# RELIABLY DEPLOYING RAILS APPLICATIONS

Reliable, repeatable deployment & provisioning

BY BEN DIXON

# Reliably Deploying Rails Applications

Hassle free provisioning, reliable deployment

Ben Dixon

This book is for sale at http://leanpub.com/deploying_rails_applications

This version was published on 2021-04-02

# Contents

# Intro

## Preface to the fifth edition (2021)

The first edition of Reliably Deploying Rails applications was released in November 2013 and was based on Ubuntu 12.04 and Ruby 1.9.2. Since then a lot has changed in the world of application deployment. The shift towards devops has hugely increased the number of people exposed to the deployment process, CI and SSL has become ubiquitous and containerisation has rapidly gained in popularity.

But even amongst all this change, the basic requirement, to be able to spin up a VPS, deploy a Rails application to it and then largely forget about it, seems to live on. I remain a huge fan of Docker, Kubernetes and the ecosystems that surround them, but I've yet to find anything that comes close to the simplicity and reliability of provisioning a VPS with Chef and then using Capistrano to deploy a Rails application to it.

Systems I built 10 years ago have been running on this setup with minimal maintenance requirements on the infrastructure side and little or no downtime. And with the like of Hetzner Cloud providing perfectly usable virtual severs for less than 5 dollars a month, you can serve a surprising amount of traffic for an astonishingly low cost.

The fifth edition updates to the stack to cover Ruby 3.0. Rails 6 and Ubuntu 20.04 LTS. It adds automatic SSL with LetsEncrypt, migrates the application server from Unicorn to Puma and standardises on userspace systemd for managing the persistence of services without requiring sudo access. It will include chapters on using Capistrano to deploy to common CI systems and modern backups with Restic.

When I wrote the first edition 8 years ago, I thought at most, some 10's of people would read it, so I wrote it mainly for the enjoyment of the exercise. Since then thousands of people have purchased it, and even more gratifyingly, hundreds have gotten in touch with suggestions, improvements and stories of how they're using it and what they're building.

I can't thank everyone who's purchased this book and gotten in touch enough, and I hope the fifth edition continues to enable people to reliably and affordably deploy Rails applications while keeping their time, focus and money on building amazing things.

Ben

## Current status of the fifth edition

As with the previous editions, the fifth edition has been released as a work in progress via Leanpub. This means that the fifth edition is not yet finished. If you purchase it now, you'll receive lifetime free updates of this and future editions as they are completed. Anybody who purchased previous editions will also receive these updates for free.

The chapter list below indicates the current status of each chapter with whether or not it has been updated yet for the fifth edition.

At a high level the "critical path" as been updated, so the sections on setting up the server and deploying to it with Capistrano are completed, with updated production ready sample code. So if your goal is "I need to get a Rails application deployed now", the book is ready for you.

What remains to complete is updating the supplementary chapters which go into the detail of what's happening behind the scenes and how to further customise the configuration. These will be completed over the coming months.

## The purpose of this book

This book will show you from start to finish how to:

- Setup a VPS from Scratch
- Automate setting up additional servers in minutes
- Use Capistrano to deploy reliably
- Automate boring maintenance tasks

If you've got applications on Heroku which are costing you a fortune, this will provide you with the tools you need to move them onto a VPS.

If you're already running your app on a VPS but the deploy process is flaky - it sometimes doesn't restart or loads the wrong version of the code - this book provides a template for making the process robust.

I've spent hundreds of hours combing through blog posts, documentation and tweaking configuration files. This has got me to the stage where deploying to a VPS is as easy as - in fact often easier than - deploying to Heroku. If you want to do the same, this book will save you a lot of time.

## Who is this book not for

This book is not for people who want to learn to deploy containerised Rails applications to the likes of Kubernetes. I remain a huge fan of containers and the ecosystem around them, but this book remains focused on the battle tested, simple as possible method of provisioning a VPS and deploying to it with Capistrano.

## If you're in a hurry

If what you need is just to get a Rails application deployed and working as quickly as possible then:

- Start with Chapter 4, installing tools

- Then complete Chapter 5, the Chef Quick Start to setup the server
- Finally complete Chapter 18, the Capistrano quickstart to configure your Rails application for deployment

The entire process should take less than half an hour and at the end of it you'll have a fully functional deployment to a VPS. Otherwise read on!

## About Me

I've been developing web applications for over fifteen years, over the last few years specialising in Rails development and deployment as well as Elixir and Kubernetes. I'm co-founder & CTO of Catapult[1] a venture backed global SaaS platform for maximising worker engagement.

Previously I was the technical lead at a health and fitness startup who provide the timetabling for many of the UK's leisure operators as well as producing a globally recognised iOS app (Speedo Fit) for swimmers.

As part of these projects I've dealt with everything from the usual rapid growth from 10's of requests per minute to 10's per second to more unusual challenges such as expanding infrastructure into China and debugging obscure indexing issues.

I spoke about deploying Rails applications at Railsconf 2014 and integrating Docker with Rails in 2015. In 2015 I also spoke at Refresh! conf in Tallinn about common mistakes people make when deploying and scaling web applications.

## Why This Book Exists

I came to Rails having worked a lot in PHP and then Python/ Django. I was converted to Rails by the way the convention over configuration orientated approach removed much of the boilerplate I was accustomed to writing, leaving me free to focus on the real functionality of the application.

The first real hiccup was when I came to deploy an application to something other than Heroku. This became a necessity to keep costs down on side projects with background jobs and on several larger production projects where more flexibility was needed.

Having Googled extensively on how to deploy Rails applications to a VPS and found many tutorials documenting the commands to be entered by hand, I went down this route. This felt slow and repetitive. Somehow however carefully I documented the process, it never quite worked the next time I needed to setup a server.

This grated compared to the rest of my work-flow where anything I needed on multiple projects was simply abstracted into a gem and re-used, but I persevered and by and large it worked. Throughout the process I talked to others at local Ruby groups and other companies and they seemed to be following a similar process.

Eventually the inevitable happened and a production server failed completely, necessitating a complete rebuild.

[1]https://www.catapult.com

Unfortunately at the time I was in rural France on a rare holiday where the only usable WiFi signal was obtained by sitting half in the wardrobe in the attic of a small cottage. After spending nearly 12 hours in a wardrobe, typing in commands over a connection barely capable of maintaining an SSH session, I realised this had to be a problem someone had already solved.

I then discovered configuration management, experimented with Chef, Puppet and Ansible and eventually settled on Chef. In a surprisingly short amount of time, I had a simple Chef configuration which allowed me to configure a Rails ready server with just a few local commands, with a near guarantee it would be functionally identical to all previous ones.

This guarantee meant I could also use a standard Capistrano configuration to deploy to it. Suddenly deploying a Rails application to a VPS for the first time went from something that took at least a day to something that took just a few minutes.

Talking to others about why they weren't using such a system, it became clear that configuration management was considered the realm of "Ops" with too steep of a learning curve for most developers who just wanted to deploy the odd application here and there. This didn't tally with my experience at all, Chef was just Ruby and the time taken to learn the basics had been barely longer than it usually took me to setup a server by hand.

I started doing more consulting work for companies wanting to get up and running with configuration management and sharing my notes on the process with colleagues and friends for their own projects. Someone suggested this would make a useful book. And here we are!

I've tried to strike a balance between providing sample code which works out of the box to get up and running quickly whilst still giving enough detail on the rationale behind it and how each component works so that anyone reading this will be able to gradually develop a template tailored to them.

## Intended Audience

This book is intended for people who develop Ruby on Rails applications and have to be involved with or, completely manage, the infrastructure and deployment of these applications.

Whether deploying applications is new to you or you've been doing it by hand for a while and want to move to a more structured approach, I hope this book will be useful to you.

## Pre-requisites

It's assumed anyone reading this is already proficient in Ruby and Rails development.

Some basic knowledge of the unix command line is assumed, in particular that you can:

- Use SSH to connect to remote servers
- Navigate around the file system from the shell (cd, ls, dir, mv, rm etc)
- Have a basic understanding of web architecture. E.g. what a server is, how to setup DNS etc.

## Structure of Part 1 - Setting up the server

Part 1 focuses on automated provisioning and uses Chef as the configuration management tool.

### Chapter 2 - The Stack

An overview of the components which will make up our production Rails configuration, their purpose and a high level look at the rationale between choosing each one.

This chapter has been updated for the fifth edition.

### Chapter 3 - Tools and Terminology

A brief look at why using a tool to automate the setting up of servers is important and some high level terminology we'll encounter when using this books automation tool of choice; Chef.

This chapter has been updated for the fifth edition.

### Chapter 4 - Installing Tools

How to install Chef and supporting tools using ChefDK and a brief look at how ChefDK works.

This chapter has been updated for the fifth edition.

### Chapter 5 - Quick Start

A chapter for the "read the instructions afterwards" types among us. This chapter provides step by step instructions for setting up a fully working, reproducible Rails server using the books sample code and a few simple commands. You'll have a server setup and ready within 30 minutes (and most of that is just waiting for Ruby to compile!).

This chapter has been updated for the fifth edition.

### Chapter 6 - Creating a new project

How to create a Chef project from scratch. Think `rails new` but for server configurations.

This chapter has been updated for the fifth edition.

### Chapter 7 - Using Knife

How to use Knife to interact with Chef so that we can setup our desired node end state and apply changes.

This chapter has been updated for the fifth edition.

## Chapter 8 - Creating a Chef Cookbook

A detailed guide to how to create custom Chef cookbooks. Cookbooks are like gems but for re-usable bits of functionality on a server. We'll begin with the commands we'd type in by hand to install a piece of software on the server and convert this into the simplest possible cookbook to automate the process. We'll then iterate on this to take advantage of some of Chefs more powerful features.

This chapter has not yet been updated for the fifth edition, the concepts and primitives remain the same, but the specific example used needs to be updated.

## Chapter 9 - Node and Role Definitions

First a look at the concept of a "node definition", a JSON file which defines everything about a server to be created. Then a look at how Chef allows us to create re-usable pieces of functionality which can then be applied to multiple servers or re-used in future projects using "roles".

This chapter has not yet been updated for the fifth edition, the concepts and primitives remain the same, but it builds heavily on Chapter 8 so will need updating following that.

## Chapter 10 - A Template for Rails Servers

A more detailed introduction to the sample configuration which was used in Chapter 4's quick-start to setup a fully working server in just a few minutes.

This chapter has not yet been updated for the fifth edition.

## Chapter 11 - Basic Server Setup

The bare bones components needed on any server, how - and when - to setup automatic package updates, how to have the systems time kept automatically updated and why so many people get caught out by locales.

This chapter has not yet been updated for the fifth edition.

## Chapter 12 - Security

Starting off with a look at some common security gotchas when deploying to a fresh VPS. Then some more detail around how to lock down SSH access, manage firewall rules and automatically set up users and public keys.

This chapter has not yet been updated for the fifth edition.

## Chapter 13 - Rails Server Setup

Making sure we have some basic packages required for installing common gems, then a more detailed look at how to install Ruby and make sure we have the correct version available to our Rails application.

This chapter has not yet been updated for the fifth edition.

## Chapter 14 - Monit

When something crashes, in a perfect world, it will fix itself. This chapter covers how to use Monit to automatically monitor the health of services and restart them when they fail. In the event this fails, how to have Monit alert us via email so we can intervene and save the day.

This chapter has not yet been updated for the fifth edition.

## Chapter 15 - Nginx

Nginx will be the first port of call when a request comes in. How to setup virtual hosts, serve static assets and a look at why we need Nginx at all.

This chapter has not yet been updated for the fifth edition.

## Chapter 16 - PostgreSQL

How to setup PostgreSQL, manage the different types of authentication, ensure we have the desired version and manage importing and exporting data.

This chapter has not yet been updated for the fifth edition.

## Chapter 17 - Redis and Memcached

A brief chapter on the two simplest parts of our stack. How to install Redis and Memcached, configuring whether or not they're bound to localhost and managing the maximum size they can reach.

This chapter has not yet been updated for the fifth edition.

# Structure of Part 2 - Deploying to the server

## Chapter 18 - Capistrano Quickstart

Like chapter 5, a chapter for those of us who prefer to read the instructions later. This chapter covers the minimum steps need to deploy an existing Rails application to our fresh new server.

This chapter has been fully updated for the fifth edition.

## Chapter 19 - Deploying With Capistrano

A much more detailed introduction to how to deploy to our new server with Capistrano 3 whilst using only core Capistrano gems.

This chapter has not yet been updated for the fifth edition.

## Chapter 20 - Writing Custom Capistrano Tasks

Our goal is to automate everything about interacting with our server. Eventually there will be something we need to do which no-one else has automated in a way we can re-use. This chapter covers writing custom Capistrano tasks. Once we've mastered this we can automate almost every interaction with our server imaginable.

This chapter has not yet been updated for the fifth edition although only minor changes are expected.

## Chapter 21 - Unicorn Configuration and Zero Downtime

How to configure deployment so that when we deploy, there's a seamless switch from one version to the next, with no gap while the new version starts. Includes a detailed section on causes of potential problems when setting this up and how to troubleshoot if it doesn't work reliably.

This chapter is due to be removed entirely in the fifth edition and replaced with a chapter on Puma. Zero downtime deployment is configured out of the box in the Quickstart chapter.

## Chapter 22 - Virtualhosts and SSL

A deeper look at Nginx virtualhosts which control where requests are routed after they reach NGinx. Then a look at how to setup SSL and manage certificate rotation.

This chapter has not yet been updated for the fifth edition. The section on SSL will be removed as this is now covered automatically in the quick start section.

## Chapter 23 - Sidekiq

Sidekiq is one of the most popular and efficient background job processing libraries in the Rails ecosystem. Here we cover how to integrate starting and stopping Sidekiq workers as part of the deployment process and how to setup monitoring so they'll be automatically restarted in the event of a failure.

This chapter has not yet been updated for the fifth edition. Deploying Sidekiq is now covered in this Quickstart, this chapter will be updated to cover some advanced configuration.

## Chapter 24 - Automated Backups

It's an unfortunate fact that eventually a server will fail. In this chapter we setup automatic database backups to Amazon S3 (or any other number of destinations) to ensure that in the event of a complete server failure, our data is recoverable.

This chapter has not yet been updated for the fifth edition, it will be replaced with a Restic based backup solution.

## Chapter 25 - Deploying with CI

This chapter has not yet been written

## Version

You are reading Version 5.1.0 of this book. The major and minor (x.x) numbers correspond to the version number in the sample code. The final number refers to minor improvements such as typos. Having purchased once, you can download the latest version of this book from Leanpub at any time.

# The Stack

## Overview

The stack used for examples in this book is just one of many possible configurations. I've picked a combination of components I've found to be the most common across clients but it's intended to be an example only.

If part of your intended stack is not included here don't despair, all of the general principles will apply.

If you know, or can find instructions on how to install the component in question, the techniques provided for automating with Chef can be adapted.

## Ubuntu 20.04 LTS

Ubuntu has become an increasingly common distribution in a server environment. The primary reason I recommend it is the level of community support available.

The sheer number of people of all abilities using it means that you are almost never the first person to have a particular problem so, in 99% of cases, a quick Stack Overflow search will point you in the right direction.

For the remaining 1% the community is extremely friendly and helpful.

## Nginx

Nginx is a web server and reverse proxy known for its high performance and small memory footprint.

A key benefit of nginx is that due to its event driven architecture, its memory usage is extremely predictable, even under heavy loads. This makes it ideal for projects that may begin on a small VPS for testing and grow to much larger machines as they scale.

Architecturally, when requests come into the server, they are handled by Nginx which then passes them back to our application server (for example Puma or Unicorn) which runs the rails application and returns the response.

## Postgresql

Postgresql is a traditional relational database. It has increasingly become the default for new rails applications using relational databases, in part because it is the default database supported by Heroku. It's my preferred database primarily because with the addition of native JSON support it is increasingly able to combine the benefits of a traditional RDMS with those of a NoSQL solution such as Mongo.

## Ruby + rbenv

This book covers and has been tested with Ruby 2.7.x and Ruby 3.0.x. Whilst a lot of the techniques may work with more exotic flavors such as JRuby, I have minimal experience with these and so will not cover them directly.

My preference for managing and installing ruby versions on production servers is rbenv. This is primarily because I find rbenvs operation simple to understand and therefore troubleshoot.

## Redis

Redis is an extremely fast key value store. It performs well in use cases such as caching and API rate limiting as well as more advanced areas such as calculating intersections between series.

It has a few gotchas such as its behavior when its max memory limit is reached which are covered in the section on configuring Redis.

## Memcached

Similar to Redis but entirely in memory (reboot the machine and you lose everything that was in Memcached). Great for caching, and like Redis, incredibly simple to install and maintain.

## Why This Stack

These components cover a majority of the applications I've encountered over the last few years. I'm generally fairly neutral in the "which is the best stack argument." There are lots of other great combinations out there and the concepts in this book will apply to most of them.

## Adapting to your stack

If your stack is not covered above, don't worry. The aim of this book is to show you how easy it is to use Chef to automate the provisioning of any Rails stack, the components here are just examples. By the end of the book you'll be able to either find community Chef recipes or write your own to setup almost anything.

Chef is used only for setting up the server, not for deployment. The second half of this book - deploying with Capistrano - will be relevant no matter which configuration management tool you use.

# Tools And Terminology

## Introduction

The ‘simplest’ way to provision a new server is to create a new VPS on something like Linode or Digital Ocean, login via SSH and start apt-get’ing the packages you need, dropping into vim to tweak config files and adding a few custom package sources where newer versions are needed.

When something new is needed we ssh back in, install or upgrade a package, building the server up in layers. Somewhere there’s a text file or wiki page with “all the commands we need” written down should we ever need to do it again.

This approach has a few key pitfalls;

1. It’s very hard to keep track of what’s been done. With the best will in the world the text file doesn’t quite get all of the commands. Nobody realises until the time comes to provision a new server (often under adverse conditions) and running all the commands in the file doesn’t quite yield a working server.
2. It’s slow (and therefore expensive). Even if the process is perfectly documented, someone is still needed to sit and run the commands. This is repetitive, boring work, but will often need to be done by an engineer whose time could be better spent working on the product itself.
3. It doesn’t scale. Having an engineer type in a list of commands might, with a lot of discipline, hold together when there’s only one server involved. Expand that to five or six servers and the cracks will soon start to show.

## Automation

The goal of this section and of this book as a whole, is to take the manual processes and automate them. As a general rule, any process which I’d expect to repeat more than once a year in the life-cycle of deploying and managing infrastructure, I try and automate.

