine reputation of Git, the learning curve of a single repository is pretty low. We can then refactor at the point of need—as soon as we start to feel the limitations of our approach, we should refactor—and move to the next level.

Of course we could have used the GitHub fork mechanism within the web interface to simplify the process of having our own Chef repository, but I wanted to show the manual process and support the ability to use other sources of Git server—such as an internally hosted Git server or an alternative public Git serve, such as *Bitbucket*.

In the next chapter, we'll build on the work done here by installing some essential tools using Chef.

CHAPTER 4

Using Chef with Tools

In the last chapter we installed Chef itself, a user, an IRC client, and Git. Now we move on to develop our infrastructure and our understanding further by installing and using Ruby, VirtualBox, and Vagrant.

Exercise 1: Ruby

Objectives

After completing this exercise, you will:

- Understand the differences between `chef-solo` and a server-based Chef setup
- Understand the node object in more detail
- Be set up to use Opscode's Hosted Chef service
- Understand the authentication mechanism used by Hosted Chef
- Have installed a modern Ruby on your system using Chef
- Understand the roles primitive in Chef
- Understand the idea of attribute precedence
- Have examined the components of a Chef run

Directions

1. Create an Opscode community login (if you don't have one already).
2. Download your user's private key.
3. Navigate to the *Hosted Chef Operations Console*.

4. Create an *organization*, if you don't already have one, selecting the free tier.
5. Download your organization's *validation key*.
6. Download the *knife.rb* configuration file for your organization.
7. Create a *.chef* directory under your *chef-repo* directory, and place your two keys and *knife.rb* inside this directory.
8. Read the `knife configure` documentation, and use it to create a *client.rb* file and validation certificate in */etc/chef*.
9. Validate your setup by running `knife client list`.
10. Look at the `chef-client` help page, and identify how to pass JSON to a `chef-client` run.
11. Run `chef-client` with the *dna.json* file created in the previous exercise.
12. Upload the cookbooks required to satisfy the run list to the Chef server.
13. Run `chef-client` again with the *dna.json* file created in the previous exercise.
14. Download the *chruby*, *ark*, and *ruby_build* cookbooks and place them in your `chef-repo`.
15. Upload the cookbooks to the Chef server.
16. Read the documentation shipped with the *chruby* cookbook to understand which attributes can be set.
17. Create a *role* that, in addition to the `git` and `irc` recipes, applies the `system` recipe from the *chruby* cookbook, and set the attributes to install the latest Ruby 1.9.3, and set it as default.
18. Upload the role and cookbooks to Hosted Chef.
19. Update the node's run list, replacing the `irc` and `git` recipes with the role you created.
20. Run `chef-client`.
21. Verify that your user has the version of Ruby you desired.

Worked Example

I already have an Opscode user and I use Hosted Chef, so I decided I'd create another user for the purpose of demonstration. I browsed to the community website, and clicked the sign up link.

On the sign up page, I filled out the form with a username, password, name, company name, country, and state, and agreed to the terms and conditions.

This took me to a welcome page that read:

> Your new Opscode account has been created, but some features of your account will not work until you verify your email address. To complete your verification, please check your email. Open the email from Opscode and click the enclosed link.

It also read:

> Your User Key
>
> Opscode uses two private keys: an organization-wide key and a user account-specific key (or "user key"). Opscode does not keep a copy of any private keys, so please store it somewhere safe. Learn more about private keys used by Chef.

A private key was displayed on the screen. However, I'd had experiences where copying and pasting the key gave unexpected results, so I elected to download the key as a file. I did this by clicking on my username at the top-right of the screen, and then clicking "Get a new private key."

This page read:

> Get a new private key
>
> If you've lost your private key, or would like to replace it, click the button below. When you get a new key, your old key will stop working. This private key replaces your old key. We do not keep a copy so please store it somewhere safe.

I clicked "Get a new key," and the key was downloaded to my local machine.

Next I checked my email, and found one that read:

> Hello TDI Example,
>
> Thank you for signing up with us!
>
> Please click this link to verify that you've signed up for this account:
>
> *https://community.opscode.com/users/tdiexample/email_addresses/30358/verification_requests/f8d32c8f-2519-ee08-37ef-e3a21ed28e14*
>
> Your Account has been created with the following information: User Name : tdiexample Email Address : *cookbooks@atalanta-systems.com*
>
> Thanks, The Opscode Team

I clicked the link and found myself on a landing page with options for what to do next:

- Read the Getting Started Guide
- Manage your org with the Operations Console
- Need Help?

I selected the middle option, which took me to the Opscode Hosted Chef Operations Console. This page invited me to create an organization. I filled out the form and selected the free tier.

On the resulting page, there was a link to download the validation key and to generate a *knife.rb*. I clicked both links and saved the resulting files. At the end, I had three files:

- *tdiexample.pem*
- *hunterhayes-validation.pem*
- *knife.rb*

I created a *.chef* directory under my `chef-repo` and moved these three files under it:

```
$ ls -1F chef-repo/.chef/
hunterhayes-validator.pem
knife.rb
tdiexample.pem
```

I read the manual page for `knife configure` and determined that `knife configure client` would read my *knife.rb* and create a *client.rb* file and a validation certificate. I ran the following to create the files:

```
$ knife configure client /tmp
Creating client configuration
Writing client.rb
Writing validation.pem
```

I then assumed administrator privileges, ensured the */etc/chef* directory existed, and copied the *client.rb* and *validation.pem* files into the */etc/chef* directory, with the following result:

```
# find /etc/chef/
/etc/chef/
/etc/chef/validation.pem
/etc/chef/client.rb
```

I returned to my `tdi` user, changed into my *chef-repo* directory, and validated my setup as follows:

```
$ cd ~/chef-repo
$ knife client list
hunterhayes-validator
```

I ran `chef-client --help` and noted that with the `-j, --json-attributes` flag, I could pass JSON to the client. Armed with this knowledge, I returned to my empowered user (`sudo sns` or `root`), and ran the following:

```
$ sudo chef-client -j .chef/dna.json
Starting Chef Client, version 11.4.4
[2013-06-27T09:25:51+01:00] INFO: *** Chef 11.4.4 ***
[2013-06-27T09:25:51+01:00] INFO: [inet6] no default interface, picking the
first ipaddress
Creating a new client identity for ubuntu using the validator key.
[2013-06-27T09:25:52+01:00] INFO: Client key /etc/chef/client.pem is not
present - registering
```

```
[2013-06-27T09:25:54+01:00] INFO: Setting the run_list to ["recipe[irc]",
"recipe[git]"] from JSON
[2013-06-27T09:25:54+01:00] INFO: Run List is [recipe[irc], recipe[git]]
[2013-06-27T09:25:54+01:00] INFO: Run List expands to [irc, git]
[2013-06-27T09:25:54+01:00] INFO: Starting Chef Run for ubuntu
[2013-06-27T09:25:54+01:00] INFO: Running start handlers
[2013-06-27T09:25:54+01:00] INFO: Start handlers complete.
resolving cookbooks for run list: ["irc", "git"]
[2013-06-27T09:25:55+01:00] INFO: HTTP Request Returned 412 Precondition
Failed: {"message"=>"Run list contains invalid items: no such cookbooks irc,
git.", "non_existent_cookbooks"=>["irc", "git"], "cookbooks_with_no_ver-
sions"=>[]}

================================================================================
Error Resolving Cookbooks for Run List:
================================================================================


Missing Cookbooks:
------------------
The following cookbooks are required by the client but don't exist on the serv-
er:
* irc
* git




Expanded Run List:
------------------
* irc
* git


[2013-06-27T09:25:55+01:00] ERROR: Running exception handlers
[2013-06-27T09:25:55+01:00] FATAL: Saving node information to /var/chef/cache/
failed-run-data.json
[2013-06-27T09:25:55+01:00] ERROR: Exception handlers complete
Chef Client failed. 0 resources updated
[2013-06-27T09:25:55+01:00] FATAL: Stacktrace dumped to /var/chef/cache/chef-
stacktrace.out
[2013-06-27T09:25:55+01:00] FATAL: Net::HTTPServerException: 412 "Precondition
Failed"
```

I checked the cookbooks I had in my *cookbooks* directory:

```
$ ls -1F cookbooks
build-essential/
chef_handler/
dmg/
git/
irc/
README.md
```

```
runit/
windows/
yum/
```

And uploaded them all using:

```
$ knife cookbook upload -a
Uploading build-essential [1.4.0]
Uploading chef_handler   [1.1.4]
Uploading dmg            [1.1.0]
Uploading git            [2.5.2]
Uploading irc            [0.1.0]
Uploading runit          [1.1.6]
Uploading windows        [1.10.0]
Uploading yum            [2.3.0]
Uploaded all cookbooks.
```

I returned to my power user and ran `chef-client` again, this time noting that the node converged, but without taking any action, as the system was already configured from the previous `chef-solo` exercise:

```
$ sudo chef-client -j .chef/dna.json
Starting Chef Client, version 11.4.4
[2013-06-27T09:41:40+01:00] INFO: *** Chef 11.4.4 ***
[2013-06-27T09:41:40+01:00] INFO: [inet6] no default interface, picking the
first ipaddress
[2013-06-27T09:41:41+01:00] INFO: Setting the run_list to ["recipe[irc]",
"recipe[git]"] from JSON
[2013-06-27T09:41:41+01:00] INFO: Run List is [recipe[irc], recipe[git]]
[2013-06-27T09:41:41+01:00] INFO: Run List expands to [irc, git]
[2013-06-27T09:41:42+01:00] INFO: Starting Chef Run for ubuntu
[2013-06-27T09:41:42+01:00] INFO: Running start handlers
[2013-06-27T09:41:42+01:00] INFO: Start handlers complete.
resolving cookbooks for run list: ["irc", "git"]
[2013-06-27T09:41:43+01:00] INFO: Loading cookbooks [build-essential, chef_han-
dler, dmg, git, irc, runit, windows, yum]
Synchronizing Cookbooks:
  - yum
  - build-essential
  - runit
  - chef_handler
  - windows
  - dmg
  - git
  - irc
Compiling Cookbooks...
Converging 5 resources
Recipe: irc::default
    * user[tdi] action create[2013-06-27T09:41:43+01:00] INFO: Processing
user[tdi] action create (irc::default line 1)
 (up to date)
   * package[irssi] action install[2013-06-27T09:41:43+01:00] INFO: Processing
package[irssi] action install (irc::default line 8)
```

```
 (up to date)
  * directory[/home/tdi/.irssi] action create[2013-06-27T09:41:43+01:00] INFO:
Processing directory[/home/tdi/.irssi] action create (irc::default line 12)
 (up to date)
             *       cookbook_file[/home/tdi/.irssi/config]       action
create[2013-06-27T09:41:43+01:00]    INFO:    Processing    cookbook_file[/home/
tdi/.irssi/config] action create (irc::default line 17)
 (up to date)
Recipe: git::default
   * package[git] action install[2013-06-27T09:41:43+01:00] INFO: Processing
package[git] action install (git::default line 24)
 (up to date)
[2013-06-27T09:41:44+01:00] INFO: Chef Run complete in 1.996144727 seconds
```

I finally returned to the `tdi` user, and downloaded the *chruby*, *ark*, and *ruby_build* cookbooks in the usual way:

```
$ for cb in ark chruby ruby_build; do knife cookbook site download $cb && tar
xvf $cb*gz -C ~/chef-repo/cookbooks/; done
```

I attempted to upload the cookbooks, beginning with the *chruby* cookbook, but discovered that I needed to upload them in order:

```
$ knife cookbook upload chruby
Uploading chruby         [0.1.5]
ERROR: Cookbook chruby depends on cookbook 'ark' version '>= 0.0.0',
ERROR: which is not currently being uploaded and cannot be found on the server.
```

I checked the dependencies in the metadata file, and first uploaded the cookbook on which *chruby* depended:

```
$ cd ~/chef-repo
$ knife cookbook upload {ark,ruby_build,chruby}
Uploading ark            [0.2.2]
Uploading ruby_build     [0.8.0]
Uploading chruby         [0.1.5]
Uploaded 3 cookbooks.
```

I read the documentation of the *chruby* cookbook, and identified that I needed to specify the Rubies I wanted to install and the version I wanted to use by default. Armed with this information, I created a role as follows:

```
$ cat developer.rb
name "developer"
description "For Developer machines"
run_list(
  "recipe[irc]",
  "recipe[git]",
  "recipe[chruby::system]"
)

default_attributes(
  "chruby" => {
```

```
    "rubies" => {
      "1.9.3-p392" => false,
      "1.9.3-p429" => true
    },
    "default" => "1.9.3-p429"
  }
)
```

I uploaded the role to the Chef server using `Knife`:

```
$ knife role from file developer.rb
```

To alter the run list, I used `knife node edit`. This required me to set an `EDITOR` environment variable:

```
$ export EDITOR=vi
$ knife node edit ubuntu
$ knife node edit centos
```

I updated the JSON to set the run list to *`role[developer]`*, and saved the file. After checking the run list, I ran `chef-client`:

```
$ knife node show centos -r
romanesco:
  run_list: role[developer]
$ sudo chef-client
Starting Chef Client, version 11.4.4
resolving cookbooks for run list: ["irc", "git", "chruby::system"]
Synchronizing Cookbooks:
  - runit
  - ruby_build
  - windows
  - irc
  - ark
  - yum
  - git
  - build-essential
  - chef_handler
  - dmg
  - chruby
Compiling Cookbooks...
Converging 22 resources
Recipe: irc::default
  * user[tdi] action create (up to date)
  * package[irssi] action install (up to date)
  * directory[/home/tdi/.irssi] action create (up to date)
  * cookbook_file[/home/tdi/.irssi/config] action create (up to date)
Recipe: git::default
  * package[git] action install (up to date)
Recipe: ruby_build::default
  * package[tar] action install (up to date)
  * package[bash] action install (up to date)
  * package[curl] action install (up to date)
```

```
  * package[git-core] action install (skipped due to not_if)
  * execute[Install ruby-build] action nothing (skipped due to not_if)
  * directory[/var/chef/cache] action create (up to date)
  * git[/var/chef/cache/ruby-build] action checkout (up to date)
Recipe: chruby::system
   * ruby_build_ruby[1.9.3-p429] action installRecipe: <Dynamically Defined Re-
source>
  * package[build-essential] action install
    - install version 11.6ubuntu4 of package build-essential

  * package[bison] action install
    - install version 2:2.5.dfsg-3ubuntu1 of package bison

  * package[openssl] action install (up to date)
  * package[libreadline6] action install (up to date)
  * package[libreadline6-dev] action install
    - install version 6.2-9ubuntu1 of package libreadline6-dev

  * package[zlib1g] action install (up to date)
  * package[zlib1g-dev] action install
    - install version 1:1.2.7.dfsg-13ubuntu2 of package zlib1g-dev

  * package[libssl-dev] action install
    - install version 1.0.1c-4ubuntu8 of package libssl-dev

  * package[libyaml-dev] action install
    - install version 0.1.4-2build1 of package libyaml-dev

  * package[libsqlite3-0] action install (up to date)
  * package[libsqlite3-dev] action install
    - install version 3.7.15.2-1ubuntu1 of package libsqlite3-dev

  * package[sqlite3] action install
    - install version 3.7.15.2-1ubuntu1 of package sqlite3

  * package[libxml2-dev] action install
    - install version 2.9.0+dfsg1-4ubuntu4 of package libxml2-dev

  * package[libxslt1-dev] action install
    - install version 1.1.27-1ubuntu2 of package libxslt1-dev

[2013-06-02T20:47:16+00:00] WARN: Cloning resource attributes for package[auto-
conf] from prior resource (CHEF-3694)
[2013-06-02T20:47:16+00:00] WARN: Previous package[autoconf]: /var/chef/cache/
cookbooks/ark/recipes/default.rb:25:in `from_file'
[2013-06-02T20:47:16+00:00] WARN: Current  package[autoconf]: /var/chef/cache/
cookbooks/ruby_build/providers/ruby.rb:84:in  `block  in  install_ruby_dependen-
cies'
  * package[autoconf] action install
    - install version 2.69-1ubuntu1 of package autoconf

  * package[libc6-dev] action install (up to date)
```

```
  * package[ssl-cert] action install
    - install version 1.0.32 of package ssl-cert

  * package[subversion] action install
    - install version 1.7.5-1ubuntu3 of package subversion

  * execute[ruby-build[1.9.3-p429]] action run
    - execute /usr/local/bin/ruby-build "1.9.3-p429" "/opt/rubies/1.9.3-p429"

  * package[build-essential] action nothing (up to date)
  * package[bison] action nothing (up to date)
  * package[openssl] action nothing (up to date)
  * package[libreadline6] action nothing (up to date)
  * package[libreadline6-dev] action nothing (up to date)
  * package[zlib1g] action nothing (up to date)
  * package[zlib1g-dev] action nothing (up to date)
  * package[libssl-dev] action nothing (up to date)
  * package[libyaml-dev] action nothing (up to date)
  * package[libsqlite3-0] action nothing (up to date)
  * package[libsqlite3-dev] action nothing (up to date)
  * package[sqlite3] action nothing (up to date)
  * package[libxml2-dev] action nothing (up to date)
  * package[libxslt1-dev] action nothing (up to date)
  * package[autoconf] action nothing (up to date)
  * package[libc6-dev] action nothing (up to date)
  * package[ssl-cert] action nothing (up to date)
  * package[subversion] action nothing (up to date)
  * execute[ruby-build[1.9.3-p429]] action nothing (up to date)
Recipe: ark::default
  * package[unzip] action install
    - install version 6.0-8ubuntu1 of package unzip

  * package[libtool] action install
    - install version 2.4.2-1.2ubuntu1 of package libtool

  * package[rsync] action install (up to date)
  * package[autoconf] action install (up to date)
  * package[make] action install (up to date)
  * package[autogen] action install
    - install version 1:5.17.1-1ubuntu2 of package autogen

Recipe: chruby::default
  * ark[chruby] action install_with_makeRecipe: <Dynamically Defined Resource>
  * directory[/usr/local/chruby-1] action create
    - create new directory /usr/local/chruby-1

  * remote_file[/var/chef/cache/chruby.tar.gz] action create
    - copy file downloaded from [] into /var/chef/cache/chruby.tar.gz
        (new content is binary, diff output suppressed)

  * execute[unpack /var/chef/cache/chruby.tar.gz] action nothing (up to date)
  * execute[autogen /usr/local/chruby-1] action nothing (skipped due to only_if)
```

```
   * execute[configure /usr/local/chruby-1] action nothing (skipped due to on-
ly_if)
  * execute[make /usr/local/chruby-1] action nothing (up to date)
  * execute[make install /usr/local/chruby-1] action nothing (up to date)
  * execute[unpack /var/chef/cache/chruby.tar.gz] action run
    - execute /bin/tar xzf /var/chef/cache/chruby.tar.gz --strip-components=1

  * execute[autogen /usr/local/chruby-1] action run (skipped due to only_if)
  * execute[configure /usr/local/chruby-1] action run (skipped due to only_if)
  * execute[make /usr/local/chruby-1] action run
    - execute make

  * execute[make install /usr/local/chruby-1] action run
    - execute make install

Recipe: chruby::default
  * link[/usr/local/chruby] action create
    - create symlink at /usr/local/chruby to /usr/local/chruby-1

  * template[/etc/profile.d/chruby.sh] action create
    - create template[/etc/profile.d/chruby.sh]
                --- /tmp/chef-tempfile20130602-3703-1u9rms9        2013-06-02
20:53:55.387078184 +0000
           +++ /tmp/chef-rendered-template20130602-3703-1jtacvw     2013-06-02
20:53:55.387078184 +0000
        @@ -0,0 +1,7 @@
        +source /usr/local/chruby/share/chruby/chruby.sh
        +source /usr/local/chruby/share/chruby/auto.sh
        +RUBIES+=(/opt/chef/embedded)
        +
        +
        +
        +chruby 1.9.3-p429

Chef Client finished, 26 resources updated
```

Chef ran, installed dependent software, and compiled and made Ruby available. I verified as follows:

```
$ ruby --version
ruby 1.9.3p429 (2013-05-15 revision 40747) [x86_64-linux]
```

Discussion

At its simplest, the process of developing infrastructure with Chef looks like this:

- Declare policy using resources.
- Collect resources into recipes.
- Package recipes and supporting code into cookbooks.
- Apply recipes from cookbook to nodes.

- Run Chef to configure nodes.

A useful abstraction in this process is the idea of a *role*. A role is a way of characterizing a class of node. If you could hold a conversation with someone and refer to a node as being a certain type of machine, you're probably talking about a node. If you were to say "zircon is a mysql slave" you'd be talking about a role called "mysql_slave".

Of all the primitives available in Chef, roles are at the top of the evolutionary tree.[1] Everything points to roles, and roles can encompass everything. In this respect, what they achieve is arguably the most important concept to understand. A role can be very simple. A common pattern is to have a *base* role, which every machine might share. This could be responsible for configuring an NTP server, ensuring Git is installed, and could include sudo and users.

Roles are composed of two sections: a *run list* and a set of *attributes*. In this respect, they mirror nodes. Nodes are objects that represent the machine that is being configured, and also contain a set of attributes and a run list.

We've already encountered the run list—it's simply a list of recipes and/or roles that should be present on the node. If a node has an empty run list, it will remain unconfigured. If a node has a run list containing the memcached recipe, the resources and actions specified in that recipe will be applied to that node. This process is known as *node convergence*. Importantly, the run list can contain recipes or roles, resulting in the ability to nest roles for certain types of infrastructure modeling.

We've also touched on the idea of attributes—attributes are data associated with the node. Some of this data is collected automatically, such as the hostname, IP address, and a large amount of other pieces of information. However, arbitrary data can be associated with the node as well. This is particularly useful for specifying configuration defaults, while enabling the user to override them with values that suit themselves. Cookbooks are typically shipped with some sane default values. Roles provide an opportunity to change that sane default. Any machines that then have the role on their run list will get the value of the attribute set in the role rather than the one set by default in the cookbook. In our case, the *chruby* cookbook set the version of Ruby to be installed to a patch version older than the one we wanted, and also elected to set the default Ruby to the one embedded with the Chef package:

```
$ cat cookbooks/chruby/attributes/default.rb
default['chruby']['version'] = '0.3.4'
default['chruby']['gpg_check'] = false
default['chruby']['use_rvm_rubies'] = false
default['chruby']['use_rbenv_rubies'] = false
default['chruby']['auto_switch'] = true
```

1. In recent times it has been argued that roles have some disadvantages, and alternative approaches have become popular. We discuss this in more detail in Chapter 7.

```
default['chruby']['rubies'] = {'1.9.3-p392' => true}
default['chruby']['default'] = 'embedded'
default['chruby']['user_rubies'] = {}
```

We didn't want those defaults, so we changed them in the role:

```
default_attributes(
  "chruby" => {
    "rubies" => {
      "1.9.3-p392" => false,
      "1.9.3-p429" => true
    },
    "default" => "1.9.3-p429"
  }
)
```

So far in our examples, we've only used either `chef-solo` or `chef-apply`. This is fine, in that it allows recipes to be executed on an individual node and gives access to the core recipe DSL, together with all the configuration primitives it provides. It's easy to get started with these tools, and it's fast. It also provides great power for little investment. However there are a number of constraints that are quickly felt.

First, `chef-solo` doesn't have a trivial implementation of persistent node data. During node convergence, the data produced by `ohai` is available, but any other data needs to be provided in the form of JSON files. This is simple enough for a few attributes for a few nodes, but it quickly becomes a pain and requires the creation of a solution to store, distribute, and update these JSON files. `chef-solo` can take the JSON from an HTTP URL, but this requires the construction and maintenance of that service.

Second, `chef-solo` requires that the cookbooks be provided to it prior to node convergence. This means that all changes to cookbooks need to be distributed to all nodes. Additionally, `chef-solo` does not have a dependency solver, so either a dependency solver needs to be written or located that can check each cookbook's metadata and ensure that the required cookbooks are delivered to the node, or every cookbook is delivered for good measure. Notwithstanding the realization that it isn't very elegant or efficient to do this—sometimes there can be large binary files in cookbooks. This is certainly an anti-pattern, but it's not uncommon, and the inability to select which cookbooks are or are not needed on a node rapidly gets painful. There are also questions around the security implications of having the infrastructure code that builds your entire environment on every server, visible in the event of a compromise. In addition to this, not only do the cookbooks need to be distributed to each node, a careful decision needs to be made about which exact versions of which cookbooks are distributed to each node. It's not unusual to run different versions of cookbooks on different nodes—either for development reasons, or simply because some nodes serve a subtly different purpose. Accommodating this requirement makes the cookbook distribution problem exponentially harder. Again, `chef-solo` can take an HTTP URL, and the cookbooks can be

cleared away afterwards, but now there's another service that needs to be built, and for which access control, security, and hosting need to be considered.

Third, one of the core ideas of Chef is that there should be a canonical, searchable source of information about the infrastructure that can be used dynamically to build infrastructure accordingly. In simple terms, we want to find things out about our infrastructure. We want to be able to ask questions such as, "Which machines have the web server role?" or "Tell me nodes in Rackspace that use the `postgresql::client` recipe". We also want to be able to look at a record of convergence: how many machines haven't had Chef run on them in the last 24 hours? How many machines are running a certain version of OpenSSL? Using a server-based implementation immediately provides this functionality—every node attribute, plus arbitrary, system-wide data, is stored and indexed, and available for querying at any stage.

The result of these constraints is that people determined to use Chef Solo end up trying to build the basic primitives of a Chef Server—node storage, search, and cookbook distribution.

In my view, it boils down to this: a significant amount of thought went into deciding how to build an outstanding automation framework. This thought was informed by deep experience of using other configuration management approaches and of having to solve infrastructure automation, at scale and complexity, across a large number of different technical environments and commercial applications. A significant amount of thought went into working out how to separate data and configuration to allow maximum power and flexibility in modeling infrastructure. A significant amount of thought went into how to model the storage of canonical infrastructure data. The result of that thought wasn't "let's write a DSL and ship JSON around via random websites or Rsync or Git." The solution was to build a REST API with a dependency solver, an index, and a publishing service. This is the function of the Chef Server.

The Chef Server is available in three forms:

The open source Chef Server
: Opscode ships a free version of the Chef Server in the same easy-to-use format as the Chef Client package.[2] This represents the reference API for Chef and provides all the core functionality that is required to build and maintain infrastructure with Chef. Certain enterprise features around security and access control are not available, and while Opscode remains committed to trickling down advanced features as they are developed, there is a time delay, and under certain circumstances, the decision may be made that a feature will not be released into the open source product at all. When running the open source Chef Server (OSC), it is incumbent upon the

2. This is for Chef 11. If you need the older, Chef 10 server, you might like to take a look at *http://fnichol.github.io/knife-server*, which simplifies the process of installing a Chef server and provides some other helpful capabilities.

infrastructure developer or sysadmin to configure and manage each instance of the server locally. If any data migrations are needed, or updates or patches required, these must be carried out. Additionally, ensuring the system scales in line with the infrastructure it supports is also the responsibility of the engineer(s) who elected to use OSC. Support is available from within the Chef community; Opscode does not directly support users of OSC.

Hosted Chef
: Hosted Chef (OHC) is a fully managed, multitenant, highly available version of a Chef Server that is hosted by Opscode. OHC is cloud-based, very scalable, supported 24/7/365. It includes enterprise features such as resource-based access control and, on account of its design, allows for multiple sandboxed servers to be run in one location. Functionally identical to OSC, Hosted Chef has the advantage of not needing any local setup or management.

Private Chef
: Private Chef (OPC) is effectively the same code base as OHC, delivered on-premise, to be run behind your firewall. Managed by the purchasing organization with support from Opscode, OPC is identical to OHC. Hosted Chef is the largest Private Chef deployment in the world.

Space does not permit a detailed discussion of setting up and running a local Chef server, however, Opscode provides Omnibus packages and a fully featured configuration toolkit. The documentation is excellent, and support from the community is equally good. For our examples, we're going to use Hosted Chef.

I've emphasized a number of times already—the Chef framework, at its core, is simply a REST API. Every single interaction with the Chef server is over HTTP using the API. This means that every time you interact with the Chef server you are using an API client. This includes the web interface, which is itself an API client. A Chef client running on a node we are managing is also an API client, as is the `Knife` command-line utility. The Chef Shell can also function as an API client. However, the need to secure API traffic is paramount, especially in a hosted, multitenant environment. For this reason, each API transaction is digitally signed, and each API client needs a valid identity in order to interact with the Chef server, and to authenticate using RSA public/private key pairs.

The authentication process is designed to ensure the API request has not been tampered with, is from the client claiming to make the request, and has arrived in reasonable time, not having been subjected to a replay attack. To achieve this, a string is compiled by combining four pieces of data to form a unique signature, and then encrypted with a private RSA key. This is decrypted on the server side and validated. The data used to form the signature includes the HTTP method, the timestamp, the API client ID, and the request body itself. This requires every API client to have its own public/private key pair.

Because Hosted Chef is multitenant, there needs to be a way to divide up API requests into meaningful groups. Hosted Chef uses the idea of *organizations* to achieve this. An organization is like a sandboxed Chef server and represents a way of grouping bits of infrastructure that you wish to manage using Chef. You can think of it as your own dedicated Chef server in the cloud. In Hosted Chef, when you read "organization," you can think "dedicated Chef server."

Each organization has its own private key. This key can be considered the master key; it is the key that enables other API clients to be granted keys. Sometimes called the *validation* key, it must be kept safe—without it, your ability to interact with Hosted Chef will be restricted. Although it can be regenerated from the web console, it still needs to be kept very secure, as it allows unlimited use of the platform, which could be very dangerous in the wrong hands.

Users of Hosted Chef also need an Opscode user account. An Opscode user account is shared across the Opscode Platform, the Hosted Chef Management Console, the community site, and Opscode's support page. This user also has a public/private key pair that is used to authenticate with the Chef server. Usually this interaction will use the `Knife` command-line tool; however, using that key, you can make direct API calls if you so desire. As an API client, `Knife` needs a configuration file: *knife.rb*. Amongst other settings, this specifies the URL of the API, and where to find the private key for the API requests.

As an infrastructure developer, you want to be able to build new machines using Chef. This means you need to be able to create new API clients for nodes you wish to configure, and key pairs for authentication. To do this, there is a special sort of API client called a *validation client*. This is used in the situation where an API client cannot yet make authenticated requests to the server because it lacks an identity and a key pair. This key is highly powerful and allows the creation of API clients.

Your Opscode user is associated with one or more organizations, allowing you to interact with the API either directly or via `Knife`. Similarly, the validation client is also tied directly to an organization.

To summarize, these five components are required to operate with Hosted Chef:

- An Opscode user, which grants access to the Hosted Chef Management Console
- An organization—effectively a sandboxed, dedicated Chef Server in the cloud
- A private key associated with your Opscode user and used by `Knife` to interact with the Chef server
- A validation client (and key) with the power to create API clients for an organization
- A `Knife` configuration file, ensuring you interact with the correct organization using the correct keys

We satisfied these requirements in our example by ensuring we had:

- Our Opscode user's private key
- Membership of an organization
- The validation key for the organization
- A *knife.rb* configuration file

As an infrastructure developer, the majority of your interaction with the Chef server is via the `Knife` command-line tool. Let's take a look at the *knife.rb* file that was generated and downloaded from the operations console:

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

We've already seen that most of Chef's configuration files are written in Ruby. This is no exception. Let's pick this file apart a little.

First we set the *current_dir* directory to the directory in which the *knife.rb* file resides. Then we set the log level and location; these can be safely left at their defaults. The *node_name* is a slightly confusing term, but in your `Knife` configuration this basically maps to your Opscode Username. We next set the path of the client key to be the same location as where we have our *knife.rb*. We also specify that the validation key is in the same place, and we explicitly name the validation client. The Chef Server URL is always the same—it's just *api.opscode.com* with the organization tacked on the end. Cache type and cache options again can be overlooked, and finally we tell `Knife` that our cookbooks are found in a directory called *cookbooks* in the directory above the location of our `Knife` config file and our keys. All this represents standard Ospcode convention, which can be met by ensuring the following are in place:

- A directory called *chef-repo*
- Another directory called *.chef* inside the *chef-repo* directory
- `Knife` config and keys located inside the *.chef* directory
- Be in your *chef-repo* directory when using `Knife`

This file, then, allows the *tdiexample* user to interact with the Chef API for the *hunterhayes* organization. Incidentally, the *tdiexample* user, being a global Opscode user, is also handy for a number of other interactions. It can be used to interact with other Chef users on the Opscode community portal, and also it is your mechanism for logging into the Hosted Chef operations console, which provides a useful web interface to your infrastructure.

A little more on the subject of organizations: organizations are a convenient way of grouping together related systems that are going to be managed using Chef. In actual fact, a system cannot be managed unless it belongs to an organization, and an Opscode user cannot do anything meaningful without also being associated with an organization. Users can belong to more than one organization, and can be invited to join the organizations belonging to other users. As each organization has a private key associated with it, `knife` needs to be configured on a per organization basis. At some stage, you may find you need to work with many organizations. In that case, something akin to the following *knife.rb* may be a convenient solution:

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

This allows you to keep all Chef-related keys in a *.chef* directory in the home directory. This has the added benefit of preventing the accidental checking-in of user keys into Git! All that is required to use `knife` is to export the `ORGNAME` and `OPSCODE_USER` environment variables in your shell, and then to be the username you used to sign up for the Opscode community pages. For example:

```
$ export ORGNAME=hunterhayes
$ export OPSCODE_USER=tdiexample
```

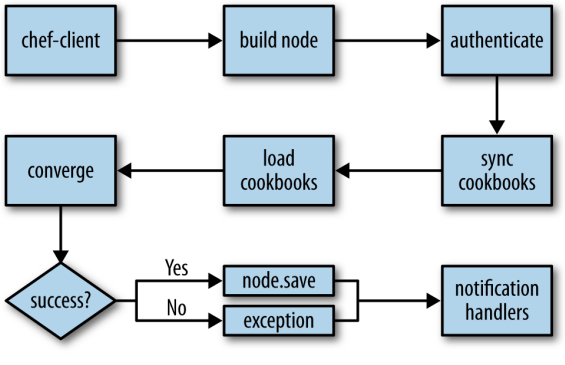
With the keys and `Knife` configuration file in place, we can now test that we can successfully speak to the Chef server. The simplest approach is to ask the Chef server which API clients it knows about. If `chef-client` has not been run on any servers, the only client it would know about is the so-called *validation client*. Since by now we've run `chef-client` on our machine, we should also see our own machine in the client list. Consequently, running `knife client list` should yield an entry, matching the organization name you set up on Hosted Chef, and the name of your machine:

```
$ knife client list
hunterhayes-validator
romanesco
```

An important workflow difference between `chef-solo` and using a Chef server is that when using a Chef server, it's necessary to publish or *upload* cookbooks to the Chef server. Then, when Chef runs, the Chef server can solve dependencies and make available whatever cookbooks are needed. The `chef-client` then downloads the required cookbooks and converges the node. The process of uploading the cookbooks to the Chef server is achieved using `knife cookbook upload`. You'll have noticed in our example, the Chef server rejected the *chruby* cookbook, when the cookbooks upon which *chruby* depended were not on the Chef server. Later in the book, I'll introduce a workflow that removes these headaches, both in terms of downloading and uploading cookbooks, but for now the important concept to grasp is simply that dependencies exist between cookbooks, and all cookbooks in the dependency chain need to be on the Chef server. While the Chef server solves dependencies for the `chef-client` run, `Knife` does not; it is necessary for you to either solve the dependencies yourself (or with a tool, as we'll see later), or rely on error messages from the Chef server.

Let's quickly run through the steps that are followed when Chef is run on a node, and compare and contrast `chef-client` and `chef-solo`:

1. Build the node
2. Synchronize cookbooks
3. Compile the resource collection
4. Converge the node
5. Notify and handle exceptions



Remember, the node is a Ruby object that represents the machine we're configuring. It contains attributes and a run list. This object is rebuilt every time, merging input from the local machine (via *Ohai*, the system profiler that provides basic information about

the node), the Chef API (which contains the last known state of the node), and attributes and run lists from roles. In the case of `chef-solo`, since there is no API to speak to, information about the node must be passed directly to `chef-solo` in the form of JSON.

Cookbooks contain a range of data—recipes, attributes, and other supporting data and code. `chef-client` requests this data via an API call. The Chef server performs some complex dependency management and serves only those cookbooks that are required for the node in question. By contrast, `chef-solo` simply ingests every cookbook, either from the local filesystem or over HTTP.

The resource collection, which we mentioned in our introductory discussion, is simply a list of resources that will be used to configure the node. In addition to the results of each evaluated recipe (and strictly speaking before), supporting code and attributes are loaded. This step is the same for `chef-solo` and `chef-client`.

Once the resource collection has been compiled, the required actions are taken by the appropriate providers. `chef-client` then saves the node status back to the server, where it is indexed for search. `chef-solo` takes no such action by default, and although community projects exist to extend `chef-solo` in this direction, my feeling is that once you start wanting to use the full power of Chef to index nodes for search and provide an API-addressable source of information in this manner, it's time to bite the bullet and use the tool in the way it was fundamentally designed to be used.

Finally, once the run has completed, action is taken dependent upon whether the run was successful or not. Chef provides the ability to write and use custom reporting and exception handlers, allowing sophisticated reporting, analytics, and notification strategies to be developed. We'll cover this in a bit more detail later, as this capability opens up some very interesting opportunities for making and verifying assertions about the Chef run.

We can see these steps in the output of the Chef run:

```
Starting Chef Client, version 11.4.4
resolving cookbooks for run list: ["irc", "git", "chruby::system"]
Synchronizing Cookbooks:
  - runit
  - ruby_build
  - windows
  - irc
  - ark
  - yum
  - git
  - build-essential
  - chef_handler
  - dmg
  - chruby
```

We don't see the node being built at this log level. Had we run with `-l debug` we'd have seen output like this:

```
[2013-06-03T12:11:36+01:00] INFO: *** Chef 11.4.4 ***
[2013-06-03T12:11:36+01:00] DEBUG: Loading plugin os
[2013-06-03T12:11:36+01:00] DEBUG: Loading plugin kernel
[2013-06-03T12:11:36+01:00] DEBUG: Loading plugin ruby
[2013-06-03T12:11:36+01:00] DEBUG: Loading plugin languages
...
```

This is `ohai` profiling the system. After all the plug-ins finish, we'd see, among other things, lines like these:

```
[2013-06-03T12:11:36+01:00] DEBUG: Building node object for romanesco
[2013-06-03T12:11:37+01:00] DEBUG: Extracting run list from JSON attributes pro-
vided on command line
[2013-06-03T12:11:37+01:00] DEBUG: Applying attributes from json file
[2013-06-03T12:11:37+01:00] DEBUG: Platform is ubuntu version 13.04
```

Returning to the output from our non-debug `chef-client` run, we see:

```
Compiling Cookbooks...
Converging 22 resources
```

We then see, for each recipe, the resources, and what was done. For example:

```
Recipe: irc::default
  * user[tdi] action create (up to date)
  * package[irssi] action install (up to date)
  * directory[/home/tdi/.irssi] action create (up to date)
  * cookbook_file[/home/tdi/.irssi/config] action create (up to date)
```

Here, Chef takes no action (idempotence); we've already applied the default irc recipe to the node, using `chef-solo`. The providers can see that the system is in the desired state, so `chef-client` does not need to do anything.

However, in the `Recipe: chruby::system` recipe, we see action being taken:

```
* package[build-essential] action install
    - install version 11.6ubuntu4 of package build-essential

* package[bison] action install
    - install version 2:2.5.dfsg-3ubuntu1 of package bison
...
* execute[ruby-build[1.9.3-p429]] action run
    - execute /usr/local/bin/ruby-build "1.9.3-p429" "/opt/rubies/1.9.3-p429"
```

We also need to the final step—handling reporting and exceptions—under debug mode to see the following:

```
[2013-06-03T12:32:07+01:00] INFO: Chef Run complete in 5.191436914 seconds
[2013-06-03T12:32:07+01:00] INFO: Running report handlers
[2013-06-03T12:32:07+01:00] INFO: Report handlers complete
```

The standard handlers are just to print to screen, but this is configurable to send email, alert via IRC or Hipchat, make a Nabaztag Rabbit's ear flap, or whatever you feel is appropriate!

The fundamental additions that are necessary to these steps when using a Chef server are those around authentication. New users tend to find this a little perplexing, but it's not actually that tricky to understand. I liken it to a scenario in which a group of people want to have a drink in a private members bar. I'm a member of such an establishment in Oxford. If I want to find somewhere quiet to sit down, have a drink, and read the newspaper, I can do so with ease. The authentication process looks like this:

```
Me: Good morning!
Doorkeeper: Good morning, sir, may I see your members' card?
Me: Certainly...<fx>presents membership card</fx>
Doorkeeper: Thank you very much, sir.
```

Now, suppose a friend of mine wants to meet me for coffee and a chat. The authentication process looks like this:

```
Friend: Good morning!
Doorkeeper: Good morning, sir, may I see your member's card?
Friend: I'm sorry, I'm not a member.
Doorkeeper: I'm sorry, sir, this is a members' only club.
Friend: Actually I'm meeting a friend here. I believe you have a guest policy?
Doorkeeper: That's correct, sir.  May I take your name?
Friend: George Romney.
Doorkeeper: Very good, sir.  And the member you are meeting?
Friend: Stephen Nelson-Smith.
Doorkeeper: Please wait a moment, sir.
Doorkeeper (to me): Sir, do you know a gentleman by the name of George Romney?
Me: Absolutely, I'm meeting him for coffee.
Doorkeeper (to friend): Come with me, please, sir.
```

Now, my friend might like the club so much, that he decides to join. In which case, I can recommend him, he can fill out the appropriate forms, pay his membership fee, and join the club. Thereafter if he wants to spend time in the club, the authentication process looks like this:

```
George: Good morning!
Doorkeeper: Good morning, sir, may I see your member's card?
George: Certainly...<fx>presents membership card</fx>
Doorkeeper: Thank you very much, sir.
```

The final option, of course, looks like this:

```
Chancer: Hello!
Doorkeeper: Good morning, sir, may I see your member's card?
Chancer: Oh, I'm sorry, I forgot it...
Doorkeeper: I'm sorry, sir, without your membership card, I can't permit you to
enter.
Chancer: Oh...but I know...umm...John Smith!
Doorkeeper (consults records): I'm sorry, I don't have a record of John Smith,
```

```
sir.
Chancer: Umm...I know...George Romney!
Doorkeeper: Please wait a moment, sir.
Doorkeeper (to George): Sir, do you know a gentleman by the name of Chancer?
George: No! Never heard of him!
Doorkeeper (to Chancer): I'm sorry, sir, we can't help you. Have a splendid day.
```

This process is very similar to the process that happens when `chef-client` authenticates against the Chef server. For a machine that is an existing API client and has a client key, the discussion looks like this:

```
Node: Hello Chef server, I'd like to use your API, please.
Server: Do you have a private key?
Node: I do! Here it is!
Server: Great, let me just use that to sign your request, and we'll be converg-
ing in no time!
```

In the case of a brand new node, which we wish to set up to speak to a Chef server, the discussion looks like this:

```
Node: Hello Chef server, I'd like to use your API, please.
Server: Do you have a private key?
Node: I'm sorry, not yet.
Server: OK...do you have an organization's validation key?
Node: I do! Here it is!
Server: Excellent, bear with me one moment while I create a key for you. OK,
here's your client key for future reference.  Let's get converging!
```

The final case looks like this:

```
Node: Hello Chef server, I'd like to use your API, please.
Server: Do you have a private key?
Node: I'm sorry, not yet.
Server: OK...do you have an organization's validation key?
Node: I'm sorry, I don't.
Server: Then I'm afraid I can't help you.
```

We can see this transaction in the debug log, too. If we run Chef again, we'll see the client key has been created and is used to sign requests:

```
[2013-06-03T12:11:36+01:00] DEBUG: Client key /etc/chef/client.pem is present -
skipping registration
[2013-06-03T12:11:36+01:00] DEBUG: Building node object for romanesco
[2013-06-03T12:11:36+01:00] DEBUG: Signing the request as romanesco
```

If I install and run Chef on a completely new machine, we see:

```
Creating a new client identity for ip-10-35-147-80.eu-west-1.compute.internal
using the validator key.
[2013-06-03T11:46:53+00:00] INFO: Client key /etc/chef/client.pem is not
present - registering

================================================================================
Chef encountered an error attempting to create the client "ip-10-35-147-80.eu-
```

```
west-1.compute.internal"
================================================================================
```

When I make the *client.rb* file available, but not the *validation.pem*, we see:

```
[2013-06-03T11:49:18+00:00] INFO: Client key /etc/chef/client.pem is not
present - registering
[2013-06-03T11:49:18+00:00] WARN: Failed to read the private key /etc/chef/vali-
dation.pem: #<Errno::ENOENT: No such file or directory - /etc/chef/valida-
tion.pem>
[2013-06-03T11:49:18+00:00] FATAL: Chef::Exceptions::PrivateKeyMissing: I can-
not read /etc/chef/validation.pem, which you told me to use to sign requests!
```

And when I make both the *client.rb* and *validation.pem* files available we see:

```
[2013-06-03T11:51:30+00:00] INFO: Client key /etc/chef/client.pem is not
present - registering
[2013-06-03T11:51:30+00:00] DEBUG: Signing the request as hunterhayes-validator
...
[2013-06-03T11:51:32+00:00] DEBUG: Signing the request as ip-10-35-147-80.eu-
west-1.compute.internal
```

The one final aspect that is different with Chef server is that upon successful completion of a Chef run, the node object is saved on the Chef server, recording the state of the machine and its attributes, indexing them for search. We can search for data using `knife search`:

```
$ knife search node 'platform:ubuntu'
2 items found

Node Name:   carrot
Environment: _default
FQDN:        ip-10-228-118-28.eu-west-1.compute.internal
IP:          54.246.56.172
Run List:    role[developer]
Roles:       developer
Recipes:     irc, git, chruby::system
Platform:    ubuntu 13.04
Tags:

Node Name:   romanesco
Environment: _default
FQDN:        romanesco
IP:          192.168.26.2
Run List:    recipe[developer]
Roles:
Recipes:     developer
Platform:    ubuntu 13.04
Tags:
```

A full discussion of the search facilities of Chef is outside the scope of this book. Refer to the Chef documentation for further examples and explanation.

The attributes system in Chef is one of the most complex facets of the Chef framework. First, a quick recap: an attribute is that which inherently belongs to and can be predicated of anything. They describe the detail of a machine we're configuring and have three underlying purposes: they can be used to indicate the current state of a node; they can be used to store the state of the node when Chef last ran and the node object was saved; and they can be used to specify desired state—the state the machine should be in after Chef runs.

Digging a little deeper, attributes have a type, corresponding to the source of the data. We can derive attributes from five places:

- The node itself (via `ohai`, or by `knife node edit`)
- Attribute files in a cookbook
- Recipes in a cookbook
- Roles
- Environments

Additionally, in each of these five places, there are up to six types of attributes that can be set. When Chef runs, all these sources and types are merged together, and Chef calculates what the definitive state of the node attribute list should be. At the end of the Chef run, this is saved and indexed for search.

The result is a rather complex matrix of precedence. The rationale for this lies in the philosophical position of the creators of Chef. The underpinning view is that the tool should provide power and flexibility. Chef provides the framework and the primitives. The infrastructure developer is the expert; they are in possession of domain knowledge, and understand deeply the various unique ways in which the configuration of the systems they manage relate to one another. All Chef needs to know is the desired state, how to achieve it, and what the functionality of that intended state should be, once achieved. The cost of this flexible philosophy is—at times—a complex implementation lurking beneath the surface. Thankfully, the design of Chef is such that for the vast majority of cases, you need never know about or use the hidden depths of flexibility, and can thrive on a few simple rules.

For the gory details, please see the Opscode documentation. However, the general rules are as follows:

- Set sane defaults in your cookbook attribute files, using the `default` method:

```
default['apache']['dir'] = '/etc/apache2'
```

- Overwrite the sane defaults either on a per role basis, using the `default_attributes` method:

```
default_attributes({ "apache" => {"dir" => "/etc/apache2"}})
```

- Or overwrite the sane defaults within a so-called wrapper cookbook, either in a recipe with the `node.default` method or in an attribute file with the `normal` method:

```
node.default["apache"]["dir"] = "/etc/apache2"

normal["apache"]["dir"] = "/etc/apache2"
```

- If you need to set an attribute on the basis of a calculation or expression in a recipe, use the `node.override!` method:

```
node.override!["something"]["calculated"] = some_ruby_expression
```

These rules of thumb will serve you more than 80% of the time. By the time you realize you need something more flexible, you'll have enough experience and understanding to work out the right approach from the documentation.

This has been a pretty content-heavy discussion. I recommend you read over the example again and digest the information presented in this section. Take a coffee break—go on, you deserve it!

Exercise 2: Virtualbox

So far the infrastructure we've built has provided the following:

- An installation of the various Chef client tools and commands
- An unprivileged *tdi* user
- The Git source code management system
- A Git repository containing a mixture of community and hand-built cookbooks
- An IRC client, preconfigured to allow you to ask for help in any of the main channels
- A modern version of Ruby

As well as providing a useful set of tools for future work, building this infrastructure has allowed us to cover many of the fundamentals of Chef. We're now going to put in place the final pieces that will allow us to iterate more quickly on cookbook development using local virtualization.