Automation relates to an approach more than it relates to any particular tool. If we’re doing anything which involves running more than one command or sshing into a remote server more than once in a month, it’s probably worth stopping and thinking, “how can this be automated?”

The benefits are often visible after a remarkably short period of time. Automating server deployments not only makes disaster recovery easier, it makes the creation of accurate test and staging environments easier, making the testing of new deployments easier and more efficient and so decreasing downtime.

Automating the copying of production databases from production to development and staging environments makes it more likely developers will use current data in their tests. This makes tests in development more meaningful and so increases productivity and decreases costs.

Finally automation is usually easier than expected. Once we've mastered Chef for automating provisioning tasks and Capistrano for automating deployment tasks and subsequent interactions with production and staging servers, it becomes clear that the time taken to automate additional tasks is rarely significantly more than the time taken to perform them once.

## Introducing Chef

Chef is an automation platform made by Opscode which uses a Ruby DSL (Domain Specific Language) to represent the commands required to provision a server in a reusable format.

### Typical Chef Usage

A traditional Chef toolchain is made up of:

- Chef Server - This is where information about the intended setup of all nodes being managed resides
- Chef Client - This runs on every single node being managed and is responsible for executing the changes required to bring the node into the desired state
- Knife - This is the command line utility used to update the Chef Server with changes to the desired state of one or more nodes

So a minimal workflow might look like this:

1. Use Knife to tell Chef Server to `bootstrap` a node. This means installing Chef Client on the remote node and creating a "node definition" file for that node on the Chef server
2. Use Knife to tell Chef Server that we want certain "recipes" or "roles" added to the "run list" - for example the "postgesql::server" "recipe" from the "postgresql" "cookbook" - of the node
3. Use Knife to set "attributes" on our "node" which customise how the recipes in our "run list" behave. For example setting the port Postgres listens on.
4. Use Knife (or Berkshelf, explained later) to upload our Cookbooks to the Chef Server
5. Use Knife to tell Chef Server to "converge" our node
6. Chef Server will then connect to the Chef Client on the target node, copy across the relevant cookbooks and have Chef Client execute them
7. These cookbooks will lead to commands being executed which setup whatever was defined in the run-list - in the example from above, Postgres.

## Terminology

The above example included some important terminology:

| | |
|---|---|
| Bootstrap | The process of setting up a node for the first time to be used with Chef |
| Node | A server being managed by Chef |
| Recipes | A definition of how to get a target machine into a particular state. This is often how to install and configure a particular piece of software |
| Attributes | Values (variables) which customise the behaviour of recipes |
| Cookbooks | A grouping of related recipes |
| Roles | Re-usable groupings of run list entries and attributes |
| Run List | The list of roles and recipes which should be applied to a node |
| Converge | The process of applying the roles & recipes to a node |

## Working without a Chef server - Chef Zero

The above may sound excessive when all we want to do is setup one VPS for one Rails application. In scenarios like these we can use Chef Zero to complete the same workflow without any need for a central Chef Server.

Chef Zero replaces Chef Solo which was used in earlier editions of this book to accomplish the same thing. If you're already using Chef Solo, upgrading is fairly painless, with repository structure remaining almost identical.

Chef Zero runs an on-demand, in-memory Chef Server instance which persists its data to a local directory in the form of JSON files. We can interact with this using knife as if it was a central Chef server while in fact everything is being run from our local development machine.

This has the substantial advantage of meaning that everything we learn about how to interact with and use Chef, is applicable to both small standalone projects using Zero and larger projects using Chef Server.

This book will focus entirely on Chef Zero but the techniques describe work identically in a Chef Server environment.

## Leveraging the Community

One possibility when working with Chef is that we create our own cookbooks and recipes which define how to install and configure everything we need. In practice it is often much more more efficient to use a Cookbook which has already been created by the community and then customise its behaviour using attributes.

In the same way there is a gem for almost anything, there is a cookbook for almost everything, from installing Postgres and NGINX to configuring locales and managing third party log aggregation tools.

### Berkshelf

Berkshelf is like Bundler for these third party cookbooks. It allows us to specify a list of dependencies and their versions, and have them be downloaded automatically from a central index.

The chapter “Creaing a New Project & Berkshelf” covers the operation of this in more detail.

# Installing Tools

## Overview

This section covers installing the tools which are used throughout this book. All subsequent chapters assume that the tools covered in this chapter have already been installed.

## Installing Tools

### Install Chef Workstation

Install Chef Workstation from https://downloads.chef.io/products/workstation and ensure that running the command `chef` from a terminal does not return a command not found error.

Chef Workstation includes:

- The command line tool `chef` which is primarily used for generating new chef assets, for example repositories (projects) and cookbooks
- The cookbook dependency manager Berkshelf, used for managing both our own and third part cookbooks
- The command line interface Knife which we use for managing the interaction between Chef and the nodes we are configuring
- Chef Zero, a tool which allows us to use Chef without a separate centralised server

At time of writing the Chef Workstation `dmg` for MacOS can be downloaded with:

```
curl https://packages.chef.io/files/stable/chef-workstation/21.2.303/mac_os_x/11.\
0/chef-workstation-21.2.303-1.x86_64.dmg --output /tmp/chef-workstation.dmg
hdiutil attach /tmp/chef-workstation.dmg
```

And the `deb` for Debian based linux distributions with:

```
curl https://packages.chef.io/files/stable/chef-workstation/21.2.303/debian/10/ch\
ef-workstation_21.2.303-1_amd64.deb --output /tmp/chef-workstation.deb
sudo dpkg -i /tmp/chef-workstation.deb
```

This version of the book has been tested with version 21.2.303 of Chef Workstation.

We can check that the installation has been successful by executing the command `chef`, if it doesn't give a command not found error, then the installation succeeded.

## Installing Knife Zero

Knife is the command line utility which we'll use to:

- Tell Chef what the desired end state of our nodes is
- Tell Chef to actually make changes to a node

This is installed by default by Chef workstation so nothing else is needed here.

Knife Zero adds some extra commands to knife to make it easy to interact with Chef Zero which is what allows us to use Chef without a centralised Chef server. We'll cover these commands in more detail in "Using Knife".

Chef Workstation sets up an isolated Ruby and Gem environment for itself. This means that we cannot simply `gem install knife-zero` even though Knife, like a majority of Chef components is distributed in the form of a Ruby Gem.

Instead, the `chef` command line utility we installed as part of Chef Workstation provides the `chef gem install` command which allows us to install any gem into the Chef Workstation enviornment.

To install Knife Zero simply execute:

```
chef gem install knife-zero
```

In a folder without a `Gemfile` and `Gemfile.lock` in it. If there is a `Gemfile` in the current folder when we execute the above command, we make encounter hard to debug "bundler not found" errors.

Note that when installing the gem to the Chef Ruby environment, we may see output like:

```
WARNING:  You don't have /home/ben/.chefdk/gem/ruby/2.7.0/bin in your PATH,
          gem executables will not run.
```

which can be safely ignored.

We can test that installation of `knife-zero` has worked by executing:

```
knife zero
```

Which should result in output such as:

```
FATAL: Cannot find subcommand for: 'zero'
Available zero subcommands: (for details, knife SUB-COMMAND --help)

** ZERO COMMANDS **
knife zero apply QUERY (options)
knife zero bootstrap [SSH_USER@]FQDN (options)
knife zero chef_client QUERY (options) | It's same as converge
knife zero converge QUERY (options)
knife zero diagnose # show configuration from file
```

We now have the core tools we're going to work with installed and are ready to proceed either directly to the "Quick Start" or, if we prefer to read the instructions, to "Create a New Project".

# Quick Start

## Overview

This section provides the minimal steps we need to follow to provision a server using the example template. This template is documented in more detail in later chapters. For anyone who opens the box, has a go and then reads the instructions afterwards, this section provides the shortest path to getting a work server up and running.

This section assumes we have followed the steps from the previous chapter "Installing Tools".

Bear in mind that if working through this chapter prior to reading the rest of section one, there will be some terminology used which has yet to be introduced.

For anyone who prefers to read the rest of the book first, this section serves as a quick reference for provisioning new servers in future. ## The Stack

The default stack we'll be setting up is:

- Ubuntu 20.04 LTS
- Ruby 2.7
- Postgres 13
- Redis 5 ## Steps ### Setup a server

We begin by creating a VPS, this has been tested on Linode, Digital Ocean & Hetzner Cloud. This server should be provisioned with Ubuntu 20.04 and booted.

If we're re-using an existing IP address (e.g. re-provisioning a server) we'll need to remove existing references to the server from `known_hosts` with `ssh-keygen -R SERVER_IP_OR_HOSTNAME`.

If our public key wasn't automatically added to the host to allow passwordless ssh access, then we can add it manually with:

```
ssh-copy-id root@SERVER_IP_OR_HOSTNAME
```

If we're on OSX and get an error that the command `ssh-copy-id` doesn't exist, we can install it using homebrew; `brew install ssh-copy-id`.

Once this is complete we can test that this works by sshing into the server as root:

```
ssh root@SERVER_IP_OR_HOSTNAME
```

If everything has gone to plan, the login will complete without asking for a password. We can then use `exit` to return to the local terminal.

## Get the example code

Back in a terminal for our local machine, we can now clone the sample code with:

```
git clone git@github.com:TalkingQuickly/rails-server-template.git
```

And move into the newly created project folder with `cd rails-server-template`.

## Fetching Cookbooks

In Chef terminology, Cookbooks are the individual modules which tell chef how to install a particular piece of software. There are widely maintained cookbooks for most common pieces of open source software. We can search on Chef Supermarket[2] for open source cookbooks or, as documented later in this book, create our own cookbooks.

Berkshelf[3] is essentially Bundler for Chef Cookbooks, so rather than downloading third party cookbooks manually, we maintain a `Berksfile` (which looks very similar to a `Gemfile`) containing our Cookbook dependencies and then use the `berks` command line tool to install and update these.

An extract from our Berksfile looks like this:

```
source "https://api.berkshelf.com"

cookbook 'ntp', '~> 3.7.0'

cookbook 'openssh', '= 1.2.2'

cookbook 'postgresql', '~> 8.0.1'
...
```

This shows the similarity to working with Bundler. We have cookbooks for system components such as `memcached` & `postgresql`. Also like bundler, we can define constraints as to which versions should be installed. When we install cookbooks, a `Berksfile.lock` is created which captures the entire dependancy tree so we can be certain we always end up with the same versions.

In the root of the sample code directory, we now execute:

```
berks vendor
```

Which downloads the relevant cookbooks to the folder `berks-cookbooks`.

In the tools chapter we briefly looked at `knife` which is the command line tool that we'll use for interacting with chef. We configure knife using the file `knife.rb` in the root of the sample code which looks like this:

[2]https://supermarket.chef.io
[3]https://github.com/berkshelf/berkshelf

```
local_mode true
chef_repo_path   File.expand_path('../' , __FILE__)

knife[:ssh_attribute] = "knife_zero.host"
knife[:use_sudo] = true
knife[:editor] = 'vim'
cookbook_path ["cookbooks", "berks-cookbooks"]
knife[:before_bootstrap] = "berks vendor"
knife[:before_converge]  = "berks vendor"
```

You can see here that we specify `berks-cookbooks` as a location that `knife` should look for cookbooks. We also configure `before_bootstrap` and `before_converge` commands. Bootstrapping is the process of getting a server (node) ready to be configured with chef, converging is the process of applying configuration from our local repository to a server (node). Before both of these actions, `knife` will automatically run `berks vendor` to ensure that any changes in our `Berksfile` and `Berksfile.lock` are applied.

We also specify which editor to use when editing Chef definitions. This can be changed to any editor available in the local path, so could be set to `code` for VSCode or `subl` if the sublime terminal helper is installed.

## Prepare the node for Chef

In a terminal in the same folder, add the node to Chef:

```
knife zero bootstrap SERVER_IP --connection-user root --node-name NODE_NAME
```

Replacing `NODE_NAME` with a descriptive name of the node. Notice the `zero` keyword which indicates we want to do this locally. In practice this means a Chef server instance is running locally and the data is being persisted to the current directory.

This will connect to the remote server, install Chef and generate the local file `nodes/NODE_-NAME.json`. This is the file where all details about the node and what should be installed on it will be stored.

If we've never connected to the node via SSH before, we may be asked to confirm the servers fingerprint by entering `Y` and pressing enter.

We'll see output such as:

```
Bootstrapping 65.21.54.211                                                    \

 [65.21.54.211] -----> Installing Chef Omnibus (stable/16)                    \

downloading https://omnitruck.chef.io/chef/install.sh                         \

  to file /tmp/install.sh.20950/install.sh                                    \

 [65.21.54.211] trying wget...                                                \

 [65.21.54.211] ubuntu 20.04 x86_64
```

Which is the Chef client being installed on the remote server. Finally we'll see something similar to:

```
[65.21.54.211] Starting Chef Infra Client, version 16.10.17
Patents: https://www.chef.io/patents
 [65.21.54.211] Creating a new client identity for test1 using the validator key.
 [65.21.54.211] resolving cookbooks for run list: []
 [65.21.54.211] Synchronizing Cookbooks:
 [65.21.54.211] Installing Cookbook Gems:
 [65.21.54.211] Compiling Cookbooks...
 [65.21.54.211] [2021-03-02T03:59:48+01:00] WARN: Node test1 has an empty run lis\
t.
 [65.21.54.211] Converging 0 resources
 [65.21.54.211]
Running handlers:
 [65.21.54.211]
 [65.21.54.211] Running handlers complete
 [65.21.54.211] Chef Infra Client finished, 0/0 resources updated in 10 seconds
```

Which is telling us that the "run list" - the list of configuration changes to apply to the node - is empty.

At this point the node has been configured for use with Chef and a JSON definition for the node has been created in `nodes/NODE_NAME.json`. We're now ready to update our node definition with details of the configuration which should be applied to the node.

## Choose roles to apply to the node

Simplistically a recipe is generally a Chef definition of how to install and configure a specific piece of software and a role is a way of grouping together recipes. So for example a Postgres role might include the cookbooks to install Postgres along with cookbooks for related monitoring and maintenance tools.

Roles are stored as json files in the `roles` directory of the sample code. So for example the `server` role can be found in `roles/server.json`.

In this quickstart we're going to add the following roles:

- **server**: Basic functionality common to most servers including firewall, clock synchronisation, unattended upgrades and fail2ban
- **nginx-server**: Installs the NGINX web server along with appropriate firewall rules and monitoring
- **postgres-server**: Installs the Postgres database server along with appropriate monitoring
- **rails-app**: Installs rbenv for managing ruby versions as well as dependencies often required for gem installation
- **redis-server**: Installs Redis, commonly used as both a cache store and a background job data store in Rails applications

We will use the `knife node run_list add` command to add recipes and roles which should be installed on the node.

To add the required roles for this server enter the following:

```
knife node run_list add NODE_NAME 'role[server],role[nginx-server],role[postgres-\
server],role[rails-app],role[redis-server]'
```

Replacing `NODE_NAME` with the node name used when we ran `knife zero bootstrap`.

We can check that the roles have been added using `knife node show NODE_NAME` which will return something like the following:

```
Environment: _default
FQDN:        rails-server-template-test
IP:          65.21.54.211
Run List:    role[server], role[nginx-server], role[postgres-server], role[rails-\
app], role[redis-server]
Roles:
Recipes:
Platform:    ubuntu 20.04
Tags:
```

You can also see that the roles have been added to the run list by viewing the JSON directly with `knife node edit NODE_NAME`:

```
{
  "name": "test1",
  "chef_environment": "_default",
  "normal": {
    "knife_zero": {
      "host": "65.21.54.21"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "role[server]",
    "role[nginx-server]",
    "role[postgres-server]",
    "role[rails-app]",
    "role[redis-server]"
  ]
}
```

See the node in the "Fetching cookbooks" section on modifying `knife.rb` to configure the editor which is used for such commands.

@TODO —— @TODO

## Configuring The Node

Now that we've added roles to the node, we can supply configuration parameters for those roles. Typical configuration parameters are things like passwords and ruby versions.

We'll begin by setting the master Postgres password. This is the password we'll need to login as the `postgres` super user to perform operations such as creating new databases and additional users.

We now use what will become one of our most accessed commands when interacting with nodes; `knife node edit`, this command, followed by the name of a node which has been configured with `knife node bootstrap` will open the user editable part of the nodes JSON in the editor defined in `knife.rb`.

So we can run `knife node edit NODE_NAME` and add the following snippet inside the `normal` object:

```
"postgresql" : {
  "password" : {
    "postgres" : "s0mePassw0rd"
  }
},
```

There's more on the significance of the `normal` key in the section on attribute precedence[4] in the Chef documentation. But for the purposes of simple deployments, this isn't something we'll have to worry about too much.

We then follow a similar process to set the Ruby version(s) to be installed. We add the following under the `normal` key, replacing `2.7.2` with our desired Ruby version:

[4]https://docs.chef.io/attribute_precedence/

```
"rbenv":{
  "rubies": [
    "2.7.2"
  ],
  "global" : "2.7.2",
  "gems": {
    "2.7.2" : [
      {"name":"bundler"}
    ]
  }
},
```

So the final node definition might look something like this:

```
{
  "name": "test1",
  "chef_environment": "_default",
  "normal": {
    "postgresql": {
      "password": {
        "postgres": "s0mePassw0rd"
      }
    },
    "rbenv": {
      "rubies": [
        "2.7.2"
      ],
      "global": "2.7.2",
      "gems": {
        "2.7.2": [
          {
            "name": "bundler"
          }
        ]
      }
    },
    "knife_zero": {
      "host": "188.166.151.185"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "role[server]",
    "role[nginx-server]",
    "role[postgres-server]",
    "role[rails-app]",
    "role[redis-server]"
  ]
}
```

Finally we save and close the file, so that Chef can update our node definition based on our changes. We'll see output along the lines of:

```
Saving updated normal on node test1
```

Note that at this point, all we've done is update the JSON file in `nodes/NODE_NAME`, no changes have been applied to the server.

## Setup users

Before we apply our "run list" (the list of roles we want applied to the server), we want to add a final piece of configuration; the creation of users. Rather than using the root user for interacting with the server when deploying, we'll create a `deploy` user who has the option to use `sudo` if they need to execute commands as root.

The `users` cookbook handles creation of users. In this example we'll create a single user called `deploy`.

We begin by creating a data bag called `users` with a single entry; `deploy`. A data bag[5] is a way of storing global variables as JSON data which can then be indexed and searched by cookbooks. Databags are stored in the `databags` directory and can be created manually but we will generally create data bags using `knife`.

To create our users data bag we execute:

```
knife data_bag create users deploy
```

And then in a separate terminal, generate an encrypted password for the user with:

```
openssl passwd -1 "plaintextpassword"
```

We can then update the contents of databag to match the following, replacing `ENCRYPTED_-PASSWORD` with the encrypted user password generated above and `YOUR_PUBLIC_KEY` with our public key (usually the contents of `~/.ssh/id_rsa.pub`):

[5]https://docs.chef.io/data_bags/

```
{
  "id": "deploy",
  "password": "ENCRYPTED_PASSWORD",
  "ssh_keys": [
    "YOUR_PUBLIC_KEY"
  ],
  "groups": [
    "sysadmin"
  ],
  "shell": "/bin/bash"
}
```

We then save and close the file to allow knife to update the data bag.

Groups specifies the groups this user will be a member of. The `sudo` cookbook we use makes `sudo` available to all users in the `sysadmin` group.

To subsequently edit the contents of the databag use `knife data_bag edit users deploy` and edit the key value pairs under `raw_data`, there's more on interacting with databags using knife here[6].

Note that by default, the sample code excludes data bag contents from the repository with a `.gitignore`

## Apply configuration to the node

So far, only our local definition of the node has been updated, no changes have been made to the actual server. To apply these changes, we use `knife zero converge`:

```
knife zero converge "name:NODE_NAME" --ssh-user root
```

Replacing `NODE_NAME` with the node name used when running `knife zero bootstrap`.

This will take a while, especially compiling Ruby. At the end of it, the node will be configured and ready for use as a Rails application host.

You sould be able to login via ssh with `ssh deploy@SERVER_IP`.

The aim is to now use Chef to make all changes to our node, after updating the node definition, we use:

```
knife zero converge "name:NODE_NAME" --ssh-user root
```

To apply the new changes. This makes re-provisioning the server in future, or provisioning additional similar nodes, completely painless.

If the node needs to be re-provisioned from scratch, then we can use the following command to install Chef on the node without over-writing the node definition:

[6]https://docs.chef.io/knife_data_bag.html

```
knife zero bootstrap NODE_IP --node-name NODE_NAME --connection-user root --no-co\
nverge
```

## Next Steps

The node is now ready to deploy a Rails application to. You can jump straight to the "Capistrano Quickstart" chapter if you're in a hurry to get setup, or work through the rest of this section to get a better feel for exactly what's happening behind the scenes.

# Creating a new project & Berkshelf

This chapter assumes you haven't dived right into the quick start and want to get a feel for how Chef works. It also assumes that the steps in the "Installing Tools" have been completed and so Chef Workstation and Knife Zero are installed and working.

## Generating a Chef Repo

A chef repository is a collection of artifacts, in particular cookbooks, roles, recipes and node definitions which are used to define the end state we want our infrastructure to be in.

If we were working with a remote Chef Server, we would have a local repository and then use knife to upload from this local repository to the remote chef management server. When - as in this book - we're not using a central management server, we instead interact directly with the local repository and use Chef Zero[7] to provide a lightweight local Chef Server. In practice, this all happens automatically due to the combination of `local_mode true` in our `knife.rb` file and `knife zero`.

To generate a new chef repository, execute `chef generate repo NAME`

This will generate a directory structure such as the following:

**Chef Repository Structure**

```
├── chefignore
├── cookbooks
│   ├── example
│   │   ├── attributes
│   │   │   └── default.rb
│   │   ├── metadata.rb
│   │   ├── README.md
│   │   └── recipes
│   │       └── default.rb
│   └── README.md
├── data_bags
│   ├── example
│   │   └── example_item.json
│   └── README.md
├── LICENSE
├── policyfiles
│   └── README.md
└── README.md
```

[7] https://github.com/chef/chef-zero

At this stage all we've really done is created a best practice folder structure for organising a Chef project. We need to configure Knife before we can interact with it in any way.

This repository will contain the entire blueprint for your infrastructure so should be added to version control and treated in the same way any business critical code in a version control is.

## Berkshelf

### Purpose of Berkshelf

A cookbook is a Chef artifact which encapsulates the functionality needed to install and configure a specific piece of software. So, for example, a Postgres cookbook might contain everything that is needed to install and configure the Postgres datbase client and server. Well written cookbooks will provide flexible configuration options, allowing us to customise the installation to suit our needs.

Wherever possible, we'll use community maintained cookbooks to install popular software. This is analogous to using libraries in Rails. If we want to add login and signup to our application, it's generally better to drop in something like Devise - which has the benefit of thousands of other users and contributors - than writing our own system from scratch. Likewise when using Chef, if we want to install a popular piece of software like Postgres, Nginx or Rbenv, we'll almost always find that there is a popular community cookbook that has the benefit of being used by thousands of other deployments and achieve better results by using this than creating our own from scratch.

Sometimes however we'll find either that there is no community cookbook or that our requirements are sufficiently bespoke that we need to create our own cookbook, either from scratch or by using the wrapper cookbook approach (covered in chapters 8 and 9 respectively).

Over time we'll build up multiple different Chef repositories relating to different projects. If we have written some custom cookbooks, we'll often be able to re-use them across projects, further increasing the overall time saved. This principal "Don't Repeat Yourself" (DRY) will be familiar to any Ruby developer used to extracting re-usable functionality into standalone gems or using such functionality provided by gems others have created.

Ruby has bundler which allows us to define the gems and the versions of these gems which we need in a Gemfile. A quick `bundle install` will then take care of pulling in these gems from their remote sources and making them available to our Ruby application.

Bundler then generates a Gemfile.lock which contains details of all the gems - including dependencies - required by our project and their exact versions. When our colleagues run `bundle install` as long as they have this Gemfile.lock file present, they will get the same versions of all gems as we did.

Berkshelf provides exactly this functionality for Chef Cookbooks. In a Berksfile we can define the cookbooks our chef repository is dependent on and the versions of these. We can then use `berks install` to grab all of these cookbooks. If any individual cookbook specifies additional dependencies using `depends` in its `metadata.rb`, Berkshelf will take care of installing these as well.

Like bundler, Berkshelf generates a .lock file - Berksfile.lock - with the relevant versions for each cookbook. As long as we share this with our colleagues (for example by checking it into our version control system), they will get the same versions that we have when they run `berks install`.

If you want to update a cookbook you can use `berks update cookbook_name` and Berkshelf will attempt to update the latest version of that cookbook and resolve any new/ existing dependencies appropriately.

### Berksfile.lock

When troubleshooting a provisioning failure, a lot of forums will contain advice along the lines of "try just deleting your Berksfile.lock". This is generally a really bad idea (as would be deleting Gemfile.lock to fix a Rails app). It's far safer - and more likely to solve the problem - to work through the cookbooks in the Berksfile, updating them one by one, so that any new problems introduced by breaking changes between versions can be managed.

## Structure of a Berksfile

A typical Berksfile looks like this:

**Berksfile**

```
source "https://api.berkshelf.com"

cookbook 'nginx', '~> 0.101.5'
cookbook 'apt', github: 'opscode-cookbooks/apt'
cookbook 'demo-commands', path: '/Users/demo/code/chef-cookbooks/demo-commands'
```

The first line `source "https://api.berkshelf.com"` is the default source for cookbooks, in the same way Ruby Gems provides a central package index for gems, Opscode provides a similar index for packages.

The simplest definition would be:

```
cookbook 'cookbook_name'
```

This would tell Berkshelf to look for the cookbook `cookbook_name` in the opscode index and download it and any dependencies.

We can also use bundler style version definitions so:

```
cookbook 'cookbook_name', '~> x.x.x'
```

If we're working with a cookbook we have stored locally - for example one we are actively developing, we can use the following syntax:

```
cookbook 'demo-commands', path: '/Users/demo/code/chef-cookbooks/demo-commands'
```

Or to reference a repository stored in a git repository:

```
cookbook 'cookbook_name', git: 'https://github.com/user/repo'
```

You can also provide an SHA-1 commit hash to ensure a particular revision of the cookbook is used like so:

```
cookbook “cookbook_name”, git: 'https://github.com/user/repo', ref: “eef7e65806e7\
ff3bdbe148e27c447ef4a8bc3881”
```

If you’re working with public Github repositories, you can use the following syntax:

```
        cookbook "keeping_secrets", github: "RiotGames/keeping_secrets-cookbook"
```

Berkshelf like bundler is extremely powerful and it’s worth spending some time reading the documentation available at: http://berkshelf.com/

## Adding a Berksfile to our repository

We’ll use a single Berksfile in the root of our repository, create this file and add the following line at the top:

**Berksfile**

```
source "https://api.berkshelf.com"
```

## Standalone Usage

With our single Berksfile in the root of the repository, we could simply run `berks install`, sit back and wait.

This would begin resolve our dependencies and store the dependency tree in a `Berksfile.lock` file.

It would also fetch the required cookbooks, storing in `~/.berkshelf` in subdirectories following the `{name}-{version}` convention we’re used to from RubyGems.

## Berkshelf and Chef Zero

When working with Chef Zero, we’ll take the additional step of automatically vendoring the cookbooks into the `berks-cookbooks` folder of our local repository whenever we perform an update to a remote node. This is described in more detail in the “Knife Configuration File” section below as well as the next chapter; “Using Knife”.

### Where to store cookbooks

In the past, it was very common to store all cookbooks for a repository either in `cookbooks` or another folder called `site-cookbooks`. The entire repository was then kept in version control.

Tools such as Berkshelf have changed this and it is now considered practice to have an individual git repository for every cookbook and then reference these from the Berksfile. This gives a very similar structure to what we've come to expect in a Ruby application.

If we're developing a cookbook locally, we should have a separate folder for our cookbook, outside of our Chef Repo structure. We can then include this folder in our Berksfile using a `path:` reference while we are actively developing it and then change this to a git reference as it matures.

## Configuring Knife

### Knife Configuration File

Add a knife configuration file to the root of the newly created directory `knife.rb` with the following contents:

```
local_mode true
chef_repo_path   File.expand_path('../' , __FILE__)
cookbook_path ["cookbooks", "berks-cookbooks", "site-cookbooks"]

knife[:ssh_attribute] = "knife_zero.host"
knife[:use_sudo] = true
knife[:editor] = 'vim'
knife[:before_bootstrap] = "rm -rf ./berks-cookboks/* && berks vendor"
knife[:before_converge]  = "rm -rf ./berks-cookboks/* && berks vendor"
```

The line `local_mode true` configures knife - the tool we use for interacting with Chef - to run in local mode. This means that instead of looking for a remote Chef server, knife will use a local, in-memory Chef Zero instance which will persist node data to the current directory.

We'll see more about what this configuration file does in the next chapter "Using Knife".

### The `berks-cookbooks` directory and .gitignore

In the above knife configuration file, the following two lines:

```
knife[:before_bootstrap] = "rm -rf ./berks-cookboks/* && berks vendor"
knife[:before_converge]  = "rm -rf ./berks-cookboks/* && berks vendor"
```

Configure knife to automatically run `berks vendor` before interacting with a node. This creates a copy of all the current cookbooks and places them in `berks-cookbooks` so that they are available to Chef Zero.

It is entirely a matter of personal preference whether this directory should be included in version control or not.

My personal preference is to include it to account for situations where a third party source is no longer available.

## Recap / Quick Reference

When creating a new Chef Project:

1. Generate the repository with `chef generate repo NAME`
2. This heirachy should be kept in version control
3. Add a `Berksfile` to the root of the generated Chef Repository
4. Add a knife configuration in a file called `knife.rb` to the root of the generated repository

## Up Next

Now that we've created a new project, we'll move onto using knife to interact with the repository and our remote server(s).

# Using Knife

## Overview

In this chapter we'll look at how knife is used to interact with Chef Zero and our repository.

We'll use these commands extensively in "Creating a Chef Cookbook" so the purpose of this chapter is to provide the background as to what they're doing as well as to function as a useful quick reference in future.

This chapter assumes that commands are being executed with a Chef Repository setup as per the chapter "Creating a New Project". In particular it expects a `knife.rb` configuration file setup for local mode.

## SSH Access

A node is a single remote server, forming part of our infrastructure, which we wish to configure.

We'll need SSH access to every node which we wish to manage. To avoid having to continually enter passwords, we'll setup public key authentication.

Begin by creating a VPS, this book has been tested on Hetzner Cloud, Linode & Digital Ocean. Provision the new server with Ubuntu 20.04 and boot it up.

When re-using an existing IP address (e.g. re-provisioning a server) we'll need to remove existing references to the server from `known_hosts` with `ssh-keygen -R SERVER_IP_OR_HOSTNAME`.

We can now copy our SSH public key across to allow passwordless access:

```
ssh-copy-id root@SERVER_IP_OR_HOSTNAME
```

On OSX if an error that the command `ssh-copy-id` doesn't exist is encountered, we can install it using homebrew; `brew install ssh-copy-id`.

Once this is complete we can test that this works by sshing into the server as root:

```
ssh root@SERVERI_IP_OR_HOSTNAME
```

If everything has gone to plan, the login will complete without asking for a remote password. We can then use `exit` to return to the local terminal.

## Setting up a node for Chef

Every node which we wish to manage with Chef, must have the Chef Client installed on it. It is this client which actually executes the commands on the node.

Once setup, Chef Server will store a representation of the current and desired state of this node. When working with Chef Zero this will take the form of a JSON file in the `nodes` folder.

The process of setting up a node is known as bootstrapping, this is initiated with:

```
knife zero bootstrap SERVER_IP_OR_HOSTNAME --ssh-user root --node-name NODE_NAME
```

This will:

- Connect to the node as the user specified by `--ssh-user`
- Install Chef Client on that node
- Generate a node definition file in `nodes/NODE_NAME.json`

Behind the scenes before any of this happened:

- A local Chef Zero instance was started and configured to persist data to the current directory

If we open the node definition directly we'll see that Chef Client has collected and stored a lot of metadata about the node, including every package which is currently installed on it.

We should not edit this file directly, editing using `knife node edit` is covered in the section below.

## Editing data in Chef & Setting Our Editor

When interacting with Chef resources, we do not generally edit the JSON files directly, instead we use `knife` to initialize an edit.

The primary benefit to this is that knife will validate the files after saving and only allow them to be saved if they pass validation.

The editor which is used for this is set in our `knife.rb` configuration file at this line:

```
knife[:editor] = '/usr/local/bin/vim'
```

This can be set to any command which will open an editor when called in the form:

```
CMD path_to_file
```

For example if you're a Sublime Text user you could set this to `/usr/local/bin/sublime` to have Sublime Text used as the default editor.

```
knife[:editor] = '/usr/local/bin/sublime'
```

Or for VSCode users:

```
knife[:editor] = '/usr/local/bin/code'
```

## Editing a node definition

A node definition is the primary way in which we define what should be installed on a node and how it should be configured. To edit a node definition use:

```
knife node edit NODE_NAME
```

This wil open, in our chosen editor, a JSON file which looks something like this:

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "knife_zero": {
      "host": "188.166.151.185"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [

  ]
}
```

Note that the file we're editing will have a path along the lines of: `/private/var/folders/v3/dnv4tkq918x81j02n` instead of the actual node definition file.

It will also contain significantly fewer lines than we saw when we inspected the node definition file directly. This is because Chef has presented us only with parts of the file we might need to edit, rather than including parts which just related to Chefs internal representation of a nodes state.

When we save and close the file, we'll see some output from Chef. If we haven't made changes we'll see something like:

```
Node not updated, skipping node save
```

If we make a change and the JSON validates, we'll see output confirming that the file has now been saved:

```
Saving updated normal on node rdrtest1
```

And if the saved JSON contains an error we'll see details of that:

```
ERROR: Chef::Exceptions::JSON::ParseError: parse error: after key and value, insi\
de map, I expect ',' or '}'
          ey_with_error": "value"     "knife_zero": {       "host": "1
                     (right here) ------^
```

In this case the error was a trailing comma.

## Editing attributes

The most common thing we'll want to edit about a node is its associated "attributes". We'll cover attributes in more detail in "Creating a Chef Cookbook" but at a high level, attributes are variables associated with a node which allow us to customise the behaviour of a cookbook.

If, for example, we have a Redis cookbook, it might define an attribute `port` which allows us to specify which port Redis should listen on.

If this cookbook used `redis` as its namespace and the port attribute was called `listen_port` then we might use `knife node edit NODE_NAME` to update the JSON as follows:

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "redis: {
                      "port" : 3561
    },
    "knife_zero": {
      "host": "188.166.151.185"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [

  ]
}
```

To set the port attribute to 3561. Note that the change has been made as an entry in the `normal` object. Here `normal` refers to the precedence assigned to the attribute being defined. This is examined in more detail in the chapter on nodes roles and attributes but as a rule of thumb, assume that attributes should be set within the normal section.

## Adding to the run list

The run list defines the list of recipes which should be applied to the node. The run list can also contain roles, which are essentially just re-usable groupings of recipes and attributes.