If you've been unable to follow the examples up to this point, as long as you have installed Chef, you should be able to get started here, as we're going to be using community cookbooks for both VirtualBox and Vagrant, both of which support Windows and OSX.

Objectives

Upon completing this exercise you will have:

- VirtualBox installed on your local machine
- Familiarity with using Lightweight Resources and Providers (LWRPs)
- An understanding of how to structure resource declarations for multiplatform support

Directions

1. Install the Chef Rubygem.
2. Download and extract the VirtualBox cookbook from the community site.
3. Solve any dependencies recursively and ensure all cookbooks are in your `chef-repo`.
4. Upload the new cookbooks to the Chef Server.
5. Open up the default recipe in the VirtualBox and look at the resources.
6. Update the *developer.rb* role and append the default VirtualBox recipe to the run list, and upload the role to the Chef server.
7. If you're on a Red Hat–derived system, ensure your kernel, kernel headers, and kernel devel packages are in sync.
8. Run `chef-client`.
9. Verify VirtualBox installed correctly by running `vboxmanage list vms`.

Worked example

I installed the Chef Ruby gem as follows:

```
$ gem install chef --no-ri --no-rdoc
Fetching: mixlib-config-1.1.2.gem (100%)
Fetching: mixlib-cli-1.3.0.gem (100%)
Fetching: mixlib-log-1.6.0.gem (100%)
Fetching: mixlib-authentication-1.3.0.gem (100%)
Fetching: mixlib-shellout-1.1.0.gem (100%)
Fetching: systemu-2.5.2.gem (100%)
Fetching: yajl-ruby-1.1.0.gem (100%)
Building native extensions.  This could take a while...
Fetching: ipaddress-0.8.0.gem (100%)
Fetching: ohai-6.16.0.gem (100%)
Fetching: mime-types-1.23.gem (100%)
```

```
Fetching: rest-client-1.6.7.gem (100%)
Fetching: net-ssh-2.6.7.gem (100%)
Fetching: net-ssh-gateway-1.2.0.gem (100%)
Fetching: net-ssh-multi-1.1.gem (100%)
Fetching: highline-1.6.19.gem (100%)
Fetching: erubis-2.7.0.gem (100%)
Fetching: chef-11.4.4.gem (100%)
Successfully installed mixlib-config-1.1.2
Successfully installed mixlib-cli-1.3.0
Successfully installed mixlib-log-1.6.0
Successfully installed mixlib-authentication-1.3.0
Successfully installed mixlib-shellout-1.1.0
Successfully installed systemu-2.5.2
Successfully installed yajl-ruby-1.1.0
Successfully installed ipaddress-0.8.0
Successfully installed ohai-6.16.0
Successfully installed mime-types-1.23
Successfully installed rest-client-1.6.7
Successfully installed net-ssh-2.6.7
Successfully installed net-ssh-gateway-1.2.0
Successfully installed net-ssh-multi-1.1
Successfully installed highline-1.6.19
Successfully installed erubis-2.7.0
Successfully installed chef-11.4.4
17 gems installed
```

Downloading and extracting the VirtualBox cookbook was a straightforward matter of using the following:

```
$ cd
$ knife cookbook site download virtualbox
$ tar xzvf virtualbox*gz -C chef-repo/cookbooks
```

I checked the metadata, as previously, and identified that I needed the `apt` cookbook, so I obtained this, and uploaded the two cookbooks to the Chef server:

```
$ cd ~/chef-repo
$ knife cookbook site download apt
$ tar xzvf apt*gz -C cookbooks
$ knife cookbook upload {apt,virtualbox}
```

I opened the default recipe and looked at the resources, noting that this recipe included conditional logic, and new resources that we hadn't yet investigated.

I updated the `developer` role, adding the `virtualbox` recipe to the run list:

```
name "developer"
description "For Developer machines"
run_list(
  "recipe[irc]",
  "recipe[git]",
  "recipe[chruby::system]",
  "recipe[virtualbox]"
```

```
)

default_attributes(
  "chruby" => {
    "rubies" => {
      "1.9.3-p392" => false,
      "1.9.3-p429" => true
    },
    "default" => "1.9.3-p429"
  }
)
```

I uploaded the role:

```
$ knife role from file roles/developer.rb
```

On my CentOS machine, I ensured I was running the latest kernel, and installed the kernel-devel package to match the kernel:

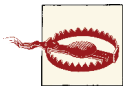
```
# yum -y update
# yum -y install kernel-devel
# uname -r
2.6.32-358.el6.x86_64
# rpm -q kernel-{devel,headers}
kernel-devel-2.6.32-358.11.1.el6.x86_64
kernel-headers-2.6.32-358.11.1.el6.x86_64
```

From previous experience, I opted to reboot the system, as I've found without doing so, the VirtualBox kernel modules don't install. When the system came back up, I ran `chef-client` and observed the resources taking action, and the repository and packages being set up accordingly. I verified that VirtualBox was operational using the `vboxmanage -version` and `vboxmanage list vms` command:

```
[root@centos ~]# VBoxManage -version
4.2.12r84980
[root@centos ~]# VBoxManage list vm
sns@ubuntu:~$ VBoxManage -version
4.2.12r84980
sns@ubuntu:~$ VBoxManage list vms
```

I also checked that the `vboxdrv` service was running:

```
sns@ubuntu:~$ sudo service vboxdrv status
VirtualBox kernel modules (vboxdrv, vboxnetflt, vboxnetadp, vboxpci) are loaded.
[root@centos ~]# service vboxdrv status
VirtualBox kernel modules (vboxdrv, vboxnetflt, vboxnetadp, vboxpci) are loaded.
```



At the time of this writing, there's a bug introduced in VirtualBox 4.12.14, which breaks the import functionality. In practice, this means that Vagrant and VirtualBox 4.12.14 won't function together. My expectation is that by the time you read this, the bug will be fixed, and you'll get version 4.12.16 or some such, and everything will work. However, if it doesn't, you'll need to downgrade to 4.12.12. There isn't an easy way to do this in the current VirtualBox cookbook, so you'll probably need to do that manually. Hopefully this issue will be fixed by the time you read this, but I include this note by way of warning. For more details, see *https://www.virtualbox.org/ticket/11895* and *https://github.com/mitchellh/vagrant/issues/1850*.

Discussion

VirtualBox is a freely available virtualization tool, originally created by innotek GmbH, purchased by Sun Microsystems (before Oracle's purchase of Sun) and now maintained and developed by Oracle. Although not ideal for heavy workloads, it's very handy for testing systems. VirtualBox emulates PC-like hardware and allows various operating systems to be installed and tested alongside one another on one host operating system. We're installing it, as it's a simple and free virtualization backend to Vagrant, which we'll introduce in the next exercise.

The VirtualBox cookbook is pretty straightforward. It simply sets up the relevant Oracle package repository and then installs the VirtualBox package. The two noteworthy items are the way multiplatform support is implemented, and the use of lightweight resource providers in the default recipe.

If we look at the default recipe, we'll see some basic conditional logic in place:

```
case node['platform_family']
when 'mac_os_x'

  sha256sum = vbox_sha256sum(node['virtualbox']['url'])

  dmg_package 'VirtualBox' do
    source node['virtualbox']['url']
    checksum sha256sum
    type 'mpkg'
  end

when 'windows'

  sha256sum = vbox_sha256sum(node['virtualbox']['url'])
  win_pkg_version = node['virtualbox']['version']
    Chef::Log.debug("Inspecting  windows  package  version:  #{win_pkg_version.in-
spect}")

  windows_package "Oracle VM VirtualBox #{win_pkg_version}" do
```

```
    action :install
    source node['virtualbox']['url']
    checksum sha256sum
    installer_type :custom
    options "-s"
  end

when 'debian'

  apt_repository 'oracle-virtualbox' do
    uri 'http://download.virtualbox.org/virtualbox/debian'
    key 'http://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc'
    distribution node['lsb']['codename']
    components ['contrib']
  end

  package "virtualbox-#{node['virtualbox']['version']}"
  package 'dkms'

when 'rhel'

  yum_key 'oracle-virtualbox' do
    url 'http://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc'
    action :add
  end

  yum_repository 'oracle-virtualbox' do
    description 'Oracle Linux / RHEL / CentOS-$releasever / $basearch - Virtual-
Box'
    url 'http://download.virtualbox.org/virtualbox/rpm/el/$releasever/$basearch'
  end

  package "VirtualBox-#{node['virtualbox']['version']}"
end
```

Platform family is a convenient method that allows infrastructure developers to test whether the node under management matches one of the listed "families"—for example *rhel* or *debian*. This is then used to execute different resources based on the value.

Now if we look at the default attribute file, we'll see similar logic to set the correct URL for the package repositories from which the packages will be downloaded:

```
default['virtualbox']['url'] = ''
default['virtualbox']['version'] = ''

case node['platform_family']
when 'mac_os_x'
  default['virtualbox']['url'] = 'http://download.virtualbox.org/virtualbox/↵
4.2.8/VirtualBox-4.2.8-83876-OSX.dmg'
when 'windows'
  default['virtualbox']['url'] = 'http://download.virtualbox.org/virtualbox/↵
4.2.8/VirtualBox-4.2.8-83876-Win.exe'
```

```
    default['virtualbox']['version'] = Vbox::Helpers.vbox_version↵
  (node['virtualbox']['url'])
  when 'debian', 'rhel'
    default['virtualbox']['version'] = '4.2'
  end
```

Within these conditional blocks, the resources make use of platform-specific providers—`apt_repository`, `windows_package`, `yum_repository`, and so on. These are examples of Lightweight Resource Providers (LWRPs).

If we think about the way Chef operates at its core, it breaks down to resources and providers. Every yin has its yang, and every resource has its provider. Like any great two-person team—Watson and Holmes, Cagney and Lacey, Bostridge and Drake—one would be ineffective without the other. Behind the scenes of every resource, there is Ruby code in the core Chef libraries, which knows how to take the actions we specified. Not only that, it knows how to take those actions on any platform. It knows how to create users on Windows, Solaris, FreeBSD, and Linux. It knows how to install packages on distributions like Debian, CentOS, Gentoo, and Suse. It also knows how to check if the action has already been taken, how to verify whether the node is already in the desired state. However, there are only a few dozen resources and providers built into Chef. Not infrequently, there comes a time when we want to abstract a repeated pattern of behavior with a declarative interface, but find that no Chef resource exists for this. Sometimes this happens when we realize we're making the same set of calls to resources, and we'd like to tidy them up. Sometimes we might need to call specialist library code to perform an action, but we'd like to address this in the recipe DSL. There are a large number of these use cases dotted throughout the community and Opscode cookbooks.

I remember many years ago, as a keen Puppet user, I wanted to be able to manage some Solaris machines that used *pkgsrc* as the main package management system. I understood I would need to create a *provider* for this, but the process was very difficult for me at the time. I needed to understand how the internals of Puppet functioned, and then I'd have had to monkey-patch Puppet, or submit pull requests, and wait for my changes to be accepted and then released. Really all I wanted to do was run `pkg-add` with a few arguments. I gave up.

Chef provides a DSL for building resources and providers, with the aim of making it easy to extend Chef with custom resources and providers, or to chain existing resources and providers together to carry out a given task. There isn't scope in the present work to cover the writing of LWRPs, and the examples used here—especially the `yum` or `apt` examples—are probably more complex than I'd like at this stage. However, you'll come across these in community cookbooks, and soon enough you'll want to write your own.

Exercise 3: Vagrant

VirtualBox is a powerful, easy-to-use, and flexible desktop virtualization solution. However, initial setup and ongoing maintenance of virtual machines (VMs) is rather a pain. *Vagrant* takes that pain away by providing a convenient command-line wrapper around creating and managing virtual machines. The Vagrant documentation provides a good summary of what Vagrant provides, and how it works:

> Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize the productivity and flexibility of you and your team.
>
> To achieve its magic, Vagrant stands on the shoulders of giants. Machines are provisioned on top of VirtualBox, VMware, AWS, or any other provider. Then, industry standard provisioning tools such as shell scripts, Chef, or Puppet can be used to automatically install and configure software on the machine.
>
> — *http://docs.vagrantup.com/v2/why-vagrant/*

Objectives

Upon completing this section, you will have:

- Vagrant installed on your local machine
- A CentOS *basebox* downloaded and available
- An understanding of how to start, stop, and interact with Vagrant boxes
- An understanding of the Vagrant plug-in architecture
- Installed the `vagrant-omnibus` plug-in
- Used `vagrant ssh` to connect to a machine
- Become familiar with the `Vagrantfile`, which configures the behavior of Vagrant
- Familiarity with the idea of a platform-based role

Directions

1. Download and extract the `vagrant` cookbook.
2. Browse to *http://downloads.vagrantup.com*, select the latest release, and then identify the URL to the download package.
3. Create a role for your platform family (e.g., *windows*, *rhel*, or *debian*).
4. Set the default[*vagrant*][*url*] to the URL of the appropriate download for your platform in your platform role.

5. Append the default `vagrant` recipe to the run list in the `developer` role, and prepend the platform role to the run list of your node.
6. Upload the roles and Vagrant cookbook to the Chef server.
7. Run `chef-client` on your machine.
8. Identify the URL of a CentOS base box for your architecture from GitHub.
9. Read the `vagrant box add` documentation.
10. Add a Vagrant box called `opscode-centos-6.4-yourarch`.
11. Read the `vagrant init` documentation.
12. Make a temporary directory, and initialize it for Vagrant use with the box you added.
13. Read the `vagrant up` documentation.
14. Launch the Vagrant box.
15. Read the `vagrant ssh` documentation.
16. Connect to the Vagrant machine, check the kernel and Chef version, then exit again.
17. Read the `vagrant plug-in` documentation.
18. Install the `omnibus-berkshelf` plug-in, read its documentation, and integrate it with Vagrant.
19. Read the `vagrant destroy` documentation.
20. Destroy and recreate the box, then connect, checking the kernel and Chef version again.

Worked Example

As the `tdi` user, I downloaded and extracted the Vagrant cookbook in the usual way:

```
$ cd
$ knife cookbook site download vagrant
$ tar xzvf vagrant*gz -C chef-repo/cookbooks
```

I checked on the Vagrant downloads page and selected version 1.2.2. I noted the packages for both *RPM* and *.deb* packages.

I created a role for the Ubuntu machine as follows:

```
name "debian"
description "Attributes specific to the Debian platform family"
run_list(
)

default_attributes(
  "vagrant" => {
    "url" => "http://files.vagrantup.com/packages↵
```

```
/7e400d00a3c5a0fdf2809c8b5001a035415a607b/vagrant_1.2.2_x86_64.deb"
  }
)
```

I created a role for the CentOS machine as follows:

```
$ cat roles/rhel.rb
name "rhel"
description "Attributes specific to the RHEL platform family"
run_list(
)

default_attributes(
  "vagrant" => {
    "url" => "http://files.vagrantup.com/packages/↵
7e400d00a3c5a0fdf2809c8b5001a035415a607b/vagrant_1.2.2_i686.rpm"
  }
)
```

I altered the developer role to be as follows:

```
$ knife role show developer
chef_type:           role
default_attributes:
  chruby:
    default: 1.9.3-p429
    rubies:
      1.9.3-p392: false
      1.9.3-p429: true
description:         For Developer machines
env_run_lists:
json_class:          Chef::Role
name:                developer
override_attributes:
run_list:
  recipe[irc]
  recipe[git]
  recipe[chruby::system]
  recipe[virtualbox]
  recipe[vagrant]
```

I edited the run list of the machine to appear as follows:

```
$ knife node show ubuntu -r
tk00.cheftraining.eu:
  run_list:
    role[debian]
    role[developer]

$ knife node show centos -r
tk01:
  run_list:
    role[rhel]
    role[developer
```

I uploaded the roles and the cookbook:

```
$ knife role from file roles/{debian,developer,rhel}.rb
Updated Role debian!
Updated Role developer!
Updated Role rhel!

$ knife cookbook upload vagrant
Uploading vagrant        [0.2.0]
Uploaded 1 cookbook.
```

I ran Chef and observed the relevant recipe being applied:

```
Recipe: vagrant::rhel
  * remote_file[/var/chef/cache/vagrant.rpm] action create
    - copy file downloaded from [] into /var/chef/cache/vagrant.rpm
        (file sizes exceed 10000000 bytes, diff output suppressed)

  * rpm_package[vagrant] action install
    - install version 1.2.2-1 of package vagrant

  * rpm_package[vagrant] action install (up to date)
Chef Client finished, 3 resources updated
```

I looked on the *Bento* page and selected a 64-bit box, and having read the `vagrant box add`, `vagrant init`, `vagrant up`, `vagrant ssh`, `vagrant plugin`, and `vagrant destroy` documentation, added a box as follows:

```
# vagrant box add opscode-centos-6.4-x86_64 https://opscode-vm.s3.amazonaws.com/
vagrant/opscode_centos-6.4_provisionerless.box
Downloading or copying the box...
Extracting box...te: 1537k/s, Estimated time remaining: 0:00:01)
Successfully added box 'opscode-centos-6.4-x86_64' with provider 'virtualbox'!
```

Next I made a temporary directory, and initialized it for use with Vagrant:

```
$ mkdir /tmp/vagrant-example
$ cd /tmp/vagrant-example
$ vagrant init opscode-centos-6.4-x86_64
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

I launched the machine:

```
# vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'opscode-centos-6.4-x86_64'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
```

```
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
```

And connected to it:

```
# vagrant ssh
Last login: Sat May 11 04:55:22 2013 from 10.0.2.2
[vagrant@localhost ~]$ uname -a
Linux localhost.localdomain 2.6.32-358.el6.x86_64 #1 SMP Fri Feb 22 00:31:26
UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
[vagrant@localhost ~]$
[vagrant@localhost ~]$ chef-client --version
-bash: chef-client: command not found
```

I installed the `vagrant-omnibus` plug-in:

```
# vagrant plugin install vagrant-omnibus
Installing the 'vagrant-omnibus' plugin. This can take a few minutes...
Installed the plugin 'vagrant-omnibus (1.0.2)'!
```

I edited the `Vagrantfile` and added the configuration directive to use the omnibus plug-in:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  # All Vagrant configuration is done here. The most common configuration
  # options are documented and commented below. For a complete reference,
  # please see the online documentation at vagrantup.com.

  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "opscode-centos-6.4-x86_64"
  config.omnibus.chef_version = :latest
...
...
```

I destroyed and recreated the machine, logged in, and verified that Chef had been installed:

```
# vagrant destroy
Are you sure you want to destroy the 'default' VM? [y/N] y
[default] Forcing shutdown of VM...
[default] Destroying VM and associated drives...
root@tk00:/tmp/example# vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'opscode-centos-6.4-x86_64'...
```

```
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Ensuring Chef is installed at requested version of 11.4.4.
[default] Chef 11.4.4 Omnibus package is not installed...installing now.
Downloading Chef 11.4.4 for el...
Installing Chef 11.4.4
warning: /tmp/tmp.PTLPHw62/chef-11.4.4.x86_64.rpm: Header V4 DSA/SHA1 Signa-
ture, key ID 83ef826a: NOKEY
Preparing...                ##################################################
chef                        ##################################################
Thank you for installing Chef!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
root@tk00:/tmp/example#
# vagrant ssh
Last login: Sat May 11 04:55:22 2013 from 10.0.2.2
[vagrant@localhost ~]$ chef-client --version
Chef: 11.4.4
```

Discussion

When it was introduced in 2010, Vagrant revolutionized the infrastructure development world. I remember recommending people take a look at it during my keynote at the second ever devopsdays conference in Hamburg, and sitting in on an open space session where a bunch of people started hacking on it. What does Vagrant do that's so awesome?

Vagrant is a tool for creating, managing, and distributing portable development environments. It enables complete machines to be automatically created, configures them repeatably, and allows the entire lifecycle to be managed from the command line or via an API. These machines (called boxes) can be shared with other team members and are portable; they can run on a wide range of platforms and allow a unified development and testing experience. It allows a user to go from nothing to a fully functioning local machine of pretty much any desired flavor, in one minute. As an infrastructure developer, this is an excellent boost to productivity and reliability. It tightens the feedback loop and allows machines to be rapidly destroyed and created, reducing the chance that one's cookbooks work because of historical side effects. It reduces the familiar cry of "It worked on my machine!" Every user, whether Linux, Windows, or Mac, can have a

machine of the same sort used in production, with the same cookbooks that are used in production.

Vagrant is well-documented, and its author, Mitchel Hashimoto, has just released his book, *Vagrant Up and Running* (O'Reilly).

Actually, we won't be using Vagrant directly very much in this book, as my recommended integration test harness actually wraps Vagrant (among other provisioning strategies), but it's a powerful and valuable tool, and I believe in understanding things from first principles, so it's worth understanding some of the fundamentals of Vagrant.

The Vagrant cookbook is nothing particularly interesting. It simply performs some platform-conditional logic, and downloads a package and installs it. It provides an LWRP for installing plug-ins, but we elected to install our plug-in manually to understand the concept.

In Chef terms, the interesting pattern we saw was that of the platform role. In a heterogenous environment, a common strategy is to build out roles of the sort:

Base
: Something that all machines get, regardless of platform or function

Platform
: Attributes or recipes that are specific to the OS (for example yum, Windows cookbooks, or URLs)

Service
: Something that describes a functional component, such as web server or database server

The Vagrant cookbook explicitly sets the URL from which to pull the package to `nil`. This is because there's no trivial way to work out what the path to the package will be—the path is made from the Git commit hash. Rather than have to maintain a complex attributes file, the cookbook maintainer has left setting the URL as an exercise for the user.

In my experience, this was a case of creating a `debian` and a `rhel` role, and setting the attribute there. Attributes in a role are at a higher precedence level than default attributes in a cookbook attributes file, and so the version from the role will take effect.

Vagrant, as a technology, is pretty easy to understand. The place to begin is the *Vagrantfile*. The Vagrantfile resides at the root of the directory of your project. Vagrant will build a virtual machine, but for what purpose? Not just because it can, but to test or demonstrate software. In Chef terms, it makes sense to keep a Vagrantfile within the cookbook to test the functionality of the cookbook. The Vagrantfile is a kind of manifest that describes how the Vagrant box you will be launching will behave. You can craft one manually, but Vagrant provides a generator in the form of the `vagrant init` command,

which will create one for you in the current directory. The Vagrantfile itself is heavily commented and pretty easy to navigate. If you need to do anything more complex or advanced, check out the documentation.

The simplest possible Vagrantfile would simply contain the following:

```
Vagrant.configure("2") { |config| config.vm.box = "opscode-centos-6.4-x86_64" }
```

This tells Vagrant to launch a machine based on the "opscode-centos-6.4-x86_64" box, with some default configuration. This must match a box on the system. The available boxes can be listed with the following:

```
# vagrant box list
opscode-centos-6.4-x86_64 (virtualbox)
```

Note that the *provider* is specified after the box name. Vagrant supports multiple alternative providers—VMware, EC2, Rackspace, LXC—we're currently using the (default) VirtualBox provider.

Vagrant boxes are the templates from which Vagrant constructs a VM. The format of a box is described in Vagrant docs, but broadly speaking, they're just archives of a specially prepared virtual machine for the provider required, together with a metadata file. We need to make Vagrant boxes available to Vagrant. Many Vagrant boxes are available on the Internet—some prepared and published by vendors, for example, Canonical or Opscode. Adding a Vagrant box is as simple as running the following:

```
vagrant box add name url
```

The name is how the machine will be referred to by the Vagrantfile or command line, and the URL is a remote or local path to the box itself, which you will need to download or create. We used the Opscode *Bento* boxes. Bento is a tool for automating the creation of VirtualBox–based Vagrant boxes, using defintions to work with Patrick Debois' Veewee utility. It tries to remain as close as possible to upstream vendor standards. You can read more at GitHub.

`Vagrant up` is the command that builds the local instance of a virtual machine.[3] It takes the template box and configures it using the Vagrantfile, and then launches the machine. The output explains the steps it goes through: it imports the machine, sorts out networking, ensures the system is clean, boots the machine, and sets up a shared directory. The two most noteworthy features are the networking and the shared directory. By default, Vagrant will use a VirtualBox configuration where the network interfaces on the virtual machine are running in *NAT* mode. That is, they are not externally routable. VirtualBox provides a port-forwarding service that allows the user to connect to the virtual machine from their local machine on a specified port; the connection will be

3. Vagrant does support alternative providers, for example, EC2 or Rackspace. Obviously in these cases, the machine being built will be remote.

forwarded to the port on the local machine. By default, Vagrant sets up a forwarder on *localhost:2222*, which connects to port 22 on the VM (i.e., it allows the user to connect to the virtual machine using ssh).

The `vagrant ssh` command uses a pre-prepared ssh key pair, which it stores in *~/.vagrant.d/insecure_private_key*. Running `vagrant ssh` will initiate a passwordless connection direct to the virtual machine, using the forwarded port.

The shared folder allows the running virtual machine to have access to the project directory in which the Vagrantfile exists. So, in the case of a cookbook, the virtual machine would be able to see the metadata, readme, recipes, templates, and so forth. By default, this will be available under */vagrant* on the local machine. We can demonstrate this by creating a file on the local system, watching it appear on the Vagrant box, and then touching a different file within the VM:

```
root@tk00:/tmp/example# ls -al
total 20
drwxr-xr-x 3 root root 4096 Jun  4 20:01 .
drwxrwxrwt 6 root root 4096 Jun  4 19:17 ..
-rw-r--r-- 1 root root    0 Jun  4 20:01 this-is-a-local-file
drwxr-xr-x 3 root root 4096 Jun  4 13:08 .vagrant
-rw-r--r-- 1 root root 4421 Jun  4 13:17 Vagrantfile

[vagrant@localhost ~]$ cd /vagrant/
[vagrant@localhost vagrant]$ ls
this-is-a-local-file  Vagrantfile

[vagrant@localhost vagrant]$ touch this-is-a-vm-file
[vagrant@localhost vagrant]$ ls -l
total 8
-rw-r--r-- 1 vagrant vagrant    0 Jun  4 19:01 this-is-a-local-file
-rw-r--r-- 1 vagrant vagrant    0 Jun  4 19:01 this-is-a-vm-file
-rw-r--r-- 1 vagrant vagrant 4421 Jun  4 12:17 Vagrantfile

root@tk00:/tmp/example# ls -l
total 8
-rw-r--r-- 1 root root    0 Jun  4 20:01 this-is-a-local-file
-rw-r--r-- 1 root root    0 Jun  4 20:01 this-is-a-vm-file
-rw-r--r-- 1 root root 4421 Jun  4 13:17 Vagrantfile
```

Vagrant is designed from the ground up to be extensible and pluggable. Much of the core functionality of Vagrant is implemented using plug-ins, and there is a large range of external plug-ins available. Rubygems lists over 100 gems beginning with "vagrant-". All of these can be installed using Vagrant's `vagrant plugin install` command. The plug-in we installed works with Vagrant boxes that do not have Chef installed, and adds a hook to `vagrant up` to install Chef using the omnibus package, just as we did in "Exercise 1: Install Chef" on page 47. This helps keep the Vagrant box slim and as close to upstream as possible, and does not require a fleet of Vagrant boxes to be created with every Chef patch release.

The final command we used was `vagrant destroy`. This simply powers off the virtual machine and deletes all traces of it. The idea is to return the host system to a clean state.

Conclusion

The objective of this and the previous chapter was to give you a hands-on, from-first-principles introduction to the fundamentals of Chef. We have covered:

- Installing Chef
- The idea of resources
- The recipe DSL
- Some common resources—package, user, file
- The idea of roles
- The node object, node attributes, and node attribute precedence
- The roles primitive
- Use of Chef Server and Chef Solo (and apply)
- The architecture of the Chef server
- The components of a Chef run
- Getting started with Opscode's Hosted Chef Service

In the process, we have introduced the following resources for additional documentation and support:

- The in-line documentation shipped with Chef
- *http://docs.opscode.com*
- *http://wiki.opscode.com*
- The #chef, #chef-hacking, #learnchef, and ##tdi IRC channels

Hopefully, if you've followed the examples as I intended, you've developed the habit of reading (or at least skimming) documentation and helping yourself. Of course we've been able to skim only the surface of the Chef framework, but my hope is that the present and previous chapters have given you a solid grounding in the fundamentals of Chef. As we work through the book, further aspects of Chef will be introduced, including Chef environments, the use of templates and service notifications, as well as enhanced workflow models to make your life as an infrastructure developer more effective.

Regardless of what else we learn, the infrastructure we've built in this series of exercises has laid the foundation for our future work; we have a modern Ruby, we have VirtualBox

and Vagrant set up and installed, and we have a configured IRC client should we need online help.

In the next chapter, we'll turn to Ruby and some of the core Ruby testing ideas, before moving on to discuss the ideas of test-driven and behavior-driven development.

CHAPTER 5

An Introduction to Test- and Behavior-Driven Development

The Principles of TDD and BDD

In Chapter 1, I argued that, to mitigate against the risks of adopting the infrastructure as code paradigm, systems should be in place to ensure that our code produces the environment needed, and to ensure that our changes have not caused side effects that alter other aspects of the infrastructure.

What we're describing here is automated testing. In his book *Managing Software Debt: Building for Inevitable Change* (Addison-Wesley), Chris Sterling uses the phrase "a supportable structure for imminent change" to describe what I am calling for. Particularly as infrastructure developers, we have to expect our systems to be in a state of flux. We may need to add components to our systems, refine the architecture, tweak the configuration, or resolve issues with its current implementation. When making these changes using Chef, we're effectively doing exactly what a traditional software developer does in response to a bug or feature request. As complexity and size grow, it becomes increasingly important to have safe ways to support change. The approach I'm recommending has its roots firmly in the historic evolution of best practices in the software development world.

A Very Brief History of Agile Software Development

By the end of the 1990s, the software industry did not enjoy a particularly good reputation—across four critical areas, customers were feeling let down. Firstly, the perception (and expectation, and experience) was often that software would be delivered late and over budget. Secondly, despite a lengthy cycle of requirement gathering, analysis, design, implementation, testing, and deployment, it was not uncommon for the customer to discover that this late, expensive software didn't really do what was needed.

Whether this was due to a failure in initial requirement-gathering or a shift in needs over the lifecycle of the software's development wasn't really the point—the software didn't fully meet the customer's requirements. Thirdly, a frequent complaint was that, once live and a part of the critical business processes, the software itself was unstable or slow. Software that fails under load or crashes every few hours is of negligible value, regardless of whether it has been delivered on budget, on time, and meeting the functional requirements. Finally, ongoing maintenance of the software was very costly. An analysis of this led to a recognition that the later in the software lifecycle that problems were identified or new requirements emerged, the more expensive they were to service.

In 2001, a small group of professionals got together to try to tackle some tough questions about why the software industry was so frequently characterized by failed projects and an inability to deliver quality code, on time and in budget. Together they put gathered a set of ideas that began to revolutionize the software development industry. Thus began the Agile movement. Its history and implementations are outside the scope of this book, but the key point is that more than a decade ago, professional developers started to put into practice approaches to tackle the seemingly inherent problems of the business of writing software.

Now, I'm not suggesting that the state of infrastructure code today is as bad as the software industry in the late 90s. However, if we're to deliver infrastructure code that is of high quality, easy to maintain, reliable, and delivers business value, I think it stands to reason that we must take care to learn from those who have already put mechanisms in place to help solve some of the problems we're facing today.

Test-Driven Development

Out of the Agile movement emerged a number of core practices that were felt to be important to guarantee not only quality software but also an enjoyable working experience for developers. Ron Jeffries summarizes these excellently in his article introducing Extreme Programming, one of a family of Agile approaches that emerged in the early 2000s. Some of these practices can be introduced as good habits, and don't require much technology to support their implementation. Of this family, the practice most crucial for creating a supportable structure for imminent change, providing insurance and warning against unwanted side effects, is that of test-driven development (TDD). For infrastructure developers, the practice is both the most difficult to introduce and implement, and also the one that promises the biggest return on investment.

TDD is a widely adopted way of working that facilitates the creation of highly reliable and maintainable code. The philosophy of TDD is encapsulated in the phrase *Red, Green, Refactor*. This is an iterative approach that follows these six steps:

1. Write a test based on requirements.
2. Run the test and watch it fail.

3. Write the simplest code you can to make the test pass.
4. Run the test and watch it pass.
5. Improve the code as required to make it perform well, be readable, and reusable, but without changing its behavior.
6. Repeat the cycle.

Kent Beck and Cynthia Andres, in *Extreme Programmng Explained* (Addison-Wesley), suggest this way of working brings benefits in four clear areas:

1. **It helps prevent scope from growing.** We write code only to make a failing test pass.
2. **It reveals design problems.** If the process of writing the test is laborious, that's a sign of a design issue; loosely coupled, highly cohesive code is easy to test.
3. **It builds trust.** The ongoing, iterative process of demonstrating clean, well-written code, with intent indicated by a suite of targeted, automated tests, builds trust with team members, managers, and stakeholders.
4. **It helps programmers get into a rhythm.** Test, code, refactor—a rhythm that is at once productive, sustainable, and enjoyable.

Behavior-Driven Development

However, in 2007, a group of Agile practitioners, including Dan North and Dave Astels, started rocking the boat with presentations and tool development work. Their key observation seemed to be that it's perfectly possible to write high quality, well-tested, reliable, and maintainable code, and miss the point altogether. As software developers, we are employed not to write code, but to help our customers to solve problems. In practice, the problems we solve pretty much always fit into one of three categories:

1. Help the customer make more money.
2. Help the customer spend less money.
3. Help the customer protect the money they already have.

Around this recognition grew up an evolution of TDD focused specifically around helping developers *write code that matters*. Just as TDD proved to be a hugely effective tool in enhancing the technical quality of software, behavior-driven development (BDD) set out to enhance the success with which software fulfilled the business' need.

The shift from TDD to BDD is subtle but significant. Instead of thinking in terms of verification of a unit of code, we think in terms of a specification of how that code should behave—what it should do. Our task is to write a specification of system behavior that is precise enough for it to be executed as code.

Importantly, BDD is about *conversations*. The whole point of BDD is to ensure that the real business objectives of stakeholders get met by the software we deliver. If stakeholders aren't involved, if discussions aren't taking place, BDD isn't happening. BDD yields benefits across many important areas.

Building the right thing

BDD helps to ensure that the right features are built and delivered the first time. By remembering the three categories of problems that we're typically trying to solve, and by beginning with the stakeholders—the people who are actually going to be using the software we write—we are able to clearly specify what the most important features are, and arrive at a definition of *done* that encapsulates the business driver for the software.

Reducing risk

BDD also reduces risk—the risk that, as developers, we'll go off at a tangent. If our focus is on making a test pass, and that test encapsulates the customer requirement in terms of the behavior of the end result, the likelihood that we'll get distracted or write something unnecessary is greatly reduced. Interestingly, a suite of acceptance tests developed this way, in partnership with the stakeholder, also forms an excellent starting point for monitoring the system throughout its lifecycle. We know how the system should behave, and if we can automate tests that prove the system is working according to specification, and put alerts around them (both in the development process so we capture defects, and when live so we can resolve and respond to service degradation), we have grounded our monitoring in the behavior of the application that the stakeholder has defined as being of paramount importance to the business.

Evolving design

It also helps us to think about the design of the system. The benefits of writing unit tests to increase confidence in our code are pretty obvious. Maturing to the point that we write these tests first helps us focus on writing only the code that is explicitly needed. The tests also serve as a map to the code and offer lightweight documentation. By tweaking our approach towards thinking about specifying behavior rather than testing classes and methods, we come to appreciate test-driven development as a practice that helps us discover how the system should work, and molds our thinking towards elegant solutions that meet the requirements.

How does all of this relate to infrastructure as code? Well, as infrastructure developers, we are providing the underlying systems that make it possible to deliver software effectively. This means our customers are often application developers or test and QA teams. Of course, our customers are also the end users of the software that runs on our systems, so we're responsible for ensuring our infrastructure performs well and remains available when needed. Having accepted that we need some kind of mechanism for testing our infrastructure to ensure it evolves rapidly without unwanted side effects, bringing the principle of BDD into the equation helps us to ensure that we're delivering

business value by providing the infrastructure that is actually needed. We can avoid wasting time pursuing the latest and greatest technology by realizing we could meet the requirements of the business more readily with a simpler and established solution.

TDD and BDD with Ruby

Ruby has always been a language in which testing, and particularly testing up-front, has been popular. Also, the development community around Ruby has historically been particularly positive about Agile software development in general, and as such has spawned a great many creative and powerful testing tools and frameworks. I think it's fair to say that as a language and environment in which to work, Ruby is probably the best served for libraries, tools, and frameworks. Within this ecosystem, I'm going to discuss three tools that, when used together, provide a full coverage of testing capabilities, from the lowest to the highest level—*Cucumber*, *RSpec*, and *Minitest*.

As this is a book about test-driven infrastructure development, I'm going to make sure we've got a reasonable understanding of testing in general and test-first development, before we go on to discuss writing infrastructure code using Chef.

For the purposes of the exercise, we're going to write a Ruby class that assesses whether a team member is a hipster. (I'm guessing everyone knows what a hipster is by now, but there's always Google if you don't!)

Minitest: Unit Testing for the 21st Century

A unit test is pretty much the simplest and lowest level kind of test we can write. It is designed to verify whether a precise, small, tightly defined piece of functionality behaves as it should. Typically, a unit test will exercise a single method. The seminal unit testing framework was *JUnit*. Conceived by Kent Beck and Erich Gamma, it built on *SUnit*, written by Kent Beck for Smalltalk. JUnit quickly became the standard approach to unit testing, to the extent that the term *xUnit* began to appear to describe a test framework in any language that broadly implemented the same approach to unit testing. Ruby's xUnit implementation was *Test::Unit*.

The pattern is pretty much always the same. You create a class as a subclass of `Test::Unit::TestCase`, write methods beginning with the word *test*, set up some state to exercise a method, and make an assertion about what the method should do.

We'll set the background with a little history lesson, and look at the original and most basic unit testing capabilities of Ruby—the faithful old workhorse `Test::Unit`.

First, ensure that the `test-unit` gem is installed:

```
$ gem install test-unit
```

Create a directory for the project and a file for a test:

```
$ mkdir tdd-principles $ cd tdd-principles $ touch test_hipster.rb
```

Now, let's write a very simple test using the traditional test/unit approach:

```
require "test/unit"

class HipsterTest < Test::Unit::TestCase

def setup @developer = HipsterAssessor.new(gears_on_bike=1) end

def test_has_fixie? assert_equal true, @developer.has_fixie? end

end
```

We're setting up the test by creating an instance of the HipsterAssessor, and passing in that the developer we are assessing has a single gear on their bicycle. We're going to test the `has\_fixie?` method, and we're setting up the expectation that the method will return true.

Let's run the test:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started E Finished in 0.000294
seconds.

1) Error: test_has_fixie?(HipsterTest): NameError: uninitialized constant Hip-
sterTest::HipsterAssessor test_hipster.rb:6:in `setup'

1 tests, 0 assertions, 0 failures, 1 errors
```

This is the standard approach for test-first programming. We've written the test. The test has failed. Now we make it pass. In this case the test wasn't able to run yet—it errored out because we're trying to instantiate a HipsterAssessor, but we've not written the code for that yet, nor is it available to the test. Let's fix that by creating a new file called *hipsterassessor.rb*, which contains the following:

```
class HipsterAssessor end
```

And let's require that file in our test by adding:

```
require './hipsterassessor'
```

Let's run the test again:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started E Finished in 0.000345
seconds.

1) Error: test_has_fixie?(HipsterTest): ArgumentError: wrong number of argu-
ments (1 for 0) test_hipster.rb:7:in `initialize' test_hipster.rb:7:in `new'
test_hipster.rb:7:in `setup'

1 tests, 0 assertions, 0 failures, 1 errors
```

Now the problem we have is that we've instantiated an assessor, but we've also passed in an argument to it, and our class definition doesn't accommodate this. Let's fix that by

adding an `initialize` method, which takes an argument. We're not going to do anything with the argument yet—at this point, we're concerned with getting to the point where we see the test fail, not return an error.

```
class HipsterAssessor

def initialize(bike_gears)

end

end
```

Run the test again:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started E Finished in 0.000322
seconds.

1) Error: test_has_fixie?(HipsterTest): NoMethodError: undefined method
`has_fixie?' for nil:NilClass test_hipster.rb:11:in `test_has_fixie?'

1 tests, 0 assertions, 0 failures, 1 errors
```

Right now the error is that we haven't written the `has_fixie?` method. Let's go ahead and write that, but without an implementation. Your *hipsterassessor.rb* should look like this now:

```
class HipsterAssessor

def initialize(bike_gears) end

def has_fixie? end

end
```

And running the test should now result in a failure, not an error:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started F Finished in 0.011201
seconds.

1) Failure: test_has_fixie?(HipsterTest) [test_hipster.rb:11]: <true> expected
but was <nil>.

1 tests, 1 assertions, 1 failures, 0 errors
```

Alright, we're getting somewhere. The test expected the method to return true, but since we've not written the code yet, we got nil. Now let's write the actual code:

```
class HipsterAssessor

def initialize(bike_gears) @gears = bike_gears end

def has_fixie? @gears == 1 end

end
```

And finally, run the test and see it pass:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started . Finished in 0.000259
seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

So, that's an example of a simple unit test. Obviously these tests can get a lot more complicated, but other than adding methods that set up some state and ensure that state is no longer there at the end of the test, there's not much more to `Test::Unit`. The end result is that while `Test::Unit` is by far the most widely used test tool in the wild, it's almost never used by itself. Most Ruby projects will pull in a large number of additional Rubygems to provide more advanced testing capabilities—test randomization, allowing more natural test descriptions, and adding the ability to set up ephemeral test fixtures to allow complex, time-consuming, or third-party libraries or processes.

In Ruby 1.9, *Minitest* replaced `Test::Unit`, built into the standard library for Ruby 1.9. This clears out some of the old and rarely used cruft from `Test::Unit`, and brings powerful, modern testing functionality right into the standard library. An important thing to note about Minitest is that the version built into Ruby lags considerably behind the current latest version; indeed the version of Minitest built into my version of Ruby is 2.5.1. I would typically recommend you install the latest version from Rubygems, but at the time of writing, Minitest 5.0.0 has only just been released with a number of breaking changes. For this reason, in the present work, I recommend taking advantage of the so-called *PessimisticVersionConstraint*.

In the Gemfile, set your Minitest line to the following:

```
gem 'minitest', '~> 4.7'
```

This will ensure that you stay on the version below the major 5.0 breaking release.

The newer Minitest syntax is backwards-compatible with `Test::Unit`, but the superclass has a different name. Let's convert it:

```
require 'minitest/autorun' require_relative 'hipsterassessor'

class HipsterTest < MiniTest::Unit::TestCase

def setup @developer = HipsterAssessor.new(gears_on_bike=1) end

def test_has_fixie? assert_equal true, @developer.has_fixie? end

end

$ ruby test_hipster.rb Run options: --seed 37275

# Running tests:

.
```

```
Finished tests in 0.000873s, 1145.4754 tests/s, 1145.4754 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

You'll notice right away that the test is much faster. You might also notice the `--seed` run option. This is because Minitest runs your tests in a random order to prevent you getting into the situation where your tests pass because of so-called *state leakage*—i.e., some state remained from the previous test, which allowed the subsequent test to pass. Randomizing the order of the tests catches this. The seed is the number used to initialize a pseudorandom number generator, which provides the randomness upon which Minitest bases its decision about ordering. You can reproduce the same state by passing the same seed manually.

That concludes our whirlwind tour of traditional, backwards-compatible xUnit-style unit testing. Although you're fairly unlikely to test your infrastructure code using this traditional approach, it's valuable to have some familiarity with it, and the general approach of iterating on failing tests until they pass is the same regardless of the testing framework being used.

RSpec: The Transition to BDD

I mentioned earlier that while there is great value in traditional unit testing, it's still possible to write code that passes unit tests but doesn't deliver value to the customer. Unit tests assert that the code behaves as it should, but what asserts how the code should behave? In order to be sure that we're building code that matters, we need some kind of specification that describes what the code should do. This is exactly the transition that is made when we start to think about behavior-driven development against test-driven development. We first specify what the behavior should be, in a written form. We then test that the code behaves as specified (which will, of course, fail). We then make the tests pass, and check against the specification. A core principle of BDD is that this specification be code itself—that the description of how our software behaves should itself be executable.

RSpec was developed around a recognition that looking at low-level code, with not entirely obvious assertion syntax and class and method definitions, was not really the ideal vehicle for expressing and communicating the intended behavior of code. It was inspired by an early Thoughtworks tool, *Agiledox*, which converted code that looked like this:

```
public class CustomerLookupTest extends TestCase { testFindsCustomerById()
{ ... } testFailsForDuplicateCustomers() { ... } ... }
```

To a specification like this:

```
CustomerLookup - finds customer by id - fails for duplicate customers - ...
```

The effect is remarkable. Immediately the intention is clear, and the brain takes it in. RSpec's output looks similar, and its input is more palatable.

Let's write some specifications for the behavior of the HipsterAssessor.

First, let's install the RSpec gem and create a directory to contain our specifications, called *spec*.