There are two primary ways of editing the run list. The first is to use `knife zero edit NODE` as before an edit the contents of the `run_list` array manually, for example:

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "knife_zero": {
      "host": "188.166.151.185"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "recipe[redis::server]",
    "role[generic_rails_application]"
  ]
}
```

Shows what the run list would look like if we were adding the `server` recipe from the `redis` cookbook and the role `generic_rails_application`.

Note that if we'd just included:

```
recipe[redis]
```

It would have included the default recipe from the redis cookbook, equivalent to entering `recipe[redis::default]`.

The other way to add or remove entries from the run list is using `knife node run_list add NODE_NAME 'ITEM(S)'`

for example:

```
knife node run_list add NODE_NAME 'recipe[redis::server],role[generic_rails_appli\
cation]'
```

Would perform exactly the same function as the corresponding edit to the node above.

We can use `knife node run_list remove NODE_NAME 'ITEM(S)'` in exactly the same mannger to remove them.

Note the single quotes around the list of recipes and roles.

## Creating and editing roles

A role is a Chef primitive which allows us to re-use a combination of run list entries and attributes.

If, for example, we had an internal standard for all of our servers that we always installed a ufw recipe (firewall) and an ssh recipe, we could create a role to do this.

We could create such a role by executing the following:

```
knife role create base_server
```

Which would open our editor of choice with some default role JSON along the lines of:

```
{
  "name": "base_server",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {

  },
  "override_attributes": {

  },
  "chef_type": "role",
  "run_list": [

  ],
  "env_run_lists": {

  }
}
```

We could amend this to:

```
{
  "name": "base_server",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {
                "openssh" : {
      "server" : {
        "password_authentication" : "no",
        "challenge_response_authentication" : "no",
        "permit_empty_passwords" : "no",
        "use_pam" : "no",
        "x11_forwarding" : "no",
        "permit_root_login" : "yes"
      }
    }
  },
  "override_attributes": {

  },
  "chef_type": "role",
  "run_list": [
                "recipe[ufw::default]",
                "recipe[openssh::default]"
  ],
  "env_run_lists": {

  }
}
```

We'll look further at what the above role would actually do in the security chapter. What's important here is that by creating this as a role, we can simple add to the run list of any node:

```
role[base_server]
```

And get the same result as if we'd added the individual attributes and run list entries to the nodes run list.

## Applying configuration to a node (converging)

So far we've used knife to interact with Chef Zero - the local in memory Chef Server instance - and to allow us to manipulate the configuration in the repository. Apart from `knife zero bootstrap` we haven't looked at any commands which will actually impact the state of the remote server.

We'll primarily rely on a single command to tell Chef that we want to take the changes we've made to our cookbooks, nodes and role definitions and apply that configuration to a sever. This command is:

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

This command will have the local Chef Zero instance connect to Chef Client on the specified node, upload the relevant cookbooks and apply the nodes definition. This means everything we've specified in the node definition, will be installed and configured as per the run list and attributes we've set.

One subtlety here is that the following line in `knife.rb`:

```
knife[:before_converge]  = "berks vendor"
```

Means that before converge is run, `berks vendor` will be run and create copies of the cookbookes in the directory `berks-cookbooks`. These should never be edited and no custom cookbooks should ever be kept in this folder.

Whenever we make changes in our repository which we want to apply to a node, we'll use `knife zero converge`.

## Working with data bags

Data bags are another Chef Primitive which allow us to store data which is then used by recipes. Generally data bags are used to store information which is

- Global: For example API keys and public keys
- Sensitive: Data bags can be encrypted, and individual items in data bags can be encrypted with separate keys. This is useful for sensitive data such as SSL certificates.

Databags are represented internally as directories within the `data_bags` folder. Each sub-item within a databag is represented as a single JSON file within this parent directory.

### Normal data bags

If we wanted to create a data bag called `users` which contained an entry `deploy`, we would use the following command:

```
knife data bag create users deploy
```

This would open our editor with a simple JSON object like so:

```
{
  "id": "deploy"
}
```

We could then add keys to the top level of the object, for example:

```
{
  "id": "deploy",
  "shell": "/bin/bash"
}
```

We'll see this in more detail in the "Security" chapter where we'll use a data bag to manage the users that are created on our server.

Once a databag has been created, an entry can be edited with:

```
knife data bag edit users deploy
```

## Encrypted data bags

Encrypted data bags behave in exactly the same way as regular databags except their contents is encrypted.

To create an encrypted entry in a data bag use:

```
knife data bag create secrets some_secrets --secret SOME_PASSWORD
```

And to edit one:

```
knife data bag edit secrets some_secrets --secret SOME_PASSWORD
```

Note that when using this approach, your secret may appear in your shells history, this should either be managed by limiting what appears in your shells history or by using `--secret-file` which allows you to specify the secret in a file.

# Other useful commands

`knife search node` will return all nodes

`knife search node "name:some_string"` will return a list of nodes with `some_string` in the name. `knife search` is hugely powerful and it's worth spending some time reading the full documenation[8].

`knife ssh 'name:some_title' --ssh-user root 'hostname'` will run the command `hostname` on any servers which match the search filter

[8]https://docs.chef.io/knife_search.html

## Quick Reference

To setup a node to work with Chef:

- If you're re-using an existing node, remove the servers IP from known_hosts `ssh-keygen -R SERVER_IP_OR_HOSTNAME`
- Copy your key across to the server: `ssh-copy-id root@SERVER_IP_OR_HOSTNAME`
- Bootstrap the node: `knife zero bootstrap SERVER_IP_OR_HOSTNAME --ssh-user root --node-name NODE_NAME`
- Edit the node definition using `knife node edit NODE_NAME`
- Apply changes to the node `knife zero converge 'name:NODE_NAME' --ssh-user root`

## Up Next

Now that we understand the basics of using `knife` to manage the lifecycle of defining our desired node state locally and then applying this to our infrastructure, we'll move onto creating a basic Chef cookbook from scratch.

# Creating a Chef Cookbook

## Overview

In this chapter we'll begin with the list of manual commands which would be used to install the latest version of Redis on an Ubuntu server. We'll then convert those into a very simple Chef cookbook and apply this to our server.

After that we'll wipe the server and iterate on our cookbook to take advantage of more of the features which make Chef so powerful.

## Prerequisites

This chapter assumes:

- You have a repository setup as per the chapter "Creating a New Project"
- You have a VPS configured with SSH access and have bootstrapped it as per the instructions in the previous chapter "Using Knife"

## Generating a Cookbook

In the chapter on creating a new project we explored the concept that we generally don't keep cookbooks within our Chef repository.

Instead we keep them in their own directory, under separate version control, and reference them from our Berksfile. This is to maximise re-usability across projects.

Personally I have a separate directory within my standard `code` directory called `chef-cookbooks` however these can be placed anywhere. The important thing is that they **should not be within the Chef repository**.

Within the folder we're going to store our cookbooks, we can then execute the following command:

**Terminal**

```
chef generate cookbook redis_server
```

Which will create the basic file and directory structure needed for a cookbook called `redis_server`.

### Underscores in cookbook names

Note that some earlier editions of this book used the name `redis-server`, with hypens instead of underscores. Using hypens in cookbook names is no longer recommend due to some problems they can cause when working with custom resources.

## Adding the cookbook to Berkshelf

Now within your Chef repository, add the following line to your `Berksfile`, replacing `/PATH/TO/COOKBOOK` with the absolute path to your cookbook directory. `cookbook 'redis_server', path: '/PATH/TO/COOKBOOK'`

You can check this path by cding into the folder and executing `pwd`.

## Structure of a Cookbook

The initial file and directory structure looks like this:

**redis-server dir listing**

```
├── Berksfile
├── README.md
├── chefignore
├── metadata.rb
├── recipes
│   └── default.rb
├── spec
│   ├── spec_helper.rb
│   └── unit
│       └── recipes
│           └── default_spec.rb
└── test
    └── integration
        ├── default
        │   └── serverspec
        │       └── default_spec.rb
        └── helpers
            └── serverspec
                └── spec_helper.rb
```

Some of these files and folders we won't need for our simple cookbook so won't be covered below. For more information on these entries, see https://docs.getchef.com/essentials_cookbooks.html

We'll also add the following directories:

- attributes
- templates

**Terminal**

```
mkdir attributes templates
```

## metadata.rb

This file contains details about the purpose of the cookbook as well as any dependencies and the version.

The initial `metadata.rb` looks like this:

**redis-server/metadata.rb**

```
name             'redis-server'
maintainer       'YOUR_COMPANY_NAME'
maintainer_email 'YOUR_EMAIL'
license          'All rights reserved'
description      'Installs/Configures redis-server'
long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
version          '0.1.0'
```

The name is important because it is how we'll refer to the cookbook when we need to include it in the `run_list` of a node or role definition. The version is also important because if we later start managing this Cookbook with Berkshelf, we'll use this version number to ensure that anyone using our configuration, uses the same version.

## The `recipes/` directory

This directory contains individual recipes. Recall that a Chef Cookbook is made up of one or more recipes. All cookbooks should have a default recipe defined in `recipes/default.rb` but may also declare others. For example we could have one recipe for installing Redis from a ppa and one for compiling it from source.

In this example we will work only with the `default.rb` recipe.

## The `templates/` directory

When we want to create a file on the remote server, we'll create `erb` files within a subfolder of this directory which Chef will then convert into a remote file. The advantage of using `erb` here is that it means we can dynamically generate and interpolate values based on attributes.

The organisation of the subfolders within the `templates` directory can be confusing. In the cookbook we've generated there is a single subfolder, `default`. It would be logical to assume that this relates to the `default.rb` recipe we saw above however this is not the case.

When organising templates, the `default` folder refers to the platform, for example Ubuntu, Debian, CentOS etc. We'll see what this means in more detail later in this section when we deal with dynamically creating configuration files.

### The `attributes/` directory

Attributes are values most commonly set in a node or role definition which allow us to customise the behavior of a recipe. Common uses would include choosing the version of a package to be installed and customising configuration file values.

In order to set default values for these attributes, we can create a file `default.rb` within the attributes folder of a cookbook. If we do not specify a value for an attribute in a node or role definition, the values from this file will be used.

We'll cover the setting of default attributes later in the "Customising Cookbooks with Attributes" section.

## Installing Redis Manually

The aim of using Chef is to replace the process of manually typing commands into a server with an automated process. It therefore makes sense to start by looking at the sort of commands we would type in manually.

The steps we would normally need to take to install an up to date version of Redis would be something like:

- Install `python-software-properties` to make adding PPA's easier
- Add a PPA which provides packages of the latest Redis version
- Install Redis
- Add a custom configuration
- Restart Redis to load our custom configuration

In order to do this, we would enter commands such as the following:

**Terminal - Remote Server**

```
sudo apt-get install -y python-software-properties
sudo add-apt-repository -y ppa:chris-lea/redis-server
sudo apt-get update
sudo apt-get install -y redis-server
sudo rm /etc/redis/redis.conf
cat <<EOF > /etc/redis/redis.conf
daemonize yes
pidfile /var/run/redis/redis-server.pid
logfile /var/log/redis/redis-server.log

port 6379
  bind 127.0.0.1
timeout 300

loglevel notice
```

```

databases 16

save 900 1
save 300 10
save 60 10000

rdbcompression yes
dbfilename dump.rdb

dir /etc/redis/
appendonly no

maxmemory 419430400
maxmemory-policy allkeys-lru

maxmemory-samples 10
EOF
sudo chown -R redis:redis /etc/redis/
sudo /etc/init.d/redis-server restart
```

The `-y` option when passed to `apt-get` commands automatically answers yes to any prompts so allows these commands to run without user intervention when used as part of a script.

In practice we might well edit the file `/etc/redis/redis.conf` with vim or similar but we'll remove it then add a new one using cat here because it serves as a better example for how to convert it into a really simple Chef cookbook.

This approach of simply entering commands manually or executing commands manually is not without its merits, in particular:

- It's just bash! Most Ruby developers have some level of familiarity with Bash and the terminal. This means the learning curve for setting up the server is minimal
- Many of the tutorials floating around the internet will explain setting up a particular piece of server side software in terms of "copy and paste" these commands

## Converting this into a simple Chef Cookbook

### Why Convert it

One of the main disadvantages of the above approach is that it tends to lack repeatability, it's easy to forget to document one of the commands which must be entered or to subsequently ssh in and execute an additional command which is not part of the script.

One of the main benefits of converting this into a Chef Cookbook is that it completely removes the process of ssh'ing into our server and forces us to create an audit trail of what we're doing to the server.

In order to execute commands on the remote server, we have to define them as part of one of our cookbooks. This builds re-usability and repeatability into our process from the start.

It's worth noting at this point though that this repeatability is only guarenteed if we apply it to a clean server every time. If we write a cookbook, apply it to the remote server then change it and apply it again, it's impossible to be sure whether the new version of the cookbook works or if the new version, combined with the commands which were previously executed, works.

## The simplest possible cookbook

If the goal of creating this Chef Cookbook is simply to remove the step of ssh'ing into the remote server then we can fulfill this simply by writing a Chef cookbook which executes bash commands. We'll begin by creating a cookbook that does exactly this, then move onto enhancing it to take advantage of some of the features which make Chef far more powerful than simple bash scripting.

Since our Cookbook is only going to have a single recipe, we edit the `default.rb` recipe directly.

Recipes are written in Ruby using a DSL for executing commands on remote servers. One of the simplest components of this DSL is the `bash` method which allows us to pass in a block specifying bash commands and the user to run them as.

To create a default recipe which executes the commands to install Redis which we outlined earlier, enter the following in `redis-server/recipes/default.rb`

**redis-server/recipes/default.rb**

```
bash 'Adding Redis PPA and Installing Package' do
  user 'root'
  code <<-EOC
    apt-get install -y python-software-properties
    add-apt-repository -y ppa:chris-lea/redis-server
    apt-get update
    apt-get install -y redis-server
  EOC
end

bash 'Removing existing redis-server config file and adding new one' do
  user 'root'
  code <<-EOC
    rm /etc/redis/redis.conf
    cat <<EOF > /etc/redis/redis.conf
    daemonize yes
    pidfile /var/run/redis/redis-server.pid
    logfile /var/log/redis/redis-server.log

    port 6379
      bind 127.0.0.1
    timeout 300

    loglevel notice

    databases 16

    save 900 1
    save 300 10
    save 60 10000

    rdbcompression yes
    dbfilename dump.rdb

    dir /etc/redis/
    appendonly no

    maxmemory 419430400
    maxmemory-policy allkeys-lru

    maxmemory-samples 10
EOF
    chown -R redis:redis /etc/redis/
  EOC
end

bash 'Restarting Redis' do
  user 'root'
  code <<-EOC
    /etc/init.d/redis-server restart
  EOC
end
```

Note the strange indentation of `EOF`. This is because when using "Here Markers" to cat multiple lines to a file, the marker must not have any trailing or leading spaces, otherwise it will be considered part of string to be added to the file.

## Adding the recipe to the node definition

Now that we've created a simple recipe to execute the commands for installing Redis, we need to add the recipe to our nodes run list.

Remember this chapter assumes that you've:

- Copied across your SSH key
- Bootstrapped the node

As per the chapter "Using Knife". And that you've added this cookbook to the Berksfile as explained earlier in this chapter.

Now, in a terminal in your chef repository add the recipe to the run list of your test node:

```
knife node run_list add NODE_NAME 'recipe[redis-server]'
```

Replacing `NODE_NAME` with the name of the traget node. You should see output similar to:

```
NODE_NAME:
  run_list: recipe[redis-server]
```

Which indicates that the operation was succesful and the run list for `NODE_NAME` now contains our new recipe.

## Applying the updated node definition

Having updated and saved the node definition as above, we can apply the updated configuration to our node by entering the following in our local terminal:

**Terminal (local)**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

We'll see output similar to the following:

```
Execute command hook in before_converge.
Resolving cookbook dependencies...
Fetching 'redis-server' from source at ../redis-server
Using redis-server (0.1.0) from source at ../redis-server
Vendoring redis-server (0.1.0) to /Users/ben/tmp/rdr_demo_1/berks-cookbooks/redis\
-server
188.166.151.185 Starting Chef Client, version 12.14.89
188.166.151.185 resolving cookbooks for run list: ["redis-server"]
188.166.151.185 Synchronizing Cookbooks:
188.166.151.185   - redis-server (0.1.0)
188.166.151.185 Installing Cookbook Gems:
188.166.151.185 Compiling Cookbooks...
188.166.151.185 Converging 3 resources
188.166.151.185 Recipe: redis-server::default
188.166.151.185   * bash[Adding Redis PPA and Installing Package] action run
188.166.151.185     - execute "bash"  "/tmp/chef-script20161009-3602-1a8gcsd"
188.166.151.185   * bash[Removing existing redis-server config file and adding ne\
w one] action run
188.166.151.185     - execute "bash"  "/tmp/chef-script20161009-3602-omzkbm"
188.166.151.185   * bash[Restarting Redis] action run
188.166.151.185     - execute "bash"  "/tmp/chef-script20161009-3602-1x7aez3"
188.166.151.185
188.166.151.185 Running handlers:
188.166.151.185 Running handlers complete
188.166.151.185 Chef Client finished, 3/3 resources updated in 17 seconds
```

You can see that the text we entered in the form `bash "description of commands"` is output when that part of the recipe is executed. That allows us to see both an indicator of progress through the recipe and makes it easier to debug if their is a problem at any given step.

A simple improvement to our cookbook might therefore be to group the commands more specifically, for example 'adding ppa' 'running apt-get update' etc.

If we now use knife to execute `redis-server -v` on our remote server:

**Terminal (local)**

```
knife ssh 'name:NODE_NAME' --ssh-user root 'redis-server -v'
```

We'll see output similar to:

```
188.166.151.185 Redis server v=3.0.7 sha=00000000:0 malloc=jemalloc-3.6.0 bits=64\
 build=869e89100d5ea8c2
```

Indicating that `redis-server` is now installed. Entering:

**Terminal (local)**

```
knife ssh 'name:NODE_NAME' --ssh-user root 'cat /etc/redis/redis.conf'
```

Will show that our custom configuration has been added.

## Is This Cookbook "Bad" or "Wrong"

It's quite likely many people who know Chef well would look at this Cookbook and say that it's a "bad" way of using Chef. What's meant by this is that it's not really taking advantage of the true power of Chef, it's simply using it as a way of executing bash scripts.

This is definitely a fair point, as we'll see in the next section, this Cookbook can be improved dramatically by leveraging more of Chefs DSL as well as additions to this DSL provided by community cookbooks.

This Cookbook is however still vastly better than relying on lists of manually entered commands or combinations of not always up to date bash scripts.

If we rely on entirely on in-house Chef Cookbooks which do nothing more complicated than automatically executing the bash commands we would previously have entered by hand, we are still in a much better place than before in the event of a server failure if this allows us to reproducibly provision a new server quickly.

That said, the Chef DSL is extremely powerful and the basics are quick to grasp so it's unlikely that anyone who's tried it once, would stick to the "nothing but bash" approach.

## Limitations of the simple Cookbook

Before looking at how we can better use the Chef DSL in our simple cookbook, it's worth looking at what the key drawbacks are of the current approach so we can see objectively how using the Chef DSL is an improvement.

Since most people already know bash, we need to have clear reasons for introducing an additional, new DSL to the mix and so adding a degree of complexity.

### Lack of Flexibility

Let's say that we have multiple nodes which need Redis installed on them. Some of these are production, standalone Redis servers which deal with high volumes of traffic and have substantial resources accordingly. Others may be staging servers with limited resources.

In this scenario we would want to set different limits on the resource usage of Redis.

At the moment there is no easy way to do this, we would need to create a copy of our recipe and amend the section where we `cat` configuration into the file to reflect the new memory limitations.

### Repetition

The above scenario also demonstrates another clear problem. Repetition.

We would now be maintaining two, nearly identical Cookbooks for installing Redis, the only difference being in the configuration file. When we update one we have to update the other. This should have any Ruby developer feeling uncomfortable since it's a clear violation of DRY.

This repetition goes even further, at the start of our Cookbook we have bash commands to install `python-software-properties`, add a ppa then issue an `apt-get update`. This is surely something we're going to want to do in many differeny cookbooks. At the moment every time we do it, we're repeating those same bash commands. If the process for adding a ppa changes in a future release of the operating system we're using, we're going to have to update that code in every cookbook we've created which uses it.

### Lack of Readability

Earlier we said that one of the benefits of simple bash scripts was readability, most developers are comfortable with simple bash scripts and so using them minimises the learning curve.

The other side of this is that simple scripts become complex scripts very quickly. Let's say we want to only restart Redis if we've changed the configuration file. The bash commands to check whether the file has changed and execute a restart if so are possible but a long way from readable.

It could be accomlished using a combiantion of `sed` and conditionals but it's not going to be pretty and it's definitely far beyond the level of bash I can glance at and understand let alone improve.

## Improving our Cookbook the Chef Way

The Chef DSL provides ways to address all of the above limitations. We'll begin by using more of the DSL in conjunction with the concept of templates to address the issues with readability and repetition and then move onto using attributes to address the lack of flexibility.

### Repetition - Adding a PPA and installing packages

We're probably going to want to add PPA's and install packages from them quite regularly therefore the Rubyist in us will want to abstract this functionality into a single location.

Chef allows us to do this using what are called "Lightweight Resource Providers", referred to from now on as LWRP's. A large part of Chefs power comes from the many community defined LWRP's which allow us to re-use other peoples code in our own recipes to perform common operations.

In this case we want to add an external PPA. Googling "Chef Add PPA" leads us to the following official cookbook from Opscode (the makers of Chef): https://github.com/opscode-cookbooks/apt.

In order to use functionality from this third party cookbook, we begin by declaring that our Cookbook is dependent on the `apt` cookbook in our `metadata.rb` by amending it to include a `depends` entry as follows:

**redis-server/metadata.rb**

```
name             'redis-server'
maintainer       'YOUR_COMPANY_NAME'
maintainer_email 'YOUR_EMAIL'
license          'All rights reserved'
description      'Installs/Configures redis-server'
long_description IO.read(File.join(File.dirname(__FILE__), 'README.md'))
version          '0.1.0'
depends          'apt'
```

We can think of this like adding a dependancy in a Ruby Gemspec.

We can see from the apt cookbooks Readme that the syntax for adding a PPA using the LWRP is as follows:

**https://github.com/opscode-cookbooks/apt/README (extract)**

```
apt_repository 'nginx-php' do
  uri          'ppa:nginx/stable'
  distribution node['lsb']['codename']
end
```

Therefore we can replace this part of our existing `default.rb`:

**rdr_redis_example/site-cookbooks/redis-server/recipes/default.rb (extract)**

```
...
bash 'Adding Redis PPA and Installing Package' do
  user 'root'
  code <<-EOC
    apt-get install -y python-software-properties
    add-apt-repository -y ppa:chris-lea/redis-server
    apt-get update
    apt-get install -y redis-server
  EOC
end
...
```

with:

**rdr_redis_example/site-cookbooks/redis-server/recipes/default.rb (extract)**

```
...
apt_repository 'redis-server' do
  uri          'ppa:chris-lea/redis-server'
  distribution node['lsb']['codename']
end

bash 'Installing Redis Server Package' do
  user 'root'
  code <<-EOC
    apt-get install -y redis-server
  EOC
end
...
```

This reduces the amount of bash to one line and more importantly, our recipe is now only concerned with which ppa is being added, not how we go about adding that ppa. The logic for adding the ppa is now centralised in the third party apt cookbook, allowing us to leverage other peoples time and expertise.

If we want to get a better idea of what's going on when we call `apt_repoistory`, for example to confirm whether or not it will automatically call `apt-get update` (it does) we can look at the source of the `apt` cookbook. Since it's a provider, it will be defined in the `providers` directory, we can look at the source directly here; https://github.com/opscode-cookbooks/apt/blob/master/providers/repository.rb.

This is definitely an improvement. But in the same way we're going to want to add PPA's quite regularly, we're going to want to install packages regularly. It would be better not to have to worry about the specifics of `apt-get` or adding flags such as `-y`.

Installing packages is such a common task that it's included in the standard Chef DSL, there's no need for a third party LWRP. The method we use is `package` and full documentation is available at: https://docs.getchef.com/resource_package.html.

Using the `package` method we can remove the remainder of our first bash script so that the beginning of our recipe reads as follows:

**redis-server/recipes/default.rb (extract)**

```ruby
apt_repository 'redis-server' do
  uri          'ppa:chris-lea/redis-server'
  distribution node['lsb']['codename']
end

package 'redis-server'
```

## Readability - Using Templates

The above improvements provide a much more elegant interface to adding the PPA and installing the package, but we still have a long bash entry which is responsible for updating the configuration file.

Chef provides a simple interface which allows us to create templates for remote files we wish to create or modify using erb and then use these templates to create or update files in specific locations.

To use this we begin by creating an entry called `redis.conf.erb` in the `templates/default` directory and paste in the contents of our configuration file.

**redis-server/templates/default/redis.conf.erb**

```
daemonize yes
pidfile /var/run/redis/redis-server.pid
logfile /var/log/redis/redis-server.log

port 6379
  bind 127.0.0.1
timeout 300

loglevel notice

databases 16

save 900 1
save 300 10
save 60 10000

rdbcompression yes
dbfilename dump.rdb

dir /etc/redis/
appendonly no
```

```

maxmemory 419430400
maxmemory-policy allkeys-lru

maxmemory-samples 10
```

At the moment, the section of our `default.rb` recipe which updates the configuration file looks like this:

**redis-server/recipes/default.rb (extract)**

```
...
bash 'Removing existing redis-server config file and adding new one' do
  user 'root'
  code <<-EOC
    rm /etc/redis/redis.conf
    cat <<EOF > /etc/redis/redis.conf
    daemonize yes
    pidfile /var/run/redis/redis-server.pid
    logfile /var/log/redis/redis-server.log

    port 6379
      bind 127.0.0.1
    timeout 300

    loglevel notice

    databases 16

    save 900 1
    save 300 10
    save 60 10000

    rdbcompression yes
    dbfilename dump.rdb

    dir /etc/redis/
    appendonly no

    maxmemory 419430400
    maxmemory-policy allkeys-lru

    maxmemory-samples 10
EOF
    chown -R redis:redis /etc/redis
  EOC
end
```

We replace this with:

**rdr_redis_example/site-cookbooks/redis-server/recipes/default.rb (extract)**

```
template "/etc/redis/redis.conf" do
  owner "redis"
  group "redis"
  mode "0644"
  source "redis.conf.erb"
end

directory "/etc/redis" do
  owner 'redis'
  group 'redis'
  action :create
end
```

## Template Directories

The directory `templates/default` doesn't refer to the default recipe, it refers to the default platform (operating system). So when we reference the template with the `source` method above, assuming we're installing on Ubuntu 14.04, it would look for the file `redis.conf.erb` in `templates/ubuntu-14.04/` then `templates/ubuntu/` then `templates/default/`. Theres more on the this here https://docs.getchef.com/essentials_cookbook_templates.html

There are several benefits to this approach:

- Once again our recipe is no longer concerned with the how of updating something, just with what should be updated. The how is abstracted away, in this case using part of the standard chef DSL; `template`
- The internals of how Chef updates file is much smarter than our simple `rm` then `cat` approach. It will compare the desired state of the file with the actual state of the file on the remote system, and only update the file on the remote system if the two are different.

The last part of this is especially important. At the moment, the final section of our `default.rb` is responsible for restarting Redis after we update the configuration file so that the new configuration will be used.

The problem with this approach is that Redis will be re-started irrespective of whether the configuration file has changed. That means that every time we run `knife zero converge` on the node, Redis will be re-started.

This means that if we were running `knife zero converge` on the node to do an unrelated maintenance task, for example adding a new public key to login with, Redis will be temporarily unavailable while it restarts.

The Chef `template` resource provides an easy way for us to perform actions only when a file, such as our `redis.conf` configuration file is changed.

At the moment the section of our `default.rb` which restarts Redis looks like this:

**redis-server/recipes/default.rb (extract)**

```
...
bash 'Restarting Redis' do
  user 'root'
  code <<-EOC
    /etc/init.d/redis-server restart
  EOC
end
...
```

We can begin by replacing this section with:

**redis-server/recipes/default.rb (extract)**

```
...
execute "restart-redis" do
  command "/etc/init.d/redis-server restart"
  action :nothing
end
...
```

This is essentially another way of executing the same command (`/etc/init.d/redis-server restart`). The key difference is the use of the `action` method. If we were to pass `:run` to the `action` method it would behave in exactly the same way as the bash section it replaces. However passing in `:nothing` means that when it is reached, it will do nothing, instead it waits to be notified that it needs to run by another resource.

We can then use our `template` resource to notify the `execute` resource to run when the file is changed by changing it to read as follows:

**redis-server/recipes/default.rb (extract)**

```
...
template "/etc/redis/redis.conf" do
  owner "redis"
  group "redis"
  mode "0755"
  source "redis.conf.erb"
  notifies :run, "execute[restart-redis]", :immediately
end
...
```

The `notifies` method is extremely powerful and it's worth taking a look at https://docs.getchef.com/resource_common.html#notifies-syntax to get a better idea of what it's capable of.

So our complete `default.rb` now looks like this:

**redis-server/recipes/default.rb**

```ruby
apt_repository 'redis-server' do
  uri          'ppa:chris-lea/redis-server'
  distribution node['lsb']['codename']
end

package 'redis-server'

template "/etc/redis/redis.conf" do
  owner "redis"
  group "redis"
  mode "0755"
  source "redis.conf.erb"
  notifies :run, "execute[restart-redis]", :immediately
end

directory "/etc/redis" do
  owner 'redis'
  group 'redis'
  action :create
end

execute "restart-redis" do
  command "/etc/init.d/redis-server restart"
  action :nothing
end
```

And we've added the configuration file to:

`redis-server/templates/default/redis.conf.erb`

## Updating The Node

### Always test with a clean node

Our recipe is now significantly more readable and tasks such as writing configuration files and adding ppa's are abstracted out of the recipe to a central location so we're not going to be repeating ourselves as our collection of recipes grows.

We still need to address the lack of flexibility but first we'll apply this updated recipe to our server and confirm that it works as we expect.

Begin by restoring a fresh Ubuntu 16.04 image to your server. It's important when testing changes to recipes and node configurations to start with a clean node rather than just re-running converge. While it often makes sense to test experimental changes by simply re-running it, the final test should always be applying it to a clean node to ensure that it's actually working on its own rather than appearing to work due to side effects of previous versions.

After restoring the base image to the node, the first time we attempt to connect via ssh it will report that the nodes fingerprint does not match which could be an indicator of a man in the middle attack. To avoid this, enter the following to remove the nodes entry from you `~/.ssh/known_hosts` file:

**Terminal (local)**

```
ssh-keygen -R YOURSERVERIP
```

You'll then need to copy your key across again:

**Terminal (local)**

```
ssh-copy-id root@YOURSERVERIP
```

And enter the servers root password when prompted.

We'll now need to bootstrap the node again. Bear in mind you must be within the Chef repository folder to do this:

**Terminal (Chef Repo)**

```
knife zero bootstrap SERVER_IP_OR_HOSTNAME --ssh-user root --node-name NODE_NAME
```

We'll be asked if we want to over-write the previous node definition with this name, we can select yes.

Then add our cookbook to the nodes run list:

**Terminal (Chef Repo)**

```
knife node run_list add NODE_NAME 'recipe[redis-server::default]'
```

And apply the updated run list with:

**Terminal (Chef Repo)**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

Once the cook completes, we can execute `redis-server -v` via ssh and confirm, as before, that Redis is installed:

**Terminal (Chef Repo)**

```
knife ssh 'name:NODE_NAME' --ssh-user root 'redis-server -v'
```

We'll see that `redis-server` is now installed. Entering:

**Terminal (Chef Repo)**

```
knife ssh 'name:NODE_NAME' --ssh-user root 'cat /etc/redis/redis.conf'
```

We'll see that our custom configuration has been added.

## Changing The Configuration File

Now we've confirmed that the recipe works in the same way it did before, we can test that Redis is only restarted when the configuration file is changed.

To test this let's first run another cook without making any changes.

First, find the PID of the Redis process by executing:

**Terminal (Chef Repo)**

```
knife ssh 'name:NODE_NAME' --ssh-user root 'ps aux | grep redis'
```

Which will give output similar to:

**Terminal output**

```
188.166.151.185 redis     3363  0.0  0.8  41728  8864 ?       Ssl  06:37   0:00 \
/usr/bin/redis-server 127.0.0.1:6379
188.166.151.185 root      3824  0.0  0.2  12652  2800 pts/0    Ss+  06:46   0:00 \
bash -c ps aux | grep redis
188.166.151.185 root      3826  0.0  0.1  11960  1948 pts/0    S+   06:46   0:00 \
grep redis
```

Where we can see the pid of the Redis process, in this case 3363.

Now in your local terminal, execute converge on the node:

**Terminal (Chef Repo)**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

Once this completes, if you run `knife ssh 'name:NODE_NAME' --ssh-user root 'ps aux | grep redis'` again, you should see that the PID is the same, Redis has not been restarted.

Now make a minor change to the configuration file (`redis-server/templates/default.redis.conf.erb`), change the value for `timeout` from `300` to `200`. Save the file and run converge again:

**Terminal**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

Notice part way through the output we see something like this:

**Terminal output**

```
188.166.151.185   * template[/etc/redis/redis.conf] action create
188.166.151.185     - update content in file /etc/redis/redis.conf from cc9070 to\
 659485
188.166.151.185     --- /etc/redis/redis.conf   2016-10-10 06:37:28.584848733 +00\
00
188.166.151.185     +++ /etc/redis/.chef-redis20161010-4604-swewyu.conf 2016-10-1\
0 06:49:41.780848733 +0000
188.166.151.185     @@ -4,7 +4,7 @@
188.166.151.185
188.166.151.185      port 6379
188.166.151.185        bind 127.0.0.1
188.166.151.185     -timeout 300
188.166.151.185     +timeout 200
188.166.151.185
188.166.151.185      loglevel notice
188.166.151.185
188.166.151.185   * execute[restart-redis] action run
188.166.151.185     - execute /etc/init.d/redis-server restart
```

This is a diff of the configuration file that was on the server and the state that our chef recipe says it should be in following the cook. This allows us to clearly see which lines have been added and removed.

We can also see from the line `execute[restart-redis] action run` that our restart block has been called.

Now check the PID for Redis using `knife ssh 'name:rdrtest1' --ssh-user root 'ps aux | grep redis'`, it will now have changed because the configuration file was changed and so the restart command was executed.

## Adding flexibility with attributes

We've now addressed the issues with repeatability and readability so it's time to look at adding some flexibility using attributes.

An attribute is a value which is used to change the behavior of a recipe. Common use cases include setting a value in a configuration file, choosing the version of a package to be installed and setting file paths.

The values of attributes can be set in several locations. In this chapter we'll set default values within our recipe, in the next chapter on node and role definitions, we'll look at overriding these on a per node or per role basis.

We'll begin by defining a few simple attributes which will allow us to:

- Set the PPA Redis is installed from

- Define the name of the Redis package
- Set the port Redis listens on
- Choose the interface Redis is bound to

## Setting Default Attributes

Begin by creating a file called `default.rb` in the `attributes` folder of the cookbook with the following content:

**redis-server/attributes/default.rb**

```ruby
default['redis-server']['bind'] = '127.0.0.1'
default['redis-server']['package'] = 'redis-server'
default['redis-server']['port'] = '6379'
default['redis-server']['ppa'] = 'ppa:chris-lea/redis-server'
```

These default values will be used if they are not overriden with different values from any source with a higher attribute precedence. For more on attribute precedence, see: https://docs.getchef.com/essentials_cookbook_attribute_files.html.

## Accessing attributes from a recipe

Within our recipe, attributes are made available as the variable `node` which is a standard Ruby hash.

We can therefore update our `default.rb` recipe as follows:

**redis-server/recipes/default.rb**

```ruby
apt_repository 'redis-server' do
  uri          node['redis-server']['ppa']
  distribution node['lsb']['codename']
end

package node['redis-server']['package']

template "/etc/redis/redis.conf" do
  owner "redis"
  group "redis"
  mode "0755"
  source "redis.conf.erb"
  notifies :run, "execute[restart-redis]", :immediately
end

directory "/etc/redis" do
  owner 'redis'
  group 'redis'
  action :create
```

```
end

execute "restart-redis" do
  command "/etc/init.d/redis-server restart"
  action :nothing
end
```

This starts to demonstrate the power of Chef recipes being written in Ruby. The attributes are available to us as a standard hash and we can write ordinary ruby code to interact with them.

So let's say we wanted to have the option to skip using the ppa completely, we could add an additional attribute to our `default.rb`:

**redis-server/attributes/default.rb (extract)**

```
...
default['redis-server']['use_ppa'] = true
...
```

And then the following to the recipe:

**redis-server/recipes/default.rb (extract)**

```
...
if node['redis-server']['use_ppa']
  apt_repository 'redis-server' do
    uri          node['redis-server]['ppa']
    distribution node['lsb']['codename']
  end
end
...
```

So that the PPA is only added if the `use_ppa` attribute is true. One of the key benefits of using Chef over the various other configuration management tools when configuring servers for Rails apps, is that we can just carry on writing the same old Ruby code we're used to.

For now we're not going to add this but it serves to demonstrate what is possible.

## Accessing Attributes from a Template

We can now customise the behavior of our recipe using attributes. The final step is to allow us to customise the values in the configuration file, in this case the port and address to bind to. In practice this is probably the most useful part to be able to customise as it's the most likely to vary between nodes.

Our configuration file template, `redis.conf.erb` is a standard erb file so we can interact with it in exactly the same we would an `erb` file in a Rails view with the nodes attributes being available once again as a hash with the name `node`.

**node V @node**

In earlier versions of Chef, the node attribute was made available in an instance variable `@node` similarly to how variables in Rails views are exposed. Although some community cookbooks still use the legacy version, the `@node` syntax is depreciated and so `node` should always be used instead. Using the older `@node` syntax will cause a depreciation warning to be displayed when the recipe is used and it will be removed in future versions of Chef.

We can therefore update our `redis.conf.erb` configuration file as follows:

**rdr_redis_example/site-cookbooks/redis-server/tempaltes/default/redis.conf.erb**