```
$ gem install rspec $ mkdir spec
```

Inside the *spec* directory, create a file called *hipster_assessor_spec.rb* with the following contents:

```
require 'rspec' require_relative '../hipsterassessor'

describe HipsterAssessor do context "assessing whether a developer is a hip-
ster" do it "can establish if the developer has a fixed-wheel bicycle" do devel-
oper  =  HipsterAssessor.new(gears_on_bike=1)  expect(developer.has_fixie?).to
be_true end end end
```

The first line simply makes the RSpec gem available, and the second line makes our `HipsterAssessor` class available. The describe block describes in a high level domain-specific language (DSL) what the class should do, and in what context it functions. If you read the code as English, it makes pretty easy reading:

You: Describe the HipsterAssessor!

Me: In the context of assessing whether a developer is a hipster, it can establish if the developer has a fixed-wheel bicycle.

Let's run the test:

```
$ rspec -fd spec/

HipsterAssessor assessing whether a developer is a hipster can establish if the
developer has a fixed-wheel bicycle

Finished in 0.00048 seconds 1 example, 0 failures
```

This is much closer to describing the behavior of the code than just testing a method.

Let's add another feature. I think the HipsterAssessor should give the developer a hipster score. To this end, it would be good to see the score and set the score.

I'm going to add the following:

```
it "reports a hipster assessment score" do developer = HipsterAsses-
sor.new(gears_on_bike=1) expect(developer.score).to be_kind_of(Numeric) end
```

This gives us the following spec:

```
require 'rspec' require_relative '../hipsterassessor'

describe HipsterAssessor do context "assessing whether a developer is a hip-
ster" do it "can establish if the developer has a fixed-wheel bicycle" do devel-
oper = HipsterAssessor.new(gears_on_bike=1) developer.has_fixie?.should == true
end

it "reports a hipster assessment score" do developer = HipsterAsses-
sor.new(gears_on_bike=1) expect(developer.score).to be_kind_of(Numeric) end end
end
```

Rather than running the whole spec each time, during development it's useful to use the `-e`, `--example` argument, which will only run the examples that match a given string:

```
$ rspec -fd -e "score" spec/ Run options: include {:full_description=>/score/}

HipsterAssessor assessing whether a developer is a hipster reports a hipster as-
sessment score (FAILED - 1)

Failures:

1) HipsterAssessor assessing whether a developer is a hipster reports a hipster
assessment score Failure/Error: expect(developer.score).to be_kind_of(Numeric)
NoMethodError: undefined method `score' for #<HipsterAssessor:0x000000020bb128
@gears=1> # ./spec/hipster_assessor_spec.rb:13:in `block (3 levels) in <top (re-
quired)>'

Finished in 0.00052 seconds 1 example, 1 failure

Failed examples:

rspec ./spec/hipster_assessor_spec.rb:11 # HipsterAssessor assessing whether a
developer is a hipster reports a hipster assessment score
```

The process should be familiar now. We need to write the code to make the test pass. We can make the test pass trivially simply by adding:

```
def score 10 end
```

Our test now passes:

```
$ rspec -fd -e "score" spec/ Run options: include {:full_description=>/(?-
mix:score)/}

HipsterAssessor assessing whether a developer is a hipster reports a hipster as-
sessment score

Finished in 0.00147 seconds 1 example, 0 failures
```

This is fine and meets our specification. This might seem a bit silly—surely the developer won't always get a score of 10? Well, this is the point of BDD. We iterate quickly, and

drive out the requirements. What's wrong with score 10? Maybe it's that it's meaningless? Maybe it's that it never varies? In which case we need to specify what the code should do. An important concept here is to ask the question, "What's the next most important thing that the system does not currently do?" In our case, let's say we want the score to vary depending on criteria. For example, let's say that having a fixie scores five points, and then add another thing to test for, let's say, empty spectacle frames. So, we add:

```
it "awards five points for having a fixie" do developer = HipsterAsses-
sor.new(gears_on_bike=1) expect(developer.score).to eq 5 end
```

And run the test (again, using the `-e` argument), which shows:

```
$ rspec -fd -e "points" spec/ Run options: include {:full_description=>/points/}

HipsterAssessor assessing whether a developer is a hipster awards five points
for having a fixie (FAILED - 1)

Failures:

1) HipsterAssessor assessing whether a developer is a hipster awards five
points for having a fixie Failure/Error: expect(developer.score).to eq 5

expected: 5 got: 10

(compared using ==) # ./spec/hipster_assessor_spec.rb:18:in `block (3 levels)
in <top (required)>'

Finished in 0.00112 seconds 1 example, 1 failure

Failed examples:

rspec ./spec/hipster_assessor_spec.rb:16 # HipsterAssessor assessing whether a
developer is a hipster awards five points for having a fixie
```

Let's change the code to make it pass. This requires a few changes, so I'll now show the whole class to date:

```
class HipsterAssessor

def initialize(bike_gears) @gears = bike_gears @score = 0 end

def has_fixie? @gears == 1 end

def score if self.has_fixie? @score = @score + 5 end @score end end
```

Let's run all the tests now. Our full spec looks like this:

```
require 'rspec' require_relative '../hipsterassessor'

describe HipsterAssessor do context "assessing whether a developer is a hip-
ster" do it "can establish if the developer has a fixed-wheel bicycle" do devel-
oper = HipsterAssessor.new(gears_on_bike=1) expect(developer.has_fixie?).to
be_true end
```

```
it "reports a hipster assessment score" do developer = HipsterAsses-
sor.new(gears_on_bike=1) expect(developer.score).to be_kind_of(Numeric) end

it "awards five points for having a fixie" do developer = HipsterAsses-
sor.new(gears_on_bike=1) expect(developer.score).to eq 5 end

end

end
```

And if we run RSpec, we get:

```
$ rspec -fd spec/

HipsterAssessor assessing whether a developer is a hipster can establish if the
developer has a fixed-wheel bicycle reports a hipster assessment score awards
five points for having a fixie

Finished in 0.00161 seconds 3 examples, 0 failures
```

The eagle-eyed amongst you will probably have noticed that our score increasing method will continue to add five points every time. Adding a test for this, and refactoring the code is just the sort of thing that would happen in real life. Of course, additionally, specs can get much more complex than this, but you should now appreciate the difference between behavior-driven and test-driven development.

In the previous section, we looked at Minitest as a drop-in replacement for `Test::Unit`. In addition to the speed improvements and general leanness of the tool, another addition is the inclusion of a spec-like DSL, which brings BDD into the core library. Let's look at how we'd express the preceding test using Minitest rather than RSpec.

Rewriting the tests also gives us a chance to refactor. I don't like that we have repeated instantiating the HipsterAssessor three times.

Both RSpec and Minitest support hooks to set up state before tests. This allows us to simplify the test. Here's the equivalent code for Minitest:

```
require 'minitest/autorun'

class HipsterTest < MiniTest::Unit::TestCase

describe HipsterAssessor do

before do @developer = HipsterAssessor.new(gears_on_bike=1) end

describe "when assessing whether a developer is a hipster" do

it "can establish if the developer has a fixed-wheel bicycle" do @develop-
er.has_fixie?.must_equal true end
```

```
it "can report a hipster assessment score" do @developer.score.must_be_in-
stance_of Fixnum end

it "can award five points for having a fixie" do @developer.score.must_equal 5
end

end

end

end
```

I've added a `before` block, which sets up the state before each test. This is a pretty common pattern, and one you'll see when we apply these principles to infrastructure code.

The syntax of Minitest is slightly different, and the matching and expectation grammar is not identical, but it's clear that at this level of simplicity, Minitest can do exactly what RSpec does. Not having to include another gem makes this an attractive option. However, RSpec is very widely used, and in the context of testing infrastructure, the Chef community hasn't yet settled on which it favors, so I've given a brief introduction to both. In terms of how this applies to testing infrastructure code, my feeling is that the community is equally undecided, and we'll cover both when we look at writing tests for Chef recipes later in the book.

Let's run the refactored test now, simply calling it with Ruby, now we're using Minitest:

```
$ ruby test_hipster.rb Loaded suite test_hipster Started ... Finished in
0.000808 seconds.

3 tests, 3 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 57543
```

Now that we've covered the basics of both RSpec and `MiniTest::Spec`, we'll move on to examine *Cucumber*.

Cucumber: Acceptance Testing for the Masses

When Dan North first started thinking about BDD back in 2003, the context was not one of replacing TDD with a different set of practices or tools, but rather about how to go about explaining the reasons for, and the underpinning ideas behind TDD itself. As we've seen in this contrived example, it seems to make sense to start with tests right at the level of the application. However, as thought around BDD began to mature, and more people started to explore the perspective it offered, so the focus of the tests started to move towards the stakeholders—those for whom the software was being built. We can see this starting to happen in our RSpec example, but it hasn't fully matured. The main thing missing is how to connect the stories—the self-contained units of work that

developers commit to in an agile project—to work that represents real value to the stakeholder. Somehow we need to be able to demonstrate that the code we're writing, and indeed testing, is applicable to the stories we've committed to delivering.

A useful template for capturing the story looks like this:

- In order to achieve some specific, measurable, definable goal
- As some kind of stake holder
- I want a feature

Moving to a BDD way of thinking brings many benefits. It helps to tease out how the software we write should behave, and it serves as an executable specification of what the software should do. This is undeniably a step in the right direction, but BDD-influenced thinkers wanted to take it a little further again. Using RSpec, or Minitest's spec capabilities, might answer questions about how it should behave, but it doesn't explain why. We never develop software in a vacuum. We rarely develop software just for fun. There's always a reason behind it—some kind of driving force behind the project. In order best to understand that, and be sure we're building the right features, for the right reasons, with the right priority, it's necessary to engage the stakeholders—the people for whom we're building the software.

An early attempt to connect these kinds of stories to RSpec was written by Dan North, but greatly improved and released by Aslak Hellesoy in 2008 as Cucumber. Cucumber takes the obvious benefits of test-first programming, and adds to it a whole series of further benefits. In his book, *The Cucumber Book*, Aslak Hellesoy and Matt Wynne (Pragmatic Bookshelf), Aslak describes Cucumber as somewhat akin to a cheerful and friendly but rather nerdy team member, with a terrifyingly precise recollection of what it is the team is building and why, and who doesn't mind the grunt work of repeatedly checking that what the team is working on is the right thing, running tests, and reporting back.

The key concept we're exploring here is that software—and of course infrastructure—begins with an idea. Usually the idea is tied in some way to making something that can be sold, used to reduce cost, improve efficiency, or add enjoyment—whatever it is, there's almost always an idea—a germ of an idea at the genesis of a software or infrastructure project. The point is that unless the person who has the idea is an incredibly gifted person, it's unlikely that they'll be able to build the idea themselves, from scratch, without getting some help. As soon as you introduce help, especially if it's technical help, you introduce the requirement to communicate. Even in an experienced agile team, with short iterations and a fast feedback cycle, it's possible to spend a two week period of time working on the wrong thing, delivering something that the developers thought was right, but which somehow got confused, miscommunicated, or misunderstood. Cucumber offers a way to ease the communication and cooperation between people and teams.

At the heart of eXtreme programming is the idea of automated acceptance tests. An acceptance test is simply some code that we can run, which captures at its heart some aspect of the functionality of the system. The idea is that the developer and a stakeholder collaborate on writing this test together to capture requirements in code, which when it passes, forms some kind of seal of approval. These are distinct from the kind of unit tests we looked at previously. Unit tests are largely written by the developer and for the developer. They help emerge and validate design and protect against errors. Acceptance tests are written by the stakeholder and the developer, for the stakeholder and the developer. A commonly used expression is that the difference between unit tests and acceptance tests is that unit tests help you build the thing right, whereas acceptance tests help you build the right thing.

Despite the obvious benefits of automated acceptance tests, in practice even among experienced XP and TDD teams, it's rarely done, or done well. One of the reasons is that finding a stakeholder with the technical ability, interest, and patience to sit at a computer writing pure Ruby code, even a DSL like RSpec, is incredibly hard. I remember working on an accounts package in PHP and pairing with a product manager, and actually writing *SimpleTest* acceptance tests. It worked really well, but I've never found a stakeholder since who is comfortable with that kind of technical involvement.

Cucumber helps to make automated acceptance testing a reality. If we think about what an acceptance test is, it's really just an example. We're saying we need this feature for this purpose. Here are a few examples of how the system would behave if we'd implemented the feature I want. If you can prove to me that these examples do what I've asked, then I'll be happy that the requirement is met. The challenge in making this happen is that in most cases, the areas of expertise of the stakeholder and the developer don't coincide. Often radically so. This is because each person is an expert in their own domain. I'm an expert at Chef, and a pretty competent Ruby and Python developer. I'm not an expert in social media advertising. The problem Cucumber sets out to solve is that of making it easy to find a shared language—a ubiquitous language—that everyone can use that describes what we're trying to build and why we're trying to build it. This language should neither be mired in the jargon of the developer, nor the person who had the idea in the first place.

Beyond making acceptance tests a reality, Cucumber also becomes documentation. Not documentation that slowly decays on a wiki—documentation that is an executable specification, that lives with and shapes the creation of the software. Documentation that can be shared, explored, grown, and that, ultimately, can be run from the command line, and should pass tests. This makes Cucumber potentially a very powerful source of truth and a barometer of health in a project. That's a pretty awesome state of affairs.

Let's look at how Cucumber works. At the highest level, it's just another command-line tool. It reads in plain text files called features, which contain scenarios that describe examples of use cases for the feature. The features and scenarios are written in what is

very close to natural language, but with a dozen or so grammar and syntax rules—a DSL called *Gherkin*. Each scenario is a sequence of steps that need to be carried out in order, setting up state, doing something, and then checking state again. These steps are then mapped onto Ruby code, which takes real action. These are called step definitions. Step definitions typically delegate to support code shipped with the test suite and call out to automation libraries for helper functions for doing things like driving a web browser or using a graphical interface. When Cucumber runs, it executes each step in turn. If all the steps complete successfully, the test is said to have passed, otherwise the user is informed that the test didn't pass, and the exact state of the test is reported.

I mentioned before that software begins with an idea. Cucumber helps us to capture what the vision behind the idea is. We need to understand what the goal is. The vision might be massive, complex, and exciting. Our task is to work with the visionary to achieve something of value that moves them in the right direction. In recent years, this has started to be called a *Minimum Marketable Feature* or *Minimum Viable Product*. Whatever you call it, we're looking for a description of something achievable that captures and advances the vision and purpose behind the software.

Let's use Cucumber to write acceptance tests for the HipsterAssessor as a way to explore how the approach works.

Enter the *HipsterAssessor* project directory and run Cucumber:

```
$ gem install cucumber $ cucumber You don't have a 'features' directory.
Please create one to get started. See http://cukes.info/ for more information.
```

OK, let's create a directory for the features, and try again:

```
$ mkdir features $ cucumber 0 scenarios 0 steps 0m0.000s
```

This illustrates two important concepts: Cucumber is made from *scenarios* and *steps*. Each test that we write represents a scenario that we describe—it tests an aspect of the broader feature that we're going to help implement. Each scenario contains steps that will tell Cucumber how to actually carry out the test and verify that the intended feature works as specified.

Features are written in a file with a *.feature* suffix, in the Gherkin language. Open your text editor and create a file called *features/assess_hipster.feature*. This is a plain text document, written with a few constraints. The constraints are minimal—the idea is that the feature we write should be in a natural language. In fact, one of the benefits of Gherkin is it supports over 40 languages, so you can write your features in Russian or Welsh if you wish. Actually, this provides a good way to demonstrate how small the DSL is:

```
$ cucumber --i18 cy-GB | feature           | "Arwedd"               | | back-
ground         | "Cefndir"                | | scenario            | "Scenar-
io"                  | | scenario_outline | "Scenario Amlinellol" | | exam-
ples         | "Enghreifftiau"       | | given            | "* ", "Anrhegedig a
" | | when              | "* ", "Pryd "         | | then             | "* ",
```

```
"Yna "             | | and                  | "* ", "A "               | |
but               | "* ", "Ond "           | | given (code)     | "Anrhegedi-
ga"         | | when (code)      | "Pryd"               | | then (code)      |
"Yna"                  | | and (code)       | "A"                     | | but
(code)       | "Ond"                |
```

That's the extent of the DSL. Using these keywords, we express our feature. In terms of grammar, the rules are very simple. The file must begin with a feature, followed by a title. This may be followed by an arbitrary number of lines of freeform text to document the feature.

```
Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster

Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points Given a developer with a fixed-
wheel bike When I request a hipster assessment Then the score should be 5

Scenario: Spectacles despite 20/20 vision scores 10 points Given a developer
with a pair of empty frames When I request a hipster assessment Then the score
should be 10
```

The preceding code block is a full feature, expressed in Gherkin. The idea behind Gherkin is to be able to provide concrete examples that illustrate the required feature. As a language, Gherkin has been optimized for readability and portability. As you can see, Gherkin is pretty much indistinguishable from natural language.

A scenario describes the behavior of the system. Each scenario shares a common pattern. First we set up some state: what is the prerequisite to test the functionality? In this case, since we're assessing developers, we need a developer. Next we take an action that we anticipate will change some state. In this case, we're going to ask for a score. Finally, we check the new state and compare it to what we expected. In this case, we expect that the HipsterAssessor will award our developer some points.

The keywords *'Scenario'*, *'Given'*, *'When'*, and *'Then'* map onto Ruby code called step definitions. If we go ahead and run Cucumber now, we'll see some progress:

```
$ cucumber Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster
```

```
Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13  Given  a  developer  with  a  fixed-wheel  bike        #  features/
assess_hipster.feature:14 When I request a hipster assessment          # fea-
tures/assess_hipster.feature:15 Then the score should be 5                 #
features/assess_hipster.feature:16

Scenario:  Spectacles  despite  20/20  vision  scores  10  points  #  features/
assess_hipster.feature:18  Given  a  developer  with  a  pair  of  empty
frames           # features/assess_hipster.feature:19 When I request a hipster
assessment                       # features/assess_hipster.feature:20 Then the
score  should  be  10                                                 #  features/
assess_hipster.feature:21

2 scenarios (2 undefined) 6 steps (6 undefined) 0m0.003s
```

You can implement step definitions for undefined steps with these snippets:

```
Given(/^a developer with a fixed\-wheel bike$/) do pending # express the regexp
above with the code you wish you had end

When(/^I request a hipster assessment$/) do pending # express the regexp above
with the code you wish you had end

Then(/^the score should be (\d+)$/) do |arg1| pending # express the regexp
above with the code you wish you had end

Given(/^a developer with a pair of empty frames$/) do pending # express the re-
gexp above with the code you wish you had end
```

If you want snippets in a different programming language, just make sure a file with the appropriate file extension exists where Cucumber looks for step definitions.

Cucumber has generated some code snippets to get us started. Step definitions by convention reside in a *step_definitions* directory, under the *features* directory. Let's create that directory, and inside there paste the suggested snippets into a file called *assess_hipster_steps.rb*.

Now if we run Cucumber we get a bit further:

```
$ cucumber Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster

Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
```

```
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:1 TODO (Cucumber::Pending) ./features/step_definitions/
assess_hipster_steps.rb:2:in `/^a developer with a fixed\-wheel bike$/' fea-
tures/assess_hipster.feature:14:in `Given a developer with a fixed-wheel bike'
When I request a hipster assessment             # features/step_definitions/
assess_hipster_steps.rb:5 Then the score should be 5                    # fea-
tures/step_definitions/assess_hipster_steps.rb:9

Scenario: Spectacles despite 20/20 vision scores 10 points # features/
assess_hipster.feature:18 Given a developer with a pair of empty
frames             # features/step_definitions/assess_hipster_steps.rb:13 TODO
(Cucumber::Pending)  ./features/step_definitions/assess_hipster_steps.rb:14:in
`/^a developer with a pair of empty frames$/' features/assess_hipster.feature:
19:in `Given a developer with a pair of empty frames' When I request a hipster
assessment                                          # features/step_definitions/
assess_hipster_steps.rb:5      Then      the      score      should      be
10                                          # features/step_definitions/
assess_hipster_steps.rb:9

2 scenarios (2 pending) 6 steps (4 skipped, 2 pending) 0m0.004s
```

Cucumber is now calling our step definitions. But our step definitions don't contain any code that does anything noteworthy, and so Cucumber stops and tells us that the first two steps of each scenario are pending—i.e., unwritten—and therefore it skipped the rest of the test.

Let's look at the structure of a step within a step definition:

```
Given /^a developer with a fixed wheel bike$/ do pending # express the regexp
above with the code you wish you had end
```

We're now in pure Ruby—well, we're in a pure Ruby DSL. `Given` is a DSL method that takes a regular expression and a block. The regular expression matches the step in the Gherkin scenario, and the contents of the block specifies what to do when this step is matched. The fact that we're using regular expressions to match the steps in the Gherkin scenario gives us two very powerful capabilities—we can use *capture groups* and *wildcards*. This is just the same as capture groups in sed—you can put parentheses around some text and store what they match in a variable for later use. Wildcards are like a more powerful and flexible form of shell globbing—we can match non-whitespace characters, digits, lowercase letters, or combinations thereof. We'll see this in action in the score step in a moment. Let's write the step definitions for real now.

The first is pretty straightforward. We're going to do the same as we did in the previous steps and instantiate a developer. Take a look at the comment that the automatically generated snippet contains. The key idea here is that we should write the code we wish

we had. We don't have any code at all. We're simply expressing the interface we'd like to see. Interestingly, when we do it this way, we tend to think with a more design-oriented head. The code we have in the RSpec test is actually a bit ugly:

```
developer = HipsterAssessor.new(gears_on_bike=1)
```

Wouldn't it be nicer to have a method on the assessor that sets the number of gears to a certain value? This would certainly be nicer if we were to think of an interface that we could use with a webform, or some other way to populate the object. Simply calling the constructor with an argument is rather clumsy. Let's write the code we wish we had:

```
Given(/^a developer with a fixed\-wheel bike$/) do @developer = HipsterAsses-
sor.new @developer.set(:gears_on_bike, 1) end
```

Now, let's fulfill the *when* step. This is just calling a method:

```
When(/^I request a hipster assessment$/) do @result = @developer.score.to_s end
```

We need to convert the score to a string because in our feature the value appears as a string, not an integer.

Now we come to the *then* step. Here we can see the power of the regular expression. Cucumber has already suggested we might be interested in the score and has suggested a capture group and wildcard. The value of this will be passed into the block as `arg1`. We should change that to something more readable.

```
Then /^the score should be (\d+)$/ do |score| expect(@result).to eq score end
```

While we're at it, let's add the developer with empty frames given, and then we can run the whole feature.

```
Given(/^a developer with a pair of empty frames$/) do @developer = HipsterAsses-
sor.new @developer.set(:glasses_prescription, nil) end
```

We should probably move this up, so it reads nicely, too. At this stage our steps look like this:

```
Given(/^a developer with a fixed\-wheel bike$/) do @developer = HipsterAsses-
sor.new @developer.set(:gears_on_bike, 1) end

Given(/^a developer with a pair of empty frames$/) do @developer = HipsterAsses-
sor.new @developer.set(:glasses_prescription, nil) end

When(/^I request a hipster assessment$/) do @result = @developer.score.to_s end

Then /^the score should be (\d+)$/ do |score| expect(@result).to eq score end
```

OK…what happens when we run Cucumber?

```
$ cucumber Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster
```

```
Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:1 uninitialized constant HipsterAssessor (NameError) ./
features/step_definitions/assess_hipster_steps.rb:2:in `/^a developer with a
fixed\-wheel bike$/' features/assess_hipster.feature:14:in `Given a developer
with a fixed-wheel bike' When I request a hipster assessment         # features/
step_definitions/assess_hipster_steps.rb:6   Then   the   score   should   be
5                 # features/step_definitions/assess_hipster_steps.rb:10

Scenario: Spectacles despite 20/20 vision scores 10 points # features/
assess_hipster.feature:18 Given a developer with a pair of empty
frames              # features/step_definitions/assess_hipster_steps.rb:14 unde-
fined method `set' for nil:NilClass (NoMethodError) ./features/step_definitions/
assess_hipster_steps.rb:15:in `/^a developer with a pair of empty frames$/' fea-
tures/assess_hipster.feature:19:in `Given a developer with a pair of empty
frames' When I request a hipster assessment                          # features/
step_definitions/assess_hipster_steps.rb:6   Then   the   score   should   be
10                                          # features/step_definitions/
assess_hipster_steps.rb:10

Failing Scenarios: cucumber features/assess_hipster.feature:13 # Scenario:
Fixed-wheel bicycle scores 5 points cucumber features/assess_hipster.feature:18
# Scenario: Spectacles despite 20/20 vision scores 10 points

2 scenarios (2 failed) 6 steps (2 failed, 4 skipped) 0m0.004s
```

OK, this is familiar—we have a failing test! We haven't connected the test to our code. Let's do that by adding the `require_relative` to the top of the steps:

```
require_relative '../../hipsterassessor'
```

Now the test runs, and the relevant part of the output is:

```
Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:3 wrong number of arguments (0 for 1) (ArgumentError) ./
hipsterassessor.rb:3:in        `initialize'        ./features/step_definitions/
assess_hipster_steps.rb:4:in          `new'         ./features/step_definitions/
assess_hipster_steps.rb:4:in `/^a developer with a fixed\-wheel bike$/' fea-
tures/assess_hipster.feature:14:in `Given a developer with a fixed-wheel bike'
When I request a hipster assessment             # features/step_definitions/
assess_hipster_steps.rb:12 Then the score should be 5                   # fea-
tures/step_definitions/assess_hipster_steps.rb:16
```

Now, here we're working slightly outside the standard pattern I would recommend because I chose to introduce testing from the unit tests out. We've actually already got

code, which we're calling, that we need to change. Let's follow through and see what happens. So at the moment, our test code is calling the constructor with no arguments:

```
Given(/^a developer with a fixed\-wheel bike$/) do @developer = HipsterAsses-
sor.new @developer.set(:gears_on_bike, 1) end
```

But in the actual class, we specify that the constructor took an argument. Let's remove that:

```
def initialize @gears = bike_gears @score = 0 end
```

While we're there, our constructor shouldn't try to set gears any more either, so let's remove that line:

```
def initialize @score = 0 end
```

Running Cucumber now yields the following:

```
$ cucumber features/assess_hipster.feature:13 Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster

Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:3   undefined   method   `set'   for   #<HipsterAssessor:
0x00000002aaf648    @score=0>    (NoMethodError)    ./features/step_definitions/
assess_hipster_steps.rb:5:in `/^a developer with a fixed\-wheel bike$/' fea-
tures/assess_hipster.feature:14:in `Given a developer with a fixed-wheel bike'
When I request a hipster assessment             # features/step_definitions/
assess_hipster_steps.rb:12 Then the score should be 5                    # fea-
tures/step_definitions/assess_hipster_steps.rb:16

Failing  Scenarios:  cucumber  features/assess_hipster.feature:13  #  Scenario:
Fixed-wheel bicycle scores 5 points

1 scenario (1 failed) 3 steps (1 failed, 2 skipped) 0m0.002s
```

We need a `set` method. At this point, we should drop down a level to RSpec or Minitest and write a test for the `set` method.

First, let's run the test and see what breaks:

```
$ ruby test_hipster.rb Run options: --seed 59310

# Running tests:
```

```
EEE

Finished tests in 0.000963s, 3113.7225 tests/s, 0.0000 assertions/s.

1) Error: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0001_can establish if the developer has a fixed-wheel bicycle: Argu-
mentError: wrong number of arguments (1 for 0) /home/tdi/tdd-principles/hipster-
assessor.rb:3:in `initialize' test_hipster.rb:8:in `new' test_hipster.rb:8:in
`block (2 levels) in <main>'

2) Error: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0002_can report a hipster assessment score: ArgumentError: wrong num-
ber of arguments (1 for 0) /home/tdi/tdd-principles/hipsterassessor.rb:3:in `in-
itialize' test_hipster.rb:8:in `new' test_hipster.rb:8:in `block (2 levels) in
<main>'

3) Error: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0003_can award five points for having a fixie: ArgumentError: wrong
number of arguments (1 for 0) /home/tdi/tdd-principles/hipsterassessor.rb:3:in
`initialize' test_hipster.rb:8:in `new' test_hipster.rb:8:in `block (2 levels)
in <main>'

3 tests, 0 assertions, 0 failures, 3 errors, 0 skips
```

Unsurprisingly, everything breaks because we're calling the constructor differently. Thankfully that's trivial to fix in our Minitest test; we only instantiate the assessor in one place. Change that to:

```
before do @developer = HipsterAssessor.new end
```

Now running the test returns failures not errors:

```
$ ruby test_hipster.rb Run options: --seed 21627

# Running tests:

FF.

Finished tests in 0.040021s, 74.9611 tests/s, 74.9611 assertions/s.

1) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0001_can establish if the developer has a fixed-wheel bicycle
[test_hipster.rb:14]: Expected: true Actual: false

2) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0003_can award five points for having a fixie [test_hipster.rb:22]:
Expected: 5 Actual: 0

3 tests, 3 assertions, 2 failures, 0 errors, 0 skips
```

We need also to write a test for the `set` method, and then turn to fixing the remaining tests. As we think about it, we realize we need a `get` method, too, and this is needed for the test:

```
it "can set a hipster credential to a given value" do @developer.set(:favor-
ite_beer, "PBR") @developer.get(:favorite_beer).must_equal "PBR" end
```

Let's run the test:

```
$ ruby test_hipster.rb Run options: --seed 44375

# Running tests:

E.FF

Finished tests in 0.018478s, 216.4764 tests/s, 162.3573 assertions/s.

1) Error: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0004_can set a hipster credential to a given value: NoMethodError: un-
defined method `set' for #<HipsterAssessor:0x00000000efe890 @score=0> test_hip-
ster.rb:26:in `block (3 levels) in <main>'

2) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0003_can award five points for having a fixie [test_hipster.rb:22]:
Expected: 5 Actual: 0

3) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0001_can establish if the developer has a fixed-wheel bicycle
[test_hipster.rb:14]: Expected: true Actual: false

4 tests, 3 assertions, 2 failures, 1 errors, 0 skips
```

Right, let's implement the `set` method. Add a `hipster\_credentials` hash to the constructor, and then the `set` method:

```
def initialize @score = 0 @hipster_credentials = {} end

def set(key, value) @hipster_credentials[key] = value end
```

Running the tests now reveals that we need a `get` method. We already exercise this in the test, so let's write the method for that:

```
def get(key) @hipster_credentials[key] end
```

Now run the tests:

```
$ ruby test_hipster.rb Run options: --seed 32953

# Running tests:

F..F

Finished tests in 0.018647s, 214.5123 tests/s, 214.5123 assertions/s.

1) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0003_can award five points for having a fixie [test_hipster.rb:22]:
Expected: 5 Actual: 0
```

```
2) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0001_can establish if the developer has a fixed-wheel bicycle
[test_hipster.rb:14]: Expected: true Actual: false

4 tests, 4 assertions, 2 failures, 0 errors, 0 skips
```

OK, now our `get` and `set` methods work. We still have other failing tests though, which we need to make pass. When we set up state in the test we need to use the `HipsterAssessor#get` and `HipsterAssessor#set`, for those cases where a fixed-wheel bicycle is mentioned. We also need to make the `has\_fixie` method use the `hipster\_credentials` hash.

In our test, we make the updates:

```
it "can establish if the developer has a fixed-wheel bicycle" do @develop-
er.set(:gears_on_bike, 1) @developer.has_fixie?.must_equal true end

it "can award five points for having a fixie" do @developer.set(:gears_on_bike,
1) @developer.score.must_equal 5 end
```

And in the class:

```
def has_fixie? @hipster_credentials[:gears_on_bike] == 1 end
```

Now all the tests pass!

```
$ ruby test_hipster.rb Run options: --seed 44033

# Running tests:

....

Finished tests in 0.000843s, 4744.1706 tests/s, 4744.1706 assertions/s.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

Let's quickly summarize the state of the test and the class. Here's the test:

```
require 'minitest/autorun' require_relative 'hipsterassessor'


describe HipsterAssessor do

before do @developer = HipsterAssessor.new end

describe "when assessing whether a developer is a hipster" do

it "can establish if the developer has a fixed-wheel bicycle" do @develop-
er.set(:gears_on_bike, 1) @developer.has_fixie?.must_equal true end

it "can report a hipster assessment score" do @developer.score.must_be_in-
stance_of Fixnum end

it "can award five points for having a fixie" do @developer.set(:gears_on_bike,
```

```
1) @developer.score.must_equal 5 end

it "can set a hispter credential to a given value" do @developer.set(:favor-
ite_beer, "PBR") @developer.get(:favorite_beer).must_equal "PBR" end

end

end
```

And here's the class:

```
class HipsterAssessor

def initialize @score = 0 @hipster_credentials = {} end

def set(key, value) @hipster_credentials[key] = value end

def get(key) @hipster_credentials[key] end

def has_fixie? @hipster_credentials[:gears_on_bike] == 1 end

def score if self.has_fixie? @score = @score + 5 end @score end end
```

Now that the tests pass, we can go back out to Cucumber.

Running Cucumber now shows the first scenario passing! The second doesn't pass:

```
$ cucumber Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster

Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.

Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:3 When I request a hipster assessment           # fea-
tures/step_definitions/assess_hipster_steps.rb:13  Then  the  score  should  be
5                 # features/step_definitions/assess_hipster_steps.rb:17

Scenario:  Spectacles  despite  20/20  vision  scores  10  points  #  features/
assess_hipster.feature:18  Given  a  developer  with  a  pair  of  empty
frames             # features/step_definitions/assess_hipster_steps.rb:8 When I
request a hipster assessment                        # features/step_definitions/
assess_hipster_steps.rb:13     Then     the     score     should     be
10                                          #  features/step_definitions/
assess_hipster_steps.rb:17

expected: "10" got: "0"
```

```
(compared using ==) (RSpec::Expectations::ExpectationNotMetError) ./features/
step_definitions/assess_hipster_steps.rb:18:in `/^the score should be (\d+)$/'
features/assess_hipster.feature:21:in `Then the score should be 10'

Failing Scenarios: cucumber features/assess_hipster.feature:18 # Scenario: Spec-
tacles despite 20/20 vision scores 10 points

2 scenarios (1 failed, 1 passed) 6 steps (1 failed, 5 passed) 0m0.004s
```

This requires us to go back to the lower level, and write a test for applying a value on the basis of phoney spectacles. Let's add that test:

```
it "can award ten points for phoney spectacles" do @developer.set(:glasses_pre-
scription, nil) @developer.score.must_equal 10 end
```

Watch it fail:

```
$ ruby test_hipster.rb Run options: --seed 36835

# Running tests:

..F..

Finished tests in 0.018249s, 273.9899 tests/s, 273.9899 assertions/s.

1) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0004_can award ten points for phoney spectacles [test_hipster.rb:29]:
Expected: 10 Actual: 0

5 tests, 5 assertions, 1 failures, 0 errors, 0 skips
```

Now we realize that we should have a test that the assessor can use to establish if the developer has phoney specs. Let's add that, too:

```
it "can establish if the developer has phoney spectacles" do @develop-
er.set(:glasses_prescription, nil) @developer.has_phoney_specs?.must_equal true
end
```

Run the tests, watch it fail:

```
$ ruby test_hipster.rb Run options: --seed 36835

# Running tests:

..F..

Finished tests in 0.018249s, 273.9899 tests/s, 273.9899 assertions/s.

1) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0004_can award ten points for phoney spectacles [test_hipster.rb:29]:
Expected: 10 Actual: 0

5 tests, 5 assertions, 1 failures, 0 errors, 0 skips tdi@tk00:~/tdd-principles$
```

```
vi test_hipster.rb tdi@tk00:~/tdd-principles$ ruby test_hipster.rb Run options:
--seed 7359

# Running tests:

.F..E.

Finished tests in 0.018551s, 323.4269 tests/s, 269.5224 assertions/s.

1) Failure: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0005_can award ten points for phoney spectacles [test_hipster.rb:34]:
Expected: 10 Actual: 0

2) Error: HipsterAssessor::when assessing whether a developer is a hip-
ster#test_0002_can establish if the developer has phoney spectacles: NoMethodEr-
ror: undefined method `has_phoney_specs?' for #<HipsterAssessor:
0x0000000136db38> test_hipster.rb:20:in `block (3 levels) in <main>'

6 tests, 5 assertions, 1 failures, 1 errors, 0 skips
```

Now add the method which checks for the specs, and then update the score method to return 10 in the case of phoney specs:

```
def has_phoney_specs? @hipster_credentials[:gears_on_bike] == nil end

def score if self.has_fixie? @score = @score + 5 elsif self.has_phoney_specs?
@score = @score + 10 end @score end
```

Once more with feeling!

```
ruby test_hipster.rb Run options: --seed 15341

# Running tests:

......

Finished tests in 0.000912s, 6581.1700 tests/s, 6581.1700 assertions/s.

6 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

And with Cucumber?

```
$ cucumber Feature: Assess hipster

In order to make sure developers are comfortable in their workplace As a manag-
er who has just hired a developer I want to be able to assess whether the new
developer is a hipster

Hipsters like to have Pabst Blue Ribbon in the fridge, listen to vinyl record-
ings of Lily Allen, and need a place to store their fixed-wheel bicycles.  As a
manager I need to be sure I can accommodate hipsters, so I want a simple web
app that gives a questionnaire which will advise me if the developer is a hip-
ster.
```

```
Scenario: Fixed-wheel bicycle scores 5 points # features/assess_hipster.feature:
13 Given a developer with a fixed-wheel bike    # features/step_definitions/
assess_hipster_steps.rb:3 When I request a hipster assessment           # fea-
tures/step_definitions/assess_hipster_steps.rb:13  Then  the  score  should  be
5                 # features/step_definitions/assess_hipster_steps.rb:17

Scenario:  Spectacles  despite  20/20  vision  scores  10  points  #  features/
assess_hipster.feature:18   Given   a   developer   with   a   pair   of   empty
frames             # features/step_definitions/assess_hipster_steps.rb:8 When I
request a hipster assessment                        # features/step_definitions/
assess_hipster_steps.rb:13      Then      the      score      should      be
10                                                  #  features/step_definitions/
assess_hipster_steps.rb:17

2 scenarios (2 passed) 6 steps (6 passed) 0m0.004s
```

So, at the end of that whistlestop tour of testing in Ruby, you should now feel confident that you understand the rationale, toolchain, and workflow of test- and behavior-driven development. Let's now move on to discuss how to go about implementing some of these ideas with respect to infrastructure coding.

CHAPTER 6
A Test-Driven Infrastructure Framework

At the time of the first edition of this book, there was only one tool and a handful of people exploring the ideas of infrastructure testing. The first edition covered that tool—a tool written by me as a proof of concept to demonstrate that the project of testing infrastructure code was achievable. This tool, *Cucumber-Chef*, was intentionally narrow in its purview, in that it attempted to explore one particular aspect of the broader infrastructure-testing landscape, in a way that reduced the commitment in terms of acquiring new machines to zero. Based around Opscode's Hosted Chef service and Amazon's EC2 platform, it set out to open the discussion and get the conversation moving.

The testing ecosystem has blossomed since the first edition of this book. Mature frameworks are emerging, significant community adoption of the testing of cookbooks and infrastructure is taking place, and helper tools and knife plug-ins specifically targeted at infrastructure testing are released regularly.

This chapter takes a high-level philosophical overview of the business of testing infrastructure code. It sets out a vision for what the landscape should look like. This is a landscape that changes day by day. At the time of this writing—early summer 2013—there is a profound level of interest in infrastructure testing. Discussions abound on the mailing lists, IRC, Twitter, and in various podcasts. It's a dynamic, exciting, and fast-moving subject area.

That said, I believe it is possible both to set a conceptual framework for what needs to be in place, and to outline a workflow based on the current best-of-breed tooling available. Having presented a conceptual framework, we will survey a selection of the currently available tools, providing examples of each tool together with a discussion of their merits and demerits, and how they fit into an overarching testing strategy.

Naturally in a fast-moving technology space such as infrastructure as code, the state of the art is in flux; however, I think we can be confident that a philosophy, methodology,

and requirements list against which we can continue to measure tools as they emerge can be synthesized.

Test-Driven Infrastructure: A Conceptual Framework

I'll start by setting out a high-level vision. I'm not a believer in luck; although I share the observation of legendary South African golfer, Gary Player, who maintained, "The harder I practice, the luckier I get." That said, I think it does no harm, as a community, or a movement, to have a mascot. The MASCOT I propose upholds the following six objectives:

Test-driven infrastructure should be:

- **M**ainstream
- **A**utomated
- **S**ide effect aware
- **C**ontinuously integrated
- **O**utside-in
- **T**est-first

Test-Driven Infrastructure Should Be Mainstream

My vision is that soon it won't even be questioned that developing infrastructure is done in a test-driven way. Although a very strong case can be made for the approach, it will never become mainstream until the barriers to entry are lowered. It's no surprise that, of modern languages, Ruby has most comprehensively embraced test-driven engineering. The quality of tooling is tremendously high with innovation and improvement seen on a regular basis. The passion and enthusiasm of the community has made testing a popular topic, and within the web development world, Ruby leads the way, and test-driven development *is* mainstream. Within our world of infrastructure as code, the tooling we have isn't yet sufficiently powerful or easy to use to encourage mass adoption, but we're on the right trajectory.

In order for testing to become mainstream, it's necessary to agree to a set of standards around which to organize. Of particular concern is community agreement about the general syntax and style of cookbooks. When developing infrastructure code in a shared environment, enforcing a house style can be a very valuable thing to implement. It encourages the team to work in a consistent way and ensures that code is maximally shareable and portable.

Test-Driven Infrastructure Should Be Automated

In order for testing to become mainstream and effective, it's essential that it's automated. This is especially the case for long-running, complex integration tests. Without a workflow that includes automation of these high-value, but labor-intensive tests, they simply won't be run with sufficient frequency to deliver consistent improvements.

Automation takes place at a number of levels. To an extent, the very act of writing test code is a kind of automation. We're encoding the steps that need to be taken to verify that a given state has been achieved, or that a given behavior is being exhibited. However, it's not just the encoding of the steps required to carry out the test that needs to be automated. We also need to automate the running of the tests with a degree of frequency that is meaningful, and a degree of feedback that is noticeable and unignorable.

To draw a parallel with the mainstream software development world, when writing tests, some tests are harder to write than others. Specifically, writing unit tests is pretty easy. Writing integration tests is harder. Writing end-to-end acceptance tests is hardest. This means that sometimes the hardest tests are simply not automated—in some cases the testing is left to the customer. The same applies when testing infrastructure. It's not difficult to write a test that asserts that a resource has been brought into the correct state. It's harder to test connectivity between two layers of infrastructure, such as between database and web server. It's hardest of all to verify that the infrastructure behaves as it should, from monitoring to backups, from top to bottom. In both worlds, the most value is in the hardest stuff.

Martin Fowler likes the sound byte, "If it hurts, do more of it." The logic behind this seemingly paradoxical statement is that there's an exponential relationship between the amount of pain experienced and the amount of time between occurrences of the thing that causes pain. This is the case for converging nodes, rebuilding servers, migrating databases, speaking to stakeholders, releasing software, and of course, running tests. Thus it stands to reason that by doing it more frequently, it will, in fact, start to hurt less.

If there's any pain associated with the frequent running of tests—unreliable tests, flakey interfaces, slow test machines, very long-running tests, or the like—it's especially important to automate them.

Our infrastructure tests should run automatically—ideally on every commit. Even better would be to move to a continuous deployment model, where every commit not only kicks off a test run, but deploys the code on a test environment and then traverses a build pipeline with appropriate yes/no gates, ultimately resulting in an update of the production infrastructure. This is the current state of the art in the software development world. If infrastructure is code, we should give serious thought to adopting the same mentality when writing Chef recipes and cookbooks.

Test-Driven Infrastructure Should Be Side-Effect Aware

In his *State of the Union* presentation at the inaugural Chefconf event in Burlingame, CA, Adam Jacob made the observation that configuration management is effectively the study of side-effects. When we write infrastructure code to capture a set of complex requirements, what we're really doing is commanding one system to take action in a way that affects another system, which in turn impacts other systems in such a way as to bring the world into a desired state. Chef takes this challenge in its stride—it aims to make systems easy to reason about, to remain predictable, and understandable in the event of a mistake.

The bigger challenge comes in the inherent portability of Chef cookbooks and recipes. Especially amongst the popular community cookbooks—such as Apache or MySQL, with dozens of contributors across a range of Linux, Unix, and Windows systems—it's entirely possible that a change or improvement introduced for one platform will have an unexpected and adverse side-effect on users on a different platform. Our test-driven infrastructure vision needs to acknowledge and mitigate against this risk.

Fundamentally, we want to be confident that seemingly trivial changes to our cookbooks don't have unwanted side-effects. This becomes more of a challenge if our cookbooks grow to support multiple platforms. The possibility that a trivial change for a system running on Red Hat breaks compatibility for FreeBSD is something that needs to be guarded against. Naturally this can be achieved manually, by spinning up a virtual machine, running Chef, and looking at the output, but automating this makes it far more likely that it will happen as a matter of course. This is especially valuable as the number and complexity of our cookbooks grow, and even more especially in an environment in which many different developers are cooperating.

Test-Driven Infrastructure Should Be Continuously Integrated

A key component of constructing a world in which our infrastructure testing is both automated and side-effect aware is that the code we write should be continuously integrated.

Another core practice from eXtreme programming, the idea of continuous integration lies in the recognition that the traditional approach of periodically integrating the code of a number of different people is invariably an error-prone, time-consuming, and painful endeavor. Ron Jeffries quips, on the C2 wiki:

> I've been working on my classes and think they are perfect. You've been working on yours and I suppose you think they're pretty good, too. Carl has been working on his, and you know how that goes.
>
> Now we have to integrate them to build a new system. Carl's code, as usual, breaks everything. It looks to me as if you have a few problems, too. My code is solid, I know that because I worked hard on it.

> What I can't understand is why you think there might be something wrong with my code, and Carl, the idiot, is after both of us.
>
> We're in for a few really unpleasant days. Maybe next time we shouldn't wait so long to integrate…
>
> — Ron Jeffries

The response is the principle that developers should be integrating and committing code very frequently. This avoids diverging or fragmented development efforts, especially where team members are not in direct communication with each other. In a community development effort, such as cookbooks, this is even more vital.

In an XP team, the process of integrating the code means gathering the latest code, and running all the tests. If tests fail, collaborating on what caused the failure and committing a fix becomes the priority task.

If we're to be serious about developing quality infrastructure code, we need to bring the same practices to bear. This means that our tests need to be run automatically on commits, and the results shared visibly and publicly.

Test-Driven Infrastructure Should Be Outside In

One of the maxims of BDD is that we take an outside-in approach. Imagine I set a group of people in a room to a programming task of moderate complexity. If you were to watch each person in the room after I'd finished explaining the task, I think you'd find that the most natural approach, and the statistically most likely approach of each person would be to open up their editor of choice and start hacking away. You might find some people opening up some kind of interactive REPL and experimenting. Those with a grounding in agile programming approaches might even start writing some basic unit tests. This kind of approach is what I call "inside-out." Straight away we're starting to write the code to solve the problem, even if we're writing tests first.

BDD encourages thinking about the problem a different way. This is the great thing about Cucumber—it allows and to an extent, even forces the developer to step right away from the implementation details and think about how the software should look, feel, behave, act. This is outside-in. We describe a feature that delivers value as an executable specification. Only once we have this feature described, and failing a test, do we start to think about how to make it pass.

The same approach makes a great deal of sense when we are doing infrastructure development. If I set a task, such as setting up an issue tracker, and asked a number of people in a room to carry this out, you'd see similar behavior. Most would start by installing Apache and PHP, and then maybe think about a user, and hack forward from there. A smaller number would start to write or even reuse Chef cookbooks and recipes. The outside-in approach starts by writing the feature that defines how the piece of infrastructure should behave.

We want to ensure that our cookbooks deliver the intended behavior—that they solve the particular problem we have in mind when we set out.

I've already covered the foundational principles of behavior-driven development, but I will re-emphasize the fact that none of our development efforts are worth a thing if they don't address a specific business value. Test-driven infrastructure means committing to build the right thing, not just build the thing right.

Test-Driven Infrastructure Should Be Test-First

The final objective of my mascot manifesto is that as we write our infrastructure, not only should we be ensuring our code is under test, but that those tests should be written before we write any Chef code. This discipline recognizes that the tests we write are actually a development tool in themselves. The benefits are clear:

- It focuses attention on precisely what the cookbook/recipe needs to do.
- It makes it very clear where the development should start.
- There is never any question about the definition of done—the test owns this.
- It encourages a lean and efficient development approach: we build only as much infrastructure as is needed to make the tests pass.
- In the spirit of Chef, it makes our code easy to reason about—the target is reproducible, predictable results.
- Dependencies are flushed out early, and their minimization is a core activity.
- It surfaces good design decisions by encouraging the creation of solutions that are simple enough to make the test pass, but no simpler.
- In the event of unexpected failures, the debugging process is targeted.
- It encourages refactoring—as we write code to make our tests pass, so we should identify hints that refactoring is needed.

I asked my family what animal they felt would be appropriate as a mascot for a test-driven infrastructure manifesto. They gave it careful deliberation before suggesting that the best choice was a tortoise. Their reasoning was that tortoises like eating cucumbers, don't dash head first into things, but take a measured and careful approach, and in fine Aesopian tradition, win the race anyway.

I'm not sure it'll catch on, but I am sure that to achieve this sextuplet of objectives, we need to overcome a number of technical hurdles.

The Pillars of Test-Driven Infrastructure

What, then, should be the conceptual framework that informs our choice of tools? How do we go about ensuring that TDI is MASCOT? If we want TDI to be mainstream, what needs to be in place? If we want our testing to be automated, what do we need to accomplish that? What does it mean for our tests to be side-effect aware? What specifically do we need to make sure this is happening? What about continuous integration—how do we go about that? Are there tools? Workflows? What do we require, and what do we lack? In order to perform the whole endeavor test-first, what do we need? Or perhaps, what does the beginning infrastructure developer lack, and if they were to be handed a starter pack, with "TDI Essentials: A Toolkit for Success" written on the front, what would it contain?

I think it makes sense to try to break the requirements down into four broad areas. Obviously we need to write the tests themselves, which requires that we have access to a testing framework, and supporting tools and documentation to help us write those tests. Naturally we need to run our tests, and indeed have them run in our absence, without our constant input. Given that we're testing infrastructure, we need to be able to set up and tear down—to build a test infrastructure for the purposes of testing. This is effectively a provisioning problem. And finally, we need to be told the results, in a meaningful way, in a timely manner, and in such a way that encourages us to take action. That is to say, the feedback we get needs to be directed, relevant, and accurate.

Let's unpack these four supporting, conceptual pillars:

- Writing
- Running
- Provisioning
- Feedback

Writing Tests

The process of testing code consists of setting up state, introducing some input that changes that state, and then comparing the resulting state with our expectations. As discussed in Chapter 5, it's apparent that this test writing needs to take place at several different levels—from the high-level behavior of the overall system we're building, to the verification that distribution-specific variables are evaluated correctly.

The main challenge here is in making this process easy. Having to write verbose, manual expectations to assert that, for example, a package was installed, is tiresome. Such expectations and assertions can be simplified and shared. The more complex, end-to-end systems are more likely to require the solutions to more involved and bespoke

challenges, however as the corpus of tests in the community grows, so will the body of experience and confidence.

It needs to be easy for infrastructure developers to assert that a resource is in the desired state. Ideally this should be in the form of providing potted assertions that can be reused, rather than requiring the developer to create this scaffolding him or herself.

Running Tests

Once infrastructure developers feel confident in writing tests, they need to establish the most effective way to run both their tests, and in cases Chef itself, on or against a range of systems.

I don't think it's unfair to claim that the mechanism by which tests are run automatically is pretty much a solved problem. There are mature job runners and continuous integration frameworks and even online services that are designed specifically for this task and are used every day by countless software development organizations.

However, orchestrating the running of tests we have written is not without its own challenges, especially if the tests are to be run on remote machines or to span multiple systems. In line with our desire for maximum automation, we also need to establish the most effective way to run tests in an unattended way, on commit, or with predictable periodicity.

Of course, a prerequisite for being able to run these tests is the ability to provision test infrastructure rapidly and painlessly. We consider this next.

Provisioning Machines

The holy grail of infrastructure testing is the ability to specify an infrastructure feature, provision some hardware, apply Chef, and verify that the intended behavior has been met, all quickly and automatically.

Primitive testing can be carried out on one's own development workstation, but pretty soon a need to provision fresh machines, run Chef against them, and then test them, becomes a clear requirement.

As soon as the infrastructure we're building has one or more of the following characteristics, we need to solve this problem:

- The infrastructure runs on a different OS from that of the developer's workstation.
- The infrastructure runs on more than one OS or distribution.

And, in fact, there is always going to be a need for a complete end-to-end test, which at the very least demands a brand new, fresh, unadulterated machine from which to start, and which may involve a large number of different machines.

Advances in desktop virtualization technology, and the ready availability of highly powered laptops and workstations does make keeping this test environment on one's local machine more achievable than it was a few years ago. Indeed the ready availability of local test machines has brought about a significant upsurge in people starting to take infrastructure testing seriously. However, we need to think beyond our local machines to facilitate unattended testing, shared infrastructure, and to accommodate the reality of a world in which some developers suffer under highly restrictive IT policies, and in which some organizations—especially charities, non-profits, and businesses in the developing world—simply don't have the same degree of power and freedom with their local machines as others.

The requirement, therefore, is to be able to provision machines, install Chef, create and apply appropriate run lists, and then run Chef to bring the machines in line with our stated policy. This is a pretty in-depth process. Assuming the Chef code has been written, we still need to make the latest version of the code available, and then converge the node or nodes. We then need to be able to run some kind of test against the converged nodes, from a machine that behaves like an external client.

Virtualization has made this process much simpler than it was even 10 years ago, and excellent network APIs exist for many cloud providers, which makes automated provisioning as a part of the testing process well within our capabilities.

Provisioning is made much simpler with the use of virtualization-based technologies. The ability to create snapshots, roll forwards and backwards, or clone or freshly provision machines makes the setting up of a platform on which or against which to run tests an achievable aim. One variable to consider, however, is the number of machines required. If your infrastructure supports three or more different underlying platforms—such as two different Linux distributions and a flavor of BSD, or Windows—the requirement is now to be able to run and work with three machines. If these machines are to be reasonably responsive and performant, resources in terms of processor power and memory need to be appropriately allocated. Modern hardware brings this within reach, with multi-core laptops with 8G of memory now not uncommon, but cases where developer workstations are insufficiently powerful are still common, so alternative approaches need to be considered.

Feedback of Results

It's actually the speed of the tests that represents one of the biggest challenges. We want the feedback time to be sufficiently quick as to be rewarding and not frustrating.

The main constraining factor when testing Chef code, which impacts the speed of tests, is the time taken to convergence of the node during a Chef run. A complex cookbook, making use of search, and using the Opscode Hosted Chef platform, could take a minute or more to run per node. Unit tests that take three minutes to run have a high likelihood

of being skipped or ignored, so working out the most effective way to converge nodes is highly significant.

Related to the running of the tests is the mechanism for extracting the results. Again, at small scale, running tests and observing the results is trivial. However, storing these results for later analysis, or running the tests and being able to see the results some hours later requires more thought.

In order to achieve continuous integration, we need to make the connection between a line of text on a console indicating a failed test, and something that an automated test runner can understand to mean that the build failed.

Finally, in line with our desire to encourage and enforce shared standards for quality, it's necessary to provide a means of both defining and assessing compliance against those standards. This has both a community and technological aspect—the standards need to be discussed and agreed, and then an approach to validating code against those standards that is flexible, fast, and automated is required.

Having drawn up a framework for Test-driven infrastructure, we now turn to building a toolkit.

CHAPTER 7

Test-Driven Infrastructure: A Recommended Toolchain

This book began with two philosophical foundations:

1. Infrastructure can and should be treated as code.
2. Infrastructure developers should adhere to the same principles of professionalism as other software developers.

It then outlined how to go about endeavoring to fulfill the second by the mechanism of practicing the first.

We've provided a thorough introduction to the core principles and primitives of Chef, and we've explored them through the means of a thorough set of worked examples.

We then set the groundwork for the program of developing the highest standards of software professionalism by presenting a directed but thorough introduction to the Ruby programming language, and the principles and practices of test-driven and behavior-driven development.

We set out a manifesto and framework around which to organize ourselves as we seek to apply these TDD and BDD principles and practices to the paradigm of infrastructure as code.

In this closing chapter, we give a clear recommendation and strategy for top-to-bottom test-driven infrastructure by illustrating and evaluating the leading tools and workflows available to assist us in our quest at this point in the evolution of this young but exciting discipline.

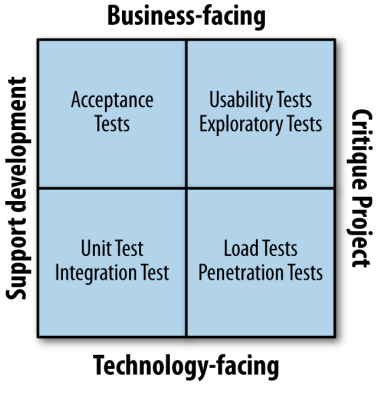
Tool Selection

> There is surely nothing quite so useless as doing with great efficiency that which should not be done at all.
>
> — Peter Drucker

Our selection of tools and recommended workflow and approach needs to be informed by a holistic perspective on testing (and building) software in general. Underpinning our every decision must be the core mantra that the purpose of our testing endeavors is to ensure that not only do we build the thing right, but that we build the right thing.

We need to check that our infrastructure code works—that it does what we intended, but also that our infrastructure delivers the functionality that is required. Beyond these considerations, our testing strategy must also account for ongoing maintainability; we need to be confident in our ability to refactor, share, and reuse our work. This moves the conversation beyond simplistic unit testing to be an all-encompassing testing strategy.

When thinking about what a testing strategy should look like, I find Brian Marick's testing quadrant diagram to be particularly helpful.



A successful infrastructure-testing strategy must encapsulate behaviors in all four of the quadrants; that is, it must include activities directed around supporting the engineering effort, both in terms of the people doing the work, and the technology and implementation, but also in terms of supporting the business it serves, in terms of the core stakeholders, but also at the highest level, in terms of verifying that value has been delivered to the business.

There are some observations associated with activities in this matrix. Activities towards the left—those that support the engineering effort—tend to lend themselves to automation. Activities towards the top—those that face the business and the stakeholder—tend to be more resource-intensive, but ultimately deliver the most value.

Tasks such as load testing, penetration testing, usability testing, and exploratory testing are really out of the scope of this book. With that in mind, of the plethora of tools and approaches available within the world of infrastructure testing, I'm aiming to recommend a subset that will assist us in our activities in quadrants one and three (i.e., tasks that support the delivery of infrastructure, rather than critique it, but face both the business and the engineering sides).

Let's quickly clarify terms before proceeding to a deeper discussion of the tooling that supports their implementation.

Unit Testing

Within quadrant three, we have traditional unit tests and integration tests. A simple definition of a unit test is:

> The execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.
>
> — by Steve McConnell
> *"Code Complete" (Microsoft Press)*

This simple definition suffices to describe what a unit test looks like. However, I think it's valuable to express explicitly what a unit test does *not* look like. A test is not a unit test if:

- The test is not automated and not repeatable.
- It is difficult to implement.
- It isn't kept around for future use.
- Only a few informed people know how to run it.
- It requires more than one step to run.
- It takes more than a few seconds.

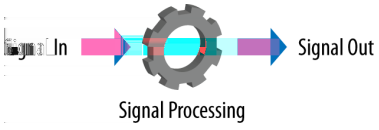
Integration Testing

Where unit tests are designed to test individual units of code (in as much isolation as possible), integration tests explore how the code units interact. This could be as simple as removing any mocks and stubs, but it could also involve crafting a special test that explicitly tests relationships between components.

Both have value, and both need to be in place.

When thinking about unit and integration tests for Chef, it makes sense to think about testing in terms of signal in, signal processing, and signal out. Signal input asks the question, "Did we send Chef the correct command?" Signal processing asks the

question, "Did Chef carry out my instructions?" Signal output asks the question: "Did my expressed intent, executed by Chef, deliver the intended result?"



Chef itself is fully tested—we don't need to test that Chef providers will do what we ask. But we do need to check that we asked Chef to do the right thing, and that what Chef did was what we actually wanted.

For testing signal input, I recommend *Chefspec*. For testing signal output, I recommend running tests using *Test Kitchen*, using a framework that allows you to be effective. I think there's significant value using the same expectation syntax for signal in and signal out, so I offer as an option the use of *Serverspec*, but also give an example of a different approach, using *Bats*. Honorable mention goes to *Minitest Handler* on account of its ease and speed of use.

Acceptance Testing

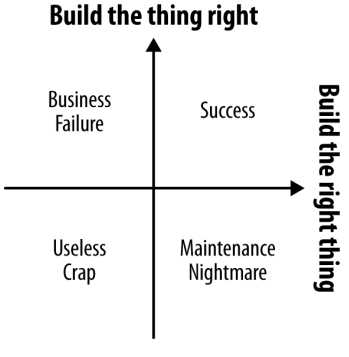
Acceptance tests describe a requirement or a feature. They are a clear indicator of success or completion—passing acceptance tests are an unambiguous definition of "done." They involve close collaboration with stakeholders and clarify the expectations of the end users. In his book, *Lean-Agile Acceptance Test-Driven Development* (Addison-Wesley), Ken Pugh gives as an example the following kind of discussion:

```
Ken: Does anyone want a fast car?
Student: Yes please
Ken: Stand by...OK, here's a fast car! It goes 0-60 in 20 seconds!
Student: That's not fast!
Ken: Oh...I thought that was fast.  Give me a test that would indicate that the
car is fast?
Student: It does 0-60 in 4.5 seconds.
Ken: Stand by...OK, here's the fast car! It does 0-60 in 4.5 seconds.  By the
way, the top speed is 60 mph.
Student: That's not fast!
Ken: Oh...OK, give me a test that would indicate that the car is fast?
Student: The top speed is 150 mph.
Ken: Stand by...OK, here's the fast car!  0-60 in 4.5 seconds, top speed 150
mph, 60-150 in 2 minutes.
```

The point being made is that without customer-facing acceptance tests, it's difficult to know if we've built the right thing. Leaving an engineer to make that decision is probably not a great idea. Something similar happens when building infrastructure. We're never building infrastructure in a vacuum, there's always a reason for the infrastructure, and the person who's going to use it almost certainly has some requirements. Leaving the requirements down to the implementor opens up a high risk of the endeavor being wasteful. To give a trivial example:

```
Me: Do you need a load balancer?
Stakeholder:  Yes!
Me: <some time later> There, a load balancer! It uses a simple round-robin algo-
rithm.
Stakeholder: Oh...I wanted to balance based on number of sessions.
Me: Oh...<replaces load balancer> There, a load balancer!
Stakeholder: Oh...I wanted to terminate SSL.
Me: Oh...
```

The following diagram, from Gojko Adžić, illustrates the importance of striving to build both the right thing and the thing right—a philosophy that is every bit as applicable in the world of infrastructure as code as it is in the world of building the software that runs on top of the infrastructure.



Speaking from personal experience, as a consultant specializing in building automated infrastructures, and having worked with dozens of clients, I've seen a number of expensive failures and presided over more than one myself. It's all too easy to spend time, and the customer's money, building a perfect infrastructure that doesn't do the right thing. I've also seen cases where the operations team has been forced into building a system that meets business requirements but is a nightmare to maintain. Succeeding in infrastructure development means striking the right balance, to land in quadrant two, and deliver success.

Striking this balance demands collaboration to drive out precise examples that encapsulate requirements, and making these examples the single source of truth. These examples become the documentation, the acceptance criteria, and the implementation plan—all in one place. This delivers the following advantages:

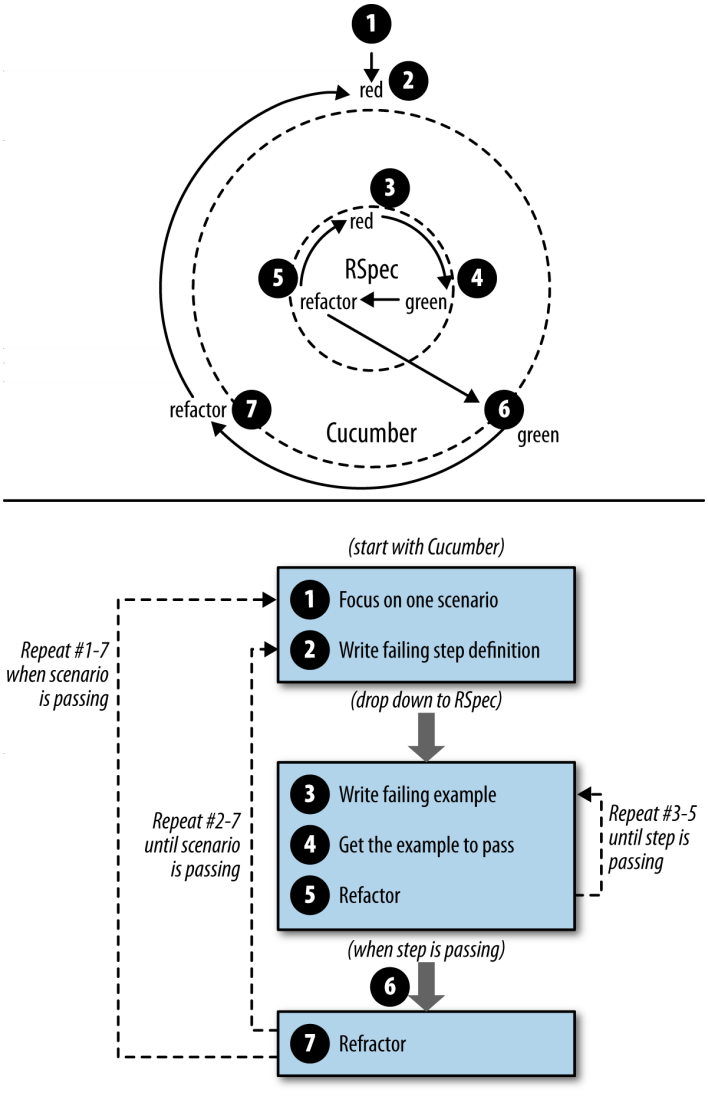
- Stakeholders and implementors have a common understanding of the requirements.
- Requirements are captured in a precise and unambiguous format.
- Documentation that enables change remains fresh and meaningful.
- An objective definition of "done" is universally understood.

The building of automated acceptance tests that represent these requirements and can demonstrate repeatably that the right thing has been built, from an external perspective, requires a different approach to test writing and a different set of tools.

For acceptance testing, I recommend *Cucumber*, paired with the orchestration capabilities of *Test Kitchen*. The enabling agent—which makes it easy for Cucumber and Test Kitchen to work together—is a theoretically simple task, but at present there isn't an obvious stand-out exemplar, so I've written one, which I've called *Leibniz*.

Testing Workflow

I think at this stage it makes sense to describe the workflow that I feel best delivers results against our desired objectives. I am much indebted to the excellent description of the Red/Green/Refactor workflow described by David Chelimsky in *"The RSpec Book"* (Pragmatic Bookshelf). This is the standard methodology used by BDD practitioners:



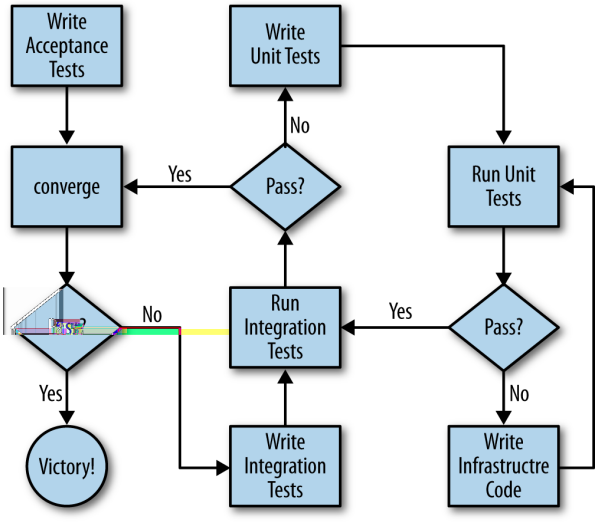
As engineers we navigate a continuously iterative cycle of testing and development, until we have met the acceptance criteria. The three phases are:

Red
: We've written a failing test, which describes the behavior of a feature we need to implement, but we haven't written the code.

Green
: We've written just enough code make the test pass.

Refactor
: Having got the feature to work and the test to pass, we refactor the code to improve its structure, maintainability, or performance, without altering its external behavior.

It's accepted practice to navigate this cycle from the outside-in; that is, to start with the acceptance tests, and move in to unit tests, and then back out again. I propose a variation on this patten for infrastructure code.



By this approach, we would structure our workflow as follows:

1. Capture examples that specify external acceptance criteria, from the perspective of a consumer of the infrastructure we are building.
2. Write executable specifications using Cucumber.
3. Watch them fail.
4. Write integration tests that describe the intended behavior of a machine once a run list has been applied to it, from the perspective of an engineer looking at the machine itself.
5. Watch them fail.
6. Write unit tests that describe the messages we pass to Chef, and the state of the resource collection, from the perspective of a recipe author.
7. Watch them fail.
8. Write the recipe to make the unit tests pass.
9. Navigate back up the hierarchy until all tests pass.

10. Refactor.

Before examining the recommended toolchain that helps us achieve this approach, we need first to discuss some supporting tooling, which will assist us in our quest.

Supporting Tools: Berkshelf

It is widely accepted and understood that effective use of Chef requires the employment of a dependency management system. This is a common requirement in the software development world. Berkshelf is the leading solution in the Chef community at present.

Overview

At the conclusion of our introduction to Ruby, we discussed Bundler—a dependency solver and portable sandboxing tool for Rubygems. If you understood the principles of Bundler, the basic idea of *Berkshelf* should be very easy to grasp. Berkshelf is, at its most basic level, Bundler for cookbooks. Let's review the twin goals of Bundler:

- Ensure that the appropriate dependencies are installed for a given problem without encountering unpleasant ordering issues or cyclical dependencies.
- Ensure code can be shared between other developers, or other machines or environments, and be confident the code and its dependencies will behave in the same way.

Berkshelf solves these problems for cookbooks, only in the place of a Gemfile, Berkshelf has a *Berksfile*.

You'll remember from our introduction to Chef that as soon as we started relying on recipes from other cookbooks and made use of the `include_recipe` resource, we needed to update the *metadata.rb* file to specify an explicit dependency on the cookbook that provided the recipe or LWRP that we wanted. That's perfectly reasonable and to be expected. However, my expectation is that you pretty soon got tired of having to solve cookbook dependencies manually and recursively. Similarly, having to upload cookbooks in the right order, one at a time, was equally tiresome. Berkshelf takes these pains away by providing a local dependency solving solution, and by functioning as a Chef API client for uploading cookbooks.

Berkshelf provides considerably more functionality than this. It's pivotal to an entire Chef development workflow, dubbed "The Berkshelf Way" by the group of developers from Riot Games, the company behind Berkshelf, who open sourced it and its component tools. We'll touch on many of these capabilities and concepts as we explore the tooling in this chapter.

Getting Started

Berkshelf is distributed as a Rubygem. This gives you the opportunity simply to install it with `gem install berkshelf`, or ensure it's installed as part of your Ruby/Developer cookbooks and/or roles. The other obvious approach is to use Bundler.

```
$ gem install berkshelf
Fetching: nio4r-0.4.6.gem (100%)
Building native extensions.  This could take a while...
Successfully installed nio4r-0.4.6
Fetching: celluloid-io-0.14.1.gem (100%)
Successfully installed celluloid-io-0.14.1
Fetching: ridley-1.0.1.gem (100%)
Successfully installed ridley-1.0.1
Fetching: safe_yaml-0.9.3.gem (100%)
Successfully installed safe_yaml-0.9.3
Fetching: test-kitchen-1.0.0.alpha.7.gem (100%)
Successfully installed test-kitchen-1.0.0.alpha.7
Fetching: berkshelf-2.0.1.gem (100%)
Successfully installed berkshelf-2.0.1
Installing ri documentation for nio4r-0.4.6
Installing ri documentation for celluloid-io-0.14.1
Installing ri documentation for ridley-1.0.1
Installing ri documentation for safe_yaml-0.9.3
Installing ri documentation for test-kitchen-1.0.0.alpha.7
Installing ri documentation for berkshelf-2.0.1
6 gems installed
```

Once Berkshelf is installed, access the help by running the following:

```
$ berks help
Commands:
   berks apply ENVIRONMENT      # Apply the cookbook version locks from Berks-
file.lock to a Chef environment
  berks configure             # Create a new Berkshelf configuration file
   berks contingent COOKBOOK    # List all cookbooks that depend on the given
cookbook
  berks cookbook NAME         # Create a skeleton for a new cookbook
   berks help [COMMAND]          # Describe available commands or one specific
command
  berks init [PATH]           # Initialize Berkshelf in the given directory
  berks install               # Install the cookbooks specified in the Berksfile
  berks list                   # List all cookbooks (and dependencies) specified
in the Berksfile
   berks outdated [COOKBOOKS]   # Show outdated cookbooks (from the community
site)
  berks package [COOKBOOK]     # Package a cookbook (and dependencies) as a tar-
ball
  berks shelf SUBCOMMAND      # Interact with the cookbook store
  berks show [COOKBOOK]        # Display name, author, copyright, and dependency
information about a cookbook
   berks update [COOKBOOKS]     # Update the cookbooks (and dependencies) speci-
fied in the Berksfile
```

```
  berks upload [COOKBOOKS]    # Upload the cookbook specified in the Berksfile
to the Chef Server
  berks version               # Display version and copyright information

Options:
  -c, [--config=PATH]    # Path to Berkshelf configuration to use.
  -F, [--format=FORMAT]  # Output format to use.
                         # Default: human
  -q, [--quiet]          # Silence all informational output.
  -d, [--debug]          # Output debug information
```

Example

Find the *irc* cookbook we created in Chapter 3. Change into its top-level directory, and have a look at the files:

```
$ ls
CHANGELOG.md  files  metadata.rb  README.md  recipes
```

Now, let's initialize the cookbook, so we can manage its dependencies with Berkshelf:

```
$ berks init
      create  Berksfile
      create  Thorfile
      create  chefignore
      create  .gitignore
         run  git init from "."
      create  Gemfile
      create  .kitchen.yml
      append  Thorfile
      create  test/integration/default
      append  .gitignore
      append  .gitignore
      append  Gemfile
      append  Gemfile
You must run `bundle install' to fetch any new gems.
      create  Vagrantfile
Successfully initialized
```

Wow, that did a lot! Some of these files will look familiar; we know about Vagrantfiles and Gemfiles, and I've already indicated that Berkshelf uses a Berksfile. We've had a look at Thor—it, too, has a file of its own. The *.gitignore* and *chefignore* files are simply there to blacklist files and directories from being uploaded to the Chef server or checked into version control. That leaves us with the *.kitchen.yml* and *test/integration/default* directory. We'll cover these later in this chapter.

Let's have a look at the Gemfile and the Berksfile:

```
$ cat Gemfile
source 'https://rubygems.org'

gem 'berkshelf'
```

```
gem 'test-kitchen', :group => :integration
gem 'kitchen-vagrant', :group => :integration

$ cat Berksfile
site :opscode

metadata
```

The Gemfile shows three dependencies. Berkshelf itself, plus two others. We'll discuss the kitchen-related files when we get to our section on *Test Kitchen*. The main thing of note here is the use of the `:integration` group. This allows us to install the core dependency, Berkshelf, on a continuous integration server, where we might want to solve dependencies, and carry out lint and static analysis tests—and perhaps fast unit tests—but where we don't want to ever run integration tests, which is the purpose of Test Kitchen. This uses Bundler's `--without` flag, allowing us to specify to install the dependencies, omitting certain groups.

The Berksfile follows the same pattern as the Gemfile. We specify a source—in this case, we're stating that by default we want to pull in dependencies from the Opscode community site. The `metadata` line delegates dependencies to the cookbook *metadata.rb* file. It's effectively saying, "I'm a cookbook. If you want to know my dependencies, check out my metadata file."

Unsurprisingly, Berkshelf follows Bundler in having an `install` command:

```
$ berks install
Using irc (0.1.0) at path: '/home/tdi/chef-repo/cookbooks/irc'
Using yum (2.2.2)
```

Again, like Bundler, Berkshelf recognizes that it already has local cookbooks that satisfy the dependency, so it "uses" them. Note that these cookbooks, and all other versions of the cookbook ever used by Berkshelf, are all stored in a conventional directory (*.berkshelf*, in this case). If there were not local copies available, it would download them from the community site.

At this stage, the similarities with Bundler evaporate, and we start to see some of the individual power and characteristics of Berkshelf. Reviewing the commands in the help text, I would draw your attention to three in particular:

- `berks configure # Create a new Berkshelf configuration file`
- `berks upload [COOKBOOKS] # Upload the cookbook specified in the Berks file to the Chef Server`
- `berks apply ENVIRONMENT # Apply the cookbook version locks from Berks file.lock to a Chef environment`

Berkshelf and Vagrant

Berkshelf provides some of the functionality we found in `Knife` to interact with a Chef server. Now, remember, everything in Chef is an API client; this means we need to configure Berkshelf as an API client. Berkshelf provides and uses its own API client library, *Ridley*. We could create a new key pair, but it's simpler just to use the key pair we used ourselves, when we used `Knife`.

The `berks configure` command will make educated guesses based on the content of your *knife.rb* file. This will be fine in our case. Let's run the command, and accept all the defaults:

```
$ berks configure
Enter value for chef.chef_server_url (default: 'https://api.opscode.com/organi-
zations/hunterhayes'):
Enter value for chef.node_name (default: 'tdiexample'):
Enter value for chef.client_key (default: '/home/tdi/chef-repo/.chef/tdiexam-
ple.pem'):
Enter value for chef.validation_client_name (default: 'hunterhayes-validator'):
Enter value for chef.validation_key_path (default: '/home/tdi/chef-repo/.chef/
hunterhayes-validator.pem'):
Enter value for vagrant.vm.box (default: 'Berkshelf-CentOS-6.3-x86_64-minimal'):
Enter value for vagrant.vm.box_url (default: 'https://dl.dropbox.com/u/31081437/
Berkshelf-CentOS-6.3-x86_64-minimal.box'):
Config written to: '/home/tdi/.berkshelf/config.json'
```

This all looks plausible. The only values I would draw your attention to are those for `vagrant.vm`. These values exist because Berkshelf is designed to interact with Vagrant, such that when running `vagrant up`, any cookbook dependencies are solved and made available on the machine under test, and the default recipe is converged. Now, we already downloaded a Vagrant box from the Opscode Bento project. We should use that in preference to the default. We can find out its name by running `vagrant box list`, and then we can edit the config file:

```
$ vagrant box list
opscode-ubuntu-10.04 (virtualbox)
opscode-ubuntu-12.04 (virtualbox)
opscode-centos-6.4 (virtualbox)
opscode-centos-5.9 (virtualbox)
```

On this particular machine, I have four machines, provided by the Vagrant/VirtualBox combination. Let's stick with the CentOS 6.4 machine. Unfortunately, the output of the `berks configure` command seems to be a bit hard to read:

```
{"chef":{"chef_server_url":"https://api.opscode.com/organizations/
hunterhayes","validation_client_name":"hunterhayes-
validator","validation_key_path":"/home/tdi/chef-repo/.chef/hunterhayes-
validator.pem","client_key":"/home/tdi/chef-repo/.chef/tdiexam-
ple.pem","node_name":"tdiexample"},"cookbook":{"copy-
right":"YOUR_NAME","email":"YOUR_EMAIL","license":"reserved"},"allowed_licen-
ses":[],"raise_license_exception":false,"vagrant":{"vm":{"box":"Berkshelf-
```

```
CentOS-6.3-x86_64-minimal","box_url":"https://dl.dropbox.com/u/31081437/
Berkshelf-CentOS-6.3-x86_64-minimal.box","forward_port":{},"network":{"bridg-
ed":false,"hostonly":"33.33.33.10"},"provision":"chef_solo"}},"ssl":{"veri-
fy":true}}
```

But we can fix this easily enough:[1]

```
$ python -mjson.tool < /home/tdi/.berkshelf/config.json > /home/tdi/.berkshelf/
config.json.readable
$ grep box /home/tdi/.berkshelf/config.json.readable
            "box": "Berkshelf-CentOS-6.3-x86_64-minimal",
              "box_url": "https://dl.dropbox.com/u/31081437/Berkshelf-CentOS-6.3-
x86_64-minimal.box",
```

Open the file in an editor, remove the `box_url` line, and update the `box` entry. This will ensure that the next time `berks init` is run, it will set the Vagrantfile to use our favored box. We're going to need to make the same edit to the Vagrantfile within the irc cookbook: remove the `box_url` entry and change the `box` entry. While we're there, we should add the config entry, which tells the Vagrant machine to install the latest Chef client from the omnibus package. This leaves our Vagrantfile looking like this:

```
$ grep -v '^$' Vagrantfile |grep -v '^ *#'
Vagrant.configure("2") do |config|
  config.omnibus.chef_version = :latest
  config.vm.hostname = "irc-berkshelf"
  config.vm.box = "opscode-centos-6.4"
  config.vm.network :private_network, ip: "33.33.33.10"
  config.ssh.max_tries = 40
  config.ssh.timeout   = 120
  config.berkshelf.enabled = true
  config.vm.provision :chef_solo do |chef|
    chef.json = {
      :mysql => {
        :server_root_password => 'rootpass',
        :server_debian_password => 'debpass',
        :server_repl_password => 'replpass'
      }
    }
    chef.run_list = [
        "recipe[irc::default]"
    ]
  end
end
```

All that remains to do is to ensure the `vagrant-berkshelf` plug-in is installed, and then run `vagrant up` to watch the magic!

1. We can do this with Ruby easily enough, too: `ruby -e "require 'json'; JSON.pretty_generate(IO.read('/home/tdi/.berkshelf/config.json'\))"`

```
$ vagrant plugin install vagrant-berkshelf
...
$ vagrant plugin install vagrant-omnibus
...
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'opscode-centos-6.4'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[Berkshelf] This version of the Berkshelf plugin has not been fully tested on
this version of Vagrant.
[Berkshelf] You should check for a newer version of vagrant-berkshelf.
[Berkshelf] If you encounter any errors with this version, please report them
at https://github.com/RiotGames/vagrant-berkshelf/issues
[Berkshelf] You can also join the discussion in #berkshelf on Freenode.
[Berkshelf] Updating Vagrant's berkshelf: '/home/tdi/.berkshelf/vagrant/
berkshelf-20130607-26262-mra02l'
[Berkshelf] Using irc (0.1.0) at path: '/home/tdi/chef-repo/cookbooks/irc'
[Berkshelf] Using yum (2.2.2)
[default] Fixed port collision for 22 => 2222. Now on port 2202.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2202 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Ensuring Chef is installed at requested version of 11.4.4.
[default] Chef 11.4.4 Omnibus package is not installed...installing now.
Downloading Chef 11.4.4 for el...
Installing Chef 11.4.4
warning: /tmp/tmp.OQLalPCu/chef-11.4.4.x86_64.rpm: Header V4 DSA/SHA1 Signa-
ture, key ID 83ef826a: NOKEY
Preparing...                ##################################################
chef                        ##################################################
Thank you for installing Chef!
[default] Setting hostname...
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
[default] -- /tmp/vagrant-chef-1/chef-solo-1/cookbooks
[default] Running provisioner: chef_solo...
Generating chef JSON and uploading...
Running chef-solo...
[2013-06-07T08:38:25+00:00] INFO: *** Chef 11.4.4 ***
[2013-06-07T08:38:25+00:00] INFO: Setting the run_list to ["recipe[irc::de-
fault]"] from JSON
[2013-06-07T08:38:25+00:00] INFO: Run List is [recipe[irc::default]]
[2013-06-07T08:38:25+00:00] INFO: Run List expands to [irc::default]
[2013-06-07T08:38:25+00:00] INFO: Starting Chef Run for irc-berkshelf
```

```
[2013-06-07T08:38:25+00:00] INFO: Running start handlers
[2013-06-07T08:38:25+00:00] INFO: Start handlers complete.
[2013-06-07T08:38:25+00:00] INFO: Processing yum_key[RPM-GPG-KEY-EPEL-6] action
add (yum::epel line 22)
[2013-06-07T08:38:25+00:00] INFO: Adding RPM-GPG-KEY-EPEL-6 GPG key to /etc/pki/
rpm-gpg/
[2013-06-07T08:38:25+00:00] INFO: Processing package[gnupg2] action install
(/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/key.rb line 32)
[2013-06-07T08:38:32+00:00] INFO: Processing execute[import-rpm-gpg-key-RPM-GPG-
KEY-EPEL-6] action nothing (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/
providers/key.rb line 35)
[2013-06-07T08:38:32+00:00] INFO: Processing remote_file[/etc/pki/rpm-gpg/RPM-
GPG-KEY-EPEL-6] action create (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/
providers/key.rb line 61)
[2013-06-07T08:38:32+00:00] INFO: remote_file[/etc/pki/rpm-gpg/RPM-GPG-KEY-
EPEL-6] updated
[2013-06-07T08:38:32+00:00] INFO: remote_file[/etc/pki/rpm-gpg/RPM-GPG-KEY-
EPEL-6] mode changed to 644
[2013-06-07T08:38:32+00:00] INFO: remote_file[/etc/pki/rpm-gpg/RPM-GPG-KEY-
EPEL-6] sending run action to execute[import-rpm-gpg-key-RPM-GPG-KEY-EPEL-6]
(immediate)
[2013-06-07T08:38:32+00:00] INFO: Processing execute[import-rpm-gpg-key-RPM-GPG-
KEY-EPEL-6] action run (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/
key.rb line 35)
[2013-06-07T08:38:33+00:00] INFO: execute[import-rpm-gpg-key-RPM-GPG-KEY-
EPEL-6] ran successfully
[2013-06-07T08:38:33+00:00] INFO: Processing yum_repository[epel] action create
(yum::epel line 27)
[2013-06-07T08:38:33+00:00] INFO: Adding and updating epel repository in /etc/
yum.repos.d/epel.repo
[2013-06-07T08:38:33+00:00] WARN: Cloning resource attributes for yum_key[RPM-
GPG-KEY-EPEL-6] from prior resource (CHEF-3694)
[2013-06-07T08:38:33+00:00] WARN: Previous yum_key[RPM-GPG-KEY-EPEL-6]: /tmp/
vagrant-chef-1/chef-solo-1/cookbooks/yum/recipes/epel.rb:22:in `from_file'
[2013-06-07T08:38:33+00:00] WARN: Current yum_key[RPM-GPG-KEY-EPEL-6]: /tmp/
vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/repository.rb:85:in `re-
po_config'
[2013-06-07T08:38:33+00:00] INFO: Processing yum_key[RPM-GPG-KEY-EPEL-6] action
add (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/repository.rb line
85)
[2013-06-07T08:38:33+00:00] INFO: Processing execute[yum-makecache] action noth-
ing (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/repository.rb line
88)
[2013-06-07T08:38:33+00:00] INFO: Processing ruby_block[reload-internal-yum-
cache] action nothing (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/
repository.rb line 93)
[2013-06-07T08:38:33+00:00] INFO: Processing template[/etc/yum.repos.d/
epel.repo] action create (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/provid-
ers/repository.rb line 100)
[2013-06-07T08:38:33+00:00] INFO: template[/etc/yum.repos.d/epel.repo] updated
content
[2013-06-07T08:38:33+00:00] INFO: template[/etc/yum.repos.d/epel.repo] mode
```

```
changed to 644
[2013-06-07T08:38:33+00:00] INFO: template[/etc/yum.repos.d/epel.repo] sending
run action to execute[yum-makecache] (immediate)
[2013-06-07T08:38:33+00:00] INFO: Processing execute[yum-makecache] action run
(/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/repository.rb line 88)
[2013-06-07T08:38:42+00:00] INFO: execute[yum-makecache] ran successfully
[2013-06-07T08:38:42+00:00] INFO: template[/etc/yum.repos.d/epel.repo] sending
create action to ruby_block[reload-internal-yum-cache] (immediate)
[2013-06-07T08:38:42+00:00] INFO: Processing ruby_block[reload-internal-yum-
cache] action create (/tmp/vagrant-chef-1/chef-solo-1/cookbooks/yum/providers/
repository.rb line 93)
[2013-06-07T08:38:42+00:00] INFO: ruby_block[reload-internal-yum-cache] called
[2013-06-07T08:38:42+00:00] INFO: Processing user[tdi] action create (irc::de-
fault line 11)
[2013-06-07T08:38:42+00:00] INFO: user[tdi] created
[2013-06-07T08:38:42+00:00] INFO: Processing package[irssi] action install
(irc::default line 18)
[2013-06-07T08:38:46+00:00] INFO: package[irssi] installing irssi-0.8.15-5.el6
from base repository
[2013-06-07T08:38:50+00:00] INFO: Processing directory[/home/tdi/.irssi] action
create (irc::default line 26)
[2013-06-07T08:38:50+00:00] INFO: directory[/home/tdi/.irssi] created directo-
ry /home/tdi/.irssi
[2013-06-07T08:38:50+00:00] INFO: directory[/home/tdi/.irssi] owner changed to
901
[2013-06-07T08:38:50+00:00] INFO: directory[/home/tdi/.irssi] group changed to
901
[2013-06-07T08:38:50+00:00] INFO: Processing cookbook_file[/home/tdi/.irssi/
config] action create (irc::default line 31)
[2013-06-07T08:38:50+00:00] INFO: cookbook_file[/home/tdi/.irssi/config] owner
changed to 901
[2013-06-07T08:38:50+00:00] INFO: cookbook_file[/home/tdi/.irssi/config] group
changed to 901
[2013-06-07T08:38:50+00:00] INFO: cookbook_file[/home/tdi/.irssi/config] cre-
ated file /home/tdi/.irssi/config
[2013-06-07T08:38:50+00:00] INFO: Chef Run complete in 25.150121171 seconds
[2013-06-07T08:38:50+00:00] INFO: Running report handlers
[2013-06-07T08:38:50+00:00] INFO: Report handlers complete
```

Well, that's pretty impressive! In the time it would have taken us to read the metadata file of a single machine—let alone upload all the cookbooks, connect to the machine, run `chef-client`, and wait for it to finish—we've built a brand new machine from scratch, installed Chef, solved dependencies, and converged a node.

We can connect to the machine as before, using `vagrant ssh`, and check out the configuration. This increase of speed in the feedback loop is vital if we're to make testing of infrastructure mainstream.

One caveat here: the current Vagrant machine is using `chef-solo` rather than `chef-client`. Frankly, for testing functionality within a single cookbook, this is frequently sufficient, and the speed of feedback is a tremendous bonus. However, if a convergence

against a Chef server is needed, Vagrant can be easily configured to use `chef-client`. Also worthy of attention is `chef-zero`—an in-memory implementation of the Chef server, designed for rapid testing against a real API. As this is a very new project, I haven't explored it in sufficient detail to be able to discuss it with authority, but I recommend at least checking out the Chef Zero project.

Berkshelf and Chef environments

The second command I wanted to draw your attention to was the `berks upload` command. You'll recall when we first began interacting with the Chef server, using `Knife`, we used `knife cookbook upload`. This was a little frustrating if we didn't upload the cookbooks in the correct order. Berkshelf combines the package set functionality of Bundler with the cookbook uploading functionality of `Knife`. This means that once a set of cookbooks has been tested on a Vagrant machine, that set of cookbooks can be uploaded to the Chef server, dependencies and all, in a single command. Just like Bundler had a *Gemfile.lock*, if we now take a look in the base directory of the cookbook, we'll see a *Berkshelf.lock* file:

```
$ cat Berksfile.lock
{
  "sha": "6ef716553a56267bb3eb743ece483db8aa94cecb",
  "sources": {
    "irc": {
      "locked_version": "0.1.0",
      "constraint": "= 0.1.0",
      "path": "."
    },
    "yum": {
      "locked_version": "2.2.2"
    }
  }
}
```

This introduces a vitally important question in Chef. Once we've tested and approved cookbooks, and pushed them to a Chef server, how can we be confident that these are the cookbooks that will be used in perpetuity, or at least until we decide to introduce a change?

At the same time, it is likely that we will be enhancing, fixing, or otherwise refactoring perhaps the same cookbooks, following the test-first paradigm explored in this book. In order to protect against the cookbooks under test interfering with our production systems, Chef provides a mechanism for specifying exactly which version of a cookbook should be used for machines in this environment. Chef also supports the idea of *freezing* cookbooks, to prevent them from being accidentally updated or altered once uploaded to a server. This mechanism is referred to as *Chef Environments*.

Let's take a quick look at the node attributes of one our machines:

```
$ knife node show romanesco
Node Name:   romanesco
Environment: _default
FQDN:        romanesco
IP:          192.168.26.2
Run List:    role[debian], role[developer]
Roles:
Recipes:
Platform:    ubuntu 13.04
Tags:
```

Unless explicitly set, a node in Chef will belong to a default environment called *default*. In the default environment, nodes will simply use the most recently uploaded cookbook on the platform, regardless of version number or quality. There is no policy at all. Obviously this is a dangerous state of affairs, so it's considered best practice to manage the versioning of your cookbooks in such a way as to make it easy for you to set a policy determining which versions of your cookbooks are to be used in which environment.

When you feel you have cookbooks and recipes that are of production quality, create an environment to enforce safe version constraints for machines whose stability is vital. Once the node attribute of the servers you feel should have these stable, reliable cookbooks has been set, they will not get any other versions, and the versions in use can be frozen, so they aren't accidentally overwritten.

A small aside on the name "environments": I feel that the term "environment" is one of those rather overloaded terms in our industry. When I work with clients, and they describe environments to me, they are usually referring to phases in the application lifecycle and use names such as "development," "staging," "uat," "perftest," or "preprod." It's pretty clear that the comparison between these environments is a function of the version of the application deployed on them, and the type of people who will be using them. By contrast, the problem domain that Chef environments addresses is related primarily to the ability to set and enforce version constraints on the infrastructure code —the code that delivers the core platform upon which the "development" or "staging" or "live" environments are deployed. I think this namespace collision is both unfortunate and confusing. We're not really talking about the same kinds of environments at all. While there may well be differences in the way in which the staging, development, and production systems are *configured*, the core functionality and behavior of the Chef code should actually be fundamentally identical between "development," "staging," and "live." For this reason, I prefer to think of Chef environments more in terms of "testing" and "stable," or perhaps, to borrow vocabuary from Maven, "RELEASE" and "SNAPSHOT." If you're familiar with Linux distribution development, you'll probably recognize this model as being that around which the Debian project package maintainers organize. This approach to environments takes cookbooks that are known to be stable, production-ready, and trusted and sets and freezes their known versions. Development of new features and bug fixing can take place in the testing environment, pending

promotion to stable. Should there be a need to test multiple combinations of multiple versions, there's no limit to the number of environments on a Chef server, so one could be created and mapped onto a project or branch.

Although this approach is the one I like most, as with pretty much all aspects of Chef, there is great flexibility and plenty of opportunity to use a different model. For example, if you are attracted to using environments in Chef in a way that models software development lifecycles akin to DEV→TEST→STAGING→PROD, this can be achieved. In this instance, use the cookbook `metadata,rb` as the place to lock dependencies. A straightforward approch to generating these dependencies is to take the output of `berks list` and simply transform the output to `depends` statements. This works particularly well with the "application cookbook" pattern, which we will discuss later in this chapter. There are clear advantages and disadvantages to both approaches. If I'm honest, I'd state that I am not convinced with the current environments implementation, and that the various approaches in place all feel a little uncomfortable. For one more approach, I recommend you take a look at Dan DeLeo's `knife boxer`. Born out of Dan's experience that "the default environments workflow makes me want to punch someone in the face," it offers an alternative approach based on Dan's rethinking of the whole environment's concept. I urge you to give thought to these alternatives, to experiment, and find the approach that works best for you. However, for the time being, we'll work with my model.

Chef has a DSL for creating and managing environments. Simply change into the environment's directory in your Chef repository and create a file named *stable.rb*. The DSL only needs a name, and zero or more cookbook constraints. These can be entered individually, or using the `cookbook_versions` method, which takes a hash of cookbook name and version:

```
name "stable"
description "Stable Cookbooks"
cookbook_versions({
  "irc"=>"= 0.1.0",
  "yum"=>"~> 2.2.0"
})
```

This specifies that in the stable environment only version 0.1.0 of the irc cookbook will be used; any version greater than or equal to 2.2.0 but less than 3.0.0 is acceptable for the yum cookbook. The version constraint syntax mirrors that of Rubygem's. To freeze a version of a cookbook, such that a developer is prevented from attempting to upload an altered version of the cookbook with the same version number, `--freeze` is appended to `knife cookbook upload`. By combining freezing and environments, you can be maximally confident that your production environments will be secure and safe.

Maintaining this environment is a case of keeping track of versions that you believe to be stable, maintaining their versions in a *stable.rb* environment file, and periodically running `knife environment from file` to upload the environment to the server. Chef

does provide an alternative mechanism via the `knife environment edit` command. This invocation, similar to `knife node edit`, allows the JSON representation of the Chef environment to be set in real time on the Chef server, over the API.

The `berks apply` command takes this complexity out of the environment management process:

```
$ knife environment create berks_stable
Created berks_stable
$ berks apply berks_stable
Using irc (0.1.0) at path: '/home/tdi/chef-repo/cookbooks/irc'
Using yum (2.2.2) at path
[tdi@tk01 irc]$ knife environment show berks_stable
chef_type:           environment
cookbook_versions:
  irc: 0.1.0
  yum: 2.2.2
default_attributes:
description:
json_class:          Chef::Environment
name:                berks_stable
override_attributes:
```

This has the effect of both setting and freezing the known stable cookbooks tested via Berkshelf.

Nodes are associated with environments by means of the `chef_environment` attribute. This must be explicitly set. The simplest way to ensure your production nodes are associated with the production environment is to specify it explicitly on the command line when provisioning or bootstrapping a machine. For more information on the process of provisioning a machine, see *http://docs.opscode.com/knife_bootstrap.html*.

Advantages and Disadvantages

Berkshelf was developed with the principal aim of simplifying the workflow required to interact with a Chef server in a production-responsible fashion. Its main advantage is that it provides slick usability with much less hassle than interacting with the server via a series of `knife` commands. A further advantage is that, within the Chef community, the Berkshelf tool, and the workflow patterns it encourages, have gained a lot of traction. You are likely to enjoy responsive support, and enthusiastic associates on the mailing lists and IRC channels.

If there's a disadvantage to Berkshelf, it's that the tool is integral to a highly opinionated set of principles around how cookbook development should take place, including a number of design patterns such as wrapper and library cookbooks. This approach is at odds with the way in which Chef has been traditionally taught and documented, and introduces a number of additional and new tools. We'll discuss this in more detail later in the chapter.

Summary and Conclusion

Berkshelf is fundamental to a whole philosophical approach to cookbook development. However, at its core, it's just a dependency solver and publishing tool. Whether you agree with the underlying philosophy about roles and wrapper cookbooks and libraries, it's a tool that will make your life easier, and should be in your toolkit. We'll assume its use henceforth.

Supporting Tools: Test Kitchen

In my preliminary comments about tool selection I identified Test Kitchen as a cornerstone. It's a great enabler, allowing us to automate the running of tests and the building of infrastructure. In this respect, it stands outside the workflow I describe but as one of its dependencies.

Overview

Test Kitchen is an orchestration tool—it runs tests across multiple nodes, converging them, verifying the resulting state across different platforms, and in complete isolation. It is designed to ensure an entirely clean state for testing. However, it isn't a testing tool, it doesn't makes sense to speak of writing tests "in" Test Kitchen. Rather, it provides a framework that enables you to verify the state of a node.

As cookbook developers, it's common to want a simple way to increase our confidence that our Chef code will work on a real platform in a real situation. For example, we'd like to be confident that our recipes will work repeatably against different operating systems or flavors of operating system, especially if our cookbooks are designed to work across a large number of platforms. My reference Linux platform is CentOS, but I try to ensure my cookbook will also work on Debian-derived systems. However, if a community member submits a pull request to add support for Arch Linux or Suse, I first want to be reassured that this enhancement doesn't introduce any regressions that the cookbook still works on CentOS and Ubuntu, and second, if I accept the pull request, I now have a responsibility to ensure that the cookbook continues to work on Arch Linux or Suse. I don't develop on or use these distributions very frequently, so the ability to be able to verify the functionality of the cookbook on all supported platforms is very advantageous.

Running these tests is expensive, in terms of time. Anything that can be done to automate and speed up the feedback loop is attractive. The foundational design goal for Test Kitchen was to provide the simplest, leanest orchestration framework possible that would deliver the requirements for continuously integrating cookbooks across multiple platforms. The simplest way to achieve this would be for the continuous integration server to be preinstalled with Rubygem, or have a Gemfile, followed by a bundle install. Then simply running a Rake or Thor task will carry out everything required to test the

cookbooks, with no need for further configuration unless the specific behavior of Test Kitchen needs to be altered. To support operation in continuous integration environments, the tasks finish with a non-zero exit code only if something in the testing process failed. Otherwise the explicit assumption is that the tests passed.

Although specifically built to facilitate continuous integration, Test Kitchen also provides a complete cookbook development testing environment for the user simply wishing to write cookbooks in an iterative and test-driven fashion.

The current version of Test Kitchen is effectively a complete rewrite of an earlier project. Although an excellent utility, the earlier version didn't meet the requirement of doing the simplest thing that could possibly work for CI. For example, it provided the apt cookbook and ran `apt-get update`, it installed Rsync, and assumed the use of Minitest Handler. All machines were created in serial, which meant the process of testing across many platforms was very time-consuming. The new version tackles these weaknesses and provides a complete framework for creating, provisioning, testing, and destroying a range of systems, rapidly, in parallel, and in a way that is designed to plug into continuous integration and deployment pipelines.

Getting Started

At the time of this writing, the 1.0 release of Test Kitchen is being prepared; by the time you read this, it'll be released. To make the tool available, simply add `test-kitchen` to your Gemfile. Since Berkshelf 2.0, Test Kitchen support is included in the Gemfile created by `berks cookbook` or `berks init`.