```
daemonize yes
pidfile /var/run/redis/redis-server.pid
logfile /var/log/redis/redis-server.log

port <%= node['redis-server']['port'] %>
bind <%= node['redis-server']['bind'] %>
timeout 300

loglevel notice

databases 16

save 900 1
save 300 10
save 60 10000

rdbcompression yes
dbfilename dump.rdb

dir /etc/redis/
appendonly no

maxmemory 419430400
maxmemory-policy allkeys-lru

maxmemory-samples 10
```

Having updated this we can converge our node with:

**Terminal (local)**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

We should see no diff in the output, the configuration file has not been changed.

## Dealing with optional configuration parameters

Our cookbook is now far more flexible, we can change the ppa which is used, the port Redis listens on and the interface it binds to without modifying the recipe itself.

It is however going to get quite tedious providing an attribute and default for all of the possible Redis configuration parameters. This is made even worse by the fact many of the parameters are optional, we may not want to include them at all.

The below extract from our Redis configuration file is a good example of this, it sets up a memory limit and defines a policy for discarding keys once this memory limit is reached. We would only want such a policy if we're using Redis as some sort of cache, where it's acceptable to discard data.

**templates/default/redis.conf.erb (extract)**

```
...
maxmemory 419430400
maxmemory-policy allkeys-lru
maxmemory-samples 10
...
```

One way of dealing with this would be to not provide default values and then only write them to the configuration file if they are provided in the node definition. We can do this by checking each one to see if it's `nil` first and only including it if not.

This however has two key drawbacks.

- We still have to know all of the possible configuration parameters in advance
- Our template file is going to be full of conditionals which will eventually hamper readability

A more elegant solution to this is to allow an optional hash to be provided with arbitrary key, value pairs and then write these to the configuration file as provided.

For example if we were to replace the entries beginning with `maxmemory` in our configuration file with the following:

**templates/default/redis.conf.erb (extract)**

```
...
<% node['redis-server']['additional_configuration_values'].each do |key, value| %>
  <%= "#{key} #{value}" %>
<% end %>
...
```

if `node['redis-server']['additional_configuration_values]` contains the following hash (we'll cover how to set this from a node or role definition in the next chapter):

```
{
  'maxmemory' => 419430400,
  'maxmemory-policy' => 'allkeys-lru',
  'maxmemory-samples' => '10'
}
```

Then this will generate exactly the same maxmemory configuration as our original configuration file.

In practice it generally makes sense to combine the two approaches. For any configuration options which should always be present, we can use the simple approach, then include the additional hash for optional values. With this in mind our final `redis.conf.erb` might look like this:

**templates/default/redis.conf.erb**

```
daemonize <%= node['redis-server']['daemonize'] %>

pidfile <%= node['redis-server']['pidfile'] %>

logfile <%= node['redis-server']['logfile'] %>

port <%= node['redis-server']['port'] %>

bind <%= node['redis-server']['bind'] %>

timeout <%= node['redis-server']['timeout'] %>

loglevel <%= node['redis-server']['loglevel'] %>

databases <%= node['redis-server']['databases'] %>

<% node['redis-server']['save'].each do |save_entry| %>
  <%= "save #{save_entry}" %>
<% end %>

rdbcompression <%= node['redis-server']['rdbcompression'] %>

dbfilename <%= node['redis-server']['dbfilename'] %>

dir <%= node['redis-server']['dir'] %>

appendonly <%= node['redis-server']['appendonly'] %>

<% node['redis-server']['additional_configuration_values'].each do |key, value| %>
  <%= "#{key} #{value}" %>
<% end %>
```

With our attribute defaults file looking like this:

**attributes/default.rb**

```ruby
default['redis-server']['appendonly'] = 'no'
default['redis-server']['additional_configuration_values'] = {}
default['redis-server']['bind'] = '127.0.0.1'
default['redis-server']['daemonize'] = 'yes'
default['redis-server']['databases'] = 16
default['redis-server']['dbfilename'] = 'dump.rdb'
default['redis-server']['dir'] = '/etc/redis/'
default['redis-server']['logfile'] = '/var/log/redis/redis-server.log'
default['redis-server']['loglevel'] = 'notice'
default['redis-server']['package'] = 'redis-server'
default['redis-server']['pidfile'] = '/var/run/redis/redis-server.pid'
default['redis-server']['port'] = '6379'
default['redis-server']['ppa'] = 'ppa:chris-lea/redis-server'
default['redis-server']['rdbcompression'] = 'yes'
default['redis-server']['save'] = [
  '900 1',
  '300 10',
  '60 10000'
]
default['redis-server']['timeout'] = 300
```

A few things to note about the final versions of our template and default attributes files.

Firstly although it is in no way required, it's good practice to order default attributes alphabetically. This makes managing larger cookbooks significantly easier.

Secondly notice that by default, we assign 'additional_configuration_values' an empty hash, this means that if it's not set anywhere else, an exception will not be raised when we try to iterate over it.

Finally notice the entry for the `save` configuration parameter which accepts an array so that an arbitrary number of save conditions can be provided.

## Conclusion

In this section we've seen in some detail how to create a Chef cookbook from scratch which leverages community code for common functionality and allows us to customise its behavior with attributes.

At this point it may be very tempting to begin creating Chef cookbooks for every part of our infrastructure. Before doing so however, it's worth considering the subsection at the start of Chapter 6 which looks at the use of Community Cookbooks.

Although it's essential to understand how cookbooks are put together and to be able to create our own for any functionality which is truly bespoke to our configuration, in a vast majority of situations, there will already be an open source cookbook we can use and it will often make sense to favour using that over creating our own.

In the next section we'll look in more detail at the node and role definitions and how these are used to customise attributes.

# Node and Role Definitions

## Overview

In 5.1 we created a cookbook for installing and configuring Redis. This cookbook allowed us to customise its behaviour using attributes.

We also briefly touched on a node definition, we added our `redis-server` cookbook to the `run_list` so that when we converged the node, the Redis recipe would be applied to our server.

We relied on our recipe to provide default values for the attributes.

In this section we'll look at two key places in which we'll set attribute values:

- Node definitions
- Role definitions

## Attribute Hierarchy

These locations fall within a hierarchy which determines which value takes precedence in the event it is provided in more than one place. For our purposes, this hierarchy will be simple, node overrides role, overrides cookbook defaults.

In practice this is a huge over simplification, there are many more locations in which attributes can be set as well as ways of setting which alter their precedence. You can read more about this here https://docs.getchef.com/essentials_cookbook_attribute_files.html but for the purposes of setting up simple (<10 nodes) configurations using Chef Solo, it's generally unnecessary to look at anything more complex.

## Node Definitions

A node definition can be thought of as the most granular level of configuration file, attributes defined here are applied to the node it relates to and only that node.

You can read more about the exact definition of a node here https://docs.getchef.com/essentials_node_object.html

At the end of section 5.1, our node definition looked like this (accessed using `knife node edit NODE_NAME`):

**rdr_redis_example/nodes/YOURSERVERIP.json**

```json
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "knife_zero": {
      "host": "128.199.233.228"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "recipe[redis-server]"
  ]

}
```

In our Redis recipe we define the following default attribute which manages the port Redis listens on:

**redis-server/attributes/default.rb (extract)**

```ruby
...
default['redis-server']['port'] = '6379'
...
```

If we wanted to override this for our node, so that Redis would listen on port 6380 instead, we could update our node definition to the following:

**Node Definition**

```json
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "redis-server": {
      "port" : "6380"
    }
    "knife_zero": {
      "host": "128.199.233.228"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "recipe[redis-server]"
  ]

}
```

Using `knife node edit NODE_NAME`

Notice that we define attributes within the "normal" key.

Now if we save the node definition and apply it to the node:

**Terminal**

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

We should see a diff including in the output similar to the following:

```
128.199.233.228     - update content in file /etc/redis/redis.conf from
f52142 to a1633e
128.199.233.228     --- /etc/redis/redis.conf   2017-01-02
07:39:49.242705879 +0000
128.199.233.228     +++ /etc/redis/.chef-redis20170102-7942-tqp42c.conf
2017-01-02 08:09:41.765322207 +0000
128.199.233.228     @@ -4,7 +4,7 @@
128.199.233.228
128.199.233.228      logfile /var/log/redis/redis-server.log
128.199.233.228
128.199.233.228     -port 6379
128.199.233.228     +port 6380
128.199.233.228
128.199.233.228      bind 127.0.0.1
```

We can see that the line `port 6379` has been removed and replaced with the updated `port 6380` meaning that Redis will now listen on the port as specified in our node definition. So far so good.

Recall that in our Redis recipe, we included the `additional_configuration_values` attribute which allowed us to pass in arbitrary key value pairs and have them written to the configuration file. This allowed us the flexibility to include optional parameters and removed the need for us to know and include logic for every option which the Redis configuration file could possible contain.

We can now update our node definition to take advantage of this by setting a maxmemory policy. The policy we'll set will cause Redis to begin discarding keys once the memory limit is reached by picking 10 random keys and discarding the least recently used.

Open the node definition using:

```
knife node edit NODE_NAME
```

And add the `additional_configuration_values` hash to the `redis-server` key.

**Node Definition**

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "redis-server": {
      "port" : "6380",
      "additional_configuration_values" : {
        "maxmemory" : "419430400",
        "maxmemory-policy" : "allkeys-lru",
        "maxmemory-samples" : "10"
      }
    }
    "knife_zero": {
      "host": "128.199.233.228"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "recipe[redis-server]"
  ]

}
```

Running `knife zero converge 'name:NODE_NAME' --ssh-user root` with this updated node definition shows a diff similar to:

**Terminal output**

```
128.199.233.228     - update content in file /etc/redis/redis.conf from
a1633e to 2982ec
128.199.233.228     --- /etc/redis/redis.conf   2017-01-02
08:09:41.765322207 +0000
128.199.233.228     +++ /etc/redis/.chef-redis20170102-8569-10g5g7.conf
2017-01-02 08:15:40.793932445 +0000
128.199.233.228     @@ -26,5 +26,8 @@
128.199.233.228
128.199.233.228      appendonly no
128.199.233.228
```

```
128.199.233.228     +   maxmemory 419430400
128.199.233.228     +   maxmemory-policy allkeys-lru
128.199.233.228     +   maxmemory-samples 10
```

Which shows that our three additional configuration lines have been added.

## Role Definitions

We can now use a node definition to set attributes and therefore customise recipe behaviour but this approach has one key weakness. It's going to lead to a lot of repetition.

In our simple node definition above we include the redis recipe, change the port and define a `maxmemory` policy. If we deploy multiple Redis servers it's likely that we're going to want each, or at least several of these, to have the same configuration. If we rely entirely on node definitions, we'll need to define exactly the same attributes and run list on each of them.

If we want to change our standard Redis configuration, we'll have to manually update every node definition. While this may not seem like too big of a task at the moment, in practice for a production configuration, we're likely to have significantly more attributes defining everything from users and sudo rights to firewall rules and alerting preferences.

Roles offer us a solution to this. Roles can be thought of as analagous to mixins in Ruby. They allow us to define a combination of attributes and `run_list` entries which we can then include in a node definition. This allows us to centralise our definitions, of say a `redis-cache-server` and then re-use this definition for every node that it applies to.

### A simple Role Definition

Begin by creating a new role `redis-cache-server`:

```
knife role create redis-cache-server
```

This will open the role definition JSON in your editor of choice. Update the `default_attributes` and `run_list` sections as below and then save it.

**Role Definition**

```
{
  "name": "redis-cache-server",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {
    "redis-server" : {
      "port" : "6380",
      "additional_configuration_values" : {
        "maxmemory" : "419430400",
        "maxmemory-policy" : "allkeys-lru",
        "maxmemory-samples" : "10"
      }
    }
  },
  "override_attributes": {

  },
  "chef_type": "role",
  "run_list": [
    "recipe[redis-server]"
  ],
  "env_run_lists": {

  }
}
```

The `name` key here is important as it is what will allow us to refer to this role from our node definition.

So far this role looks a lot like our node definition, it's a standard JSON object and includes a `run_list` which references our default `redis-server` recipe.

Notice that instead of including attributes within the `normal` key as we do in a node definition, they are instead nested inside the `default_attributes` key. As the name suggests, this key allows us to specify default attributes.

If the node definition doesn't explicitly define these attributes, the ones specified in the role definition will take precedence over the recipes own defaults.

## Including The Role in a Node Definition

We include roles in node definitions in the same way we include recipes, by adding it to the run list. We can therefore add this role to our nodes run list as follows:

```
knife node run_list add NODE_NAME 'role[redis-cache-server]'
```

At this point we can remove the recipe we previously added:

```
knife node run_list remove NODE_NAME 'recipe[redis-server]'
```

We can also edit the node definition to remove some of the attributes we set there (using `knife node edit NODE_NAME`) and updating the node definition to:

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "redis-server": {
      "port": 6380
    },
    "knife_zero": {
      "host": "128.199.233.228"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "role[redis-cache-server]"
  ]

}
```

Since we don't define any of the `maxmemory` attributes in this version of the node definition, the default attributes specified in the role definition will be used. For any attributes not specified in either the node or the role definition, the default attributes from our cookbook will be used.

We can test this by executing:

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

And seeing that there is no diff in the output because the configuration file has not changed (all of the previous attributes, despite being removed from our node definition, flow thorugh from the defaults in our role definition).

We can then edit the role definition:

```
knife role edit redis-cache-server
```

To the following:

```
{
  "name": "redis-cache-server",
  "description": "",
  "json_class": "Chef::Role",
  "default_attributes": {
    "redis-server": {
      "port": "6380",
      "additional_configuration_values": {
        "maxmemory": "419430400",
        "maxmemory-policy": "allkeys-lru",
        "maxmemory-samples": "20"
      }
    }
  },
  "override_attributes": {

  },
  "chef_type": "role",
  "run_list": [
    "recipe[redis-server]"
  ],
  "env_run_lists": {

  }
}
```

Changing `maxmemory-samples` to 20 then running:

```
knife zero converge 'name:NODE_NAME' --ssh-user root
```

To apply the changes. We'll see in the output from the diff:

```
128.199.233.228     - update content in file /etc/redis/redis.conf from
2982ec to 615128
128.199.233.228     --- /etc/redis/redis.conf   2017-01-02
08:15:40.793932445 +0000
128.199.233.228     +++ /etc/redis/.chef-redis20170102-9752-fc75m5.conf
2017-01-02 08:33:20.352719091 +0000
128.199.233.228     @@ -28,6 +28,6 @@
128.199.233.228
128.199.233.228        maxmemory 419430400
128.199.233.228        maxmemory-policy allkeys-lru
128.199.233.228     -  maxmemory-samples 10
128.199.233.228     +  maxmemory-samples 20
```

That the default has flowed through from the role definition and `maxmemory-samples` has been set to 20.

Finally we can over-ride one of these keys in the node definition:

```
{
  "name": "rdrtest1",
  "chef_environment": "_default",
  "normal": {
    "redis-server": {
      "port": 6380,
      "additional_configuration_values" : {
        "maxmemory-samples" : "40"
      }
    },
    "knife_zero": {
      "host": "128.199.233.228"
    },
    "tags": [

    ]
  },
  "policy_name": null,
  "policy_group": null,
  "run_list": [
    "role[redis-cache-server]"
  ]

}
```

In this case setting `maxmemory-samples` to `40` instead of the `20` we've set in the role definition. After converging, we see the following output:

```
128.199.233.228     - update content in file /etc/redis/redis.conf from
615128 to 22664d
128.199.233.228     --- /etc/redis/redis.conf   2017-01-02
08:33:20.352719091 +0000
128.199.233.228     +++
/etc/redis/.chef-redis20170102-10453-1x9k2bs.conf       2017-01-02
08:37:44.953492480 +0000
128.199.233.228     @@ -28,6 +28,6 @@
128.199.233.228
128.199.233.228        maxmemory 419430400
128.199.233.228        maxmemory-policy allkeys-lru
128.199.233.228     -  maxmemory-samples 20
128.199.233.228     +  maxmemory-samples 40
```

Showing that the value in the node definition has overriden the value in the role definition.

## Breaking a System Into Roles

When you're only working with a single server, it can be hard to see the benefit of roles, but it's worth thinking about how to break the system down into logical re-usable components from the start in order to make it easier when it's necessary to scale onto multiple servers.

This is not just relevant to scaling to add capacity. The benefits start to become apparent even when there's a requirement to add an additional staging or testing server. The production and staging servers simply reference the same roles and we can be confident that we're getting consistent configurations across them.

The benefits become extremely clear when as we begin to break up our system into something closer to a "one service per node" configuration, for example one node as a database server, one node as a web frontend, one node for Redis and so-on.

Each of these single purpose nodes references just the roles applicable to it, whereas staging servers can reference all of the roles. This mix and match approach allows us flexibility with minimal repetition.

On a typical Rails project, we might end up with roles similar to the following:

- `server` - basic configuration to be applied to all servers you manage. For example locking down SSH, ensuring a firewall is enabled, adding public keys for authentication and installing any favourite management tools. This role may also setup your base monitoring configuration
- `ruby` - installs ruby and provides details of where it should be installed and how. For example rbenv user v system install
- `rails-app` - any packages generally required by a rails app, for example ImageMagick headers, nodejs for asset compilation etc
- `postgres-server` - installs postgres and configures authentication methods. Sets up monitoring and alerting for the postgres process.
- `nginx-server` - installs nginx and suitable monitoring

It's worth noting here that roles can include other roles in a run_list. So it's possible to create extremely granular roles and then group these together using other roles which are then included in node definitions.

## Limitations of Roles

One key limitation of roles is that there is no concept of versioning. We'll see in the next section that we can use Berkshelf (like bundler for chef cookbooks) to install we're always using the version of a cookbook we expect. But there is no such equivalent for roles. To illustrate why this could be a problem, consider the following example.

We create a simple role `ruby` which is responsible for installing ruby and setting the default version of Ruby to 1.9.3. We then create a node which uses this role and does not override the default version and install a Rails application on that node which relies on Ruby 1.9.3.

Our organisation moves to Ruby 2.1.1 and so the role is updated to provide Ruby 2.1.1 as the default. We provision several new servers using this role, install Rails applications on them that are designed for Ruby 2.1.1 and all is well.

Subsequently our first server, running Ruby 1.9.3 fails. Luckily we use Chef so setting up a new server is easy and we set up a new vps, run `knife solo bootstrap` and rejoice! But nothing works, we eventually SSH into the server and realise that our role which previously installed Ruby 1.9.3 now installs Ruby 2.1.1.

This is a simple example, it could be remedied simple by requiring that all node definitions explicitly define which Ruby version is required.

But it serves to illustrate the underlying problem that there is no mechanism in place to ensure that the role we were applying has not changed since the last time we applied it. This has the potential to impede the repeatability we're striving towards.

For this reason, some have gone as far as to describe setting attributes in roles as an antipattern, for example http://dougireton.com/blog/2013/02/16/chef-cookbook-anti-patterns/

For many configurations, simply being aware of the limitation and factoring into how we define roles and manage the chef repositories codebase is sufficient.

In the event that is not sufficient, it's possible to use extremely lightweight cookbooks, known as wrapper cookbooks, to set attributes, and then take advantage of the fact they are versioned. Doing this is beyond the scope of this book however there is a good explanation of the process available here http://realityforge.org/code/2012/11/19/role-cookbooks-and-wrapper-cookbooks.html.

# A Template for Rails Servers

## Overview

In the previous sections we've covered how to build a Chef cookbook from the ground up. Hopefully this has shown how simple it is to use Chef to automate, in a reusable manner, pretty much any process we would normally SSH in to complete.

One of the most important things to take from this is that there's no "magic" involved, Chef is just executing the commands we would normally execute by hand. If it can be done in the terminal, we can write a Chef recipe to automate it.

We can maintain complete control of how our stack is provisioned, while at the same time maintaining the convenience and automation of something like Heroku or pre made images which hide the process completely.

In addition to writing our own Chef recipes, there are a wealth of pre-written community recipes available for you to make use of. These range from mainstream components like PostgreSQL or Nginx to community created recipes for installing and configuring Wordpress automatically.

While these recipes are a great resource, I strongly recommend getting comfortable writing your own recipes by hand first.

If we rely entirely on applying recipes written by other people, as soon as we come across something there isn't a pre-made recipe for, the temptation will be to SSH in and do it manually. This is disastrous as when we next come to provision a node using our Chef definition, it won't include this step.

The aim should be to get sufficiently comfortable throwing together a Chef recipe, that it feels as easy to put together a Chef recipe to complete the task as it would be to do it manually.

Secondly, being comfortable with the structure of a Chef cookbook will enable to use third party ones far more efficiently. Most Ruby developers get to a point where they realise taking a quick look at the source of a troublesome gem to see what it's doing under the hood is often better than hours on Stack Overflow.

In the same way when working with Chef, being able to quickly look at the internals of a third party cookbook will usually provide an explanation for any behavior you weren't expecting in just a few minutes and ensure you're getting the most out of them.

## The Example Configuration

In the remainder of this book we'll take as a basis the example chef repository available at:

https://github.com/TalkingQuickly/rails-server-template

The chapter “Quick Start” contains instructions for how to provision a server using this code. The rest of this section assumes you’re working with servers based on this template.

It provides role definitions for each of the following services:

- PostgreSQL
- Redis
- Memcache

So these can easily be swapped in and out.

This sample code is illustrative of how a normal chef project will look, a mixture of mainstream, community cookbooks with a few bespoke ones specific to a particular setup.

# Basic Server Setup

## Overview

In this section we'll look at the basic setup needed to get a server ready for use in a production environment. This includes installing tools which make interacting with the server easier, deciding on which package updates - if any - we want installing automatically, making sure the systems clock is always accurate and ensuring that appropriate locales are installed and configured. Because security is such an important topic, that will be considered separately in the chapter "Security".

A majority of the configuration discussed in this chapter is covered by the "server" role located in `roles/server.json`.

## Unattended Upgrades

Unattended upgrades refer to an Ubuntu package which allow particular package updates to be installed automatically in the background.

More about this process is available here: https://help.ubuntu.com/community/AutomaticSecurityUpdates

### Considerations

Automatic upgrades is something of a controversial topic. On the one hand having package updates installed automatically can ensure that our system remains up to date and secure even if we don't remember to regularly monitor new package releases. On the other hand, it's always possible that a package update will break existing functionality.

My personal preference is to enable automatic upgrades for security related package updates only. Whilst it's completely possible that a security update will have unforseen consequences which breaks existing functionality, in practice it happens very rarely if ever, so in my view the risks associated with running an unpatched system for longer than is necessary is greater than the risk of introducing incomatibility.

### Configuring

Unattended upgrade configuraiton is provide by the OpsCode`apt` cookbook available https://github.com/opscode-cookbooks/apt. This is new functionality for the `apt` cookbook and so at the time of writing, there is no documentation available in the Readme. We can however get a good understanding of the functionality available by looking at the section of the attributes `default.rb` which relates to this functionality:

**https://github.com/opscode-cookbooks/apt/blob/master/attributes/default.rb (extract)**

```
...
default['apt']['unattended_upgrades']['enable'] = false
default['apt']['unattended_upgrades']['update_package_lists'] = true
codename = node.attribute?('lsb') ? node['lsb']['codename'] : 'notlinux'
default['apt']['unattended_upgrades']['allowed_origins'] = [
  "#{node['platform'].capitalize} #{codename}"
]
default['apt']['unattended_upgrades']['package_blacklist'] = []
default['apt']['unattended_upgrades']['auto_fix_interrupted_dpkg'] = false
default['apt']['unattended_upgrades']['minimal_steps'] = false
default['apt']['unattended_upgrades']['install_on_shutdown'] = false
default['apt']['unattended_upgrades']['mail'] = nil
default['apt']['unattended_upgrades']['mail_only_on_error'] = true
default['apt']['unattended_upgrades']['remove_unused_dependencies'] = false
default['apt']['unattended_upgrades']['automatic_reboot'] = false
default['apt']['unattended_upgrades']['dl_limit'] = nil
...
```

In conjunction with the relevant recipe https://github.com/opscode-cookbooks/apt/blob/master/recipes/unattended-upgrades.rb and the template for the main configuration file; https://github.com/opscode-cookbooks/apt/blob/master/templates/default/50unattended-upgrades.erb. Looking through the attribute defaults and then corresponding recipes is a good habbit to get into for many community cookbooks which we'll often find have functionality which is not fully documented in the Readme.

The key configuration values for our purposes are:

#### `['apt']['unattended_upgrades']['enable']`

This determins whether unattended_upgrades will be enabled or not.

#### `['apt']['unattended_upgrades']['allowed_origins']`

This determines which package origins will be automatically updated. Looking at the Ubuntu wiki, it shows that the following values can be used to only allow security updates:

**https://help.ubuntu.com/community/AutomaticSecurityUpdates (extract)**

```
// Automatically upgrade packages from these (origin, archive) pairs
Unattended-Upgrade::Allowed-Origins {
    // ${distro_id} and ${distro_codename} will be automatically expanded
    "${distro_id} stable";
    "${distro_id} ${distro_codename}-security";
    // "${distro_id} ${distro_codename}-updates";
    // "${distro_id} ${distro_codename}-proposed-updates";
};
```

#### `['apt']['unattended_upgrades']['automatic_reboot']`

This determines whether or not the system should be rebooted automatically if this is required to install an update. Setting this to true will inevitably lead to downtime when the system is rebooted to install upgrades, if you set it to `false` then be sure to configure email alerts so that you'll know when a restart is required.

In practice this means that system will be restarted if the file `/var/run/reboot-required` is found after an automatic upgrades run completes. If this option is being used it's also important we're confident that all services will reload of their own accord after a system restart, but this resiliency should be of our underlying goals for this system anyway.

#### `['apt']['unattended_upgrades']['mail']`

This should be an email address which alerts will be sent to if there are problems with automatic updates. If an email address is provided then emails will only be sent if there is a working mail configuration on the node.

### The example configuration

The example configuration sets the following values for unattended upgrades in the `server` role.

**roles/server.json (extract)**

```
...
"default_attributes": {
    "apt" : {
      "unattended_upgrades" : {
        "enable" : true,
        "allowed_origins" : [
          "${distro_id} stable",
          "${distro_id} ${distro_codename}-security"
        ],
        "automatic_reboot" : true
      }
    },
...
```

Which enables automatic upgrades for security updates only and allows automatic system reboots if these are required.

## Automatic System Time via NTP

The `server` role includes `"recipe[ntp::default]",` in its run list. This has the effect of installing NTP which will periodically synchronize the system clock with a remote time server.

There are several areas where the system time is particularly important. The simplest of these is logging. Particularly when debugging problems involving multiple machines, it's extremely useful to know that the timestamps on the log entries will match across machines to an accuracy of less than a second.

## Locales

Locales are used to customize the behavior of a piece of software to a particular language or country. This ranges from defining character encodings to interface languages and default currency symbols.

### Why Locales Matter

To use a specific example, if we are using Postgres as our database provider, it will use the systems locale when it creates the initial database cluster. The same is true when we create new databases. Whilst some locale related settings can be changed after database creation, some cannot be. It's beyond the scope of this book to look at how locale impacts the behavior of each component of the server but it's important to understand that if we are using different locales in development and production, our systems are not identical and will behave differently in some respects.

Another more visible area in which locales will often surface is when interacting with the remote server via SSH. By default on OSX and many other systems, when we connect to a remote server via SSH, it sets a number of environment variables prefixed with `LC_*` which have the effect of setting the remote server to use the same locale as the client machine for that SSH session.

To give an example scenario where this may be problematic:

- A server is setup with only the locale `en_US.utf8`
- Our locale development machine is setup with the locale `en_GB.utf8`

When we connect to the remote server, if the `LC_*` environment variables are being set, then it will try to set the servers locale for that session to one which isn't installed (`en_GB.utf8`) and we'll see an error or warning message. Furthermore if we then start a Postgres shell to create a database, Postgres will pickup the invalid `LC_*` variables and raise warning messages about the invalid locale configuration.

One solution to this is to disable the sending of `LC_*` variables when establishing SSH sessions. The process for doing this varies depending on your client, for the terminal app in OSX the option is called "Set locale environment variables on startup" in the advanced menu. Or you can modify the OpenSSH client configuration by removing the following line:

```
SendEnv LANG LC_*
```

From ~/`.ssh/config`.

Another solution is to simply install the additional locales you require on the remote server. This is supported using the example configuration and documented below however if doing this, it's important to bear in mind that when executing commands via SSH, it's possible the locale being used is not the default one.

## Configuring

The primary cookbook concerned with locations is: `"recipe[locales::default]"` in the run list of the `server` role. This is responsible for installing and setting the systems active locale. It is configured in the `server` role as follows:

**roles/server.json (extract)**

```
...
"locales" : {
      "packages" : ["locales"],
      "default" : "en_US.utf8"
    },
...
```

This defines the package(s) which provide locale functionality, for Ubuntu this is always `locales`. It then allows us to choose the locale which should be installed and set as the default locale.

# Security

## Overview

In this chapter we'll cover some basic steps which can be taken on all servers to help protect against unauthorised access. We'll also look at common pitfalls when securing production systems and how to avoid them.

## No Magic Bullet

Depending on the type of application, the environment we're hosting in and the regulations associated with the releveant industry, security requirements will vary.

There is no one secure server configuration which works for everyone. In this section we'll cover the basic steps which can be applied to all new servers to provide a base layer of security. This is unlikely to be exhaustive for any one application.

These are basic security tips, it's essential you do your own research and if necessary engage an independent security expert to ensure the measures you're taking are suitable for your requirements.

## Common Pitfalls

When we talk about security, it brings to mind firewalls and auto banning brute force attacks. In practice many, potentially most security breaches are due to not following simple security best practices rather than server configuration errors, in particular:

### Mistake 1 - Not Updating Gems

Security updates are regularly issued for gems, it's a pain staying up to date. Not doing so can be fatal. As a minimum we shoudl be subscribed to receive security advisories for the Rails gem. If a major vulnerability is discovered within Rails, it's essential you have a plan in place to upgrade within days if not hours of this being released.

There has been at least one core Rails vulnerability in recent memory which affected almost all production Rails applications and was trivial to exploit and scan for. Within hours of the vulnerability being announced, log files showed a high volume of people running automated scans to detect and exploit the vulnerability. It's fair to say that almost anyone who did not apply the update and has a live public facing site, either already has or will be compromised.

## Mistake 2 - Hard coding credentials

As an app grows and the number of people who have or have had access to the source code grows, the danger of having hard coded login credentials becomes clearer.

Common culprits are:

- AWS Keys
- Mail server Passwords
- Error logging services
- Database logins

Your development repository should contain no production credentials and it's worth scouring initializers to make sure none remain. All credentials should be moved to standalone config files or environment variabels which are maintained independently on production, staging and development environments. The basics of this are covered when we look at deploying with Capistrano.

It's also worth noting that "use environment variables" is not a magic bullet. Whilst it's good practice to abstract credentials into environment variables which are then read in at run time, the loading of these environment variables still has to take place at some point. If, for example, `dotenv` or some equivilent is being used to set the environment variables from a text file of some sort, then equal care must be taken over who has access to the production version of this text file and whether it ever exists in version control.

Remember even once these are removed, a record will still exist in version control so any passwords which have been hard coded should be changed. Alternatively it's possible to purge all references to a scpecific file from the history of most most version control systems.

## Mistake 3 Re-using Passwords

Combined with hard coding credentials, re-using passwords this can be fatal. I've seen hard coded mail server credentials which were identical to the VPS providers login credentials.

It's also tempting to use identical credentials across production and staging environments, this is bad for two main reasons:

- It means anyone you give access to staging, can access production. There may well be times (such as trialling a contractor) when you want to give someone access to a staging environment but aren't yet ready to give access to production.
- It makes it far easier to accidentally make a change on production when you thought you were working on staging. Anyone who's had a near miss and started typing "rake db:drop" on production thinking they were on staging knows this is bad for the blood pressure.

## SSH Hardening

### Considerations

#### Password Authentication

By default when you use `ssh user@yourserpip` authentication will first be attempted by public key. This means looking at your private key, by default in `~/.ssh/id_rsa`, and checking to see whether there is a matching public key in `~/.ssh/authorised_keys` on the remote machine.

If there is not, you will then be prompted to enter the password for the user you are trying to connect as.

One of the first steps I take on any new server is to disable password login via ssh. Anyone whos run a publicly available sever and has taken a look at the logs will have seen the regular and persistent attempts to brute force ssh login. Disabling it removes this attack vector completely.

It's easy to think that the risk is minimal anyway if suitable long and unique passwords combined with fail2ban are used. Unfortunately it only takes one test account to be created with a weak or default password and the forgotten about to expose your server to significant risk of intrusions.

#### Root Access

Whilst disabling password authentication for SSH access is fairly uncontroversial, whether or not SSHing in directly as `root` should be allowed seems to be a topic which will split most rooms (of devops folk, most other audiences seem bemused by the question).

My personal view is that as long as password authentication is disabled for SSH access, there is very little, if any, additional protection offered by disabling root access. But your milage may vary.

### Configuration

SSH configuration is managed by the opscode OpenSSH cookbook available at https://github.com/opscode-cookbooks/openssh. The default configuration in the `server` role is as follows:

**roles/server.json (extract)**

```
...
"openssh" : {
  "server" : {
    "password_authentication" : "no",
    "challenge_response_authentication" : "no",
    "permit_empty_passwords" : "no",
    "use_pam" : "no",
    "x11_forwarding" : "no",
    "permit_root_login" : "yes"
  }
...
```

Which should be fairly self explanatory based on the considerations above.

## Firewall

The firewall is responsible for determining which ports are accessible to which machines on which network interfaces. It can be thought of as our first line of defence, when a connection attempt of any kind comes through, it must first be allowed through the firewall.

### Considerations

In this configuration we'll use `ufw` which stands for "Uncomplicated Firewall" which is a friendly layer on top of the very well known iptables. The approach taken here is a fairly traditional "by default allow nothing except SSH".

An important component of this is stopping to think what the minimum level of access required is when adding exceptions to this general rule. For example it's common - but a very bad idea - for people to publicly expose their database server so that they can connect to it using a local management tool. In practice almost all database management tools support connecting via an SSH tunnel so there's no need to publicly expose the database if SSH access is available.

Another example is multi machine configurations. If we take a simple example where we have a web frontend (web1) on one VPS and a database server (db1) on another. It may be tempting to simply setup db1 to allow all connections from web1, irrespective of port or interface. It's worth first considering:

- Does db1 really need to allow connections on all ports or just the port the database server is listening on?
- Does db1 really need to allow connections on all interfaces? Many VPS providers include a private network connection between VM's on the same account, in which case we should do all of our database communication over this private interface and only allow connections from the private IP of web1.

Being cautious from the start about the ports and origins allowed through the firewall help us to maintain a minimal attack surface as our deployment grows.

### Configuring

Firewall configuration is managed by the community `ufw` cookbook available at https://github.com/opscode-cookbooks/ufw. By default the entry `"recipe[ufw::default]"` in the run list of the `server` role will install UFW and configure it to allow only SSH traffic on port 22.

An example of allowing an additional port through the firewall can be seen in the `nginx` role:

**roles/nginx-server.json (extract)**

```
...
"default_attributes": {
  "firewall" : {
    "rules" : [
      {"allow http on port 80" : {"port" : 80}}
    ]
  }
...
```

UFW rules are contained in array made up of hashes in the format:

**UFW rule format**

```
{
        "description" : {
                "port" : 80,
                "source" => "192.168.1.45",
                "action" => "allow",
                "protocol" => "tcp"
        }
}
```

The UFW cookbook Readme https://github.com/opscode-cookbooks/ufw contains a more complete reference to the options available.

### Attribute Precedence

As seen above, the nginx role defines the rule for port 80 within `default_attributes`. There's an importance nuance here around what what will happen if we attempt to define firewall rules in both roles and node definitions. If we define attributes at role level, in `default_attributes`, then the arrays containing the rules from each role will be merged.

If we were to then define additional rules at node definition level, we might expect that these would be merged with the rules defined in the roles. This is not however what will happen. Because attributes at node definition level override attributes set at default level, the rules defined at node level will replace all of those defined and role definition level.

In practice this is quite intuitive, otherwise it would become very confusing trying to keep track of what was coming through from roles and what was being defined at node level. My personal preference is to use roles for single node configurations and then define everything at node level for multi host configurations.

## Debugging

UFW logs every connection which is blocked to `/var/log/ufw.log`, tailing this with:

**terminal**