```
$ gem install test-kitchen
```

The primary context in which Test Kitchen operates is a single cookbook. The expectation is that it will be used to test and maintain the functionality of a given individual cookbook across multiple platforms, ensuring that the contract it claims to provide to infrastructure developers using the cookbook is honored.

Test Kitchen is driven entirely by a YAML file: a simple data representation format, which describes the configuration of systems and the tests we wish to run. If you've used *TravisCI*, this will be very familiar as an approach. The idea is to have an expressive way to define our testing strategy statically. It allows the developer to define that these tests should be run on these platforms, in these places. For example, we might wish to run all tests on EC2 with one exception, which we want to run on Rackspace. The file that describes this—*.kitchen.yml*—is, therefore, a testing manifest, and is explicitly not executable code.

Test Kitchen additionally has a command-line interface and is built upon Thor, meaning each command is also accessible as a Thor task, executable by a job runner or continuous delivery server.

Running `kitchen` without arguments gives the various options available:

```
 kitchen
Commands:
  kitchen console                          # Kitchen Console!
  kitchen converge [(all|<REGEX>)] [opts]  # Converge one or more instances
  kitchen create [(all|<REGEX>)] [opts]    # Create one or more instances
  kitchen destroy [(all|<REGEX>)] [opts]   # Destroy one or more instances
  kitchen driver                           # Driver subcommands
   kitchen driver create [NAME]              # Create a new Kitchen Driver gem
project
   kitchen driver discover                     # Discover Test Kitchen drivers
published on RubyGems
  kitchen driver help [COMMAND]            # Describe subcommands or one specif-
ic subcommand
  kitchen help [COMMAND]                   # Describe available commands or one
specific command
   kitchen init                               # Adds some configuration to your
cookbook so Kitchen can rock
  kitchen list [(all|<REGEX>)]             # List all instances
  kitchen login (['REGEX']|[INSTANCE])     # Log in to one instance
  kitchen setup [(all|<REGEX>)] [opts]     # Setup one or more instances
  kitchen test [all|<REGEX>)] [opts]       # Test one or more instances
  kitchen verify [(all|<REGEX>)] [opts]    # Verify one or more instances
  kitchen version                          # Print Kitchen's version information
```

The basic unit of reasoning in Test Kitchen is called an *instance*. An instance is composed of a *platform* and a *suite*. A platform is a combination of operating system, version, Chef version, architecture, and name. Conceivably it could also include a specification as to whether the instance is a physical or virtual machine. A suite is a run list with optional node attributes. It represents something we wish to test, for example, a Redis cookbook using a package or building from source.

Test Kitchen will then build a pairwise matrix of platforms and suites, resulting in the final set of instances that will be managed.

There are five lifecycle events in the existence of an instance:

`create`
: Brings an instance into existence and boots it, providing a system ready for work to begin

`converge`
: Installs Chef, creates a sandbox of what is needed for testing—roles, databags, attribute data—and uploads it to the instance. Next, Chef is run, either in `chef-solo` or `chef-zero` form.

`setup`
: Sets up a gem, called Busser, on the instance, which is responsible for preparing whatever test harness runners and plug-ins are needed to test the cookbook. The mechanism has no dependencies, and uses the embedded Ruby provided by Chef.

`verify`
: Runs any test suites that have been written. It will take no action if no tests are found. In the event of a test failure, the action will return with a non-zero exit code, suitable for signalling a broken build to a continuous integration service.

`destroy`
: Simply destroys the instance and returns the host system to a clean state.

Additionally, there is a master action—`test`—designed for clean CI purposes, which will run the destroy, create, converge, setup, and verify tasks, before finally running destroy once more.

Test Kitchen has the concept of drivers, which determine how and where the infrastructure required for the tests will be built. By default, the driver used is Vagrant, but Test Kitchen also supports cloud-based systems and is easily extensible.

Summary and Conclusion

We will cover detailed use of Test Kitchen shortly, with examples, when we look at using Serverspec and Bats for integration testing, but in summary, let me state that Test Kitchen is shaping up to be the one-stop-shop for cookbook testing. It is very actively developed and has considerable community traction. Support for Windows systems is under active development, and while improvements and enhancements are happening on a daily basis, the core design and API has been stable for a number of months.

Test Kitchen is the tool you should have at the very heart of your workflow. Because of its integration with Berkshelf and Vagrant, it replaces these as your primary interface to provisioning systems. It can easily be configured to use alternative provisioning backends, instead of Vagrant, and with the `chef-zero driver`, provides a complete client/server testing experience with a very fast feedback loop.

The Busser architecture makes Test Kitchen an effectively unlimited framework in terms of flexibility. The growing ecosystem of plug-ins can be observed by performing a search on *rubygems.org* for the string "busser-".

The high-level tasks available on the command line make the iterative process of creating, converging, verifying, and destroying simple and effective. And the ability to develop on a preferred platform and then test across a range of platforms all from the same interface is extremely convenient.

For further documentation and examples, I recommend looking at the project homepage on GitHub, and at Fletcher Nichol's cookbooks, particularly the *rbenv* and *razor* cookbooks.

Acceptance Testing: Cucumber and Leibniz

The first edition of this book introduced the fundamental idea of applying behavior-driven development (BDD) and the acceptance testing paradigm to infrastructure code. As the world of test-driven infrastructure has matured, the approach of the first infrastructure BDD tool, Cucumber-Chef, has been superseded by a more modular approach, which can be implemented by writing examples using Gherkin/Cucumber, and orchestrating the provisioning of infrastructure and running of tests using a separate tool—one such example is the newly released Leibniz project by the current author.

Overview

Testing classes and methods is trivial. Mature unit testing frameworks exist that make it very easy to write simple code test-first. As the complexity of the system under test increases and the requirement to test code that depends on other services arises, the frameworks become more sophisticated, allowing for the creation of mock services and the ability to stub out slow-responding or third-party interfaces. As a relevant aside, see "Mocks Aren't Stubs" by Martin Fowler for an excellent discussion of the difference between mocking and stubbing.

Writing integration tests that exercise the code end-to-end is an order of magnitude more involved. A successful integration testing strategy will require the use of specialist testing libraries for testing network services, GUI components, or JavaScript.

Testing code that builds an entire infrastructure is a different proposition altogether. Not only do we need sophisticated libraries of code to verify the intended behavior of our systems, we need to be able to build and install the systems themselves. Consider the following test:

```
Scenario: Bluepill restarts Unicorn
  Given I have a newly installed Ubuntu machine managed by Chef
  And I apply the Unicorn role
  And I apply the Bluepill role
  And the Unicorn service is running
  When I kill the Unicorn process
  Then within 2 seconds the Unicorn process should be running again
```

To test this manually we would need to find a machine, install Ubuntu on it, bootstrap it with Chef, apply the role, run Chef, log onto the machine, check Unicorn is running, kill Unicorn, then finally check that it has restarted. This would be tremendously time-consuming and expensive—so much so that nobody would do it. Indeed, almost no one

does because despite the benefits of being able to be sure that our recipe does as it is supposed to, the cost definitely outweighs the benefit.

The answer is, of course, automation. The explosion of adoption of virtualization, both on workstations and servers, and the widespread adoption of public and private cloud computing, makes it much easier to provision new machines, and most implementations expose an API to make it easy to bring up machines programmatically. Similarly of course, Chef is designed from the ground up as a RESTful API. Libraries exist and can be built upon to access remote machines and perform various tests. What is required is a way to integrate the Chef management, the machine provisioning, and the verification steps with a testing framework that enables us to build our infrastructure in a behavior-driven way.

Cucumber provides the ideal framework for capturing requirements in a form in which they can be tested. It provides a very high-level domain specific language for achieving this. By following a few simple language rules, it's possible to write something that is highly readable and understandable by the business, but which itself is an executable specification—something that functions as an automated acceptance.

Cucumber achieves this by wiring the high-level requirements to Ruby code that sets up state and makes assertions. In Cucumber terminology, we capture *features*, which are mapped onto tests in *steps*. These steps have the responsibility of setting up the state we need prior to making assertions against the requirements, perhaps making changes to the state, in line with the requirements, before finally tearing down whatever state was needed in order to be able to run the tests.

The significant difference when compared to unit testing, especially in our specific context, is that the number of steps and relative complexity is considerably higher. We need to write steps that build machines, install Chef, set up run lists, make cookbooks available, maybe make changes, maybe disable services. We then need to carry out external probes: for example, using a web page, logging onto a machine, or speaking to a service over the network. These kinds of steps are difficult to write and time-consuming. However, they do provide excellent value—they truly demonstrate whether the infrastructure code we have developed has delivered the functionality that is needed.

My first foray into this space was to write an integrated tool that generated examples tests, built infrastructure, handled all aspects of the Chef provisioning process, and finally reported results. That tool—*Cucumber-Chef*—is still widely used, but with the benefit of a few years' more experience, I now feel a slightly different model is called for.

With recent releases of both Vagrant and Test Kitchen, we now have mature tooling for provisioning infrastructure and running Chef, fully customizable to our needs, whether those are containerized app or OS deployments with Linux Containers, local virtualization solutions with VirtualBox or VMware, private cloud infrastructures with Openstack or Openshift, or public cloud infrastructures with Amazon AWS, Rackspace cloud,

or Microsoft Azure. In the same way that Chef provides primitives for automating the components of an infrastructure upon which we deploy our applications, what is needed is a set of primitives for building stacks of machines and delivering desired state through configuration management. In the spirit of the Unix philosophy, we should write programs that do one thing and do it well, and write programs to work together.

Cucumber admirably fits into this philosophy—it runs executable specifications and reports their result. Vagrant and Test Kitchen similarly. What is missing is a tool that ties them together, which would make it easy, in the context of Cucumber steps, to provision and test infrastructure. Leibniz provides this capability.

Leibniz provides an integration layer between Cucumber and Test Kitchen, in the form of steps that can be used in feature files to describe and provision infrastructure for acceptance testing.

Getting Started

We already know how to get started with Cucumber, as we covered it in the Hipster Assessor. Leibniz is a very simple Rubygem, which provides steps to Cucumber to provision machines via Test Kitchen.

Therefore we need only add the following three things to a Gemfile:

```
gem 'cucumber'
gem 'rspec-expectations'
gem 'leibniz'
```

However, where should the Gemfile be? That may seem like a ridiculous question, but think for a moment. As a cookbook author, especially a cookbook that is widely used in the community, the task of developing, testing, and releasing code is somewhat akin to that of a Rubygem, or even of working on an aspect of a core library within Ruby. This is code that is used by people to perform a task. It's building-block code. The way we test a library in Ruby is very different from the way we test a Rails application. The Rails application provides a service to an external user. Sure it might actually just be an internal API, but it sets up a contract with and is consumed by an external agency. That's not quite the same as *StringIO* within the Ruby standard library. Let me come at this from a different perspective.

When we are building infrastructure with Chef, it's essential to think from the outside in. Why are we actually building this infrastructure? What service does it provide? As Jamie Winsor, developer at Riot Games, creators of Berkshelf and makers of *League of Legends*, says, "Nobody plays CentOS, or Nginx. They play League of Legends!"

With this in mind, I would argue that the kind of acceptance testing that I advocate makes most sense not so much in the context of the Nginx cookbook, as in a cookbook that describes the top-level service that consumes the Nginx cookbook. This pattern is known as *The Application Cookbook*.

The application cookbook pattern is characterized by having decided the top-level service that we provide and creating a cookbook for that service. That cookbook wraps all the dependent services that are needed to deliver the top-level service. For example, an "awesome" web application might need components such as an app server, a database server, a load balancer. Each of these components is given a recipe that includes—and if necessary alters the behavior of—cookbooks that provide infrastructure modeling primitives such as Nginx, MySQL, and Redis.

This looks a lot like the kind of thing that might be accomplished using a Chef role, but has some significant advantages.

First of all, cookbooks can be explicitly versioned and tracked in a way that roles can't. Roles function as a (potentially dangerous) global variable that, when changed, will impact every node that has the role on its run list. Cookbooks can be explicitly versioned, frozen, and pinned, depending on use case.

Second, the behavior that the role describes, and encapsulates its meaning, should be tested. Where do we keep the tests? Where do we keep any documentation or change log? If the need should arise (and we should avoid it) to incorporate logic to control the behavior of the role, we have the power and flexibility to do so, and to test that logic. None of these options look easy when using the role DSL and a run list.

Third, we can use precisely the same toolkit for solving dependencies, interacting with the Chef API, and performing local testing, without having to maintain an additional primitive and its state.

If we look at the function of a role, it really does three things:

1. Contains and manipulates run lists
2. Alters recipe behavior using attributes
3. Provides simple taxonomy to label and tag nodes

The use of an application cookbook removes the need for the first and the second, although one consideration is that with a single cookbook/recipe on the run list, it's not possible to find, via the Chef API, which recipes will be run on a node. This can be found, however, using the `knife audit` command.

Nodes simply get either the top-level `awesome` recipe, if the node includes absolutely everything in one place, or it is given the recipe that corresponds to the logical function in the application, such as `awesome::cache_server`.

If there is a need to alter the behavior of an upstream cookbook, attributes can be set in a recipe, and if functionality needs to be added, tested, or tweaked, this can be achieved by wrapping upstream cookbooks in a manner that looks much like object inheritance. This has the twin advantages again of being testable, but also of avoiding constant forking of upstream cookbooks.

Tagging can be achieved by using the explicit tagging capabilities of Chef, or with a custom attribute set with a recipe in a cookbook. On occasions where cookbooks search for machines having a certain role, this can be supported by using an empty "marker" role, or by modifying the recipe to use a different way to categorize and find nodes.

Finally, I think that keeping as much as possible in cookbooks allows us to design our cookbooks in accordance with good object-oriented design principles. This is because we can treat cookbooks, recipes, and resources much more like objects than we can a mixture of data and code, which is what we have with the combination of roles and cookbooks.

At this point I urge you to buy and read the excellent *Practical Object-Oriented Design in Ruby* by Sandi Metz (Addison-Wesley). Let me summarize very briefly some key takeaways as directly applicable to infrastructure as code:

- Change is inevitable. We can't predict how things will change, but they will. We should design our infrastructure code in such a way as to accommodate the inevitability of change.
- Tying tests to the implementation makes refactoring difficult, so testing the external interface, outside-in, is the best way to build for change.
- We should favor loose coupling and build to test, valuing highly ease of change and embracing refactoring.
- Dependencies are inevitable. We will need to express and use dependencies in our designs, but should think carefully about them.
- Building our cookbooks to be pluggable and reusable, with clearly defined behavior, will help keep dependencies healthy.
- Object-orientation is all about message-sending. We should follow the principles of encapsulation and trust; our cookbooks don't need to know a lot about each other.

With this in mind, I would advocate that when modeling infrastructure, the first thing we should do is create a cookbook that presents the external service in a way that can be reasoned about and tested.

Example

The use of Cucumber and Leibniz is actually fundamentally pretty trivial. The value is first in the conversations, and second in the downward descent into the lower regions of the testing workflow. It's here that the design will emerge, and that the nuts and bolts infrastructure code takes place.

All we're doing at the top-most level is writing a test that will exercise the external interface of the infrastructure we're building.

Of course, such words cover a multitude of complications, and the actual process of writing those steps is not actually so easy. Nevertheless, I'll show an example of testing an application cookbook, from the outside in, beginning with Cucumber, and ending with the test passing.

Let's start with the requirements.

We're going to begin with a trivially simple infrastructure project. I usually find that it makes sense to make it into a bit of a story, to get into the mood of capturing requirements. In practice, I'm going to have you serve a simple website. But let's make it a bit more fun.

The scenario I am painting for you is that we, as infrastructure developers, have been approached by a small graphic design agency. This sort of thing happens quite often at Atalanta Systems—because we provide outsourced sysadmin and infrastructure development services, it's not uncommon for even very small companies to approach us and ask us to help them with their infrastructure.

The owner of the company has sent you an email, which reads:

> Hi there,
>
> I run a small graphic design agency. It's been running for a year or two, mostly on the basis of word-of-mouth and referral. However, we'd like to expand our horizons a little, and so we'd like to put together a simple website that describes what we do, with a few case studies or references. A friend of mine suggested you might be a good person to speak to about putting together whatever is necessary to get this running in the cloud. We can handle the design of the content, and we've hired a web designer who is going to pull it together. However, we're not really technically minded, so we'd appreciate some help with actually getting it live in a reliable and secure fashion. Can you help?
>
> Best,
>
> Miles Hunt

This sounds pretty trivial to you; all that's needed is a web server and a mechanism of getting their content onto it. Of course we don't yet know anything about whether the design agency is using a CMS, and we don't know about the various non-functional requirements, such as how frequently it should be backed up, how many users are expected, what a reasonable response time might be, and so on.

The very first step, therefore, is to find the stakeholder, and book some time with her. You arrange a meeting and bring your laptop with you to the meeting. This is important because in the meeting you're going to talk about the rationale for the project and the acceptance criteria, and these need to go into the feature specification. You could take notes on paper and then go away, but part of the beauty of Cucumber is that you can sit down with non-technical people and start writing the test right there and then.

I found one of my children roaming around the house looking for something to do, so I sat him down and made him pretend to be a person wanting a website, like our fictional depiction of Miles Hunt.

I opened up a buffer in Emacs, and I wrote:

```
Feature:
```

We talked for a bit, and we agreed that the minimum viable feature for the project was that a prospective customer could browse to the website and read about the services offered by the design agency. As a result, we added "Potential customer can read about services" to the feature, and described the feature as follows:

```
Feature: Potential customer can read about services

In order to generate more leads for my business
As a business owner
I want web users to be able to read about my services
```

We then talked about a possible example that would demonstrate that the most fundamental requirements had been met. We agreed that the following would make sense:

```
Scenario: User visits home page
  Given a url http://wonderstuff-design.me
  When a web user browses to the URL
   Then the user should see "Wonderstuff Design is a boutique graphics design
agency."
```

We agreed that if this test passed, we'd feel that significant progress had been made, so we didn't write any more scenarios at this stage.

As we discussed earlier, Gherkin is a plain text DSL for mapping high-level stakeholder requirements to source code that sets up state and verifies it against those requirements. When starting an infrastructure project, I'd recommend setting aside some time to talk through the reasons for the requirement, and to understand what the simplest thing would be that would deliver value and move the project forward.

I'm not a big fan of capturing dozens of detailed stories at the start; I'd rather get two or three down first and get started on that. You can always go back for more later.

It doesn't matter if the form in which you take down the initial requirement doesn't end up being exactly the form you use—you can go back and check language later; the most important thing to do is have the conversation and capture the output of that conversation. For this reason, I asked you to write the feature before anything else.

Having captured the requirement, we need to work out how to test it.

Let's start by creating a cookbook to encapsulate the services we need:

```
$ berks cookbook wonderstuff
      create  wonderstuff/files/default
      create  wonderstuff/templates/default
```

```
      create  wonderstuff/attributes
      create  wonderstuff/definitions
      create  wonderstuff/libraries
      create  wonderstuff/providers
      create  wonderstuff/recipes
      create  wonderstuff/resources
      create  wonderstuff/recipes/default.rb
      create  wonderstuff/metadata.rb
      create  wonderstuff/LICENSE
      create  wonderstuff/README.md
      create  wonderstuff/Berksfile
      create  wonderstuff/Thorfile
      create  wonderstuff/chefignore
      create  wonderstuff/.gitignore
         run  git init from "./wonderstuff"
      create  wonderstuff/Gemfile
      create  .kitchen.yml
      append  Thorfile
      create  test/integration/default
      append  .gitignore
      append  .gitignore
      append  Gemfile
      append  Gemfile
You must run `bundle install' to fetch any new gems.
      create  wonderstuff/Vagrantfile
```

Now let's update the Gemfile and then run Bundle:

```
$ cat Gemfile
source 'https://rubygems.org'

gem 'berkshelf'
gem 'test-kitchen', :group => :integration
gem 'kitchen-vagrant', :group => :integration
gem 'cucumber', :group => :integration
gem 'rspec-expectations', :group => :integration
gem 'leibniz', :group => :integration
```

Now, we already know from our Hipster Assessor, that we need to create a features directory and a steps directory, and then create a feature containing the acceptance criteria:

```
$ mkdir -p wonderstuff/features/step_definitions
$ cat <<EOF > wonderstuff/features/readable_services.feature
> Feature: Potential customer can read about services
>
> In order to generate more leads for my business
> As a business owner
> I want web users to be able to read about my services
>
> Scenario: User visits home page
>   Given a url http://wonderstuff-design.me
>   When a web user browses to the URL
```

```
>   Then the user should see "Wonderstuff Design is a boutique graphics design
agency."
> EOF
```

Now, let's think about this a little bit. We've captured the basic requirement, now let's think about what's involved in testing this infrastructure. We're going to need a machine, an operating system, Chef, a cookbook, a run list, and then we need to run Chef. Leibniz exists to make this easy for us. To use Leibniz, all we need to do is add a background description, containing a table detailing the infrastructure we want to build:

```
Background:

  Given I have provisioned the following infrastructure:
  | Server Name | Operating System | Version | Chef Version | Run List       |
  | wonderstuff | ubuntu            |   12.04 |       11.4.4 | wonderstuff::de-
fault |
  And I have run Chef
```

What this will do is launch a machine using Test Kitchen, with the preceding specification, and make available an object that provides instance data from Test Kitchen.

Let's look again at the example we took from Corin, I mean, Miles Hunt:

```
Scenario: User visits home page
  Given a url http://wonderstuff-design.me
  When a web user browses to the URL
   Then the user should see "Wonderstuff Design is a boutique graphics design
agency."
```

This seems fine—it describes the behavior as needed. Let's run our test, which currently reads:

```
Feature: Potential customer can read about services

  In order to generate more leads for my business
  As a business owner
  I want web users to be able to read about my services

  Background:

    Given I have provisioned the following infrastructure:
| Server Name | Operating System | Version | Chef Version | Run List       |
| wonderstuff | ubuntu            |   12.04 |       11.4.4 | wonderstuff::de-
fault |
    And I have run Chef

  Scenario: User visits home page

    Given a url http://wonderstuff-design.me
    When a web user browses to the URL
     Then the user should see "Wonderstuff Design is a boutique graphics design
agency."
```

Now let's run our test:

```
$ cucumber
Feature: Potential customer can read about services

  In order to generate more leads for my business
  As a business owner
  I want web users to be able to read about my services

  Background:                                                    # features/
readable_services.feature:7
    Given I have provisioned the following infrastructure: ↵
# features/readable_services.feature:9
|  Server  Name  |  Operating  System  |  Version  |  Chef  Version  |  Run
List            |
| wonderstuff | ubuntu             | 12.04    | 11.4.4        | wonderstuff::de-
fault |
     And I have run Chef                                         # features/
readable_services.feature:12

                   Scenario:          User          visits          home
page                                                             # features/
readable_services.feature:14
                     Given          a          url          http://wonderstuff-
design.me                                          # features/readable_serv-
ices.feature:16
            When      a      web      user      browses      to      the
URL                                                # features/readable_serv-
ices.feature:17
   Then the user should see "Wonderstuff Design is a boutique graphics design
agency." # features/readable_services.feature:18

1 scenario (1 undefined)
5 steps (5 undefined)
0m0.003s

You can implement step definitions for undefined steps with these snippets:

Given(/^I have provisioned the following infrastructure:$/) do |table|
  # table is a Cucumber::Ast::Table
  pending # express the regexp above with the code you wish you had
end

Given(/^I have run Chef$/) do
  pending # express the regexp above with the code you wish you had
end

Given(/^a url http:\/\/wonderstuff\-design\.me$/) do
  pending # express the regexp above with the code you wish you had
end

When(/^a web user browses to the URL$/) do
  pending # express the regexp above with the code you wish you had
```

```
end

Then(/^the user should see "(.*?)"$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

If you want snippets in a different programming language,
just make sure a file with the appropriate file extension
exists where Cucumber looks for step definitions.
```

This should look familiar. We're now going to write the steps that map the Gherkin code to real Ruby that will provision and exercise our infrastructure.

```
require 'leibniz'
require 'faraday'

Given(/^I have provisioned the following infrastructure:$/) do |specification|
  @infrastructure = Leibniz.build(specification)
end

Given(/^I have run Chef$/) do
  @infrastructure.destroy
  @infrastructure.converge
end

Given(/^a url "(.*?)"$/) do |url|
  @host_header = url.split('/').last
end


When(/^a web user browses to the URL$/) do
  connection = Faraday.new(:url => "http://#{@infrastructure↵
['wonderstuff'].ip}",
                           :headers => {'Host' => @host_header}) do |faraday|
    faraday.adapter Faraday.default_adapter
  end
  @page = connection.get('/').body

end

Then(/^the user should see "(.*?)"$/) do |content|
  expect(@page).to match /#{content}/
end
```

We begin by requiring the Leibniz library to give us access to the steps that allow us to interact with Test Kitchen. We also require the Faraday library, which is a powerful and pleasant-to-use Ruby HTTP client library.

The first two steps come from Leibniz, and do pretty much exactly what they say: they build infrastructure according to the specification in the table, run the destroy task to ensure a clean environment, and then run the converge task.

The third step simply takes the URL and extracts what will be necessary to pass as the Host header to the web server. Given that we're not going to have a real DNS entry, this is a tidy way to have a scenario devoid of testing and implementation detail, which translates to a trivial Ruby method.

The fourth step instantiates an instance of the Faraday HTTP client, passing as its arguments the IP address of the machine we provisioned, and the Host header we calculated. We then perform an HTTP GET and capture the body.

Finally we assert that the page will match the content we specified in the scenario.

A very simple example, but one that exercises the system from top to bottom and demonstrates the principles at play.

Let's run the test:

```
$ cucumber
Feature: Potential customer can read about services

  In order to generate more leads for my business
  As a business owner
  I want web users to be able to read about my services

  Background:                                                          # features/
readable_services.feature:7
    Given I have provisioned the following infrastructure: # features/step_defi-
nitions/visit-home-page-steps.rb:4
        | Server Name | Operating System | Version | Chef Version | Run
List            |
      | wonderstuff | ubuntu             | 12.04    | 11.4.4        | wonder-
stuff::default |
    And I have run Chef                                                # features/
step_definitions/visit-home-page-steps.rb:8

                  Scenario:         User         visits         home
page                                                                 # features/
readable_services.feature:14
                           Given       a       url       "http://wonderstuff-
design.me"                                    # features/step_definitions/
visit-home-page-steps.rb:13
                    When    a    web    user    browses    to    the
URL                                                  # features/step_defini-
tions/visit-home-page-steps.rb:18
      Connection refused - connect(2) (Faraday::Error::ConnectionFailed)
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:763:in `initialize'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:763:in `open'
        /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:763:in `block in con-
nect'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/timeout.rb:55:in `timeout'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/timeout.rb:100:in `timeout'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:763:in `connect'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:756:in `do_start'
```

```
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:745:in `start'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:1285:in `request'
      /opt/rubies/1.9.3-p429/lib/ruby/1.9.1/net/http.rb:1027:in `get'
               /home/tdi/.gem/ruby/1.9.3/gems/faraday-0.8.7/lib/faraday/adapter/
net_http.rb:73:in `perform_request'
               /home/tdi/.gem/ruby/1.9.3/gems/faraday-0.8.7/lib/faraday/adapter/
net_http.rb:38:in `call'
         /home/tdi/.gem/ruby/1.9.3/gems/faraday-0.8.7/lib/faraday/connection.rb:
247:in `run_request'
         /home/tdi/.gem/ruby/1.9.3/gems/faraday-0.8.7/lib/faraday/connection.rb:
100:in `get'
       ./features/step_definitions/visit-home-page-steps.rb:23:in `/^a web user
browses to the URL$/'
      features/readable_services.feature:17:in `When a web user browses to the
URL'
    Then the user should see "Wonderstuff Design is a boutique graphics design
agency." # features/step_definitions/visit-home-page-steps.rb:27

Failing Scenarios:
cucumber features/readable_services.feature:14 # Scenario: User visits home page

1 scenario (1 failed)
5 steps (1 failed, 1 skipped, 3 passed)
1m5.946s
```

We have a failing acceptance test—unsurprisingly because we haven't built anything. I'm now going to race through the steps of adding integration tests and unit tests, without comment, as we'll discuss these in detail shortly. Once we have the unit and integration tests passing, we'll run the Cucumber test again, and we should be all green!

Next we write the integration tests:

```
require 'spec_helper'
describe 'Wonderstuff Design' do

  it 'should install the lighttpd package' do
    expect(package 'lighttpd').to be_installed
  end

  it 'should enable and start the lighttpd service' do
    expect(service 'lighttpd').to be_enabled
    expect(service 'lighttpd').to be_running
  end

  it 'should render the Wonderstuff Design web page' do
    expect(file('/var/www/index.html')).to be_file
    expect(file('/var/www/index.html')).to contain 'Wonderstuff Design is a bou-
tique graphics design agency.'
  end

end
```

And run it:

```
$ kitchen verify
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Verifying <default-ubuntu-1204>
       Removing /opt/busser/suites/serverspec
Uploading /opt/busser/suites/serverspec/spec_helper.rb (mode=0664)
Uploading /opt/busser/suites/serverspec/localhost/cisco_spec.rb (mode=0664)
Uploading /opt/busser/suites/serverspec/localhost/cisco_spec.rb~ (mode=0664)
-----> Running serverspec test suite
/opt/chef/embedded/bin/ruby -I/opt/busser/suites/serverspec -S /opt/chef/embed-
ded/bin/rspec /opt/busser/suites/serverspec/localhost/cisco_spec.rb
Package `lighttpd' is not installed and no info is available.
Use dpkg --info (= dpkg-deb --info) to examine archive files,
and dpkg --contents (= dpkg-deb --contents) to list their contents.
FFF

Failures:

  1) Wonderstuff Design should install the lighttpd package
     Failure/Error: expect(package 'lighttpd').to be_installed
         dpkg -s lighttpd && ! dpkg -s lighttpd | grep -E '^Status: .+ not-
installed$'
     # /opt/busser/suites/serverspec/localhost/cisco_spec.rb:5:in `block (2 lev-
els) in <top (required)>'

  2) Wonderstuff Design should enable and start the lighttpd service
     Failure/Error: expect(service 'lighttpd').to be_enabled
           ls /etc/rc3.d/ | grep -- lighttpd || grep 'start on' /etc/init/
lighttpd.conf
       grep: /etc/init/lighttpd.conf: No such file or directory
     # /opt/busser/suites/serverspec/localhost/cisco_spec.rb:9:in `block (2 lev-
els) in <top (required)>'

  3) Wonderstuff Design should render the Wonderstuff Design web page
     Failure/Error: expect(file('/var/www/index.html')).to be_file
       test -f /var/www/index.html
       # /opt/busser/suites/serverspec/localhost/cisco_spec.rb:14:in `block (2
levels) in <top (required)>'

Finished in 0.02524 seconds
3 examples, 3 failures

Failed examples:

rspec /opt/busser/suites/serverspec/localhost/cisco_spec.rb:4 # Wonderstuff De-
sign should install the lighttpd package
rspec /opt/busser/suites/serverspec/localhost/cisco_spec.rb:8 # Wonderstuff De-
sign should enable and start the lighttpd service
rspec /opt/busser/suites/serverspec/localhost/cisco_spec.rb:13 # Wonderstuff De-
sign should render the Wonderstuff Design web page
```

Now we write the unit tests:

```
require 'spec_helper'

describe "wonderstuff::default" do
  let(:chef_run) do
    runner = ChefSpec::ChefRunner.new(
      log_level: :error,
      cookbook_path: COOKBOOK_PATH,
    )
    Chef::Config.force_logger true
    runner.converge('recipe[wonderstuff::default]')
  end

  it "installs the lighttpd package" do
    expect(chef_run).to install_package 'lighttpd'
  end

  it "creates a webpage to be served" do
     expect(chef_run).to create_file_with_content '/var/www/index.html', 'Wonder-
stuff Design is a boutique graphics design agency.'
  end

  it "starts the lighttpd service" do
    expect(chef_run).to start_service 'lighttpd'
  end

  it "enables the lighttpd service" do
    expect(chef_run).to set_service_to_start_on_boot 'lighttpd'
  end
end
```

And run them:

```
$ rspec -fd
Using wonderstuff (0.1.0) at path: '/home/tdi/wonderstuff'

wonderstuff::default
  installs the lighttpd package (FAILED - 1)
  creates a webpage to be served (FAILED - 2)
  starts the lighttpd service (FAILED - 3)
  enables the lighttpd service (FAILED - 4)

Failures:

  1) wonderstuff::default installs the lighttpd package
     Failure/Error: expect(chef_run).to install_package 'lighttpd'
       No package resource named 'lighttpd' with action :install found.
      # ./spec/unit/recipes/default_spec.rb:14:in `block (2 levels) in <top (re-
quired)>'

  2) wonderstuff::default creates a webpage to be served
        Failure/Error:  expect(chef_run).to  create_file_with_content  '/var/www/
index.html', 'Wonderstuff Design is a boutique graphics design agency.'
       File content:
```

```
        does not match expected:
       Wonderstuff Design is a boutique graphics design agency.
     # ./spec/unit/recipes/default_spec.rb:18:in `block (2 levels) in <top (re-
quired)>'

  3) wonderstuff::default starts the lighttpd service
     Failure/Error: expect(chef_run).to start_service 'lighttpd'
       No service resource named 'lighttpd' with action :start found.
     # ./spec/unit/recipes/default_spec.rb:22:in `block (2 levels) in <top (re-
quired)>'

  4) wonderstuff::default enables the lighttpd service
     Failure/Error: expect(chef_run).to set_service_to_start_on_boot 'food'
       expected chef_run: recipe[wonderstuff::default] to set service to start
on boot "lighttpd"
     # ./spec/unit/recipes/default_spec.rb:26:in `block (2 levels) in <top (re-
quired)>'

Finished in 0.00969 seconds
4 examples, 4 failures

Failed examples:

rspec ./spec/unit/recipes/default_spec.rb:13 # wonderstuff::default installs
the lighttpd package
rspec ./spec/unit/recipes/default_spec.rb:17 # wonderstuff::default creates a
webpage to be served
rspec ./spec/unit/recipes/default_spec.rb:21 # wonderstuff::default starts the
lighttpd service
rspec ./spec/unit/recipes/default_spec.rb:25 # wonderstuff::default enables the
lighttpd service
```

Now we write the cookbook:

```
$ cat recipes/default.rb
package 'lighttpd'

service 'lighttpd' do
  action [:enable, :start]
end

cookbook_file '/var/www/index.html' do
  source 'wonderstuff.html'
end

$ cat files/default/wonderstuff.html
<html>
<body>
<p>Wonderstuff Design is a boutique graphics design agency.</p>
</body>
</html>
```

Now we run the unit tests again:

```
$ rspec -fd
Using wonderstuff (0.1.0) at path: '/home/tdi/wonderstuff'

wonderstuff::default
  installs the lighttpd package
  creates a webpage to be served
  starts the lighttpd service
  enables the lighttpd service

Finished in 0.01352 seconds
4 examples, 0 failures
```

Now we run the integration tests:

```
$ kitchen verify 12
-----> Starting Kitchen (v1.0.0.dev)
-----> Setting up <default-ubuntu-1204>
-----> Setting up Busser
       Creating BUSSER_ROOT in /opt/busser
       Creating busser binstub
       Plugin serverspec already installed
       Finished setting up <default-ubuntu-1204> (0m3.21s).
-----> Verifying <default-ubuntu-1204>
       Removing /opt/busser/suites/serverspec
Uploading /opt/busser/suites/serverspec/spec_helper.rb (mode=0664)
Uploading /opt/busser/suites/serverspec/localhost/cisco_spec.rb (mode=0664)
Uploading /opt/busser/suites/serverspec/localhost/cisco_spec.rb~ (mode=0664)
-----> Running serverspec test suite
/opt/chef/embedded/bin/ruby -I/opt/busser/suites/serverspec -S /opt/chef/embed-
ded/bin/rspec /opt/busser/suites/serverspec/localhost/cisco_spec.rb
...

Finished in 0.04747 seconds
3 examples, 0 failures
       Finished verifying <default-ubuntu-1204> (0m2.12s).
-----> Kitchen is finished. (0m6.40s)
```

And finally, we run Cucumber again:

```
$ cucumber
Feature: Potential customer can read about services

  In order to generate more leads for my business
  As a business owner
  I want web users to be able to read about my services

  Background:                                                    # features/
readable_services.feature:7
    Given I have provisioned the following infrastructure: # features/step_defi-
nitions/visit-home-page-steps.rb:4
         | Server Name | Operating System | Version | Chef Version | Run
List            |
       | wonderstuff | ubuntu             | 12.04   | 11.4.4       | wonder-
stuff::default |
```

```
Using wonderstuff (0.1.0) at path: '/home/tdi/wonderstuff'
     And I have run Chef                                              # features/
step_definitions/visit-home-page-steps.rb:8

                    Scenario:          User          visits          home
page                                                                  # features/
readable_services.feature:14
                              Given       a       url       "http://wonderstuff-
design.me"                                          # features/step_definitions/
visit-home-page-steps.rb:13
                      When     a     web     user     browses     to     the
URL                                                      # features/step_defini-
tions/visit-home-page-steps.rb:18
    Then the user should see "Wonderstuff Design is a boutique graphics design
agency." # features/step_definitions/visit-home-page-steps.rb:27
      expected "<?xml version=\"1.0\" encoding=\"iso-8859-1\"?>\n<!DOCTYPE html
PUBLIC  \"-//W3C//DTD  XHTML  1.0  Transitional//EN\"\n                \"http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\n<html       xmlns=\"http://
www.w3.org/1999/xhtml\" xml:lang=\"en\" lang=\"en\">\n <head>\n   <title>403 -
Forbidden</title>\n </head>\n <body>\n   <h1>403 - Forbidden</h1>\n </body>\n</
html>\n" to match /Wonderstuff Design is a boutique graphics design agency./
(RSpec::Expectations::ExpectationNotMetError)
        ./features/step_definitions/visit-home-page-steps.rb:28:in `/^the user
should see "(.*?)"$/'
       features/readable_services.feature:18:in `Then the user should see "Won-
derstuff Design is a boutique graphics design agency."'

Failing Scenarios:
cucumber features/readable_services.feature:14 # Scenario: User visits home page

1 scenario (1 failed)
5 steps (1 failed, 4 passed)
1m11.921s
```

Aha! What happened!

Upon investigation, we discover that we didn't set the ownership and group of the html page, so the user under which *lighttpd* runs won't be able to read it!

Now, at this point it is very important to write a failing test that catches the mistake:

```
  it "creates a webpage to be served" do
    expect(chef_run).to create_file_with_content '/var/www/index.html', 'Wonder-
stuff Design is a boutique graphics design agency.'
    expect(file).to be_owned_by('www-data', 'www-data')
  end
```

Let's run the test:

```
$ rspec -fd
Using wonderstuff (0.1.0) at path: '/home/tdi/wonderstuff'

wonderstuff::default
  installs the lighttpd package
```

```
  creates a webpage to be served (FAILED - 1)
  starts the lighttpd service
  enables the lighttpd service

Failures:

  1) wonderstuff::default creates a webpage to be served
     Failure/Error: expect(file).to be_owned_by('www-data', 'www-data')
     NameError:
          undefined local variable or method `file' for #<RSpec::Core::Exam-
pleGroup::Nested_1:0x00000003ddd418>
      # ./spec/unit/recipes/default_spec.rb:19:in `block (2 levels) in <top (re-
quired)>'

Finished in 0.013 seconds
4 examples, 1 failure

Failed examples:

rspec ./spec/unit/recipes/default_spec.rb:17 # wonderstuff::default creates a
webpage to be served
```

Now let's make the test pass by updating the resource in the recipe:

```
package 'lighttpd'

service 'lighttpd' do
  action [:enable, :start]
end

cookbook_file '/var/www/index.html' do
  source 'wonderstuff.html'
  owner 'www-data'
  group 'www-data'
end
```

Now, let's run Cucumber one last time:

```
$ cucumber
Feature: Potential customer can read about services

  In order to generate more leads for my business
  As a business owner
  I want web users to be able to read about my services

  Background:                                                          # features/
readable_services.feature:7
    Given I have provisioned the following infrastructure: # features/step_defi-
nitions/visit-home-page-steps.rb:4
        | Server Name | Operating System | Version | Chef Version | Run
List            |
       | wonderstuff | ubuntu             | 12.04    | 11.4.4        | wonder-
stuff::default |
Using wonderstuff (0.1.0) at path: '/home/tdi/wonderstuff'
```

```
    And I have run Chef                                              # features/
step_definitions/visit-home-page-steps.rb:8

                  Scenario:         User         visits         home
page                                                               # features/
readable_services.feature:14
                           Given       a       url       "http://wonderstuff-
design.me"                                       # features/step_definitions/
visit-home-page-steps.rb:13
                    When     a     web     user     browses     to     the
URL                                                  # features/step_defini-
tions/visit-home-page-steps.rb:18
    Then the user should see "Wonderstuff Design is a boutique graphics design
agency." # features/step_definitions/visit-home-page-steps.rb:27

1 scenario (1 passed)
5 steps (5 passed)
1m10.105s
```

Although a truly trivial example, I hope this gives a sense of the workflow and the ideas behind writing acceptance tests for application cookbooks. Indeed, even on this simple example we see the benefit of a true acceptance test—our unit tests passed, our integration tests passed, but what we delivered was useless crap. Only with the true exercising of the system we built did we discover our mistake!

I will be documenting far more complex examples on the website or on my blog. I would welcome your enthusiastic contributions and discussions on the Chef users' mailing list.

As an appetite whetter, I offer the following feature:

```
Feature: Highly Available Jenkins

Infrastructure developers should be able to enjoy uninterrupted access to their
build jobs.

  Background:

  Given I have provisioned the following infrastructure:

| Server Name      | Operating System | Architecture | Version | Chef Version |
Run List         |
| lb1              | CentOS           | 64 bit       | 6.4     | 11.4.4       |
tedious::ha      |
| lb2              | CentOS           | 64 bit       | 6.4     | 11.4.4       |
tedious::ha      |
| jenkins1         | CentOS           | 64 bit       | 6.4     | 11.4.4       |
tedious::jenkins |
| jenkins2         | CentOS           | 64 bit       | 6.4     | 11.4.4       |
tedious::jenkins |

  And 'http://tedio.us' resolves to the virtual IP of the loadbalancer
```

```
  Scenario: Jenkins should be available

    Infrastructure developers should be able to reach and use a Jenkins server.

    When I try to download the Jenkins CLI tool
    Then the download will succeed
    And when I query the version number
    Then the version number will be returned

  Scenario: Jenkins should be load-balanced

     Infrastructure developers should be able to use Jenkins in the event of a
Jenkins server failure

    Given I am using a Jenkins server
    When that Jenkins server is switched off
    Then I should be able to reach an alternative Jenkins server

  Scenario: Load balancers should be in a redundant pair

    Given that I am using Jenkins
    When one of the load balancers is switched off
    Then I should still be able to use Jenkins
```

Advantages and Disadvantages

The advantages of writing executable specifications, using Gherkin and Cucumber, have been expressed throughout this book. As an approach, it is widely regarded as offering outstanding value.

The main disadvantage is simply that it's not easy. The tooling is immature compared to the other resources discussed in this section. In its current evolutionary state, the requirement to do one's own heavy lifting is not inconsiderable. However, the more people that engage in the process, and the more the tooling matures, the greater the differential between effort in and value out.

Other than this, I would like to address two particular objections to the approach.

The first is the argument that "a good monitoring system" takes care of the requirement for externally facing acceptance tests.

While I certainly agree that a monitoring system should be measuring the extent to which one's system meets its acceptance criteria, I think this is missing the point to a significant degree.

Doubtless, a monitoring system that doesn't measure and alert on the fundamental purpose of the business is not a very valuable monitoring system. Indeed, it's for this reason that I have long advocated that acceptance tests can be used as an input to, or in certain cases, even directly as one's monitoring system. However, the tests that comprise

that monitoring system still need to be written, the requirements still need to be captured, and that is a collaborative effort that belongs squarely in the same conversation and workflow as the rest of the program of cookbook testing. Ducking the issue, or delegating it to a separate monitoring discussion, is to introduce segregation and siloization where there should be none.

Furthermore, certain acceptance tests, or even smoke tests, could be destructive, expensive, or impose hostile load burdens, or probe security issues. These don't belong in a production monitoring system, but they are very much requirements and specifications that need to be considered when beginning to build infrastructure as code.

If we accept the view that acceptance tests are the same as, or function as, monitoring checks, then these monitoring checks should be the first thing we write. This is the purest interpretation of the mandate to develop outside-in.

On this point, it's illuminating to think about outside-in as being fractal. Post Chef-run convergence testing is outside-in from the perspective of the Chef run, although it's not truly at the level of a test from outside the node itself. Thus the kind of approach that Cucumber and Leibniz offer can be viewed as a higher-order testing approach.

Ultimately, I think it's as plain as this: for most organizations building infrastructure at scale using Chef, the business is the application. If the application doesn't function—if customers cannot login and perform critical business actions—then all other monitoring is for naught.

The second objection is that when building infrastructure, the stakeholders are frequently technical, so the domain language is shared, and the value in capturing requirements in Gherkin is diminished.

This is a deceptively attractive sounding position. It is indeed frequently the case that the stakeholders for infrastructure projects are technical architects, developers, or even system administrators. In these cases the ubiquitous language shared between stakeholders and implementers is imbued with technical concepts to a more significant degree than when designing software to be consumed by users.

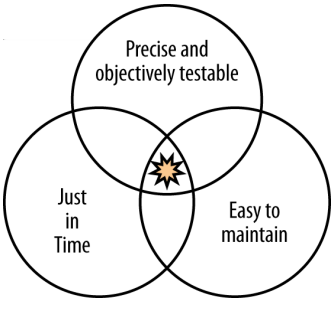
However, this does not remove the need for acceptance tests or documentation. It is still imperative that, as engineers, we both build the thing right and build the right thing. To do so, we need to ensure the following are in place:

- A common and unambiguous understanding of what needs to be delivered
- Explicit and univocal specification of requirements to minimize rework
- A concrete and measurable definition of done
- Documentation to support future change and maintenance

Traditional project management approaches invested large amounts of time and money in big upfront specifications and testing phases, which, to an extent, assisted (although some might argue hindered) the achievement of these prerequisites. However, in today's fast-paced, continuously delivering universe, such an approach simply isn't an option anymore. Nonetheless, the necessity of these foundational cornerstones is still apparent.

Now, more than ever, there is an urgent need for efficient specification, and lean planning; for reliable, always-right, and easily changeable documentation; for objective mechanisms to verify that the system meets requirements. How can this be achieved in a world of constant improvement, rapid change, auto-scaling, and cloud-bursting?

Gojko Adzic offers the following visualization:



The intersection of just-in-time delivery, highly maintainable documentation, and precise and objectively testable specifications lies in the very thing this book holds as pivotal —in requirements as executable acceptance tests. And this is required as much for highly technical stakeholders as for any other consumer of services.

Summary and Conclusion

Full acceptance testing of complex multi-node systems, conducted from a perspective outside of the systems under test, is the holy grail of test-driven infrastructure. The tooling is not yet up to scratch for solving problems of this complexity, but it's without a doubt an area where much experimentation and research is being carried out.

With hindsight, my decision to begin my crusade to bring test-driven and behavior-driven development practices into the world of infrastructure as code, with the purest form of outside-in acceptance testing, was wildly optimistic. However, I stand by my view that this is the correct methodology, and we should be pushing at the boundaries of the possible.

Leibniz is a brand new project, but initial feedback on the concept has been positive, and the problems it attempts to solve are real, topical, and tractable. Doubtless, its implementation and approach will change rapidly, so please consider this section very much an appetite-whetter and discussion-starter rather than a definitive description of a mature framework.

The other area where we are sure to see important developments is in the emergence of mature orchestration frameworks functioning at a level higher than the node. Opscode's "*push jobs*" looks to be the beginning of a process of releasing primitives for sophisticated orchestration capability within the Opscode product roadmap. At the same time, Riot Games has been promising to open source *Motherbrain*, their orchestration system. Alongside this, the engineers at Heavy Water have also been experimenting with proofs of concept playing in this space. Add to this the role already played by the popular *MCollective* framework in current orchestration frameworks, and it's clear this is an area of great potential.

Integration Testing: Test Kitchen with Serverspec and Bats

We introduced Test Kitchen as a foundational tool for orchestration and test running earlier in the chapter. We now turn to a detailed worked example of building infrastructure tests using both Serverspec and Bats.

Within a cookbook, the `kitchen init` command will generate a core testing structure, consisting of a YAML file, *.kitchen.yml*, which describes the various run configurations to test, and a directory for tests and supporting material, *test/integration/default*.

A default *.kitchen.yml* file contains the following:

```
---
driver_plugin: vagrant
driver_config:
  require_chef_omnibus: true

platforms:
- name: ubuntu-12.04
  driver_config:
    box: opscode-ubuntu-12.04
                        box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_ubuntu-12.04_provisionerless.box
- name: ubuntu-10.04
  driver_config:
    box: opscode-ubuntu-10.04
                        box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_ubuntu-10.04_provisionerless.box
- name: centos-6.4
  driver_config:
    box: opscode-centos-6.4
                        box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_centos-6.4_provisionerless.box
- name: centos-5.9
  driver_config:
    box: opscode-centos-5.9
                        box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_centos-5.9_provisionerless.box
```

```
suites:
- name: default
  run_list: ["recipe[ruby-install]"]
  attributes: {}
```

The driver plug-in is Vagrant, as mentioned. We hand off installation of Chef to the Omnibus plug-in, enabling us to keep our base boxes as minimal as possible. We then specify the platforms we're interested in testing against. By default, the assumption is that a cookbook developer wants to test against CentOS and Ubuntu, on both CentOS 5 and 6 and Ubuntu 10.04 and 12.04. These can easily be altered, as they are simply references to a Vagrant box name and source URL, exactly as would go into a Vagrantfile. Finally, we list suites of tests we want to run against each platform—in this case, by default, we want to apply the default recipe from the `ruby-install` cookbook. The possibility of specifying node attributes in the suite is also available.

It's entirely possible that you decide you don't want to test against all four of these systems—in which case, simply delete the ones that aren't relevant.

Running `kitchen list` will give a quick status review of your test kitchen:

```
$ bundle exec kitchen list
Instance             Last Action
default-ubuntu-1204  <Not Created>
default-ubuntu-1004  <Not Created>
default-centos-64    <Not Created>
default-centos-59    <Not Created>
```

Let's create a cookbook that will install the Pound load balancer. We want to be sure it will work on CentOS 5 as well as CentOS 6. We don't have any CentOS 5 machines, but we want to support this platform for the sake of the community, and as responsible cookbook developers, we want to be sure that as we develop on a Mac and deploy on CentOS 6, we don't introduce any regressions that would cause a problem on an earlier version of CentOS.

```
$ berks cookbook pound
      create  pound/files/default
      create  pound/templates/default
      create  pound/attributes
      create  pound/definitions
      create  pound/libraries
      create  pound/providers
      create  pound/recipes
      create  pound/resources
      create  pound/recipes/default.rb
      create  pound/metadata.rb
      create  pound/LICENSE
      create  pound/README.md
      create  pound/Berksfile
      create  pound/Thorfile
      create  pound/chefignore
```

```
      create  pound/.gitignore
         run  git init from "./pound"
      create  pound/Gemfile
      create  .kitchen.yml
      append  Thorfile
      create  test/integration/default
      append  .gitignore
      append  .gitignore
      append  Gemfile
      append  Gemfile
You must run `bundle install' to fetch any new gems.
      create  pound/Vagrantfile
```

Now let's slim down the platforms:

```
---
driver_plugin: vagrant
driver_config:
  require_chef_omnibus: true

platforms:
- name: centos-6.4
  driver_config:
    box: opscode-centos-6.4
                      box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_centos-6.4_provisionerless.box
- name: centos-5.9
  driver_config:
    box: opscode-centos-5.9
                      box_url:      https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_centos-5.9_provisionerless.box

suites:
- name: default
  run_list: ["recipe[pound]"]
  attributes: {}
```

Now we can create the base machines using the `kitchen create` command:

```
$ kitchen create all
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Creating <default-centos-64>
       [kitchen::driver::vagrant command] BEGIN (vagrant up --no-provision)
       Bringing machine 'default' up with 'virtualbox' provider...
       [default] Importing base box 'opscode-centos-6.4'...
       [default] Matching MAC address for NAT networking...
       [default] Setting the name of the VM...
       [default] Clearing any previously set forwarded ports...
       [Berkshelf] Skipping Berkshelf with --no-provision
       [default] Fixed port collision for 22 => 2222. Now on port 2204.
       [default] Creating shared folders metadata...
       [default] Clearing any previously set network interfaces...
       [default] Preparing network interfaces based on configuration...
```

```
       [default] Forwarding ports...
       [default] -- 22 => 2204 (adapter 1)
       [default] Running any VM customizations...
       [default] Booting VM...
       [default] Waiting for VM to boot. This can take a few minutes.
       [default] VM booted and ready for use!
       [default] Setting hostname...
       [default] Configuring and enabling network interfaces...
       [default] Mounting shared folders...
       [default] -- /vagrant
       [kitchen::driver::vagrant command] END (0m37.01s)
       [kitchen::driver::vagrant command] BEGIN (vagrant ssh-config)
       [kitchen::driver::vagrant command] END (0m1.27s)
       Vagrant instance <default-centos-64> created.
       Finished creating <default-centos-64> (0m38.95s).
-----> Creating <default-centos-59>
       [kitchen::driver::vagrant command] BEGIN (vagrant up --no-provision)
       Bringing machine 'default' up with 'virtualbox' provider...
         [default] Box 'opscode-centos-5.9' was not found. Fetching box from
specified URL for
       the provider 'virtualbox'. Note that if the URL does not have
       a box for this provider, you should interrupt Vagrant now and add
       the box yourself. Otherwise Vagrant will attempt to download the
       full box prior to discovering this error.
       Downloading or copying the box...
       Extracting box...3k/s, Estimated time remaining: --:--:--)
       Successfully added box 'opscode-centos-5.9' with provider 'virtualbox'!
       [default] Importing base box 'opscode-centos-5.9'...
       [default] Matching MAC address for NAT networking...
       [default] Setting the name of the VM...
       [default] Clearing any previously set forwarded ports...
       [Berkshelf] Skipping Berkshelf with --no-provision
       [default] Fixed port collision for 22 => 2222. Now on port 2205.
       [default] Creating shared folders metadata...
       [default] Clearing any previously set network interfaces...
       [default] Preparing network interfaces based on configuration...
       [default] Forwarding ports...
       [default] -- 22 => 2205 (adapter 1)
       [default] Running any VM customizations...
       [default] Booting VM...
       [default] Waiting for VM to boot. This can take a few minutes.
       [default] VM booted and ready for use!
       [default] Setting hostname...
       [default] Configuring and enabling network interfaces...
       [default] Mounting shared folders...
       [default] -- /vagrant
       [kitchen::driver::vagrant command] END (6m6.32s)
       [kitchen::driver::vagrant command] BEGIN (vagrant ssh-config)
       [kitchen::driver::vagrant command] END (0m1.26s)
       Vagrant instance <default-centos-59> created.
       Finished creating <default-centos-59> (6m14.04s).
-----> Kitchen is finished. (6m54.05s)
```

This will download the boxes if needed. After the creation has finished, `kitchen list` will show the updated status:

```
$ bundle exec kitchen list
Instance           Last Action
default-centos-64  Created
default-centos-59  Created
```

Now that we have the platforms available, we can attempt to converge the nodes with `kitchen converge`. Even though we haven't written a recipe yet, this will install Chef using the Omnibus plug-in, and prove that we have machines ready to test:

```
$ bundle exec kitchen converge
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Converging <default-centos-64>
       [local command] BEGIN (if ! command -v berks >/dev/null; then exit 1; fi)
       [local command] END (0m0.01s)
          [local command] BEGIN (berks install --path /tmp/default-centos-64-
cookbooks20130613-31209-12t45x0)
       Using pound (0.1.0) at path: '/home/tdi/pound'
       [local command] END (0m0.99s)
       Uploaded pound/chefignore (985 bytes)
       Uploaded pound/Berksfile.lock (179 bytes)
       Uploaded pound/Vagrantfile (3259 bytes)
       Uploaded pound/recipes/default.rb (127 bytes)
       Uploaded pound/metadata.rb (267 bytes)
       Uploaded pound/Gemfile.lock (2830 bytes)
       Uploaded pound/Berksfile (24 bytes)
       Uploaded pound/README.md (112 bytes)
       Uploaded pound/Gemfile (136 bytes)
       Uploaded pound/Thorfile (241 bytes)
       Uploaded pound/LICENSE (72 bytes)
       Starting Chef Client, version 11.4.4
       [2013-06-13T04:53:31+00:00] INFO: *** Chef 11.4.4 ***
              [2013-06-13T04:53:31+00:00] INFO: Setting the run_list to
["recipe[pound]"] from JSON
       [2013-06-13T04:53:31+00:00] INFO: Run List is [recipe[pound]]
       [2013-06-13T04:53:31+00:00] INFO: Run List expands to [pound]
       [2013-06-13T04:53:31+00:00] INFO: Starting Chef Run for default-centos-64
       [2013-06-13T04:53:31+00:00] INFO: Running start handlers
       [2013-06-13T04:53:31+00:00] INFO: Start handlers complete.
       Compiling Cookbooks...
       Converging 0 resources
        [2013-06-13T04:53:31+00:00] INFO: Chef Run complete in 0.001873196 sec-
onds
       [2013-06-13T04:53:31+00:00] INFO: Running report handlers
       [2013-06-13T04:53:31+00:00] INFO: Report handlers complete
       Chef Client finished, 0 resources updated
       Finished converging <default-centos-64> (0m3.05s).
-----> Converging <default-centos-59>
       [local command] BEGIN (if ! command -v berks >/dev/null; then exit 1; fi)
       [local command] END (0m0.01s)
```

```
          [local command] BEGIN (berks install --path /tmp/default-centos-59-
cookbooks20130613-31209-dcec44)
       Using pound (0.1.0) at path: '/home/tdi/pound'
       [local command] END (0m0.99s)
       Uploaded pound/chefignore (985 bytes)
       Uploaded pound/Berksfile.lock (179 bytes)
       Uploaded pound/Vagrantfile (3259 bytes)
       Uploaded pound/recipes/default.rb (127 bytes)
       Uploaded pound/metadata.rb (267 bytes)
       Uploaded pound/Gemfile.lock (2830 bytes)
       Uploaded pound/Berksfile (24 bytes)
       Uploaded pound/README.md (112 bytes)
       Uploaded pound/Gemfile (136 bytes)
       Uploaded pound/Thorfile (241 bytes)
       Uploaded pound/LICENSE (72 bytes)
Starting Chef Client, version 11.4.4
       [2013-06-13T04:53:34+00:00] INFO: *** Chef 11.4.4 ***
              [2013-06-13T04:53:34+00:00]  INFO:  Setting  the  run_list  to
["recipe[pound]"] from JSON
       [2013-06-13T04:53:34+00:00] INFO: Run List is [recipe[pound]]
       [2013-06-13T04:53:34+00:00] INFO: Run List expands to [pound]
       [2013-06-13T04:53:34+00:00] INFO: Starting Chef Run for default-centos-59
       [2013-06-13T04:53:34+00:00] INFO: Running start handlers
       [2013-06-13T04:53:34+00:00] INFO: Start handlers complete.
Compiling Cookbooks...
Converging 0 resources
       [2013-06-13T04:53:34+00:00] INFO: Chef Run complete in 0.002037 seconds
       [2013-06-13T04:53:34+00:00] INFO: Running report handlers
       [2013-06-13T04:53:34+00:00] INFO: Report handlers complete
Chef Client finished, 0 resources updated
       Finished converging <default-centos-59> (0m2.99s).
-----> Kitchen is finished. (0m7.15s)
```

After converge, the state is reported as:

```
$ bundle exec kitchen list
Instance           Last Action
default-centos-64  Converged
default-centos-59  Converged
```

Note that Test Kitchen uploaded the cookbooks, and would have solved any dependencies if we'd needed. The next step is to get it to run some tests. It's critical at this stage to understand how Test Kitchen achieves this. While it's perfectly possible to use Test Kitchen to run Minitest Handler tests, it's essentially designed to run what I call "Post Chef-run" tests. That is, after the Chef run has completely finished, inspect the state of the converged node, and report back. Minitest Handler is a nice approach and brings post-converge testing with minimal setup, but it does rely on being able to peek into Chef's internals, and the tests won't even attempt to run if Chef doesn't finish converging cleanly. The Test Kitchen approach is to allow the node to converge fully and after Chef has finished, inspect the state.

Test Kitchen achieves this testing using the concept of a *Busser*. Unless you're from North America, this term could be puzzling. The best explanation I can give is to refer you to a classic early episode of Seinfeld, called "The Busboy." If you've never watched Seinfeld before, stop what you're doing right now, go watch Seinfeld, and then come back. Seriously—work can wait. Back? Great, so in "The Busboy," the three friends Jerry, George, and Elaine, are eating in a restaurant, where the adjacent table catches fire. George explains to the manager of the restaurant that the fire was caused by the *busboy* leaving the menu too close to the candle. Elaine comments that she'll never eat at the restaurant again, and the manager, taking this seriously, fires the busboy. I won't spoil the rest of the episode, but you get the picture: a busboy, or busser, is a waiter's helper in a restaurant. It's his responsibility to ensure that the fruits of the chef, the produce from the kitchen, can be enjoyed by the patrons of the establishment. In Test Kitchen, the metaphor is similar. Busser is a Rubygem that is responsible for ensuring whatever is needed to run tests after a `kitchen converge` is in place. Specifically, it installs any required testing Gems, and generally helps get the remote node ready to receive test files and run them.

In theory, we could use anything we liked to test the system, after Chef has run. If you were particularly keen on Perl or Python, there would be no reason not to write tests in Perl or Python to verify the state; as long as it reports back test results, it doesn't matter what is used. Busser is fully pluggable, and creation of plug-ins is very easy. We'll look at two Busser plug-ins to demonstrate the principle.

When thinking about these sorts of tests, I think it makes sense to consider the steps you might take if you were asked to manually examine a machine to verify that something had been set up or installed. In the case of Pound, what would we do? Off the top of my head, if someone gave me a computer, told me that another sysadmin had installed Pound, and asked me to verify it, I'd probably do some of the following:

- Check to see if a Pound service was running.
- See if I could find a Pound config file.
- Look at the Pound config file to see if it looked sane.
- Look at what backends were configured.
- Make an HTTP request to a backend and note the response.
- Make an HTTP request to Pound, and compare the response to the request from the backend.

These are the sorts of tests we want Test Kitchen to run. How, then, shall we construct these tests? Well, as I mentioned, the possibilities are effectively limitless, but I will draw attention to two particularly interesting options, then mention alternatives you might like to consider.

Introducing Bats

The first is to use *Bats*, the Bourne-again shell (Bash) testing framework. About as simple as a test framework can be, a Bats test is simply a shell function with a description. Here's an example from Fletcher Nichol's rbenv cookbook:

```
@test "global Ruby can install nokogiri gem" {
  export RBENV_VERSION=$global_ruby
  run gem install nokogiri --no-ri --no-rdoc
  [ $status -eq 0 ]
}
```

Just like any test framework, we set up some state, and then make an assertion. In this case, the assertion is simply the exit status of a shell command. An exit status of 0 is interpreted as a test passing, while any non-zero exit status is interpreted as a test failure. Assertions can be any valid shell command, but the Bats framework also provides a helper method, *run*, which will run a command and store the exit status and output. These are available as three variables:

`$status`
: The exit code of the command passed as an argument to *run*.

`$output`
: The combined contents of the shell's standard out and standard error.

`$lines`
: An array that stores multiple lines of output.

This makes the final assertion as simple as utilizing the bash shell's `[ ]` testing mechanism; examples might include:

```
[ $status -eq 0 ]
[ $(echo "$output" | grep "^$global_ruby$") = "$global_ruby" ]
[ ${lines[0]} = "$global_ruby" ]
```

If you come from a Linux or Unix system administration background, you'll find this a powerful, quick, and effective way to investigate state. If this looks somewhat arcane to you, but you can see its inherent simplicity and power, there are a number of excellent introductory works on shell scripting, study of which would yield reward. Alternatively, of course, you could simply ignore this option, and move on to a testing mechanism that suits your background and purposes.

Introducing Serverspec

The second option I want to draw your attention to is *Serverspec*. Serverspec is a set of custom matchers and expectations for RSpec, designed specifically to test configured infrastructure. Although it can be configured to use SSH and connect to a remote machine, for our purposes, we're simply going to run the test after the Chef run has finished and return the result.

The project offers the following examples. I would point out that these examples use the old RSpec expectation format, which is no longer the preferred or recommended approach. Later, we'll use the current approach, but I leave these examples per the documentation, so you can see examples of each.

```
describe 'Apache package' do
  it 'should be installed' do
end

package('httpd') do
  it { should be_installed }

describe service('httpd') do
  it { should be_enabled   }
  it { should be_running   }
end

describe port(80) do
  it { should be_listening }
end

describe file('/etc/httpd/conf/httpd.conf') do
  it { should be_file }
  it { should contain "ServerName www.example.jp" }
end
```

This approach has the advantage of being familiar for anyone who has done any development in Ruby and has any exposure to RSpec. It also has the advantage that we're already using RSpec expectations in Chefspec, and RSpec expectations are commonly used with Cucumber; this gives us the opportunity to standardize on a single testing format. Additionally, there is a large number of very useful, pre-defined matchers, which makes the task of creating some immediately useful tests very easy to achieve, quickly. Finally, the project has broad cross-config-management support, being used by Puppet and CFengine users, so the community support and development effort is healthy.

The final two options I'll mention are simply writing your own tests in either Minitest or RSpec. In this case, you simply write tests using the standard library or importing any gems you need. This has the advantage of minimum fuss and maximum flexibility. If you're comfortable in the world of Ruby and Ruby testing, this will be no different from your day-to-day test-writing.

The Busser is responsible for installing any software that is required for running tests. It does this via a plug-in mechanism and by filesystem layout convention. Busser will load the plug-in that corresponds to the name of the directory. The format is as follows:

```
/path/to/my/cookbook/test/integration/<SUITE-NAME>/<BUSSER-PLUGIN>
```

So, to run Bats tests for the default suite, simply drop tests in:

```
/path/to/my/cookbook/test/integration/default/bats
```

Let's write some tests for the Pound cookbook using Bats. Create a file *pound.bats* under */test/integration/default/bats/* with the following content:

```
match() {
  local p=$1 v
  shift  for v
    do [[ $v = $p ]] && return
  done
  return 1
}

@test "The Pound service is running" {
  run service pound status
  echo "$output" | grep -Eq 'pound.*is running'
}

@test "Two Pound backends are active" {
  run poundctl -c /var/lib/pound/pound.cfg
  match "*Backend*8000*active*" "${lines[@]}"
  match "*Backend*8001*active*" "${lines[@]}"
}

@test "Pound has an HTTP listener" {
  run poundctl -c /var/lib/pound/pound.cfg
  match "*http Listener*" "${lines[@]}"
}

@test "Pound does not have an HTTPS listener" {
  run poundctl -c /var/lib/pound/pound.cfg
  ! match "*HTTPS Listener*" "${lines[@]}"
}

@test "Server is listening on port 80" {
  run nmap -sT -p80 localhost
  match "80/tcp open  http" "${lines[@]}"
}

@test "Server accepts HTTP requests" {
  echo "GET / HTTP/1.1" | nc localhost 80
}
```

Obviously being able to write this test assumes some degree of familiarity with the system you're going to configure. Naturally you could write much more basic tests at first and evolve more complex ones as you discover functionality you want to test.

Let's quickly run through the test. First, we set up a function that will check for a match in the lines of an array. In the first test, we're just checking that we see the Pound service running. This isn't very cross-platform, as we're relying on the format of the service command, which may be different on alternative versions or distributions, but it illustrates a simple `grep`. The next three tests all use the `match` function, and the built-in `run` function. `Poundctl` is a command-line utility that will dump out the running

configuration of the service—we're just checking against its output. The final two tests use the `netcat` and `nmap` commands to do primitive network testing. These could be much more complex if needed. The latter of the two tests simply makes use of the return code—if `netcat` cannot reach the machine on port 80, it will have a non-zero exit code.

These two tests illustrate a further important feature of Test Kitchen. Fairly often we find that for test purposes we would like to have some handy commands—for example `netcat`, `lsof`, or `telnet`. We might not normally have these in our base build, but we want them to be available for running our post–Chef-run tests. Test Kitchen allows these prerequisites to be installed after the Chef run by dropping off a file called *prepare_recipe.rb*, containing recipe DSL code, which is executed using a slightly modified `chef-apply`. In our case we would add:

```
$ cat test/integration/default/bats/prepare_recipe.rb
%w{ nc nmap }.each { |pkg| package pkg }
```

Of course, to be truly cross-platform, we'd need to take into account the different naming conventions of various Linux distributions, but the principle is clear.

Having written the tests, we now want to run them. Based on the five lifecycle phases, as described earlier, Test Kitchen provides a number of commands that control the lifecycle of a test suite. In order, they are:

`kitchen create`
: Creates the base machine

`kitchen converge`
: Installs and runs Chef with the run list specified in the *.kitchen.yml* file

`kitchen setup`
: Instructs Busser to set up whatever is needed to run tests

`kitchen verify`
: Runs the tests to verify that the state of the machine is as desired and/or expected

`kitchen destroy`
: Destroys the machine entirely, leaving the host OS in a clean state

These tasks can be called individually—one at a time—but later commands in the lifecycle will attempt to call previous steps. So, running `kitchen verify` will create, converge, and set up a machine before verifying. The tasks take an argument of which instance to control. The commands perform a regular expression match, which makes it convenient to run actions against a specified subset of machines reported by `kitchen list`. With no pattern, the default is to take action against *all* the instances. This can be made explicit with the `all` keyword.

Let's run them one at a time to see how they function:

```
$ kitchen create 6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Creating <default-centos-64>
       [kitchen::driver::vagrant command] BEGIN (vagrant up --no-provision)
       Bringing machine 'default' up with 'virtualbox' provider...
       [default] Importing base box 'opscode-centos-6.4'...
       [default] Matching MAC address for NAT networking...
       [default] Setting the name of the VM...
       [default] Clearing any previously set forwarded ports...
       [Berkshelf] Skipping Berkshelf with --no-provision
       [default] Fixed port collision for 22 => 2222. Now on port 2204.
       [default] Creating shared folders metadata...
       [default] Clearing any previously set network interfaces...
       [default] Preparing network interfaces based on configuration...
       [default] Forwarding ports...
       [default] -- 22 => 2204 (adapter 1)
       [default] Running any VM customizations...
       [default] Booting VM...
       [default] Waiting for VM to boot. This can take a few minutes.
       [default] VM booted and ready for use!
       [default] Setting hostname...
       [default] Configuring and enabling network interfaces...
       [default] Mounting shared folders...
       [default] -- /vagrant
       [kitchen::driver::vagrant command] END (0m37.19s)
       [kitchen::driver::vagrant command] BEGIN (vagrant ssh-config)
       [kitchen::driver::vagrant command] END (0m1.25s)
       Vagrant instance <default-centos-64> created.
       Finished creating <default-centos-64> (0m39.43s).
-----> Kitchen is finished. (0m40.49s)
```

We now have a CentOS 6.4 machine ready for action. We can connect to the machine and look around using `kitchen login`:

```
$ kitchen login 6
Last login: Sat May 11 04:55:22 2013 from 10.0.2.2
[vagrant@default-centos-64 ~]$ uname -a
Linux default-centos-64.vagrantup.com 2.6.32-358.el6.x86_64 #1 SMP Fri Feb 22
00:31:26 UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
[vagrant@default-centos-64 ~]$ exit
logout
Connection to 127.0.0.1 closed.
```

Now let's converge the node:

```
$ kitchen  converge 6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Converging <default-centos-64>
-----> Installing Chef Omnibus (true)
       --2013-06-15 02:56:11--  https://www.opscode.com/chef/install.sh
Resolving www.opscode.com...        184.106.28.83
       Connecting to www.opscode.com|184.106.28.83|:443...
       connected.
```

```
HTTP request sent, awaiting response...        200 OK
       Length: 6510 (6.4K) [application/x-sh]
       Saving to: “STDOUT”

 0% [                                       ] 0           --.-K/s
100%[======================================>] 6,510       --.-K/s   in 0s

       2013-06-15 02:56:11 (1.28 GB/s) - written to stdout [6510/6510]

       Downloading Chef  for el...
       Installing Chef
         warning: /tmp/tmp.69MKcEOA/chef-.x86_64.rpm: Header V4 DSA/SHA1 Signa-
ture, key ID 83ef826a: NOKEY
Preparing...                #####  ##########################################
[100%]
   1:chef                           ##########################################
[100%]
       Thank you for installing Chef!
       [local command] BEGIN (if ! command -v berks >/dev/null; then exit 1; fi)
       [local command] END (0m0.01s)
          [local command] BEGIN (berks install --path /tmp/default-centos-64-
cookbooks20130615-18465-o79vfq)
       Using pound (0.1.0) at path: '/home/tdi/pound'
       [local command] END (0m1.63s)
       Uploaded pound/chefignore (985 bytes)
       Uploaded pound/Berksfile.lock (179 bytes)
       Uploaded pound/Vagrantfile (3259 bytes)
       Uploaded pound/recipes/default.rb (226 bytes)
       Uploaded pound/metadata.rb (281 bytes)
       Uploaded pound/Gemfile.lock (2830 bytes)
       Uploaded pound/Berksfile (24 bytes)
       Uploaded pound/README.md (112 bytes)
       Uploaded pound/Gemfile (136 bytes)
        Uploaded pound/test/integration/default/bats/.kitchen/logs/celluloid.log
(0 bytes)
         Uploaded pound/test/integration/default/bats/.kitchen/logs/kitchen.log
(3033 bytes)
          Uploaded pound/test/integration/default/bats/pound.bats-disabled (942
bytes)
       Uploaded pound/test/integration/default/bats/prepare_recipe.rb (42 bytes)
       Uploaded pound/test/integration/default/bats/pound.bats (942 bytes)
       Uploaded pound/test/.kitchen/logs/celluloid.log (0 bytes)
       Uploaded pound/test/.kitchen/logs/kitchen.log (3735 bytes)
       Uploaded pound/Thorfile (241 bytes)
       Uploaded pound/LICENSE (72 bytes)
       Starting Chef Client, version 11.4.4
       [2013-06-15T02:56:35+00:00] INFO: *** Chef 11.4.4 ***
              [2013-06-15T02:56:35+00:00]  INFO:  Setting  the  run_list  to
["recipe[pound]"] from JSON
       [2013-06-15T02:56:35+00:00] INFO: Run List is [recipe[pound]]
       [2013-06-15T02:56:35+00:00] INFO: Run List expands to [pound]
       [2013-06-15T02:56:35+00:00] INFO: Starting Chef Run for default-centos-64
```

```
       [2013-06-15T02:56:35+00:00] INFO: Running start handlers
       [2013-06-15T02:56:35+00:00] INFO: Start handlers complete.
       Compiling Cookbooks...
       Converging 0 resources
        [2013-06-15T02:56:35+00:00] INFO: Chef Run complete in 0.001689128 sec-
onds
       [2013-06-15T02:56:35+00:00] INFO: Running report handlers
       [2013-06-15T02:56:35+00:00] INFO: Report handlers complete
       Chef Client finished, 0 resources updated
       Finished converging <default-centos-64> (0m24.62s).
-----> Kitchen is finished. (0m25.69s)
```

Test Kitchen installs Chef, and then runs it. It also uses Berkshelf to solve any dependencies. Now let's run the setup task:

```
$ kitchen  setup  6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Setting up <default-centos-64>
Fetching: thor-0.18.1.gem (100%)
Fetching: busser-0.4.1.gem (100%)
       Successfully installed thor-0.18.1
       Successfully installed busser-0.4.1
       2 gems installed
-----> Setting up Busser
       Creating BUSSER_ROOT in /opt/busser
       Creating busser binstub
       Plugin bats installed (version 0.1.0)
-----> Running postinstall for bats plugin
             create  /tmp/bats20130615-2256-hylsr3/bats
             create  /tmp/bats20130615-2256-hylsr3/bats.tar.gz
       Installed Bats to /opt/busser/vendor/bats/bin/bats
             remove  /tmp/bats20130615-2256-hylsr3
       Finished setting up <default-centos-64> (0m8.30s).
-----> Kitchen is finished. (0m9.37s)
```

We're ready to run the tests now:

```
$ kitchen  verify 6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Verifying <default-centos-64>
       Suite path directory /opt/busser/suites does not exist, skipping.
       Uploading /opt/busser/suites/bats/prepare_recipe.rb (mode=0664)
       Uploading /opt/busser/suites/bats/pound.bats (mode=0664)
       Uploading /opt/busser/suites/bats/pound.bats-disabled (mode=0664)
-----> Running bats test suite
-----> Preparing bats suite with /opt/busser/suites/bats/prepare_recipe.rb
       [2013-06-15T02:58:27+00:00] INFO: Run List is []
       [2013-06-15T02:58:27+00:00] INFO: Run List expands to []
       Recipe: (chef-apply cookbook)::(chef-apply recipe)
          * package[nc] action install[2013-06-15T02:58:27+00:00] INFO: Process-
ing package[nc] action install ((chef-apply cookbook)::(chef-apply recipe) line
1)
        [2013-06-15T02:58:37+00:00] INFO: package[nc] installing nc-1.84-22.el6
```

```
from base repository

           - install version 1.84-22.el6 of package nc

           * package[nmap] action install[2013-06-15T02:58:40+00:00] INFO: Pro-
cessing  package[nmap]  action  install  ((chef-apply  cookbook)::(chef-apply
recipe) line 1)
                [2013-06-15T02:58:40+00:00]  INFO:  package[nmap]  installing
nmap-5.51-2.el6 from base repository

           - install version 5.51-2.el6 of package nmap
       1..6
       not ok 1 The Pound service is running
       # /opt/busser/suites/bats/pound.bats:12
       not ok 2 Two Pound backends are active
       # /opt/busser/suites/bats/pound.bats:17
       not ok 3 Pound has an HTTP listener
       # /opt/busser/suites/bats/pound.bats:7
       ok 4 Pound does not have an HTTPS listener
       not ok 5 Server is listening on port 80
       # /opt/busser/suites/bats/pound.bats:7
       not ok 6 Server accepts HTTP requests
       # /opt/busser/suites/bats/pound.bats:38
        Command [/opt/busser/vendor/bats/bin/bats /opt/busser/suites/bats] exit
code was 1
>>>>>> Verify failed on instance <default-centos-64>.
>>>>>> Please see .kitchen/logs/default-centos-64.log for more details
>>>>>> ------Exception-------
>>>>>> Class: Kitchen::ActionFailed
>>>>>> Message: SSH exited (1) for command: [sudo -E /opt/busser/bin/busser
test]
>>>>>> ----------------------
```

Alright! Failing tests! We see the extra tools being installed, and then the tests running and failing. Note that we could have done this in one go, by calling the `kitchen verify` step, rather than each individual step.

We can make these tests pass very easily. Open up the default recipe and add the following:

```
include_recipe 'yum::epel'

package 'Pound'

service 'pound' do
  action [:enable, :start]
end
```

Add the dependency on the yum cookbook to the metadata and run `kitchen converge`. Take a deep breath, there's a lot of output to read:

```
$ kitchen converge 6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Converging <default-centos-64>
       [local command] BEGIN (if ! command -v berks >/dev/null; then exit 1; fi)
       [local command] END (0m0.01s)
          [local command] BEGIN (berks install --path /tmp/default-centos-64-
cookbooks20130615-31535-12jfpi7)
       Using pound (0.1.0) at path: '/home/tdi/pound'
       Using yum (2.2.2)
       [local command] END (0m1.75s)
       Uploaded yum/metadata.json (11004 bytes)
       Uploaded yum/CONTRIBUTING.md (10811 bytes)
       Uploaded yum/resources/key.rb (831 bytes)
       Uploaded yum/resources/repository.rb (1585 bytes)
       Uploaded yum/files/default/tests/minitest/support/helpers.rb (1280 bytes)
       Uploaded yum/files/default/tests/minitest/test_test.rb (1709 bytes)
       Uploaded yum/files/default/tests/minitest/default_test.rb (828 bytes)
       Uploaded yum/recipes/ius.rb (1537 bytes)
       Uploaded yum/recipes/remi.rb (1140 bytes)
       Uploaded yum/recipes/test.rb (1150 bytes)
       Uploaded yum/recipes/repoforge.rb (1716 bytes)
       Uploaded yum/recipes/yum.rb (748 bytes)
       Uploaded yum/recipes/elrepo.rb (1028 bytes)
       Uploaded yum/recipes/default.rb (625 bytes)
       Uploaded yum/recipes/epel.rb (1181 bytes)
       Uploaded yum/metadata.rb (1492 bytes)
Uploaded yum/Berksfile (81 bytes)
       Uploaded yum/providers/key.rb (2242 bytes)
       Uploaded yum/providers/repository.rb (4235 bytes)
       Uploaded yum/templates/default/yum-rhel6.conf.erb (1367 bytes)
       Uploaded yum/templates/default/yum-rhel5.conf.erb (900 bytes)
       Uploaded yum/templates/default/repo.erb (803 bytes)
       Uploaded yum/README.md (8405 bytes)
       Uploaded yum/CHANGELOG.md (2797 bytes)
       Uploaded yum/attributes/remi.rb (1146 bytes)
       Uploaded yum/attributes/elrepo.rb (970 bytes)
       Uploaded yum/attributes/default.rb (1076 bytes)
       Uploaded yum/attributes/epel.rb (1448 bytes)
       Uploaded yum/LICENSE (10850 bytes)
       Uploaded pound/chefignore (985 bytes)
       Uploaded pound/Berksfile.lock (179 bytes)
       Uploaded pound/Vagrantfile (3259 bytes)
       Uploaded pound/recipes/default.rb (223 bytes)
       Uploaded pound/metadata.rb (280 bytes)
       Uploaded pound/Gemfile.lock (2830 bytes)
       Uploaded pound/Berksfile (24 bytes)
       Uploaded pound/README.md (112 bytes)
       Uploaded pound/Gemfile (136 bytes)
        Uploaded pound/test/integration/default/bats/.kitchen/logs/celluloid.log
(0 bytes)
         Uploaded pound/test/integration/default/bats/.kitchen/logs/kitchen.log
(3033 bytes)
```

```
         Uploaded pound/test/integration/default/bats/pound.bats-disabled (942
bytes)
       Uploaded pound/test/integration/default/bats/prepare_recipe.rb (42 bytes)
       Uploaded pound/test/integration/default/bats/pound.bats (942 bytes)
       Uploaded pound/test/.kitchen/logs/celluloid.log (0 bytes)
       Uploaded pound/test/.kitchen/logs/kitchen.log (3735 bytes)
       Uploaded pound/Thorfile (241 bytes)
       Uploaded pound/LICENSE (72 bytes)
       Starting Chef Client, version 11.4.4
       [2013-06-15T05:43:35+00:00] INFO: *** Chef 11.4.4 ***
              [2013-06-15T05:43:36+00:00]  INFO:  Setting  the  run_list  to
["recipe[pound]"] from JSON
       [2013-06-15T05:43:36+00:00] INFO: Run List is [recipe[pound]]
       [2013-06-15T05:43:36+00:00] INFO: Run List expands to [pound]
       [2013-06-15T05:43:36+00:00] INFO: Starting Chef Run for default-centos-64
       [2013-06-15T05:43:36+00:00] INFO: Running start handlers
       [2013-06-15T05:43:36+00:00] INFO: Start handlers complete.
       Compiling Cookbooks...
       Converging 4 resources
       Recipe: yum::epel
         * yum_key[RPM-GPG-KEY-EPEL-6] action add[2013-06-15T05:43:36+00:00] IN-
FO: Processing yum_key[RPM-GPG-KEY-EPEL-6] action add (yum::epel line 22)
          [2013-06-15T05:43:36+00:00] INFO: Adding RPM-GPG-KEY-EPEL-6 GPG key
to /etc/pki/rpm-gpg/
        (up to date)
       Recipe: <Dynamically Defined Resource>
          * package[gnupg2] action install[2013-06-15T05:43:36+00:00] INFO: Pro-
cessing  package[gnupg2]  action  install  (/tmp/kitchen-chef-solo/cookbooks/yum/
providers/key.rb line 32)
        (up to date)
                    *  execute[import-rpm-gpg-key-RPM-GPG-KEY-EPEL-6]  action
nothing[2013-06-15T05:43:37+00:00] INFO: Processing execute[import-rpm-gpg-key-
RPM-GPG-KEY-EPEL-6] action nothing (/tmp/kitchen-chef-solo/cookbooks/yum/provid-
ers/key.rb line 35)
        (skipped due to not_if)
         * remote_file[/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6] action create
          [2013-06-15T05:43:37+00:00] INFO: Processing remote_file[/etc/pki/rpm-
gpg/RPM-GPG-KEY-EPEL-6]  action  create  (/tmp/kitchen-chef-solo/cookbooks/yum/
providers/key.rb line 61)
         [2013-06-15T05:43:38+00:00] INFO: remote_file[/etc/pki/rpm-gpg/RPM-GPG-
KEY-EPEL-6] updated

              - copy file downloaded from [] into /etc/pki/rpm-gpg/RPM-GPG-KEY-
EPEL-6
               ---  /tmp/chef-tempfile20130615-578-gjrfug            2013-06-15
05:43:38.676574342 +0000
          +++ /tmp/chef-rest20130615-578-1bv08zs  2013-06-15 05:43:38.676574339
+0000
        @@ -0,0 +1,29 @@
        +-----BEGIN PGP PUBLIC KEY BLOCK-----
        +Version: GnuPG v1.4.5 (GNU/Linux)
        +
```

```
        +mQINBEvSKUIBEADLGnUj24ZVKW7liFN/JA5CgtzlNnKs7sBg7fVbNWryiE3URbn1
        +JXvrdwHtkKyY96/ifZ1Ld3lE2gOF61bGZ2CWwJNee76Sp9Z+isP8RQXbG5jwj/4B
        +M9HK7phktqFVJ8VbY2jfTjcfxRvGM8YBwXF8hx0CDZURAjvf1xRSQJ7iAo58qcHn
        +XtxOAvQmAbR9z6Q/h/D+Y/PhoIJp1OV4VNHCbCs9M7HUVBpgC53PDcTUQuwcgeY6
        +pQgo9eT1eLNSZVrJ5Bctivl1UcD6P6CIGkkeT2gNhqindRPngUXGXW7Qzoefe+fV
        +QqJSm7Tq2q9oqVZ46J964waCRItRySpuW5dxZO34WM6wsw2BP2MlACbH4l3luqtp
 +Xo3Bvfnk+HAFH3HcMuwdaulxv7zYKXCfNoSfgrpEfo2Ex4Im/I3WdtwME/Gbnwdq
        +3VJzgAxLVFhczDHwNkjmIdPAlNJ9/ixRjip4dgZtW8VcBCrNoL+LhDrIfjvnLdRu
        +vBHy9P3sCF7FZycaHlMWP6RiLtHnEMGcbZ8QpQHi2dReU1wyr9QgguGU+jqSXYar
        +1yEcsdRGasppNIZ8+Qawbm/a4doT10TEtPArhSoHlwbvqTDYjtfV92lC/2iwgO6g
        +YgG9XrO4V8dV39Ffm7oLFfvTbg5mv4Q/E6AWo/gkjmtxkculbyAvjFtYAQARAQAB
        +tCFFUEVMICg2KSA8ZXBlbEBmZWRvcmFwcm9qZWN0Lm9yZz6JAjYEEwECACAFAkvS
        +KUICGw8GCwkIBwMCBBUCCAMEFgIDAQIeAQIXgAAKCRA7Sd8qBgi4lR/GD/wLGPv9
        +qO39eyb9NlrwfKdUEo1tHxKdrhNz+XYrO4yVDTBZRPSuvL2yaoeSIhQOKhNPfEgT
        +9mdsbsgcfmoHxmGVcn+lbheWsSvcgrXuz0gLt8TGGKGGROAoLXpuUsb1HNtKEOwP
        +Q4z1uQ2nOz5hLRyDOV0I2LwYV8BjGIjBKUMFEUxFTsL7XOZkrAg/WbTH2PW3hrfS
        +WtcRA7EYonI3B80d39ffws7SmyKbS5PmZjqOPuTvV2F0tMhKIhncBwoojWZPExft
        +HpKhzKVh8fdDO/3P1y1Fk3Cin8UbCO9MWMFNR27fVzCANlEPljsHA+3Ez4F7uboF
        +p0OOEov4Yyi4BEbgqZnthTG4ub9nyiupIZ3ckPHr3nVcDUGcL6lQD/nkmNVIeLYP
        +x1uHPOSlWfuojAYgzRH6LL7Idg4FHHBA0to7FW8dQXFIOyNiJFAOT2j8P5+tVdq8
        +wB0PDSH8yRpn4HdJ9RYquau4OkjluxOWf0uRaS//SUcCZh+1/KBEOmcvBHYRZA5J
        +l/nakCgxGb2paQOzqqpOcHKvlyLuzO5uybMXaipLExTGJXBlXrbbASfXa/yGYSAG
        +iVrGz9CE6676dMlm8F+s3XXE13QZrXmjloc6jwOljnfAkjTGXjiB7OULESed96MR
        +XtfLk0W5Ab9pd7tKDR6QHI7rgHXfCopRnZ2VVQ==
        +=V/6I
        +-----END PGP PUBLIC KEY BLOCK-----[2013-06-15T05:43:38+00:00] INFO: re-
mote_file[/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6] mode changed to 644

          - change mode from '' to '0644'

         [2013-06-15T05:43:38+00:00] INFO: remote_file[/etc/pki/rpm-gpg/RPM-GPG-
KEY-EPEL-6] sending run action to execute[import-rpm-gpg-key-RPM-GPG-KEY-
EPEL-6] (immediate)
                    *  execute[import-rpm-gpg-key-RPM-GPG-KEY-EPEL-6]  action
run[2013-06-15T05:43:38+00:00] INFO: Processing execute[import-rpm-gpg-key-RPM-
GPG-KEY-EPEL-6]  action  run  (/tmp/kitchen-chef-solo/cookbooks/yum/providers/
key.rb line 35)
       [2013-06-15T05:43:38+00:00] INFO: execute[import-rpm-gpg-key-RPM-GPG-KEY-
EPEL-6] ran successfully

          - execute rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6

       Recipe: yum::epel
          * yum_repository[epel] action create[2013-06-15T05:43:38+00:00] INFO:
Processing yum_repository[epel] action create (yum::epel line 27)
        [2013-06-15T05:43:38+00:00] INFO: Adding and updating epel repository
in /etc/yum.repos.d/epel.repo
          [2013-06-15T05:43:38+00:00] WARN: Cloning resource attributes for
yum_key[RPM-GPG-KEY-EPEL-6] from prior resource (CHEF-3694)
             [2013-06-15T05:43:38+00:00]  WARN:  Previous  yum_key[RPM-GPG-KEY-
EPEL-6]: /tmp/kitchen-chef-solo/cookbooks/yum/recipes/epel.rb:22:in `from_file'
            [2013-06-15T05:43:38+00:00]  WARN:  Current    yum_key[RPM-GPG-KEY-
```

```
EPEL-6]: /tmp/kitchen-chef-solo/cookbooks/yum/providers/repository.rb:85:in `re-
po_config'
        (up to date)
       Recipe: <Dynamically Defined Resource>
         * yum_key[RPM-GPG-KEY-EPEL-6] action add[2013-06-15T05:43:38+00:00] IN-
FO: Processing yum_key[RPM-GPG-KEY-EPEL-6] action add (/tmp/kitchen-chef-solo/
cookbooks/yum/providers/repository.rb line 85)
        (up to date)
         * execute[yum-makecache] action nothing[2013-06-15T05:43:38+00:00] IN-
FO: Processing execute[yum-makecache] action nothing (/tmp/kitchen-chef-solo/
cookbooks/yum/providers/repository.rb line 88)
        (up to date)
                          *   ruby_block[reload-internal-yum-cache]   action
nothing[2013-06-15T05:43:38+00:00] INFO: Processing ruby_block[reload-internal-
yum-cache] action nothing (/tmp/kitchen-chef-solo/cookbooks/yum/providers/repos-
itory.rb line 93)
        (up to date)
         * template[/etc/yum.repos.d/epel.repo] action create
         [2013-06-15T05:43:38+00:00] INFO: Processing template[/etc/yum.repos.d/
epel.repo] action create (/tmp/kitchen-chef-solo/cookbooks/yum/providers/reposi-
tory.rb line 100)
         [2013-06-15T05:43:38+00:00] INFO: template[/etc/yum.repos.d/epel.repo]
updated content
         [2013-06-15T05:43:38+00:00] INFO: template[/etc/yum.repos.d/epel.repo]
mode changed to 644

           - create template[/etc/yum.repos.d/epel.repo]
               ---  /tmp/chef-tempfile20130615-578-1rq8217           2013-06-15
05:43:38.819576393 +0000
           +++ /tmp/chef-rendered-template20130615-578-z3junu        2013-06-15
05:43:38.819576393 +0000
        @@ -0,0 +1,8 @@
        +# Generated by Chef for default-centos-64.vagrantup.com
        +# Local modifications will be overwritten.
        +[epel]
        +name=Extra Packages for Enterprise Linux
                        +mirrorlist=http://mirrors.fedoraproject.org/mirrorlist?
repo=epel-6&arch=$basearch
        +gpgcheck=1
        +gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
        +enabled=1

         [2013-06-15T05:43:38+00:00] INFO: template[/etc/yum.repos.d/epel.repo]
sending run action to execute[yum-makecache] (immediate)
          * execute[yum-makecache] action run[2013-06-15T05:43:38+00:00] INFO:
Processing    execute[yum-makecache]    action    run    (/tmp/kitchen-chef-solo/
cookbooks/yum/providers/repository.rb line 88)
       [2013-06-15T05:43:57+00:00] INFO: execute[yum-makecache] ran successfully

           - execute yum -q makecache
         [2013-06-15T05:43:57+00:00] INFO: template[/etc/yum.repos.d/epel.repo]
sending create action to ruby_block[reload-internal-yum-cache] (immediate)
```

```
                          *   ruby_block[reload-internal-yum-cache]   action
create[2013-06-15T05:43:57+00:00]  INFO:  Processing  ruby_block[reload-internal-
yum-cache] action create (/tmp/kitchen-chef-solo/cookbooks/yum/providers/reposi-
tory.rb line 93)
        [2013-06-15T05:43:57+00:00] INFO: ruby_block[reload-internal-yum-cache]
called

           - execute the ruby block reload-internal-yum-cache

       Recipe: pound::default
          * package[Pound] action install[2013-06-15T05:43:57+00:00] INFO: Pro-
cessing package[Pound] action install (pound::default line 11)
               [2013-06-15T05:44:00+00:00]  INFO:  package[Pound]  installing
Pound-2.6-2.el6 from epel repository

           - install version 2.6-2.el6 of package Pound

           * service[pound] action enable[2013-06-15T05:44:03+00:00] INFO: Pro-
cessing service[pound] action enable (pound::default line 13)
       [2013-06-15T05:44:03+00:00] INFO: service[pound] enabled

           - enable service service[pound]

         * service[pound] action start[2013-06-15T05:44:03+00:00] INFO: Process-
ing service[pound] action start (pound::default line 13)
       [2013-06-15T05:44:03+00:00] INFO: service[pound] started

           - start service service[pound]

        [2013-06-15T05:44:03+00:00] INFO: Chef Run complete in 27.503439435 sec-
onds
       [2013-06-15T05:44:03+00:00] INFO: Running report handlers
       [2013-06-15T05:44:03+00:00] INFO: Report handlers complete
       Chef Client finished, 8 resources updated
       Finished converging <default-centos-64> (0m32.78s).
-----> Kitchen is finished. (0m33.85s)
```

Test Kitchen passed off the dependency to Berkshelf, the node was converged, the EPEL repo was installed, then Pound was installed, and the service was started. In 30 seconds. Now let's look at the tests:

```
$ kitchen  verify 6
-----> Starting Kitchen (v1.0.0.alpha.7)
-----> Verifying <default-centos-64>
       Removing /opt/busser/suites/bats
       Uploading /opt/busser/suites/bats/prepare_recipe.rb (mode=0664)
       Uploading /opt/busser/suites/bats/pound.bats (mode=0664)
       Uploading /opt/busser/suites/bats/pound.bats-disabled (mode=0664)
-----> Running bats test suite
-----> Preparing bats suite with /opt/busser/suites/bats/prepare_recipe.rb
       [2013-06-15T05:48:23+00:00] INFO: Run List is []
       [2013-06-15T05:48:23+00:00] INFO: Run List expands to []
```

```
       Recipe: (chef-apply cookbook)::(chef-apply recipe)
          * package[nc] action install[2013-06-15T05:48:23+00:00] INFO: Process-
ing package[nc] action install ((chef-apply cookbook)::(chef-apply recipe) line
1)
        (up to date)
           * package[nmap] action install[2013-06-15T05:48:26+00:00] INFO: Pro-
cessing  package[nmap]  action  install  ((chef-apply  cookbook)::(chef-apply
recipe) line 1)
        (up to date)
       1..6
       ok 1 The Pound service is running
       ok 2 Two Pound backends are active
       ok 3 Pound has an HTTP listener
       not ok 4 Pound does not have an HTTPS listener
       # /opt/busser/suites/bats/pound.bats:5
       ok 5 Server is listening on port 80
       ok 6 Server accepts HTTP requests
        Command [/opt/busser/vendor/bats/bin/bats /opt/busser/suites/bats] exit
code was 1
>>>>>> Verify failed on instance <default-centos-64>.
>>>>>> Please see .kitchen/logs/default-centos-64.log for more details
>>>>>> ------Exception-------
>>>>>> Class: Kitchen::ActionFailed
>>>>>> Message: SSH exited (1) for command: [sudo -E /opt/busser/bin/busser
test]
>>>>>> ----------------------
```

Five out of six tests pass. However, we do have an HTTPS listener. We'll need to disable that in the configuration. This allows us to introduce another feature of the Chef DSL: *templates*.

Templates

Templates are much like the parlor game "Consequences," one version of which has the following rules: The object is to construct an amusingly random narrative based around a chance encounter between two people. Three or more players get together. Each player takes a sheet of blank paper and writes one section of the narrative. The paper is then folded and passed on. Here are the sections:

1. A description beginning with the word *the* (e.g., *The beautiful*, *The very talkative*)
2. A man's name
3. A second description, as above
4. A woman's name
5. Where they met
6. What he gave her
7. What she said

8. What he did
9. What the consequence was
10. What the world said about it

After all the sections have been completed, each part of the narrative is read aloud, inserting the words "met," "at," and so on, where appropriate. Much hilarity ensues.

We all understand these instructions; we interpret the sections and replace each one with appropriate words. What we're doing is following a template. Ruby (and indeed Chef) has this concept. Let's illustrate this with the same game. A template that matches these instructions might look like the following:

```
The <%= @description_one.downcase %> <%= @man %> met the <%= @descrip-
tion_two.downcase %> <%= @woman %> at <%= @location %>.
He gave her <%= @gift.downcase %>, and she said "<%= @woman_saying %>".  He did
<%= @man_action.downcase %>.
The consequence was <%= @consequence %>, and the world said "<%= @world_saying
%>"
```

This is an *Embedded Ruby*, or *ERB*, template. This allows Ruby code to be embedded within a pair of `<%` and `%>` delimiters. These embedded code blocks are then evaluated in place (they are replaced by the result of their evaluation). In this example, we're using *expression result substitution*, which is denoted by `<%= %>` delimiters. The result of the Ruby expression is printed. The `@something` variables are instance variables, which you'll remember are variables that describe the attributes of an instance of a class, in object-oriented programming. They're always preceded with the `@` sign—you can remember them by the connection between the @ symbol, and the attributes they describe. In this case we're seeing the attributes of an ERB template. These variables are said to be *passed into* the template—in this case as strings, which is why in places we can call the `String#downcase` method.

I wrote this silly example to introduce ERB templates:

```
require 'ztk'

male_descriptions = [
   "dashingly handsome",
   "tanned, muscular",
   "shockingly rude",
   "diffident, bespectacled"
]

men = [
   "Karl Barth",
   "Nelson Mandela",
   "Tigran Petrosian",
   "Ian Botham"
]
```

```
female_descriptions = [
   "doughty, tweedy",
   "ravishing",
   "brilliantly intelligent",
   "austere, high-minded"
]

women = [
   "Zola Budd",
   "Margaret Thatcher",
   "Audrey Hepburn",
   "Marie Antoinette"
]

locations = [
   "the pub",
   "freshers' fair",
   "Chefconf",
   "the opera"
]

gifts = [
   "jam trousers",
   "chocolate cake",
   "fish knives",
   "a blank cheque"
]

woman_sayings = [
   "I have a dream!",
   "The future is much like the present, only longer",
   "The wisest men follow their own direction",
   "I despise the pleasure of pleasing people that I despise"
]

man_actions = [
   "Danced a jig",
   "joined a Buddhist monastery",
   "fell dead on the spot",
   "sold all his possessions"
]

consequences = [
   "world peace",
   "global warming",
   "a sharp rise in interest rates",
   "entirely unremarkable"
]

world_sayings = [
   "I don't suffer from insanity. I enjoy every minute of it.",
```

```
    "I'll be the in to your sane.",
    "Use it or lose it is a cliche because it's true.",
    "That which does not kill us makes us stronger."
]

output = ZTK::Template.render( "consequences.erb",
    {
        :description_one => male_descriptions.sample,
        :man => men.sample,
        :description_two => female_descriptions.sample,
        :woman => women.sample,
        :location => locations.sample,
        :gift => gifts.sample,
        :woman_saying => woman_sayings.sample,
        :man_action => actions.sample,
        :consequence => consequences.sample,
        :world_saying => world_sayings.sample
    }
)

puts output
```

All this does is feed random strings into the template and print the output. It uses the handy ZTK template class from Zachary Patten, co-author of *Cucumber-Chef*. Let's give it a few spins:

```
> ruby consequences.rb
The dashingly handsome Karl Barth met the austere, high-minded Audrey Hepburn
at Chefconf.
He gave her jam trousers, and she said "The wisest men follow their own direc-
tion".  He fell dead on the spot.
The consequence was entirely unremarkable, and the world said "Use it or lose
it is a cliche because it's true."

> ruby consequences.rb
The diffident, bespectacled Nelson Mandela met the brilliantly intelligent Mar-
garet Thatcher at the pub.
He gave her fish knives, and she said "The wisest men follow their own direc-
tion".  He joined a Buddhist monastery.
The consequence was entirely unremarkable, and the world said "I don't suffer
from insanity.  I enjoy every minute of it."

> ruby consequences.rb
The shockingly rude Tigran Petrosian met the doughty, tweedy Audrey Hepburn at
freshers' fair.
He gave her a blank cheque, and she said "The future is much like the present,
only longer".  He sold all his possessions.
The consequence was global warming, and the world said "That which does not
kill us makes us stronger."
```

The Chef template resource behaves very similarly. We can pass in variables that are rendered as instance variables in the template, or we can use attributes on the node, directly within `<%= %>` tags.

Copy the config file from the machine under test into *templates/default/pound.cfg.erb/*.

```
User "pound"
Group "pound"
Control "/var/lib/pound/pound.cfg"

ListenHTTP
    Address 0.0.0.0
    Port 80
End

ListenHTTPS
    Address 0.0.0.0
    Port    443
    Cert    "/etc/pki/tls/certs/pound.pem"
End

Service
    BackEnd
        Address 127.0.0.1
        Port    8000
    End

    BackEnd
        Address 127.0.0.1
        Port    8001
    End
End
```

We need to disable the HTTPS functionality. We could simply delete it and serve the configuration file as a static asset. However, as a cookbook maintainer, it's usually wise to provide attributes to make the cookbook flexible and easy to configure. Looking at this file, I can see a number of candidates for abstraction into attributes—the user and group, the ports, whether or not to even run SSL, where the SSL certificate is found, the control file, and even the address of the backends. All these are data that we would like to be able to control. Let's leave the backend config for now, but configure the rest. Create a default attributes file in the cookbook:

```
default['pound']['user'] = 'pound'
default['pound']['group'] = 'pound'
default['pound']['port'] = '80'
default['pound']['control'] = '/var/lib/pound/pound.cfg'
default['pound']['ssl']['enabled'] = false
default['pound']['ssl']['cert'] = '/etc/pki/tls/certs/pound.pem'
default['pound']['ssl']['port'] = '443'
```

Now we need to get these values into the template:

```
User "<%= node['pound']['user'] %>"
Group "<%= node['pound']['group'] %>"
Control "<%= node['pound']['control'] %>"

ListenHTTP
    Address 0.0.0.0
    Port <%= node['pound']['port'] %>
End

<% if node['pound']['ssl']['enabled'] -%>
ListenHTTPS
    Address 0.0.0.0
    Port    <%= node['pound']['ssl']['port'] %>
    Cert    "<%= node['pound']['ssl']['cert'] %>"
End
<% end -%>

Service
    BackEnd
        Address 127.0.0.1
        Port    8000
    End

    BackEnd
        Address 127.0.0.1
        Port    8001
    End
End
```

Finally we need to render the config file in the recipe by adding the following resource:

```
template '/etc/pound.cfg' do
  source 'pound.cfg.erb'
end
```

Now let's converge the node and run the tests again. If you look carefully at the output you should see:

```
[2013-06-15T06:18:54+00:00] INFO: template[/etc/pound.cfg] backed up to /var/
chef/backup/etc/pound.cfg.chef-20130615061854
        [2013-06-15T06:18:54+00:00] INFO: template[/etc/pound.cfg] updated con-
tent

           - update template[/etc/pound.cfg] from bc2726 to d81205
        --- /etc/pound.cfg      2013-06-15 06:17:31.154571676 +0000
           +++ /tmp/chef-rendered-template20130615-3712-1nfupzj      2013-06-15
06:18:54.694571487 +0000
        @@ -16,11 +16,6 @@
             Port 80
         End

        -ListenHTTPS
        -    Address 0.0.0.0
```

```
-    Port    443
-    Cert    "/etc/pki/tls/certs/pound.pem"
-End

 Service
     BackEnd
```

So Chef has removed the HTTPS block. Now the tests should pass:

```
1..6
ok 1 The Pound service is running
ok 2 Two Pound backends are active
ok 3 Pound has an HTTP listener
not ok 4 Pound does not have an HTTPS listener
# /opt/busser/suites/bats/pound.bats:5
ok 5 Server is listening on port 80
ok 6 Server accepts HTTP requests
```

What? What happened? We just saw the file change! Why didn't the test pass? Well, what would you do as a sysadmin, after making a change to the configuration? You'd restart the service! We didn't ask Chef to do that, so it didn't. This allows us to introduce the idea of notifications. We want to restart the service if the config file changes. All resources can send and receive messages using the `notifies` metaparameter. Update the template resource as follows:

```
template '/etc/pound.cfg' do
  source 'pound.cfg.erb'
  notifies :restart, 'service[pound]'
end
```

Converge the node again, and see what happens:

```
* template[/etc/pound.cfg] action create[2013-06-15T06:24:17+00:00] INFO: Pro-
cessing template[/etc/pound.cfg] action create (pound::default line 17)
        (up to date)
        [2013-06-15T06:24:17+00:00] INFO: Chef Run complete in 5.792886113 sec-
onds
       [2013-06-15T06:24:17+00:00] INFO: Running report handlers
       [2013-06-15T06:24:17+00:00] INFO: Report handlers complete
       Chef Client finished, 0 resources updated
       Finished converging <default-centos-64> (0m11.22s).
```

Curiouser and curiouser. Why didn't the service get restarted? This is a common gotcha in Chef and requires careful attention. The config file didn't change so we didn't trigger a restart. Conceivably our system could now be in a broken state and not recoverable without either manually logging onto the machine and removing the file, so Chef can replace it, or by making a change in the template. The lesson to learn is to make sure you pay careful attention to your resources and messages.

I logged on with `kitchen login` and deleted the file, before finally converging the node again. This time we see the following message:

```
[2013-06-15T06:29:56+00:00] INFO: template[/etc/pound.cfg] sending restart ac-
tion to service[pound] (delayed)
          * service[pound] action restart[2013-06-15T06:29:56+00:00] INFO: Pro-
cessing service[pound] action restart (pound::default line 13)
       [2013-06-15T06:29:57+00:00] INFO: service[pound] restarted

           - restart service service[pound]
```

And now all our tests pass!

```
1..6
      ok 1 The Pound service is running
      ok 2 Two Pound backends are active
      ok 3 Pound has an HTTP listener
      ok 4 Pound does not have an HTTPS listener
      ok 5 Server is listening on port 80
      ok 6 Server accepts HTTP requests
```

A brief discussion on services and templates is needed at this stage. This basic pattern —install a package, render a dynamic config file using a template, and manage a service —is what I call the holy trinity of configuration management. About 80% of the configuration management you'll need to do will be a variation on this theme.

Installing the package is the obvious part of the trinity—we want to provide some kind of functionality and that requires us to install some software. Software is frequently distributed in packages, and Chef knows how to install them. Nothing much to say here.

Templates represent the most obviously flexible way to manage files on a node. Because we can pass in data either from an external place or insert values from the node attributes, it's the perfect way to separate configuration from data. When combined with Chef's ability to search for data, it opens up effectively limitless opportunity for dynamic configuration. The most obvious example would be the case where the backends for a load balancer could be determined in real time by searching for all machines with an application server recipe or role, and returning the IP address.

When a template changes, we want to be able to restart the service it configures. In order to do that we need to explicitly declare the service, but then having declared it, we can send it a message in the event of a change to the template. This is what's going on in the preceding template resource:

```
notifies :restart, 'service[pound]'
```

We do that using the `notifies` metaparameter. It's called a "metaparameter" because all services can send (and receive) notifications. The syntax is as follows:

```
resource "name" do
  ...
  notifies :restart, "resource[something]"
end
```

In our case we used:

```
template '/etc/pound.cfg' do
  source 'pound.cfg.erb'
  notifies :restart, 'service[pound]'
end
```

The ordering sounds a bit funny—you don't notify a restart. I find it helps to think of the resource doing the notifying as a rather keen but desperately unreliable child to whom you have entrusted a message:

```
"Wilfrid, please will you tell Atty to feed her Guinea Pigs?"
"OK! .... <scurries off>"
"Wait a second Wilfrid... tell me the message..."
"Feed the Guinea Pigs!"
"And who are you going to tell?"
"Atty."
"Jolly good."
```

Similarly, we say, "What's the message? And what resource is getting the message?"

Before we move on, I want to demonstrate the same procedure using Serverspec instead of Bats. First, destroy your instance using `kitchen destroy`, and then comment out the default recipe so no action is taken.

Now, rename the Bats file to *pound.bats-disabled*. We do this because the tests run in alphabetical order, and will stop as soon as a failure is reached. This means we'd never see our Serverspec tests!

Create a directory for the Serverspec tests, and add the following file:

```
$ cat test/integration/default/serverspec/spec_helper.rb
require 'serverspec'
require 'pathname'
include Serverspec::Helper::Exec
include Serverspec::Helper::DetectOS

RSpec.configure do |c|
  c.before :all do
    c.os = backend(Serverspec::Commands::Base).check_os
  end
end
```

This file is needed to ensure the helpers and operating system detection is in place. Now create a subdirectory inside Serverspec, called *localhost*, and add the following test:

```
$ cat test/integration/default/serverspec/localhost/pound_spec.rb
require 'spec_helper'

describe 'Pound Loadbalancer' do

  it 'should be listening on port 80' do
    expect(port 80).to be_listening
  end
```

```
  it 'should be running the pound service' do
    expect(service 'pound').to be_running
  end

  it 'should have two active backends' do
           expect(command   'poundctl   -c   /var/lib/pound/pound.cfg').to   re-
turn_stdout /.*Backend.*800[01].*active/
  end

  it 'should have an HTTP listener' do
           expect(command   'poundctl   -c   /var/lib/pound/pound.cfg').to   re-
turn_stdout /.*http Listener.*/
  end

  it 'should not have an HTTPS listener' do
         expect(command   'poundctl   -c   /var/lib/pound/pound.cfg').not_to   re-
turn_stdout /.*HTTPS Listener.*/
  end

  it 'should accept HTTP connections on port 80' do
    expect(command "echo 'GET / HTTP/1.1' | nc localhost 80").to return_stdout /
Content-Length:.*/
  end
end
```

We're testing more or less the same thing, but with a different testing framework. This time we can run `kitchen verify` in one go, which will create the machine, install Chef, run Chef, and run the tests:

```
-----> Running serverspec test suite
       /opt/chef/embedded/bin/ruby -I/opt/busser/suites/serverspec -S /opt/chef/
embedded/bin/rspec /opt/busser/suites/serverspec/localhost/pound_spec.rb
FFFF.       F

       Failures:

         1) Pound Loadbalancer should be listening on port 80
            Failure/Error: expect(port 80).to be_listening
       netstat -tunl | grep -- :80\
            # /opt/busser/suites/serverspec/localhost/pound_spec.rb:6:in `block
(2 levels) in <top (required)>'

         2) Pound Loadbalancer should be running the pound service
            Failure/Error: expect(service 'pound').to be_running
       service pound status
       pound: unrecognized service
                 # /opt/busser/suites/serverspec/localhost/pound_spec.rb:10:in
`block (2 levels) in <top (required)>'

         3) Pound Loadbalancer should have two active backends
                  Failure/Error: expect(command 'poundctl -c /var/lib/pound/
pound.cfg').to return_stdout /.*Backend.*800[01].*active/
```

```
       poundctl -c /var/lib/pound/pound.cfg
       sh: poundctl: command not found
                 # /opt/busser/suites/serverspec/localhost/pound_spec.rb:14:in
`block (2 levels) in <top (required)>'

         4) Pound Loadbalancer should have an HTTP listener
                  Failure/Error: expect(command 'poundctl -c /var/lib/pound/
pound.cfg').to return_stdout /.*http Listener.*/
       poundctl -c /var/lib/pound/pound.cfg
       sh: poundctl: command not found
                 # /opt/busser/suites/serverspec/localhost/pound_spec.rb:18:in
`block (2 levels) in <top (required)>'

         5) Pound Loadbalancer should accept HTTP connections on port 80
            Failure/Error: expect(command "echo 'GET / HTTP/1.1' | nc localhost
80").to return_stdout /Content-Length:.*/
       echo 'GET / HTTP/1.1' | nc localhost 80
                 # /opt/busser/suites/serverspec/localhost/pound_spec.rb:26:in
`block (2 levels) in <top (required)>'

       Finished in 0.05315 seconds
       6 examples, 5 failures
```

The output of the failures is much more verbose, but very much as expected. Now uncomment the recipe, converge the node, and run the tests again:

```
-----> Running serverspec test suite
       /opt/chef/embedded/bin/ruby -I/opt/busser/suites/serverspec -S /opt/chef/
embedded/bin/rspec /opt/busser/suites/serverspec/localhost/pound_spec.rb
.....       .

       Finished in 0.11034 seconds
       6 examples, 0 failures
       Finished verifying <default-centos-64> (0m12.07s).
```

We already covered the basics of RSpec in Chapter 5. All Serverspec adds is a set of matchers that check the state of various common resources across a range of operating systems. The resources are documented at Serverspec's website, although the example code given uses the deprecated expectation syntax. My examples use the recommended and current approach, and I recommend you follow this format.

Integration Testing: Minitest Handler

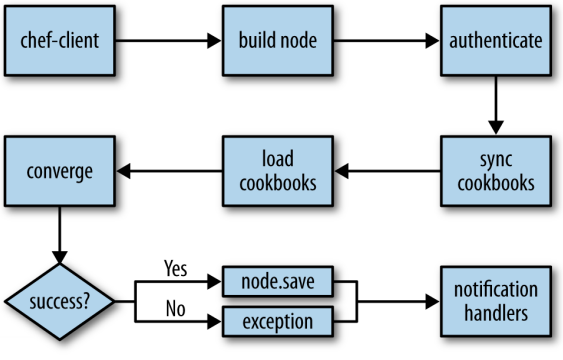
One of the early approaches to integration testing with Chef is Minitest Handler. Considered by some to be no longer as relevant, given the advent of the latest breed of tools, it is nevertheless a popular and useful tool.

Overview

We can view unit tests as being simple, discrete, isolated tests, exercising one piece of functionality, and integration tests as tests that exercise examples of those units of functionality talking to one another. We described this in Chef terms, as *signal in* and *signal out*.

Minitest Handler allows Minitest suites to be run after recipes have been applied to a node to verify the status of the system. In this respect, it's a good approach to testing signal out.

Unlike most of the other tools we discuss in this chapter, the process of writing and running tests via Minitest Handler is managed through a combination of a cookbook and a Chef run itself.



When Chef runs, and configures a node, the last stage of the process is to run so-called report and exception handlers. In simple terms, these provide an interface through which we can collect and display information about the result of a Chef run. The report handler displays information about what happened; the exception handler displays information about what went wrong. The design of the system is such that anyone can write a custom handler that takes data from the Chef run and formats it, sends it, processes it, or displays it in whichever way suits the user.

While these are frequently used to provide notification, for example via IRC or Campfire, they can, of course, be used to do anything at all. Minitest Handler uses this feature to run Minitest suites at the end of the Chef run. This is achieved by adding an entry to the run list to ensure the tests run.

The Minitest Handler cookbook sets up everything needed to use it for the running of tests. The naming is slightly confusing, so I'll clarify quickly:

- *Minitest Handler* is a cookbook that sets up your system to enable you to write Minitest examples to verify the state of your system after your Chef node has converged.
- *Minitest Chef Handler* is a Rubygem that provides the handler itself, and library code for assertions, matchers, and helpers to make the writing and running of these tests possible.

The cookbook carries out the following tasks:

- Installs the latest Minitest gem
- Installs the Chef Minitest gem
- Places test files from cookbooks on the target node as part of a Chef run

For each recipe we wish to test, we must create a corresponding test under the *files/default/tests/minitest* directory. The naming of the test is significant and follows the name of the recipe. So, if we had a recipe called *server.rb*, the test file would be located at *files/default/tests/minitest/server_test.rb*.

These tests are just the same as the Minitest spec examples we wrote when we were developing the Hipster assessor. The only difference is that instead of testing an instance of a Ruby class we wrote, we're testing the results of a Chef run.

Running the tests is simply a matter of running Chef and looking at the output printed to the screen. In this respect, Minitest Handler is one of the simplest tools to start using.

Getting Started

Berkshelf has a command-line option that will add Minitest Handler support to a cookbook it creates. This is the best way to get started. Let's create a cookbook to install GNU Screen—a common screen multiplexer:

```
$ berks cookbook --chef-minitest screen
      create  screen/files/default
      create  screen/templates/default
      create  screen/attributes
      create  screen/definitions
      create  screen/libraries
      create  screen/providers
      create  screen/recipes
      create  screen/resources
      create  screen/recipes/default.rb
      create  screen/metadata.rb
      create  screen/LICENSE
      create  screen/README.md
      create  screen/Berksfile
      create  screen/Thorfile
      create  screen/chefignore
```

```
      create  screen/.gitignore
         run  git init from "./screen"
      create  screen/files/default/tests/minitest/support
      create  screen/files/default/tests/minitest/default_test.rb
      create  screen/files/default/tests/minitest/support/helpers.rb
      create  screen/Gemfile
      create  .kitchen.yml
      append  Thorfile
      create  test/integration/default
      append  .gitignore
      append  .gitignore
      append  Gemfile
      append  Gemfile
You must run `bundle install' to fetch any new gems.
      create  screen/Vagrantfile
```

The key sections here are as follows:

```
 create  screen/files/default/tests/minitest/support
 create  screen/files/default/tests/minitest/default_test.rb
 create  screen/files/default/tests/minitest/support/helpers.rb
```

This sets up an example test and a helper file, which includes the Chef-Minitest code, and provides a convenient place for us to put any of our own functions to support our tests. The helper file created for us by Berkshelf looks like this:

```
module Helpers
  module Screen
    include MiniTest::Chef::Assertions
    include MiniTest::Chef::Context
    include MiniTest::Chef::Resources
  end
end
```

And the example test looks like this:

```
$ cat default_test.rb
require File.expand_path('../support/helpers', __FILE__)

describe 'screen::default' do

  include Helpers::Screen

  # Example spec tests can be found at http://git.io/Fahwsw
  it 'runs no tests by default' do
  end

end
```

This introduces the important ideas of Modules and Mixins, which we mentioned earlier. Modules are a particularly excellent feature of Ruby. They serve two purposes; they implement namespaces, so that as program complexity and size grows, we don't get into a situation where multiple methods with the same name, but serving very different

purposes, clash with each other. Instead, we use modules to make it clear which we mean:

```
module Trig

  def sin(degrees)
  end

  def tan(degrees)
  end

  def cos(degrees)
  end

end

module Catholic

  def sin(naughty_thing)
  end

  def confess(naughty_thing)
  end

  def pray(saint)
  end
end
```

If there was a situation in which the programmer wanted to use both Trig and Catholic modules, all that would be required would be to specify the dependency on the module, and then use the namespace:

```
require 'trig'
require 'catholic'

dice_roll = rand(5)+1

def do_maths_homework

  def find_opposite(theta, hypotenuse)
    Trig::sin(theta) * hypotenuse
  end

  def submit_homework
  ...
  end

  result = find_opposite(30, 100)

  if dice_roll > 3

    Catholic::sin("Lie about using a computer")
```

```
      submit_homework
      Catholic::confess("I claimed I didn't use a computer, but I did!")
    else
      submit_homework
    end

  end
```

The second, and more immediately relevant benefit of modules, is the concept of "mixing in." We've already seen in my silly example earlier that modules can have methods. If you `include` a module in a class, all the methods from that class automatically become available to the class. This is known as the *mixin* facility. We see this facility in use throughout Ruby's core.

```
>= ri Array
= Array <= Object


------------------------------------------------------------------------------
= Includes:
Enumerable (from ruby core)

(from ruby core)
------------------------------------------------------------------------------
Arrays are ordered, integer-indexed collections of any object. Array indexing
starts at 0, as in C or Java.  A negative index is assumed to be relative to
the end of the array---that is, an index of -1 indicates the last element of
the array, -2 is the next to last element in the array, and so on.
------------------------------------------------------------------------------
```

The Array class mixes in the Enumerable module:

```
>= ri Enumerable
= Enumerable

(from ruby core)
------------------------------------------------------------------------------
The Enumerable mixin provides collection classes with several traversal and
searching methods, and with the ability to sort. The class must provide a
method each, which yields successive members of the collection. If
Enumerable#max, #min, or #sort is used, the objects in the collection must
also implement a meaningful <=> operator, as these methods rely on an ordering
between members of the collection.
------------------------------------------------------------------------------
```

Here we see the Enumerable mixin and our class can interact. As long as we define an "each" method, and include Enumerable, we'll get a whole bunch of extra stuff.

Normally to include a mixin, we explicitly call `include mymixin`, and that's exactly what we see in the example test. By including `Helpers::Screen`, we get access to the functionality within that module.

The last relevant step that the Berkshelf generator took was to add a line to our Berksfile, ensuring that we have access to the cookbook and its content:

```
$ cat Berksfile
site :opscode
group :integration do
  cookbook 'minitest-handler'
end

metadata
```

Berkshelf has provided everything we need: the cookbook itself, an example test, and helper code. To run the tests, all we need to do is run `vagrant up`:

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[Berkshelf] This version of the Berkshelf plugin has not been fully tested on
this version of Vagrant.
[Berkshelf] You should check for a newer version of vagrant-berkshelf.
[Berkshelf] If you encounter any errors with this version, please report them
at https://github.com/RiotGames/vagrant-berkshelf/issues
[Berkshelf] You can also join the discussion in #berkshelf on Freenode.
[Berkshelf] Updating Vagrant's berkshelf: '/home/tdi/.berkshelf/vagrant/
berkshelf-20130618-27272-1cv4qos'
[Berkshelf] Using minitest-handler (0.2.1)
[Berkshelf] Using screen (0.1.0) at path: '/home/tdi/screen'
[Berkshelf] Using chef_handler (1.1.4)
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Setting hostname...
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
[default] -- /tmp/vagrant-chef-1/chef-solo-1/cookbooks
[default] Running provisioner: chef_solo...
Generating chef JSON and uploading...
Running chef-solo...
[2013-06-18T06:06:02+00:00] INFO: *** Chef 10.14.2 ***
[2013-06-18T06:06:03+00:00] INFO: Setting the run_list to ["recipe[minitest-
handler::default]", "recipe[screen::default]"] from JSON
[2013-06-18T06:06:03+00:00] INFO: Run List is [recipe[minitest-
handler::default], recipe[screen::default]]
[2013-06-18T06:06:03+00:00] INFO: Run List expands to [minitest-
handler::default, screen::default]
[2013-06-18T06:06:03+00:00] INFO: Starting Chef Run for screen-berkshelf
```

```
[2013-06-18T06:06:03+00:00] INFO: Running start handlers
[2013-06-18T06:06:03+00:00] INFO: Start handlers complete.
[2013-06-18T06:06:03+00:00] INFO: Processing chef_gem[minitest] action nothing
(minitest-handler::default line 2)
[2013-06-18T06:06:03+00:00] INFO: Processing chef_gem[minitest] action install
(minitest-handler::default line 2)
[2013-06-18T06:06:03+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion nothing (minitest-handler::default line 9)
[2013-06-18T06:06:03+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion install (minitest-handler::default line 9)
[2013-06-18T06:06:34+00:00] INFO: Processing chef_gem[minitest] action nothing
(minitest-handler::default line 2)
[2013-06-18T06:06:34+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion nothing (minitest-handler::default line 9)
[2013-06-18T06:06:34+00:00] INFO: Processing directory[minitest test location]
action delete (minitest-handler::default line 18)
[2013-06-18T06:06:34+00:00] INFO: Processing directory[minitest test location]
action create (minitest-handler::default line 18)
[2013-06-18T06:06:34+00:00] INFO: directory[minitest test location] created di-
rectory /var/chef/minitest
[2013-06-18T06:06:34+00:00] INFO: directory[minitest test location] owner
changed to 0
[2013-06-18T06:06:34+00:00] INFO: directory[minitest test location] group
changed to 0
[2013-06-18T06:06:34+00:00] INFO: directory[minitest test location] mode
changed to 775
[2013-06-18T06:06:34+00:00] INFO: Processing ruby_block[load tests] action cre-
ate (minitest-handler::default line 29)
[2013-06-18T06:06:34+00:00] INFO: Processing directory[/var/chef/minitest/
minitest-handler] action create (dynamically defined)
[2013-06-18T06:06:34+00:00] INFO: directory[/var/chef/minitest/minitest-
handler] created directory /var/chef/minitest/minitest-handler
[2013-06-18T06:06:34+00:00] INFO: Processing directory[/var/chef/minitest/
screen] action create (dynamically defined)
[2013-06-18T06:06:34+00:00] INFO: directory[/var/chef/minitest/screen] created
directory /var/chef/minitest/screen
[2013-06-18T06:06:34+00:00] INFO: Enabling minitest-chef-handler as a report
handler
[2013-06-18T06:06:34+00:00] INFO: ruby_block[load tests] called
[2013-06-18T06:06:34+00:00] INFO: Chef Run complete in 31.002947638 seconds
[2013-06-18T06:06:34+00:00] INFO: Running report handlers
Run options: -v --seed 30025

# Running tests:

screen::default#test_0001_runs no tests by default =
0.00 s =
.



Finished tests in 0.001883s, 531.0494 tests/s, 0.0000 assertions/s.
```

```
1 tests, 0 assertions, 0 failures, 0 errors, 0 skips

[2013-06-18T06:06:34+00:00] INFO: Report handlers complete
```

Naturally, this used the default Vagrantfile created by Berkshelf, which might not use the Vagrant box you want, and at the time of this writing, installs Chef 10 rather than Chef 11. But this is mere detail—we already know how to swap out Vagrant boxes.

Berkshelf made the Minitest Handler cookbook available, and the existence of the tests under the *files/default/tests/minitest* location meant that the tests were picked up and run, with the test results visible at the conclusion of the Chef run.

Example

Let's write a couple of trivial tests for our screen cookbook before looking at some more involved examples.

I think the two obvious things we'd want to test when installing Screen would be that the package was installed and that a standard, customized screen config was made available to users. We can make assertions about this as follows. Edit the *files/default/tests/minitest/default.rb* file:

```
require File.expand_path('../support/helpers', __FILE__)

describe 'screen::default' do

  include Helpers::Screen

   it "installs Screen" do
    package("screen").must_be_installed
  end

  it "provides a global, customized default configuration" do
    file("/usr/local/etc/screenrc").must_exist
     file('/usr/local/etc/screenrc').must_match /^caption string .*%\?%F%{= Bk}%
\?.*$/
    file('/usr/local/etc/screenrc').must_match /^hardstatus string '%{= kG}.*$/
  end

end
```

We can run these tests with `vagrant provision`:

```
[2013-06-18T08:32:52+00:00] INFO: Running report handlers
Run options: -v --seed 16917

# Running tests:

screen::default#test_0001_installs Screen =
```

```
5.73 s = F
screen::default#test_0002_provide a global, customized default configuration =
0.00 s = F


Finished tests in 5.735119s, 0.3487 tests/s, 0.3487 assertions/s.

  1) Failure:
screen::default#test_0001_installs       Screen       [/var/chef/minitest/screen/
default_test.rb:8]:
Expected package 'screen' to be installed


  2) Failure:
screen::default#test_0002_provide a global, customized default configuration
[/var/chef/minitest/screen/default_test.rb:12]:
Expected path '/usr/local/etc/screenrc' to exist


2 tests, 2 assertions, 2 failures, 0 errors, 0 skips

[2013-06-18T08:32:58+00:00] INFO: Report handlers complete
[2013-06-18T08:32:58+00:00] ERROR: Running exception handlers
[2013-06-18T08:32:58+00:00] ERROR: Exception handlers complete
[2013-06-18T08:32:58+00:00] FATAL: Stacktrace dumped to /tmp/vagrant-chef-1/
chef-stacktrace.out
[2013-06-18T08:32:58+00:00] FATAL: RuntimeError: MiniTest failed with 2 fail-
ure(s) and 0 error(s).
Failure:
screen::default#test_0001_installs       Screen       [/var/chef/minitest/screen/
default_test.rb:8]:
Expected package 'screen' to be installed

Failure:
screen::default#test_0002_provide a global, customized default configuration
[/var/chef/minitest/screen/default_test.rb:12]:
Expected path '/usr/local/etc/screenrc' to exist
```

Now let's write the code to make the test pass:

```
$ cat recipes/default.rb
package "screen"

cookbook_file "/etc/screenrc" do
  source "screenrc"
end

$ cat files/default/screenrc
caption string "%?%F%{= Bk}%? %C%A %D %d-%m-%Y %{= kB} %t%= %?%F%{= Bk}%:%{= wk}
%? %n "
hardstatus alwayslastline
hardstatus string '%{= kG}[ %{G}%H %{g}][%= %{= kw}%?%-Lw%?%{r}(%{W}%n*%f%t%?
(%u)%?%{r})%{w}%?%+Lw%?%?%= %{g}][%{B} %d/%m %{W}%c %{g}]'
```

```
defscrollback 30000
escape ^Zz
```

Now if we run `vagrant provision`, Chef should apply our recipe and then run the tests, and they should pass:

```
[default] Running provisioner: chef_solo...
Generating chef JSON and uploading...
Running chef-solo...
[2013-06-18T08:55:34+00:00] INFO: *** Chef 10.14.2 ***
[2013-06-18T08:55:34+00:00] INFO: Setting the run_list to ["recipe[minitest-
handler::default]", "recipe[screen::default]"] from JSON
[2013-06-18T08:55:34+00:00] INFO: Run List is [recipe[minitest-
handler::default], recipe[screen::default]]
[2013-06-18T08:55:34+00:00] INFO: Run List expands to [minitest-
handler::default, screen::default]
[2013-06-18T08:55:34+00:00] INFO: Starting Chef Run for screen-berkshelf
[2013-06-18T08:55:34+00:00] INFO: Running start handlers
[2013-06-18T08:55:34+00:00] INFO: Start handlers complete.
[2013-06-18T08:55:34+00:00] INFO: Processing chef_gem[minitest] action nothing
(minitest-handler::default line 2)
[2013-06-18T08:55:34+00:00] INFO: Processing chef_gem[minitest] action install
(minitest-handler::default line 2)
[2013-06-18T08:55:34+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion nothing (minitest-handler::default line 9)
[2013-06-18T08:55:34+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion install (minitest-handler::default line 9)
[2013-06-18T08:56:02+00:00] INFO: Processing chef_gem[minitest] action nothing
(minitest-handler::default line 2)
[2013-06-18T08:56:02+00:00] INFO: Processing chef_gem[minitest-chef-handler] ac-
tion nothing (minitest-handler::default line 9)
[2013-06-18T08:56:02+00:00] INFO: Processing directory[minitest test location]
action delete (minitest-handler::default line 18)
[2013-06-18T08:56:02+00:00] INFO: Processing directory[minitest test location]
action create (minitest-handler::default line 18)
[2013-06-18T08:56:02+00:00] INFO: directory[minitest test location] created di-
rectory /var/chef/minitest
[2013-06-18T08:56:02+00:00] INFO: directory[minitest test location] owner
changed to 0
[2013-06-18T08:56:02+00:00] INFO: directory[minitest test location] group
changed to 0
[2013-06-18T08:56:02+00:00] INFO: directory[minitest test location] mode
changed to 775
[2013-06-18T08:56:02+00:00] INFO: Processing ruby_block[load tests] action cre-
ate (minitest-handler::default line 29)
[2013-06-18T08:56:02+00:00] INFO: Processing directory[/var/chef/minitest/
minitest-handler] action create (dynamically defined)
[2013-06-18T08:56:02+00:00] INFO: directory[/var/chef/minitest/minitest-
handler] created directory /var/chef/minitest/minitest-handler
[2013-06-18T08:56:02+00:00] INFO: Processing directory[/var/chef/minitest/
screen] action create (dynamically defined)
[2013-06-18T08:56:02+00:00] INFO: directory[/var/chef/minitest/screen] created
directory /var/chef/minitest/screen
```

```
[2013-06-18T08:56:02+00:00] INFO: Enabling minitest-chef-handler as a report
handler
[2013-06-18T08:56:02+00:00] INFO: ruby_block[load tests] called
[2013-06-18T08:56:02+00:00] INFO: Processing package[screen] action install
(screen::default line 10)
[2013-06-18T08:56:09+00:00] INFO: package[screen] installing
screen-4.0.3-16.el6 from base repository
[2013-06-18T08:56:13+00:00] INFO: Processing cookbook_file[/usr/local/etc/
screenrc] action create (screen::default line 12)
[2013-06-18T08:56:13+00:00] INFO: cookbook_file[/usr/local/etc/screenrc] cre-
ated file /usr/local/etc/screenrc
[2013-06-18T08:56:13+00:00] INFO: Chef Run complete in 38.732347486 seconds
[2013-06-18T08:56:13+00:00] INFO: Running report handlers
Run options: -v --seed 23177

# Running tests:

screen::default#test_0001_installs Screen =
0.16 s = .
screen::default#test_0002_provides a global, customized default configuration =
0.00 s = .


Finished tests in 0.168615s, 11.8613 tests/s, 23.7227 assertions/s.

2 tests, 4 assertions, 0 failures, 0 errors, 0 skips
[2013-06-18T08:56:13+00:00] INFO: Report handlers complete
```

Although very simple, this should give a good sense of how easy it is to use the Minitest Handler process to carry out integration tests with nothing more than Vagrant and Berkshelf.

Moving on to a more complex example, consider the following tests from the Opscode apache cookbook:

```
it 'installs apache' do
  package(node['apache']['package']).must_be_installed
end
it 'starts apache' do
  apache_service.must_be_running
end
it 'enables apache' do
  apache_service.must_be_enabled
end
it 'creates the conf.d directory' do
  directory("#{node['apache']['dir']}/conf.d").must_exist.with(:mode, "755")
end
it 'creates the logs directory' do
  directory(node['apache']['log_dir']).must_exist
end
it 'enables the default site' do
  file("#{node['apache']['dir']}/sites-enabled/000-default").must_exist
```

```
  file("#{node['apache']['dir']}/sites-available/default").must_exist
end
it 'ensures the debian-style apache module scripts are present' do
  %w{a2ensite a2dissite a2enmod a2dismod}.each do |mod_script|
    file("/usr/sbin/#{mod_script}").must_exist
  end
end
it 'reports server name only, not detailed version info' do
    assert_match(/^ServerTokens  Prod  *$/,  File.read("#{node['apache']['dir']}/
conf.d/security"))
end
```

These tests demonstrate one very important feature of Minitest Handler—the tests are all executed in the context of a Chef run. This has profound implications for testing. At any point we have access to three important objects from Chef: the `run_status`, the node itself, and the `run_context`. This is potentially very useful to us; in these examples, we're using node attributes in our test. However, it's also important to understand that the tests we're carrying out are often based on knowledge Chef has rather than external validation of desired state. Now, of course, we implicitly trust Chef, but it's worth stating explicitly that, in certain cases, what these tests are doing is inspecting Chef's knowledge rather than carrying out probes on a configured server.

The final example I'll cover is one where we use a helper method:

```
it 'listens on port 80' do
  apache_configured_ports.must_include(80)
end

it 'only listens on port 443 when SSL is enabled' do
  unless ran_recipe?('apache2::mod_ssl')
    apache_configured_ports.wont_include(443)
  end
end
```

Here we have an example of helper code. This could go in the helper module we already discussed:

```
def apache_configured_ports
   port_config = File.read("#{node['apache']['dir']}/ports.conf")
   port_config.scan(/^Listen ([0-9]+)/).flatten.map{|p| p.to_i}
 end

def ran_recipe?(recipe)
  node.run_state[:seen_recipes].keys.include?(recipe)
end
```

Herein we see examples of the kind of heavy lifting that is necessary to make the writing of tests more accessible to infrastructure developers. A line must carefully be walked between providing reusable helper methods that make the writing of tests fast and easy, and creating chunks of code that encourage lazy and brittle test writing. The right balance will emerge as the discipline and community matures, but for now, the

infrastructure developer is well-served by matchers and expectations built into minitest-chef-handler, and creative programming will furnish helper methods that over time may emerge as reusable patterns.

Minitest Handler with Test Kitchen

Before looking at the advantages and disadvantages and drawing a conclusion, I want to demonstrate how to run Minitest Handler tests using Test Kitchen.

Here's an example *.kitchen.yml* file:

```
$ cat .kitchen.yml
---
driver_plugin: vagrant
driver_config:
  require_chef_omnibus: true

platforms:
- name: ubuntu-10.04
  driver_config:
    box: opscode-ubuntu-10.04
                      box_url:     https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_ubuntu-10.04_provisionerless.box
- name: centos-5.9
  driver_config:
    box: opscode-centos-5.9
                      box_url:     https://opscode-vm.s3.amazonaws.com/vagrant/
opscode_centos-5.9_provisionerless.box

suites:
- name: default
  run_list: ["recipe[minitest-handler]", "recipe[screen]"]
  attributes: {}
```

All we need to do is ensure that the minitest-handler recipe is included on the run list for whichever suite we care about. As long as we have minitest-handler in the Berksfile (or in the cookbook metadata), the cookbook will be made available and applied, and the tests will be run on a `kitchen converge` action:

```
[2013-06-18T09:33:35+00:00] INFO: Chef Run complete in 26.619225 seconds
       [2013-06-18T09:33:35+00:00] INFO: Running report handlers
Run options: -v --seed 2739

# Running tests:

screen::default#test_0001_installs Screen = 0.00 s = .
screen::default#test_0002_provides a global, customized default configuration =
0.00 s = .


Finished tests in 0.003959s, 505.1781 tests/s, 1010.3562 assertions/s.
```

```
2 tests, 4 assertions, 0 failures, 0 errors, 0 skips
       [2013-06-18T09:33:35+00:00] INFO: Report handlers complete
Chef Client finished, 5 resources updated
       Finished converging <default-centos-59> (0m30.77s).
-----> Kitchen is finished. (1m0.12s)
```

Advantages and Disadvantages

The immediate advantage of Minitest Handler is that the barrier to entry is very low. If you're using Berkshelf, and you should be, the generator will create all you need to start writing and running tests. There's good coverage in terms of assertions and matchers, and the feedback cycle is quick.

I have two main concerns with this approach, though. First, I'm not entirely comfortable with expecting new users to learn and use two different expectation syntaxes. On the assumption that unit testing will be done using Chefspec, it's rather irksome that the integration tests use a different approach. The second area where I feel a certain skepticism is in the reliance upon, or use of, Chef's internal knowledge. I feel that we're not really doing true integration testing here. In places, we're relying on magical knowledge from within the framework that had responsibility for bringing our infrastructure in line with policy. For these reasons, I feel much more comfortable recommending an integration framework that is entirely ignorant of Chef, as this provides the opportunity to standardize on RSpec expectation syntax and to run tests that have absolutely no knowledge of, or dependence upon, the configuration management framework.

A final potential gotcha should be noted. The most recent release of the Minitest Handler cookbook altered the mechanism that makes the test files available to the host upon which the tests are being run. This means that tests will not be run on machines using a client/server model rather than `chef-solo`. At the time of this writing, there is work ongoing to resolve this issue, but for now this should be noted as a consideration for this testing approach.

Summary and Conclusion

Minitest Handler is easy to use, capable, and fast. It needs minimal setup and offers immediate value. However, I feel that the central tool in the infrastructure developer's kit is going to be Test Kitchen, and having invested in making the Test Kitchen framework available, I see little use for Minitest Handler, and prefer to use tests run by the kitchen Busser.

Unit Testing: Chefspec

The purest, fastest, and most lightweight unit-testing approach belongs to Chefspec—a popular and powerful tool enabling the infrastructure developer to create RSpec examples for cookbook code.

Overview

Well-written unit tests have the following characteristics:

- Exercise every aspect of the code under test
- Run in isolation, shielded from external forces, with any external functions (including the operating system) mocked out, to give complete control over the environment
- Written in such a way as to be easy for any developer to run
- Run very quickly, giving fast feedback
- Checked into the same version control system as the code they test

Chefspec allows the infrastructure developer to write RSpec examples for cookbooks that meet these characteristics. The Chef run itself is mocked, allowing us to assert that the Chef providers are called with the correct parameters. Any input data, including attribute data from Ohai, roles, cookbooks, or recipes can be set on whatever platform is required, giving comprehensive coverage. Because the node is never actually converged, and because there is never any genuine API traffic, the tests are very fast and give extremely rapid feedback.

It's important to emphasize that, as I argued in the first edition, there is little point in writing tests that verify the Chef resources and providers behave as they should. We trust that behavior implicitly. Chef is tested, and Chef is production quality code, widely deployed across hundreds of thousands of machines all over the world. We don't need to test that when we ask Chef to install Apache that Chef does indeed install Apache. If the Chef run completes without error, and you asked it to install Apache, Apache will be installed. That's the whole point of Chef as a declarative interface to infrastructure resources.

However, what we do need to test is that we asked Chef to do the right thing. Chefspec provides this capability—it allows us to check what is in the resource collection and what actions would be taken. We can compare that against what we expected. This is useful on a couple of levels. First, as we grow our test coverage, so we will catch regressions and foolish errors. Especially when developing for multiple operating systems or distributions, the task of ensuring that no unwanted side effects have been introduced is very valuable. Second, the discipline of writing the tests (especially writing the tests first) helps the infrastructure developer think through the feature being added. By thinking about the intended outcome, and by writing a test to capture that, the features are emerged incrementally, and in accordance with demand.

When writing Chefspec tests it makes sense to think of the cookbooks as a black box. We're interested in how the code handles various inputs. Just as when writing unit tests for traditional software, where we would write tests to verify the behavior of the code

when given different arguments, so we do the same with Chef. In Chef we can provide input to our cookbooks from attributes (whether from Ohai, or cookbooks, roles or environments), search results, and databag look-ups. We could also, of course, provide input from arbitrary helper methods calling external services, or making calculations during the Chef run.

Given that one of the great advantages of the Chef framework is the ease with which we can write data-driven cookbooks, it's very helpful to be able to exercise our code by feeding it data, allowing us to test edge cases and verify our reasoning and understanding about how Chef will behave, but without having to provision a large number of different machines to run Chef a large number of times.

Getting Started

Chefspec is, again, distributed as a Rubygem. Simply add it to the Gemfile, and run `bundle update`.

Once installed, Chefspec provides an extension to the `knife cookbook` command, which will create a basic RSpec boilerplate. Let's create a cookbook that installs the handy network utility *netcat*.

For the purposes of illustration, we'll create this cookbook using `Knife` rather than Berkshelf.

```
$ knife cookbook create netcat -o .
WARNING: No knife configuration file found
** Creating cookbook netcat
** Creating README for cookbook: netcat
** Creating CHANGELOG for cookbook: netcat
** Creating metadata for cookbook: netcat
$ knife cookbook create_specs netcat -o .
WARNING: No knife configuration file found
** Creating specs for cookbook: netcat
```

This creates a spec directory and populates it with an example test:

```
$ cat netcat/spec/default_spec.rb
require 'chefspec'

describe 'netcat::default' do
  let (:chef_run) { ChefSpec::ChefRunner.new.converge 'netcat::default' }
  it 'should do something' do
    pending 'Your recipe examples go here.'
  end
end
```

Again, note the naming convention. We're initially testing the *default* cookbook; create a file named *default_spec.rb*. Running the test is a simple matter of running the `rspec` command in the top-level directory of the cookbook:

```
$ rspec

Pending:
  netcat::default should do something
    # Your recipe examples go here.
    # ./spec/default_spec.rb:5

Finished in 0.00028 seconds
1 example, 0 failures, 1 pending
```

Example

Let's look again at the boilerplate example that we created with `knife cookbook create_specs`:

```
require 'chefspec'

describe 'netcat::default' do
  let (:chef_run) { ChefSpec::ChefRunner.new.converge 'netcat::default' }
  it 'should do something' do
    pending 'Your recipe examples go here.'
  end
end
```

The first line simply pulls in Chefspec, much like our Thor example, when we pulled in library code from elsewhere. Next we do exactly as we did in the Hipster test—set up a describe block. I always like to imagine having a conversation here:

```
Me: "Describe the default recipe in the netcat cookbook."
You: "It installs netcat!"
```

It's helpful to remember when we're writing these tests that we're describing the behavior of the system, in terms of examples that demonstrate the intended functionality of the thing we're building.

The next thing we need to do is create an instance of a *Chef Runner*. A Chef Runner is the object responsible for running Chef in the context of our tests. Incidentally, here's a cutely recursive way to learn what a Chef Runner is:

```
$ cd ~/src/chefspec
$ rspec -fd spec/chefspec/chef_runner_spec.rb
ChefSpec::ChefRunner
  #initialize
    should create a node for use within the examples
    should set the chef cookbook path to a default if not provided
    should set the chef cookbook path to any provided value
     should support the chef cookbook path being passed as a string for back-
wards compatibility
    should default the log_level to warn
    should set the log_level to any provided value
    should alias the real resource actions
    should capture the resources created
```

```
    should execute the real action if resource is in the step_into list
    should accept a block to set node attributes
    should allow evaluate_guards to be falsey
    should allow evaluate_guards to be truthy
...
...
```

I throw that in as an example of how tests can function as documentation, and shed light on our understanding of how software functions.

So we need an instance of a Chef Runner. There are a couple of ways to do this, but the approach adopted here is the most commonly used:

```
let (:chef_run) { ChefSpec::ChefRunner.new.converge 'netcat::default' }
```

This line of code introduces a couple of useful Ruby ideas, so I'll cover them briefly.

The *let* method defines a *memoized* helper method. What does this mean? Well, memoization is a simple pattern, which simply means "cache the result of the method." This is a handy technique for storing values of a function instead of recomputing them each time the function is called.

Suppose we had a method to run, which we know is always going to return the same result. Suppose we also knew that running this method was rather slow, or resource intensive. Wouldn't it make sense to cache the result the first time we ran it? That's the basic idea behind memoization. Here's a dumb example:

```
def album_and_song
  "#{album} - #{song}"
end
```

Although not hugely expensive, this method does require the string to be reconstructed every time. The classic way to memoize in Ruby is to use the conditional assignment operator, `||=`.

```
def album_and_song
  @album_and_song ||= "#{album} - #{song}"
end
```

This means, if `@album_and_song` is not initialized, or if it is set to nil or false, it will be assigned to the value of the expression to the right—the result of the string interpolation creating the album/song combination. However, if it's already set to a truthy value (anything other than nil or false), it will remain unchanged.

This is a handy technique when writing tests that instantiate something we want to use. We can define a method that describes the thing we want to instantiate, use memoization behind the scenes, and henceforth just use the method without ever having to worry about instantiating it.

The `let` method does exactly this—it gives us an instance of something we need to use but with some handy advantages. Specifically, using `let()` over instance variables is

safer because it creates a method rather than a variable, and so if we ever mistype the name we'll get a clear `NameError` rather than `nil`, which you'll quickly learn is hard to track down. `let` is also lazy-evaluated—that is, it is not evaluated until the first time the method it defines is invoked, so the code runs only if the example calls it; by contrast, the obvious alternative, which is to use an instance variable in a `before(:each)`, will run before every example, which is wasteful.

Now we have the Chef Runner available to use, and we set it up to converge the default recipe in our netcat cookbook. All we need to do is use the `chef_run` object to make assertions.

The simplest thing we could assert would be that the Chef Runner will install the netcat package:

```
it 'installs the netcat package' do
  expect(chef_run).to install_package('netcat')
end
```

Let's try running the test:

```
$ rspec
F

Failures:

  1) netcat::default installs the netcat package
     Failure/Error: expect(chef_run).to install_package('netcat')
       No package resource named 'netcat' with action :install found.
     # ./spec/default_spec.rb:6:in `block (2 levels) in <top (required)>'

Finished in 0.05371 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/default_spec.rb:5 # netcat::default installs the netcat package
```

As we expected, we have a failure. We've asserted that when we converge the default recipe, the Chef runner will be asked to install the netcat package. But we haven't written the default recipe yet, so the test fails. Let's fix that:

```
$ cat recipes/default.rb
package 'netcat'
```

Now when we run RSpec, the test passes:

```
$ rspec
.

Finished in 0.01144 seconds
1 example, 0 failures
```

This is great, but remember, we've tested only signal in. We need to test signal out. Also, we might want to support installing netcat on multiple platforms, so it would be sensible to test it on multiple platforms. Let's fire up Test Kitchen again, and see what happens when we converge the recipe for real on both CentOS and Ubuntu:

```
$ kitchen converge
[tdi@tk01 netcat]$ kitchen converge
-----> Starting Kitchen (v1.0.0.dev)
-----> Creating <default-ubuntu-1204>
...
...
-----> Converging <default-ubuntu-1204>
...
Converging 1 resources
Recipe: netcat::default
  * package[netcat] action install[2013-06-18T11:47:02+00:00] INFO: Processing
package[netcat] action install (netcat::default line 1)

    - install version 1.10-39 of package netcat

[2013-06-18T11:47:06+00:00] INFO: Chef Run complete in 4.114389784 seconds
       Finished converging <default-ubuntu-1204> (0m17.76s).
-----> Creating <default-centos-64>
...
...
-----> Converging <default-centos-64>
...
       Converging 1 resources
       Recipe: netcat::default
          * package[netcat] action install[2013-06-18T11:48:41+00:00] INFO: Pro-
cessing package[netcat] action install (netcat::default line 1)

           * No version specified, and no candidate version available for netcat


================================================================================
       Error executing action `install` on resource 'package[netcat]'

================================================================================


       Chef::Exceptions::Package
       -------------------------
       No version specified, and no candidate version available for netcat
...
Chef::Exceptions::Package: No version specified, and no candidate version avail-
able for netcat
>>>>>> Converge failed on instance <default-centos-64>.
>>>>>> Please see .kitchen/logs/default-centos-64.log for more details
>>>>>> ------Exception-------
>>>>>> Class: Kitchen::ActionFailed
>>>>>> Message: SSH exited (1) for command: [sudo -E chef-solo --config /tmp/
```

```
kitchen-chef-solo/solo.rb  --json-attributes  /tmp/kitchen-chef-solo/dna.json  --
log_level info]
>>>>>> ----------------------
```

Here we see the value of having Test Kitchen to hand. We didn't even write any tests, but we were able to see, with a single command, whether our recipe would even converge on both platforms. And we learned it wouldn't. The reason for this is that, although Chef providers know how to take appropriate action on all supported platforms, Chef isn't clever enough to know that Debian calls netcat "netcat," whereas CentOS calls it "nc." We need to put that logic in the recipe.

You'll remember that when we run Chef on a node, one of the first things that happens is Ohai runs, profiling the system, and providing useful information to the recipe DSL, such as the platform version or the family of operating system. Chefspec has the ability to mock this data, using a little library called Fauxhai. Fauxhai is effectively an open source store of Ohai data for multiple platforms, which Chefspec can use in order to pretend to be a machine running on, for example, Solaris 10.

We make use of this capability by providing a platform and version to the constructor when we instantiate a Chef runner. Our current Chef runner looks like this:

```
let (:chef_run) { ChefSpec::ChefRunner.new.converge 'netcat::default' }
```

If we want the runner to look like a CentOS machine, we instead call it like this:

```
let(:chef_run) do
  runner = ChefSpec::ChefRunner.new(
    platform: 'centos',
    version: '6.3'
  )
  runner.converge 'netcat::default'
end
```

However, we want to do this for more than one platform. This is where RSpec *contexts* come in handy.

A context is an important concept in RSpec. In RSpec, we are generally concerned with an *Example Group*. This is a set of tests that describe the behavior of the item under test. The two keywords used to build and test example groups are `describe()` and `it()`. For example:

```
describe "MusicPlayer" do
  it "lists available tracks" do
  end
end
```

Describe blocks can be nested to provide a richer description of behavior. For example:

```
describe "MusicPlayer" do
  describe "when in select music mode" do
    it "lists available tracks" do
    end
```

```
  end
end
```

RSpec provides the `context()` method as an alias for `describe()`. This allows us to word our examples to set the context in which the item under test is used. For example, we could express the previous example as:

```
describe "MusicPlayer" do
  context "when in select music mode" do
    it "lists available tracks" do
    end
  end
end
```

We can use the same pattern in our Chefspec examples, by using a context for each platform:

```
require 'chefspec'

describe 'netcat::default' do
  context 'centos' do
    let(:chef_run) do
      runner = ChefSpec::ChefRunner.new(
                                        platform: 'centos',
                                        version: '6.3'
                                        )
      runner.converge 'netcat::default'
    end
    it  'installs the nc package' do
      expect(chef_run).to install_package('nc')
    end
  end

  context 'ubuntu' do
    let(:chef_run) do
      runner = ChefSpec::ChefRunner.new(
                                        platform: 'ubuntu',
                                        version: '12.04'
                                        )
      runner.converge 'netcat::default'
    end
    it  'installs the netcat package' do
      expect(chef_run).to install_package('netcat')
    end
  end
end
```

Now let's run the test:

```
$ rspec
.

Failures:
```

```
  1) netcat::default centos installs the nc package
     Failure/Error: expect(chef_run).to install_package('nc')
       No package resource named 'nc' with action :install found.
     # ./spec/default_spec.rb:13:in `block (3 levels) in <top (required)>'

Finished in 0.04224 seconds
2 examples, 1 failure

Failed examples:

rspec ./spec/default_spec.rb:12 # netcat::default centos installs the nc package
```

So, as we already know, the recipe doesn't try to install "nc" when the machine is a CentOS machine. We need to fix this in the recipe:

```
package 'nc' do
  package_name case node['platform_family']
               when 'debian'
                 'netcat'
               else
                 'nc'
               end
end
```

Now the test passes:

```
$ rspec
..

Finished in 0.04242 seconds
2 examples, 0 failures
```

And when we converge the node:

```
$ kitchen converge
-----> Starting Kitchen (v1.0.0.dev)
-----> Converging <default-ubuntu-1204>
       Resolving cookbook dependencies with Berkshelf
Using netcat (0.1.0)
       Removing non-cookbook files in sandbox
        Uploaded /tmp/default-ubuntu-1204-sandbox-20130618-30672-y0aiqp/solo.rb
(168 bytes)
           Uploaded /tmp/default-ubuntu-1204-sandbox-20130618-30672-y0aiqp/cook-
books/netcat/recipes/default.rb (180 bytes)
           Uploaded /tmp/default-ubuntu-1204-sandbox-20130618-30672-y0aiqp/cook-
books/netcat/metadata.rb (276 bytes)
           Uploaded /tmp/default-ubuntu-1204-sandbox-20130618-30672-y0aiqp/cook-
books/netcat/README.md (1447 bytes)
       Uploaded /tmp/default-ubuntu-1204-sandbox-20130618-30672-y0aiqp/dna.json
(31 bytes)
Starting Chef Client, version 11.4.4
[2013-06-18T12:42:55+00:00] INFO: *** Chef 11.4.4 ***
[2013-06-18T12:42:55+00:00] INFO: Setting the run_list to ["recipe[netcat]"]
```

```
from JSON
[2013-06-18T12:42:55+00:00] INFO: Run List is [recipe[netcat]]
[2013-06-18T12:42:55+00:00] INFO: Run List expands to [netcat]
[2013-06-18T12:42:55+00:00] INFO: Starting Chef Run for default-ubuntu-1204
[2013-06-18T12:42:55+00:00] INFO: Running start handlers
[2013-06-18T12:42:55+00:00] INFO: Start handlers complete.
Compiling Cookbooks...
Converging 1 resources
Recipe: netcat::default
  * package[nc] action install[2013-06-18T12:42:55+00:00] INFO: Processing pack-
age[nc] action install (netcat::default line 1)
 (up to date)
[2013-06-18T12:42:55+00:00] INFO: Chef Run complete in 0.027679985 seconds
[2013-06-18T12:42:55+00:00] INFO: Running report handlers
[2013-06-18T12:42:55+00:00] INFO: Report handlers complete
Chef Client finished, 0 resources updated
       Finished converging <default-ubuntu-1204> (0m2.29s).
-----> Converging <default-centos-64>
       Resolving cookbook dependencies with Berkshelf
Using netcat (0.1.0)
       Removing non-cookbook files in sandbox
         Uploaded /tmp/default-centos-64-sandbox-20130618-30672-131b8ou/solo.rb
(166 bytes)
        Uploaded /tmp/default-centos-64-sandbox-20130618-30672-131b8ou/cookbooks/
netcat/recipes/default.rb (180 bytes)
        Uploaded /tmp/default-centos-64-sandbox-20130618-30672-131b8ou/cookbooks/
netcat/metadata.rb (276 bytes)
        Uploaded /tmp/default-centos-64-sandbox-20130618-30672-131b8ou/cookbooks/
netcat/README.md (1447 bytes)
         Uploaded /tmp/default-centos-64-sandbox-20130618-30672-131b8ou/dna.json
(31 bytes)
       Starting Chef Client, version 11.4.4
       [2013-06-18T12:43:18+00:00] INFO: *** Chef 11.4.4 ***
        [2013-06-18T12:43:18+00:00] INFO: Setting the run_list to ["recipe[net-
cat]"] from JSON
       [2013-06-18T12:43:18+00:00] INFO: Run List is [recipe[netcat]]
       [2013-06-18T12:43:18+00:00] INFO: Run List expands to [netcat]
       [2013-06-18T12:43:18+00:00] INFO: Starting Chef Run for default-centos-64
       [2013-06-18T12:43:18+00:00] INFO: Running start handlers
       [2013-06-18T12:43:18+00:00] INFO: Start handlers complete.
       Compiling Cookbooks...
       Converging 1 resources
       Recipe: netcat::default
          * package[nc] action install[2013-06-18T12:43:18+00:00] INFO: Process-
ing package[nc] action install (netcat::default line 1)
        [2013-06-18T12:43:20+00:00] INFO: package[nc] installing nc-1.84-22.el6
from base repository

           - install version 1.84-22.el6 of package nc

        [2013-06-18T12:43:22+00:00] INFO: Chef Run complete in 4.099788551 sec-
onds
```

```
       [2013-06-18T12:43:22+00:00] INFO: Running report handlers
       [2013-06-18T12:43:22+00:00] INFO: Report handlers complete
       Chef Client finished, 1 resources updated
       Finished converging <default-centos-64> (0m5.62s).
-----> Kitchen is finished. (0m8.97s)
[tdi@tk01 netcat]$ kitchen list
Instance             Driver   Provisioner  Last Action
default-ubuntu-1204  Vagrant  Chef Solo    Converged
default-centos-64    Vagrant  Chef Solo    Converged
```

Now everything works fine!

Advantages and Disadvantages

I started out as a skeptic, when it came to Chefspec. The system doesn't do a real converge and is really only decoration atop Chef's own recipe DSL. Chef, by virtue of being a declarative system, is inherently providing the most basic test of all. If I declare the state I want, and I run Chef, Chef will take action to make my wishes take effect. The Chef run will either succeed, in which case my desired state will take effect, or it will fail, with an error message and a stack trace.

However, the fact is that this is a clumsy and ineffective way of catching mistakes. Chefspec allows us to get feedback almost instantly, without having to take action on a real node, and very rapidly pays for itself in terms of time saved. Because the Chef run takes place in memory, and the provider actions are always set to not truly take effect, the speed of the test is remarkable. To give a sense of the speed, it's possible to run 10,000 tests in 30 seconds. Realistically you might have as many as 50 tests in a single cookbook, and they should all run in about a second. This is a much more effective and efficient way to catch mistakes. When Chefspec is partnered with Guard, for immediate feedback whenever the filesystem changes, the feedback is even quicker.

One common mistake I see people make is to forget to create a cookbook file or template, or to give it a subtly incorrect name, or perhaps to fail to put it in the *default* directory. Chefspec catches such errors without us having to go through the cycle of cookbook edit, cookbook upload, run Chef, and then wait for a bunch of resources to be applied, only to discover a simple error.

The ability to test multiplatform logic without ever needing to fire up machines of different types is also hugely advantageous. Fauxhai allows us to mock any platform and test the logic of our recipes even if we only ever develop on a Mac or Windows machine.

Perhaps the biggest business benefit that Chefspec delivers is in supporting the effort of refactoring a recipe. A common example would be perhaps reaching a decision to split up a large and complex cookbook into smaller, more logical components. This could deliver results in terms of faster Chef runs, and enhanced readability and maintainability. However, when refactoring, it's surprisingly easy to miss a resource out —perhaps a seemingly insignificant file, or package resource. I've certainly experienced

exactly this scenario: the recipe computes, the Chef run completes, with no indication of a problem. On a machine that has already been configured with Chef, the error may not ever be discovered because the effect of running Chef previously was to configure the resource, and simply removing the resource from the recipe won't undo the state of the machine where Chef previously took action. This means that only when Chef is run against a new machine does the missing resource cause an issue, often to the bafflement of the developer. Chefspec catches these regressions. If we write a test for the resource, and then accidentally change or delete the resource in the recipe, Chefspec will fail, immediately, never leaving a real machine in an incorrect or unknown state.

Sometimes the errors that Chefspec could catch or prevent are surprisingly inconvenient or damaging. Imagine the case of a cookbook responsible for configuring ssh access, or firewall or network settings. It's very easy to make a silly mistake—forget to write out a config, or set an incorrect permission—and when working with a remote machine, access to the whole system could be lost, with costly consequences.

Yet another example is the use of search to write out hosts entries, or perhaps load balancer configuration. A simple typing mistake—specifying the wrong index, or a subtly incorrect query—could result in a badly misconfigured system. With Chefspec, we stub out the search but make explicit the expectation that search should be called against a specific node with a specific query. If by some means, this query is incorrect in the recipe, our tests will fail, and we'll avoid misconfiguring the system. I've certainly had the experience where I've accidentally pressed a key in an open buffer, saved the buffer, and uploaded a recipe with a syntax error. Running Chefspec, under Guard, alerts in this situation immediately, resulting in far fewer silly mistakes.

However, it's not just catching silly mistakes or regressions that delivers value. There's something deeply satisfying, something addictively enjoyable about watching a recipe's journey from red to green. It introduces a sense of achievement, a yardstick for progress, and delivers a delicious experience of knowing when you're done, an experience that is painfully absent in most forms of knowledge work.

The simple fact is that writing your cookbooks test-first and using Chefspec as part of your development workflow will result in you writing better cookbooks.

Summary and Conclusion

Chefspec has deservedly earned a strong following within the Chef community already. It provides excellent return on investment, delivers rapid feedback, and enhances code quality and maintainability. The project is actively developed and well-documented. Unit testing at the level of resources and recipes is an essential part of the infrastructure developer's workflow, and Chefspec is the tool to use for this purpose.

Chefspec is a very powerful tool and can be used to perform very complex tests involving sophisticated mocking and stubbing, stepping into LWRPs to test their internal actions,

and working with Berkshelf. It's also highly extensible—third-party additions exist, and if you write cookbooks including library or LWRP resources, you can create and ship custom matchers for other people to use. Although already providing rapid feedback, this can be improved and made near-instantaneous by using *Guard*—a command-line tool that watches for filesystem events and runs tests as soon as a file is changed. Sadly these subjects are beyond the scope of this book, but examples and documentation can be found online, or guidance can be found via the usual channels.

Static Analysis and Linting Tools

As a wrapper around the testing workflow I recommended earlier, there is tremendous value in having mechanisms in place to help maintain code quality and standards, and reduce waste and rework owing to trivial mistakes. This brief section discusses tools that support this effort.

Overview

I'm often asked "How can I get started with testing? What's the simplest thing I can do that adds value?" The lowest level of syntax, style, and lint testing is probably the answer.

Writing Chef recipes is, in some respects, similar to the slower pace of the early computer programmers. Running Chef on a node could take a few minutes to complete, only to yield an error that was the result of a foolish mistake. If this happens two or three times, we could easily have wasted 10 minutes or more. It's not uncommon to introduce peculiar little bugs such as a misnamed action argument or a typo on an attribute. This all builds up. When added to the already stated desire to start to define and check against community-agreed coding standards, it seems that what would be ideal would be some kind of static analysis of our Chef code before we run it.

There are a number of related tools that provide elements of this functionality. We'll look at:

- *Foodcritic*
- *Knife Cookbook Test*
- *Tailor*
- *Strainer*

Foodcritic is a linting tool for cookbooks. It sets out its two primary objectives as follows:

> To make it easier to flag problems in your Chef cookbooks that will cause Chef to blow up when you attempt to converge. This is about faster feedback. If you automate checks for common problems you can save a lot of time.
>
> To encourage discussion within the Chef community on the more subjective stuff—what does a good cookbook look like? Opscode has avoided being overly prescriptive, which

> by and large I think is a good thing. Having a set of rules to base discussion on helps drive out what we as a community think is good style.

Foodcritic ships with more than 30 default rules and can be easily extended. Both Etsy and CustomInk have contributed extensive and valuable rules, which extend the coverage, and the Foodcritic documentation gives clear instructions on how to add your own, either to be considered as default rules or pertinent to your own organization's standards.

Foodcritic is an excellent tool, but it doesn't actually test the syntax of your Ruby. Thankfully, `Knife` already has built-in functionality for this. It's simple but effective, using Ruby syntax checking to verify every file in a cookbook ending in *.rb* and *erb*.

The final obvious area to test is the style of your cookbooks against community Ruby standards. An ideal tool for this is Tailor. The project describes itself as follows:

> Tailor parses Ruby files and measures them with some style and static analysis "rulers." Default values for the Rulers are based on a number of style guides in the Ruby community as well as what seems to be common. More on this here.
>
> Tailor's goal is to help you be consistent with your style throughout your project, whatever style that may be.

Strainer grew out of the realization that with the combination of a linter, syntax checker, and style guide, one potentially has three separate commands to run to test one's code. That's not very convenient or efficient. Strainer allows a collection of testing tools to be grouped together under one file and run with one command. This makes it very easy to plumb the whole collection of tools together, and run as a single job on a continuous integration server.

Getting Started

`Knife cookbook test` is already included for you if you installed Chef. To check syntax, simply run:

```
$ knife cookbook test mycookbook
```

You may need to specify your cookbook path with the `-o, --cookbook-path` option.

Assuming you have already run `berks init` in your cookbook directory, you will already have a Gemfile. Foodcritic, Tailor, and Strainer are all shipped as Rubygems, so add a line in your Gemfile for each gem, and then run `bundle install`. For now we'll remove the kitchen paraphernalia, and concentrate purely on the linting and static analysis aspects. Our Gemfile, therefore, looks like this:

```
$ cat Gemfile
source 'https://rubygems.org'

gem 'berkshelf'
gem 'foodcritic'
```

```
gem 'tailor'
gem 'strainer'
```

Running `bundle install` yields:

```
$ bundle install
Fetching gem metadata from https://rubygems.org/........
Fetching gem metadata from https://rubygems.org/..
Resolving dependencies...
Using i18n (0.6.1)
Using multi_json (1.7.6)
Using activesupport (3.2.13)
Using addressable (2.3.4)
Using builder (3.2.2)
Using gyoku (1.0.0)
Using nokogiri (1.5.9)
Using akami (1.2.0)
Using timers (1.1.0)
Using celluloid (0.14.1)
Using hashie (2.0.5)
Using chozo (0.6.1)
Using multipart-post (1.2.0)
Using faraday (0.8.7)
Using json (1.8.0)
Using minitar (0.5.4)
Using mixlib-config (1.1.2)
Using mixlib-shellout (1.1.0)
Using retryable (1.3.3)
Using erubis (2.7.0)
Using mixlib-log (1.6.0)
Using mixlib-authentication (1.3.0)
Using net-http-persistent (2.8)
Using net-ssh (2.6.7)
Using solve (0.4.4)
Using ffi (1.8.1)
Using gssapi (1.0.3)
Using httpclient (2.2.0.2)
Using little-plugger (1.1.3)
Using logging (1.6.2)
Using rubyntlm (0.1.1)
Using rack (1.5.2)
Using httpi (0.9.7)
Using nori (1.1.5)
Using wasabi (1.0.0)
Using savon (0.9.5)
Using uuidtools (2.1.4)
Using winrm (1.1.2)
Using ridley (0.12.4)
Using thor (0.18.1)
Using yajl-ruby (1.1.0)
Using berkshelf (1.4.5)
Using gherkin (2.11.8)
Using rak (1.4)
```

```
Using polyglot (0.3.3)
Using treetop (1.4.14)
Using foodcritic (2.1.0)
Using log_switch (0.4.0)
Using strainer (2.1.0)
Using tins (0.8.0)
Using term-ansicolor (1.2.2)
Using text-table (1.2.3)
Using tailor (1.2.1)
Using bundler (1.3.5)
Your bundle is complete!
Gems in the group integration were not installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Once installed, running `foodcritic` without options will yield the following (or similar) options:

```
> foodcritic
foodcritic [cookbook_paths]
     -r, --[no-]repl                     Drop into a REPL for interactive rule
editing.
     -t, --tags TAGS                    Only check against rules with the speci-
fied tags.
     -f, --epic-fail TAGS               Fail the build if any of the specified
tags are matched.
     -c, --chef-version VERSION        Only check against rules valid for this
version of Chef.
    -C, --[no-]context                Show lines matched against rather than the
default summary.
    -I, --include PATH                Additional rule file path(s) to load.
     -S, --search-grammar PATH          Specify grammar to use when validating
search syntax.
    -V, --version                     Display the foodcritic version.
```

Foodcritic has the idea of rules, against which your cookbook code is tested. Examples range from stylistic—`FC019: Access node attributes in a consistent manner`—to syntactical—`FC010: Invalid search syntax`—to portable: `FC024: Consider adding platform equivalents`.

To get started, simply navigate to a directory or folder containing a cookbook and run:

```
> foodcritic .
```

If you wish to include extra rules, clone the CustomInk and Etsy repositories into a convenient location, and include the location with the `-I --include` argument.

The `tailor` command line will, by default, look in a *lib* directory for Ruby files, and check style against a standard set of guidelines. These guidelines are configurable, either on the command line or in a configuration file. For testing cookbooks, the following command line will provide a sensible testing regime:

```
$ tailor */**/*.rb
```

Note that this will not check ERB templates, and it won't find any files more than one directory deep. You can compare what Tailor tested against what you have in your cookbook by running the following (on a Linux/Unix system):

```
$ find . -name \*.rb
```

Example

To explore Foodcritic, let's pick a cookbook from the community site at random, and see how it measures up:

```
PS C:\Users\stephen\src> knife cookbook site download monit
Downloading monit from the cookbooks site at version 0.7.0 to C:/Users/
stephen/src/monit-0.7.0.tar.gz
Cookbook saved: C:/Users/stephen/src/monit-0.7.0.tar.gz
PS C:\Users\stephen\src> tar xzvf .\monit-0.7.0.tar.gz
...
PS C:\Users\stephen\src> cd .\monit
PS C:\Users\stephen\src\monit> foodcritic .
FC012: Use Markdown for README rather than RDoc: ./README.rdoc:1
FC023: Prefer conditional attributes: ./recipes/default.rb:5
FC027: Resource sets internal attribute: ./recipes/default.rb:14
FC043: Prefer new notification syntax: ./libraries/monitrc.rb:8
FC043: Prefer new notification syntax: ./recipes/default.rb:20
FC045: Consider setting cookbook name in metadata: ./metadata.rb:1
PS C:\Users\stephen\src\monit>
```

In this case, the Monit cookbook is using an out-of-date README format. Additionally, when dropping off the default Monit config, it wraps the resource in an `if` condition rather than using the `cookbook\_file only\_if` metaparameter. The Monit service explicitly sets the enabled attribute to true, when this would be better set by using the action parameter. On two occasions, deprecated notification syntax is used and finally, the name of the cookbook is not explicitly set in the metadata.

Let's fix each of these in turn. First, on closer inspection, there's already a *README.md*, but it isn't written in Markdown. Fixing that is pretty simple in this case. Now let's remove the old *rdoc* version.

Looking at the conditional logic in the default recipe:

```
if platform?("ubuntu")
  cookbook_file "/etc/default/monit" do
    source "monit.default"
    owner "root"
    group "root"
    mode 0644
  end
end
```

The cookbook metadata doesn't specify which platforms it supports, but it seems to assume based on which Monit is available in the default package repositories. Rather

than leave it to guesswork, it would be better to remove the platform check altogether and explicitly state that the cookbook supports only Ubuntu. As other platforms are tested, they can and should be added to both the README and the metadata. While we're at it, we can add a name parameter to the metadata, so if the name of the directory containing the cookbook changes, `knife cookbook` commands still function. The metadata now reads:

```
name "monit"
maintainer       "Alex Soto"
maintainer_email "apsoto@gmail.com"
license          "MIT"
description       "Configures monit.  Originally based off the 37 Signals Cook-
book."
long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
version          "0.7"
supports "ubuntu"
```

The notification syntax is next. It currently reads:

```
notifies :restart, resources(:service => "monit"), :immediately
```

This should be:

```
notifies :restart, "service[monit]", :immediately
```

Finally, let's change the service resource to start and enable Monit:

```
service "monit" do
  action [:enable, :start]
  supports [:start, :restart, :stop]
end
```

Having made these changes, let's run Foodcritic again:

```
PS C:\Users\stephen\src\monit> foodcritic .

PS C:\Users\stephen\src\monit>
```

We now have a clean cookbook, which meets all the default Foodcritic rules.

So what about `knife cookbook test`? You get this for free, it's just available within Chef. We can test our irc cookbook:

```
$ knife cookbook test irc
checking irc
Running syntax check on irc
Validating ruby files
Validating templates
```

Running Tailor against our irc cookbook gives promising results:

```
$ tailor **/*.rb
#--------------------------------------------------------------------------------#
#                           Tailor Summary                                       |
#--------------------------------------------------------------------------------#
```

```
#   File                                                                  | Probs |
#-------------------------------------------------------------------------------#
# recipes/default.rb                                                      |     0 |
#-------------------------------------------------------------------------------#
#   TOTAL                                                                 |     0 |
#-------------------------------------------------------------------------------#
```

However, running against another randomly selected cookbook from the community site yields complaints about line length:

```
$ tailor **/*.rb
#-------------------------------------------------------------------------------#
# File:
#   attributes/default.rb
#
# File Set:
#   default
#
# Problems:
#  1.
#    * position:  20:114
#    * property:  max_line_length
#    * message:   Line is 114 chars long, but should be 80.
#  2.
#    * position:  21:99
#    * property:  max_line_length
#    * message:   Line is 99 chars long, but should be 80.
#
#-------------------------------------------------------------------------------#
#-------------------------------------------------------------------------------#
# File:
#   recipes/default.rb
#
# File Set:
#   default
#
# Problems:
#  1.
#    * position:  22:99
#    * property:  max_line_length
#    * message:   Line is 99 chars long, but should be 80.
#
#-------------------------------------------------------------------------------#
#-------------------------------------------------------------------------------#
#                           Tailor Summary                                      |
#-------------------------------------------------------------------------------#
#   File                                                                  | Probs |
#-------------------------------------------------------------------------------#
# attributes/default.rb                                                   |     2 |
# recipes/default.rb                                                      |     1 |
#-------------------------------------------------------------------------------#
#   Error                                                                 |     3 |
#-------------------------------------------------------------------------------#
```

```
#   TOTAL                                                                 |     3 |
#-------------------------------------------------------------------------------#
```

Strainer is designed to funnel a range of disparate tests into one place. With a single configuration file, we can encapsulate all the tests we want to run, into a single command that is trivial for a continuous integration server to run. All that is required is the creation of a *Strainerfile* in the root of the cookbook:

```
$ cat Strainerfile
# Strainerfile
knife test: bundle exec knife cookbook test $COOKBOOK
foodcritic: bundle exec foodcritic -f any $SANDBOX/$COOKBOOK
tailor: bundle exec tailor $SANDBOX/$COOKBOOK/**/*.rb
```

Now in a single command we can see the health of our cookbook:

```
$ bundle exec strainer test
# Straining 'irc (v0.1.0)'
knife test           | bundle exec knife cookbook test irc
knife test            | /home/tdi/.gem/ruby/1.9.3/gems/bundler-1.3.5/lib/bundler/
rubygems_integration.rb:214:in `block in replace_gem': chef is not part of the
bundle. Add it to Gemfile. (Gem::LoadError)
knife test           | from /home/tdi/.gem/ruby/1.9.3/bin/knife:22:in `<main>'
knife test            | Terminated with a non-zero exit status. Strainer assumes
this is a failure.
knife test           | FAILURE!
foodcritic             | bundle exec foodcritic -f any /home/tdi/chef-repo/cook-
books/irc
foodcritic               | FC008: Generated cookbook metadata needs updating: /
home/tdi/chef-repo/cookbooks/irc/metadata.rb:2
foodcritic               | FC008: Generated cookbook metadata needs updating: /
home/tdi/chef-repo/cookbooks/irc/metadata.rb:3
foodcritic           | Terminated with a non-zero exit status. Strainer assumes
this is a failure.
foodcritic           | FAILURE!
tailor                | bundle exec tailor /home/tdi/chef-repo/cookbooks/irc/**/
*.rb
tailor                                                                        |
#-------------------------------------------------------------------------------#
tailor                    | #                                    Tailor  Summa-
ry                                    |
tailor                                                                        |
#-------------------------------------------------------------------------------#
tailor                                                          |    #
File                                                               | Probs |
tailor                                                                        |
#-------------------------------------------------------------------------------#
tailor                                                      |    #    irc/recipes/
default.rb                                                |     0 |
tailor                                                                        |
#-------------------------------------------------------------------------------#
tailor                                                        |    #           TO-
TAL                                                              |     0 |
```

```
tailor                                                                    |
#-------------------------------------------------------------------------#
tailor              | SUCCESS!
```

Aha, we just need to ensure that Chef is in the Gemfile. I know from experience that if we don't set a version constraint, Bundler installs a prehistoric version of Chef, which breaks everything. I don't fully understand why, but in the spirit of full disclosure, I tell you. This sort of thing will happen—you'll bash your head on the desk for a few hours wondering why things aren't working as they should, but at times like this, I think it's valuable to reflect on quite how pioneering this discipline is. Many of the ideas we're putting into practice, and the tools we're using, are very new. The community is responsive, supportive, and fun. The cost of this is that sometimes things don't always go as smoothly as we'd like.

Update the Gemfile, include Chef, and run `bundle install`. Once the bundle has installed, we can run Strainer one more time:

```
$ bundle exec strainer test
# Straining 'irc (v0.1.0)'
knife test          | bundle exec knife cookbook test irc
knife test          | checking irc
knife test          | Running syntax check on irc
knife test          | Validating ruby files
knife test          | Validating templates
knife test          | SUCCESS!
foodcritic            | bundle exec foodcritic -f any /home/tdi/chef-repo/cook-
books/irc
foodcritic              | FC008: Generated cookbook metadata needs updating: /
home/tdi/chef-repo/cookbooks/irc/metadata.rb:2
foodcritic              | FC008: Generated cookbook metadata needs updating: /
home/tdi/chef-repo/cookbooks/irc/metadata.rb:3
foodcritic          | Terminated with a non-zero exit status. Strainer assumes
this is a failure.
foodcritic          | FAILURE!
tailor               | bundle exec tailor /home/tdi/chef-repo/cookbooks/irc/**/
*.rb
tailor                                                                    |
#-------------------------------------------------------------------------#
tailor                    | #                                    Tailor Summa-
ry                                  |
tailor                                                                    |
#-------------------------------------------------------------------------#
tailor                                                        |    #
File                                                             | Probs |
tailor                                                                    |
#-------------------------------------------------------------------------#
tailor                                                    |    #    irc/recipes/
default.rb                                              |     0 |
tailor                                                                    |
#-------------------------------------------------------------------------#
tailor                                                      |    #           TO-
```

```
TAL                                                                    |     0 |
tailor                                                                        |
#-------------------------------------------------------------------------------#
tailor               | SUCCESS!
```

Our irc cookbook has failed on `FC008`. Fixing this is left as an exercise for the reader!

Advantages and Disadvantages

The advantage of this set of tools is that they are absolutely the lowest barrier to entry possible. They can be built right into a simple continuous integration or continuous delivery pipeline. For an example of how simple this is to achieve with a public service such as TravisCI, see Nathen Harvey's blog posts on Foodcritic and TravisCI and Knife Test and TravisCI.

Once you've got the discipline of running regular tests, checking your style and syntax against community standards, you can start to layer in more complex testing.

The only disadvantage is that there can be some tension in finding a community style that suits all the members of your team, and then enforcing it. Thankfully, Tailor is pretty much infinitely configurable, so as long as you can find a style that you all agree on, and isn't massively at odds with the rest of the Ruby or Chef community, you're probably going to derive benefit from monitoring, measuring, and enforcing adherence to that style.

Summary and Conclusion

If you do nothing else, do this. The cost of implementation is low, and the return on investment is high. Get yourself set up with the basics of a continuous integration pipeline, where your static analysis and linting tests are run on every commit, and then start to layer on more advanced testing.

To Conclude

The workflow and tooling recommended in this chapter represent a snapshot in time. It is very much my hope that by emphasizing the philosophical aspects of test-driven infrastructure and the rationale behind the current selection of tools, there is value in this book that extends way beyond a specific set of recommendations.

However, to summarize, my current recommended toolchain and workflows are, in brief, as follows:

1. Build upon a solid foundation by using a combination of Berkshelf and Test Kitchen to orchestrate and manage the infrastructure and cookbooks that build it.

2. Write acceptance tests first, using Gherkin as the requirements capturing language, Cucumber as the test runner, and Leibniz as the interface to the provisioning engine of Test Kitchen and Berkshelf.
3. Write integration tests next, using Test Kitchen as the test runner, and using whichever test framework most suits your experience and skillset.
4. Write unit tests last, using Chefspec, and think seriously about the art and science of unit testing, and making appropriate use of RSpec's mocking and stubbing capabilities to keep tests isolated and fast.
5. Wrap all your cookbook development endeavors in a process that reinforces agreed standards of code quality and style, using Strainer as the collecting and running mechanism, and using Knife Cookbook Test, Foodcritic, and Tailor.
6. Automate the running of your static, linting, and unit tests, using Guard, and also a form of continuous integration such as Travis CI or Jenkins.
7. Automate the running of cookbook integration tests by driving Test Kitchen from within a continuous integration system such as Jenkins, or if using an appropriate driver, Travis.
8. Treat your acceptance tests as a foundation for monitoring the day-to-day behavior of your built systems, plugging relative components into your monitoring and alerting systems.

CHAPTER 8
Epilogue

This is a substantial book, covering a large and rapidly expanding subject area. Constraints have been imposed at various stages for purely practical reasons. In this final section, I want to enumerate very briefly what some of these constraints are, what has been specifically left out of scope, what I hope to be able to include in further incarnations, and where additional guidance, documentation, and support may be found.

An immediate constraint is that while I had every intention of making this book 100% compatible with Microsoft Windows, doing so would have expanded the examples and setup by a significant factor. It's not that Windows is in any way a less supported citizen, it's just that there are nuances involved, both in terms of its automation as a server platform and its use as a development platform, which led me to focus my attentions on Linux as the primary use case in this text. As a technologist, I am very enthusiastic about the Microsoft technology stack, and as a consultant and trainer, I have worked extensively with Windows infrastructure automation. As an area of interest, it is something I intend to devote more dedicated time and material to in the near future.

An explicit and hopefully obvious constraint is that this is a practical and philosophical book about the process by which we develop infrastructure code. It's not a complete introduction or tutorial for Chef, nor is it an advanced or comprehensive discussion of the framework. That said, I have explicitly assumed absolutely no familiarity with the framework, and the reader who works her way through the book will find herself rapidly able to be effective in Chef. In that respect, the present volume serves admirably well as an introduction to Chef when used alongside the existing documentation and materials provided by both Opscode and the community.

There are areas to which, owing to the constraints of time and space, I was unable to devote attention. The whole process by which we automate the running of tests and their feedback—the building of a continuous integration system, and the path towards a build pipeline for infrastructure code—is a highly relevant and most fascinating subject. Opscode is increasingly positioning itself as a specialist in the field of continuous

delivery, and its consultants have valuable and unique insights to share. This is certainly an area where I intend to focus time in both research and writing, and I would not be surprised to see contributions to the discussions and literature on the subject coming from Opscode—either formally or informally.

Similarly, I feel some of the social aspects of agile and lean development are very highly relevant to the discipline of infrastructure as code. I would love to have been able to discuss and demonstrate code review processes using, for example, *Gerrit* or *Reviewboard*, and to explore some of the principles around which I feel effective teams organize, such as pair programming and flow-based workflow management.

The publishing industry has changed beyond all recognition in the last 20 years. The emergence of digital delivery and multimedia-enriched content has made the task of an author somewhat different. In my heart, I believe that writing (and publishing) exists because there are problems to be solved, and people who want to help solve them. The fact that we can't solve all these problems in a single book, and the fact that the present problem domain is so volatile, should not discourage us. As an author, I am committed to continue to educate, entertain, and synthesize, so where I've been unable to cover all that I would have liked to, I am confident that content on these subjects will, nevertheless, be forthcoming.

This book includes a comprehensive bibliography and has referenced and encouraged the user to make use of the excellent community in which the Chef framework is developed. I would urge the reader to engage with the community, via the mailing lists, IRC, the frequent conferences and user groups, and by creating and consuming online content. It's my sincere hope that this book has whet your appetite, and that you will add your voice to the conversation.

APPENDIX A
Bibliography

There are many excellent books on test-driven and behavior-driven development, plus several on the tools that underpin the approaches discussed in this book. Here's a selection of books that have informed my own views, and books that will reward further study.

Books on TDD and ATDD

- *Test-Driven Development: By Example* by Kent Beck (Addison-Wesley Professional, 2002)
- *Test-Driven Development: A Practical Guide* by David Astels (Prentice Hall, 2003)
- *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin; Janet Gregory (Addison-Wesley Professional, 2008)
- *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration* by Ken Pugh, Aslak Hellesoy, et al.(Addison-Wesley Professional, 2010)
- *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development* by Markus Gärtner (Addison-Wesley Professional, 2012)
- *Specification by example: How successful teams deliver the right software* by Adžić, Gojko (Manning, 2011)
- _ Bridging the communication gap: Specification by Example and Agile Acceptance Testing_, by Adžić, Gojko (Neuri Ltd., 2009)

Books and Articles on BDD

- *Instant Cucumber BDD How-to* by Wayne Ye (Packt Publishing, 2013)

- Introducing BDD by Dan North
- What's in a story? by Dan North

Books on Agile Testing in General

- *Beautiful Testing*, ed. by Adam Goucher and Tim Riley (O'Reilly, 2009)
- *Impact Mapping: Making a Big Impact with Software Products and Projects* by Gojko Adzic, Marjory Bisset, and Nikola Korac (Provoking Thoughts, 2012)

Chef Articles and Presentations

- Guide on Authoring Cookbooks
- Slideshare: The Berkshelf Way

Books on Tools

- *The RSpec Book* by David Chelimsky et al. (Pragmatic Bookshelf, 2010)
- *Learning GNU Emacs*, Third Edition by Debra Cameron, James Elliott, Marc Loy, Eric S. Raymond, and Bill Rosenblatt (O'Reilly, 2004)
- *Version Control with Git*, Second Edition by Jon Loeliger and Matthew McCullough (O'Reilly, 2012)
- *Jenkins: The Definitive Guide* by John Ferguson Smart (O'Reilly, 2011)
- *Jenkins Continuous Integration Cookbook* by Alan Berg (Packt Publishing, 2012)

Books on Ruby

- *The Ruby Way: Solutions and Techniques in Ruby Programming*, Second Edition by Hal Fulton and Russ Olsen (Addison-Wesley Professional, 2006) (A third edition is scheduled for publication in December 2013.)
- *Why the Lucky Stiff's (Poignant) Guide to Ruby*
- *Programming Ruby*, Second Edition by Dave Thomas, with Chad Fowler and Andy Hunt (Pragmatic Programmers, 2005)
- *The Ruby Programming Language* by David Flanagan and Yukihiro Matsumoto (O'Reilly, 2008)

- *Eloquent Ruby* by Russ Olsen (Addison-Wesley Professional, 2011)
- *The Well-Grounded Rubyist* by David A. Black (Manning Publications, 2009)
- *Metaprogramming Ruby* by Paolo Perrotta (Pragmatic Bookshelf, 2010)
- *Design Patterns in Ruby* by Russ Olsen (Addison-Wesley Professional, 2007)
- *Practical Object-Oriented Design in Ruby* by Sandi Metz (Addison-Wesley Professional, 2012)

Books on Bash and Shell Scripting

- *Classic Shell Scripting* by Arnold Robbins and Nelson H. F. Beebe (O'Reilly, 2005)
- *Shell Scripting* by Steve Parker (Wrox, 2011)
- *Learning the bash Shell (A Nutshell Handbook)* by Cameron Newham and Bill Rosenblatt (O'Reilly, 1998)
- *bash Cookbook* by Carl Albing, JP Vossen, and Cameron Newham (O'Reilly, 2007)
- See also Bash Guide (excellent for beginners) and BashFAQ (for FAQ/cookbooks)

General Programming Books

- *Extreme Programming Explained* by Kent Beck and Cynthia Andres (Addison Wesley, First edition, 1999, and Second edition, 2004)
- *Mastering Regular Expressions* by Jeffrey E.F. Friedl (O'Reilly, 1997)
- *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin (Prentice Hall, 2008)

Other Great Books

- *Web Operations*, ed. John Allspaw and Jesse Robbins (O'Reilly, 2010)
- *Continuous Delivery* by Jez Humble and David Farley (Addison Wesley, 2010)
- *The Art of Capacity Planning: Scaling Web Resources* by John Allspaw (O'Reilly, 2008)
- *The Art of Agile Development* by James Shore (O'Reilly, 2007)
- *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley, 2003)

- *Kanban: Successful Evolutionary Change for Your Technology Business* by David Anderson (Blue Hole Press, 2010)
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley, 2009)
- *Exploring Requirements: Quality Before Design* by Donald C. Gause and Gerald M. Weinberg (Dorset House Publishing, 2011)
- *Lean Software Development: An Agile Toolkit* by Mary Poppendieck and Tom Poppendieck (Addison-Wesley Professional, 2003)
- *User Stories Applied: For Agile Software Development* by Mike Cohn (Addison-Wesley, 2004)
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Kent Beck,et al.(Addison-Wesley, 1999)
- *Agile Software Development, Principles, Patterns, and Practices* by Robert C. Martin (Pearson, 2011)
- *The Visible Ops Handbook* by Kevin Behr, Gene Kim and George Spafford (IT Process Institute, 2005)
- *Introduction to Real ITSM* by Rob England (CreateSpace, 2008)
- *Devops for Developers* by Michael Hüttermann (Apress, 2012)
- *High Performance Web Sites: Essential Knowledge for Front-End Engineers* by Steve Souders (O'Reilly, 2007)
- *Even Faster Web Sites: Performance Best Practices for Web Developers* by Steve Souders (O'Reilly, 2009)
- *Scalable Internet Architectures* by Theo Schlossnagle (Developer's Library, 2007)
- *Release It!: Design and Deploy Production-Ready Software* by Michael T. Nygard (Pragmatic Programmers, 2007)
- *Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications* by Cal Henderson (O'Reilly, 2006)
- *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud* by George Reese (Theory in Practice) (O'Reilly, 2009)
- *High Performance MySQL: Optimization, Backups, Replication, and More* by Baron Schwartz, Peter Zaitsev et al. (O'Reilly, 2008)
- *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble David Farley (Addison-Wesley Signature Series, 2010)
- *MySQL High Availability: Tools for Building Robust Data Centres* by Charles Bell, Mats Kindahl and Lars Thalmann (O'Reilly, 2010)

- *Continuous Integration* by Paul M Duvall, Steve Matyas and Andrew Glover (Addison-Wesley, 2007)
- *Lean IT* by Stephen C Bell and Michael A Orzen (Productivity Press, 2010)
- *Management Challenges for the 21st Century* by Peter F. Drucker (Butterworth-Heinemann, 2007)

Index

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

C

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

About the Author

Stephen Nelson-Smith (@LordCope) is principal consultant at Atalanta Systems, a fast-growing agile infrastructure consultancy, and Opscode training and solutions partner in Europe. One of the foundational members of the emerging Devops movement, he has been implementing configuration management and automation systems for five years for clients ranging from Sony, the UK government and Mercado Libre to startups amongst the burgeoning London 'Silicon Roundabout' community. A UNIX sysadmin, Ruby and Python programmer, and lean and agile practitioner, his professional passion is ensuring operations teams deliver value to the business. He is the author of the popular blog http://agilesysadmin.net, and lives in Hampshire, UK, where he enjoys outdoor pursuits, his family, reading, and opera.

Colophon

The animal on the cover of *Test-Driven Infrastructure with Chef, Second Edition* is an edible-nest swiftlet (*Aerodramus fuciphagus*). This small bird, of the swift family, is found in southeast Asia.

The bird itself is 11–12 cm long and weighs around 15–18 grams. The top plumage is a blackish-brown with paler underparts; its bill and feet are black. It has a slightly forked tail and long, narrow wings. When in caves—usually for breeding—these birds are known to use loud, rattling calls for echolocation.

This swiftlet's diet consists of flying insects that get caught in its wings. It feeds in large flocks, often with other species of swift and swallow. The swiftlet's nest, shaped like brackets, is made from solidified saliva and is among the most expensive animal products consumed by humans, going for an average of $2,500 per kg in Asia. It is used primarily as an ingredient in bird's nest soup, where the nest is soaked and steamed in water. The nests are said to be an aphrodisiac with medicinal qualities. Because of extensive commercial harvesting, the IUCN has labeled several populations—in the Andaman and Nicobar Islands—as critically threatened. To combat the effects of harvesting, the use of artificial bird houses is growing.

The cover image is from *Cassells Natural History*. The cover font is URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.