```
sudo tail -f /var/log/ufw.log
```

can avoid many hours of frustrating debugging of issues caused by systems not being able to communicate as expected.

## Users

User management is provided by the OpsCode users cookbook https://github.com/opscode-cookbooks/users. This cookbook is included by the `server` role (`recipe[users::sysadmins]`) which allows us to add users in the sysadmins group to the system via the `users` databag.

### Managing Users

Each user is defined by a json file in the users databag (`data_bags/users`). A definition looks like this:

**data_bags/users/deploy.json**

```
{
  "id": "deploy",
  // generate this with: openssl passwd -1 "plaintextpassword"
  "password": "$1$jil6kjrT$iZ5o0DkBc8rBOljXxo.4j1",
  // the below should contain a list of ssh public keys which should
  // be able to login as deploy
  "ssh_keys": [
    "rsa_public_key"
  ],
  "groups": [ "sysadmin"],
  "shell": "\/bin\/bash"
}
```

The `sysadmins` recipe which is included in the `server` role will search for any user in the `sysadmin` group and add them to the system.

A users password should be generated using the `openssl` utility:

```
openssl passwd -1 "plaintextpassword"
```

Whilst this password will not be needed for SSH access, since password based SSH authentication should be disabled, it will be needed if we choose to require users to enter a password when executing commands with `sudo` (see the section below).

## Sudo

The sudo utility allows regular users to execute commands as root. Which users have access to sudo and the behaviour of the sudo utility is managed by the Sudo cookbook; https://github.com/opscode-cookbooks/sudo and included in the server role (recipe[sudo::default]).

### Considerations

One of the main considerations when configuring sudo access is whether or not to require a user to enter their password to execute commands as root.

The arguments in favour of requiring this are compelling. If we imagine that we're running a rails application as the user deploy and our Rails app is in some way compromised such that an attacked can execute arbitrary system commands. If a password is required for sudo, whilst the attacked can still do a lot of damage, the damage is limited to actions which can be performed by the unprivileged deploy user.

If however no password is required for sudo access, an attacker exploiting the same vulnerability would be able to execute arbitrary commands as root. Making their access to the system effectively unlimited.

The primary reason for considering not requiring a password for sudo access on our system is that Capistrano, the deployment tool we'll use in part two of this book, does not work well without passwordless sudo enabled. It is possible to be far more granular with which commands can be executed with sudo with or without a password however I have never been able to come up with a satisfactory set of rules which were in reality any more secure than simply allowing passwordless sudo.

In general for a single tenanted system, I allow passwordless sudo, for multi-tenant systems I do not. As discussed earlier, whether or not the risks associated with passwordless sudo are acceptable is entirely dependent on the specific requirements of the application being deployed.

### Configuration

The sample configuration provides the following default values for the sudo cookbook in the server role:

**roles/server.json (extract)**

```
"authorization": {
  "sudo": {
    // everyone in the group sysadmin gets sudo rights
    "groups": ["sysadmin"],
    // the deploy user specifically gets sudo rights
    "users": ["deploy"],
    // whether a user with sudo rights has to enter their
    // password when using sudo
    "passwordless": true
  }
}
```

This should be fairly self explanatory. First we define the groups which should be allowed to use `sudo`. In this case everyone in the `sysadmin` group which is the group to which all users we create in the previous section belong, are given sudo access. In addition we can specify particular users who, irrespective of the group they belong to, will be given sudo rights.

Finally the `passwordless` attribute determines whether or not the user has to enter their password to execute commands with `sudo`. If it is `true` then the user can execute commands using `sudo` without entering their password.

# Ruby & Gem Dependencies

## Overview

In this chapter we'll look at installing components specific to running Rails applications, in particular Ruby and packages needed to build native extensions for commonly used gems.

It's extremely important that our production environment is running the same version of Ruby as our development environment. If this is not the case we're likely to run into tough to debug errors and unpredictable behavior.

Generally the Ubuntu package repository contains a fairly old version of Ruby. One solution to this is to use third party repositories to update the system version of Ruby to match your local version.

Whilst this can work I prefer to make use of tools which are designed for easy switching of Ruby versions. The key benefit of this is that it's possible to have multiple versions installed alongside each other and switch quickly back and forth between them.

This is a particular benefit if you perform an update and discover an unexpected issue. Rather than having to completely rollback the installation, you can simply switch back to the previous version.

## Rbenv v RVM

'My tool is better than your tool' tends not to add much to development discussions. RVM and Rbenv both have their advantages and disadvantages.

The official rbenv page on why to choose rbenv over rvm is here:

https://github.com/sstephenson/rbenv/wiki/Why-rbenv%3F

In a nutshell I prefer rbenv because I find its operation simple to understand and therefore simple to troubleshoot. As a result it's rbenv which is covered in this book, if you're keen to use rvm, there are several good third party cookbooks for installing it so adapting the template to use one of these shouldn't be too complex.

## How rbenv works

### $PATH

To understand rbenv we first need to understand $PATH. At first $PATH seems quite intimidating, it's constantly referred to in troubleshooting threads and Stack Overflow answers ("ahhh, it's obviously not in your $PATH") but with very little explanation.

$PATH is actually very simple. If you type "echo $PATH" at your local terminal, you will see something like the following:

```
/Users/ben/.rvm/gems/ruby-1.9.3-p286-falcon/bin:/Users/ben/.rvm/gems/ruby-1.9.3-p\
286-falcon@global/bin:/Users/ben/.rvm/rubies/ruby-1.9.3-p286-falcon/bin:/Users/be\
n/.rvm/bin:/Users/ben/.rvm/gems/ruby-1.9.3-p286-falcon/bin:/Users/ben/.rvm/gems/r\
uby-1.9.3-p286-falcon@global/bin:/Users/ben/.rvm/rubies/ruby-1.9.3-p286-falcon/bi\
n:/Users/ben/.rvm/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/bin:/usr/sbin:/sbi\
n:/usr/local/bin:/opt/X11/bin:/Users/ben/.rvm/bin
```

We can see that this is just a list of directories, separated by colons. As you can see, on this machine I'm using rvm so I have a lot of rvm specific directories in mine. If you were running rbenv you might see something like this:

```
/usr/local/rbenv/shims:/usr/local/rbenv/bin:/usr/local/sbin:/usr/local/bin:/usr/s\
bin:/usr/bin:/sbin:/bin:/usr/bin/X11:/usr/games
```

When you enter a command in the terminal, such as "rake" or "irb," in the console, the system searches through each of the directories in $PATH, in the order they are displayed, for an executable file with that name.

## What Rbenv is doing

As you can see in the second output above, rbenv has added a directory to the beggining of my $PATH variable:

```
/usr/local/rbenv/shims
```

Since $PATH is evaluated from left to right, this means that when I enter a command such as bundle or ruby, the first directory it will look in is `/usr/local/rbenv/shims`. If it finds a matching command there, it will execute it.

So if, for example, there is an executable file with the name "bundle" in this directory, it will execute it.

If we look at the directory listing (`ls /usr/local/rbenv/shims`) we see:

```
-rwxr-xr-x 1 root root  393 Apr 17  2013 bundle*
-rwxr-xr-x 8 root root  393 Apr 17  2013 erb*
-rwxr-xr-x 8 root root  393 Apr 17  2013 gem*
-rwxr-xr-x 8 root root  393 Apr 17  2013 irb*
-rwxr-xr-x 8 root root  393 Apr 17  2013 rake*
-rwxr-xr-x 8 root root  393 Apr 17  2013 rdoc*
-rwxr-xr-x 8 root root  393 Apr 17  2013 ri*
-rwxr-xr-x 8 root root  393 Apr 17  2013 ruby*
-rwxr-xr-x 8 root root  393 Apr 17  2013 testrb*
```

If we were to look at each of these files (e.g. `cat /usr/local/rbenv/shims/ruby`) we'd see that each one is just a simple shell script. This shell script performs some processing on the input and then passes our command back to rbenv to process.

Rbenv will then determine which ruby version to execute the command with from the following sources, in order:

1. The RBENV_VERSION environment variable
2. The first .ruby-version file found by searching the directory which the script being executed is in, and all parent directories until reaching the filesystem root
3. The first .ruby-version file found by searching the current working directory and all parent directories until reaching the filesystem root
4. ~/.rbenv/version which is the users global ruby. Or /usr/local/rbenv/version if it is a system wide install.
5. If none of these are available, rbenv will default to the version of Ruby which would have been run if rbenv were not installed (e.g. system ruby)

Actual ruby versions are installed in `~/.rbenv/versions` or, if you've installed rbenv system wide rather than on a per user basis (more on this later) in `/usr/local/rbenv/versions`. Version names in the sources above are simply names of folders in this directory.

We'll be installing both rbenv and ruby versions using a chef recipe below but for more on using rbenv in your development environment, see the official documentation which is excellent and available at https://github.com/sstephenson/rbenv

## The rbenv Cookbook

We'll be using fnichols excellent `chef-rbenv` cookbook which will be in your `cookbooks/chef-rbenv` folder.

In our `rails-app.json` role we include the rbenv::system recipe which contains the following default attributes:

**roles/rails-app.json (extract)**

```
"default_attributes": {
        "rbenv":{
          "rubies": [
            "2.1.2"
          ],
          "global" : "2.1.2",
          "gems": {
            "2.1.2" : [
              {"name":"bundler"}
            ]
          }
        }
```

The rbenv::system performs a system wide install of rbenv, this means that it's installed to /usr/local/rbenv rather than ~/rbenv. This is generally my preference as it reduces issues caused by, for example, cron jobs running as root.

The `rubies` attribute contains an array of ruby versions to be installed and made available. If you're planning on hosting multiple applications on the same server, each of which requires a different ruby version, you can specify them here and ensure the correct one is used by specifying it in a .ruby-version file in your project's root directory.

The `global` attribute specifies the global system ruby (`/usr/local/rbenv/version`) which rbenv will fall back to as per the hierarchy defined above. This should be one of the rubies specified in the previous array.

The `gems` attribute should contain a key for each of rubies being installed. This should contain an array of gems to be installed for that version of ruby. In general I install just bundler and allow each applications deployment process to take care of installing its specific gems (covered in section 2 of this book).

## Gem Dependencies

The `rails-server` role also includes `recipe[rails_gem_dependencies-tlq::default]` which is a very simple cookbook I created to install packages required when compiling native extensions for common gems.

The recipe itself is extremely simple:

**rails_gem_dependencies-tlq/recipes/default.rb**

```ruby
package 'curl'
package 'libcurl3'
package 'libcurl3-dev'
package 'libmagickwand-dev'
package 'imagemagick'
apt_repository("node.js") do
  uri "http://ppa.launchpad.net/chris-lea/node.js/ubuntu"
  distribution node['lsb']['codename']
  components ["main"]
  keyserver "keyserver.ubuntu.com"
  key "C7917B12"
end
package 'nodejs'
```

It installs some standard packages then adds a ppa which provides an up to date version of `nodejs` which we can use as our javascript runtime when compiling assets on the remote server.

# Monit

It is inevitable that unexpected behaviors will occur which will cause some processes to fail. This could be anything from our database process to nginx to the application server or background job workers.

Where possible we want the system to take care of itself, to detect that a process has either failed or is performing incorrectly and restart it accordingly. Where this is not possible, we want to be alerted immediately so we can take action to rectify it.

Monit allows us to do just this, it can monitor system parameters such as processes and network interfaces. If a process fails or moves outside of a range of defined parameters, Monit can restart it, if a restart fails or there are too many restarts in a given period, Monit can alert us by email. If needed, using email to SMS services, we can have these alerts delivered by text message.

In the example template, Monit is managed by two cookbooks; `monit-tlq` and `monit_configs-tlq`.

## Monit-tlq

This cookbook takes care of installing and configuring Monit and contains only one recipe `default`.

It begins by installing the Monit package and updates Monits main configuration file `/etc/monit/monitrc` from the template `monit-rc.erb`.

As usual the attributes are documented in `attributes/default.rb` but it's worth looking at the configuration file to see just how simple Monits setup is:

**monit-tlq/templates/default/monit-rc.erb (excerpt)**

```
set daemon <%= node[:monit][:poll_period] || 30 %>
```

This sets the interval in seconds which Monit will performs its checks at. In Monit terminology this is known as a cycle, this is important because when we're defining configurations for individual processes we want to monitor, we'll specify many parameters in terms of number of cycles. A good starting point for most systems is between 30 and 120 seconds.

**monit-tlq/templates/default/monit-rc.erb (excerpt)**

```
set logfile syslog facility log_daemon
```

This line tells Monit to log error and status messages to `/var/log/syslog`.

**monit-tlq/templates/default/monit-rc.erb (excerpt)**

```
<% if node[:monit][:enable_emails] %>
  <% if node[:monit][:notify_emails] %>
    <% node[:monit][:notify_emails].each do |email| %>
      <% if node[:monit][:minimise_alerts] %>
set alert <%= email %> but not on {instance, pid, ppid, resource}
      <% else %>
set alert <%= email %>
      <% end %>
    <% end %>
  <% end %>

  <% if node[:monit][:mailserver] %>
  set mailserver <%= node[:monit][:mailserver][:host] %> port <%= node[:monit][:m\
ailserver][:port] %>
    username "<%= node[:monit][:mailserver][:username] %>"
    password "<%= node[:monit][:mailserver][:password] %>"
    using tlsv1
    with timeout 30 seconds
    using hostname "<%= node[:monit][:mailserver][:hostname] %>"
  <% end %>
<% end %>
```

This section sets up a mailserver and the users which should be alerted when appropriate conditions are met. We'll examine the two variations on the `set alert` lines in more detail later but in short, the addition of `but not in {instance, pid}` prevents emails from being sent due to events which usually don't require any manual intervention.

**monit-tlq/templates/default/monit-rc.erb (excerpt)**

```
set httpd port 2812 and
    use address localhost
    allow localhost
    allow <%= "#{node[:monit][:web_interface][:allow][0]}:#{node[:monit][:web_int\
erface][:allow][1]}"%>
```

Here we set up Monits web interface which allows us to view system status and manually start and stop processes where appropriate.

The username and password (basic auth) are set in the final allow line. By default in the configuration above, the web interface is bound to localhost (e.g. not externally accessible). While it is possible to add additional allow lines to have the web interface accept connections from additional IP's or ranges and then allow these connections through the firewall, it is not recommended.

The recommended approach to making the web interface externally accessible is to proxy it through Nginx, the process for this is covered later in this chapter.

**monit-tlq/templates/default/monit-rc.erb (excerpt)**

```
include /etc/monit/conf.d/*.conf
```

Finally we tell Monit to include any other configuration files in `/etc/monit/confi.d/*.conf` - note the `.d` convention meaning that it's a directory.

Rather than having a single monolithic configuration file for everything we want to monitor, we'll have an individual config file for each of our services we wish to monitor. This modular approach has the benefit of making it easy to debug the monitoring of a particular service as well as making re-use of config files across servers with different combinations of services much easier.

## Which configuration goes where

The processes and services we're going to monitor fall into two categories; system components and app components.

System components are those which are installed when provisioning, for example Nginx, our database, Redis, memcached and ssh.

App components are generally processes specific to the Rails app(s) we are running on the server. Examples of these include our app server (unicorn) and background workers such as Sidekiq.

The rule followed for the sample configuration is that a components monitoring should be managed at the time it is added to the server. Specifically if chef is used to install something (generally system components) then a suitable Monit configuration should be added by chef. If on the other hand a component is added via Capistrano (Rails apps and background workers) then the Monit configuration should be defined within the app and managed from Capistrano with the rest of the apps resources.

Monit configurations for system components can be found in the various recipes contained in the cookbook `monit_configs-tlq`. Monit configs for app components are covered in the second section of this book.

## The importance of a custom monitoring configuration

Whilst my Monit configurations are a good starting point, the monitoring requirements will vary based the applications specific uptime requirements. I suggest you fork my recipes and customise them to suit your own requirements.

## System level monitoring

Out of the box, Monit can keep track of load averages, memory usage and cpu usage and alert you when they move outside of certain ranges.

Take as an example the below `system` definition from `monit_configs-tlq`.

**monit_configs-tlq/templates/default/system.conf.erb**

```
check system localhost
  if loadavg (1min) > 4 then alert
  if loadavg (5min) > 3 then alert
  if memory usage > 75% then alert
  if cpu usage (user) > 70% for 5 cycles then alert
  if cpu usage (system) > 30% for 5 cycles then alert
  if cpu usage (wait) > 20% for 5 cycles then alert
```

The first line `check system localhost` tells Monit to, on each cycle (which we defined earlier as a period of time), perform each of the checks listed below it.

The first three are quite simple, they perform a check, if the criteria are met then "alert" is called which means an email will be sent as per your alert configuration in the main Monit config.

The second three demonstrate the addition of another criteria `for 5 cycles`. This means that Monit will only perform the specified action (in this case alert) if the conditions are met for 5 checks in a row. This allows us to avoid being alerted to brief operation as normal spikes, instead receiving alerts only if a spike continues for an extended amount of time.

## Load Average

The load average refers to the systems load, taken over three time periods, 1, 5 and 15 minutes. You may well recognised these as they're displayed on many server control panels as well as on utilities such as `top`.

Intuitively, it might seem that a load average of 1.0 is perfect, that the system is loaded to exactly its maximum capacity. In practice this is a dangerous position to be in as there is no headroom, if there is any additional load, you'll start to see slow downs. A 15 minute load average of 0.7 on a single core server is a good rule of a thumb for the `maximum` before you should start to look at reducing the servers load or upgrading it. A 1.0 load averaging on a single core server needs investigating urgently and anything above 1.0 means there is a problem.

On systems with more than one core, we can roughly multiply your maximum acceptable load by the number of cores. So on a 4 core system a load average of 4.0 would be 100% load, 3.0 would be 75% load etc.

So on a 4 core system, we might choose to have the following line:

```
if loadavg (15min) > 2.8 then alert
```

Which would tell Monit to alert us if 15 minute load average was above 2.8 for a single cycle.

## Memory Usage

This one is fairly simple, Monit can alert us when the servers memory usage exceeds a certain threshold. Nothing will kill a servers performance faster than swapping so there should always be some available RAM.

A rule of thumb is that more than 70% - 80% memory usage on an ongoing basis is an indicator the server either needs uprating or some processes moving off it.

This would be reflected by the following Monit line:

```
if memory usage > 75% then alert
```

Or, to reduce the number of alerts caused by brief spikes, the following could be used

```
if memory usage > 75% for 20 cycles then alert
```

Which would only alert us if it was above 75% for 20 cycles. If our interval is configured as 30 seconds, this would alert us if memory usage was above 75% continually, for 10 minutes.

## Monitoring Pids

Monitoring based on pid forms the bulk of the monitoring on most production servers. Every process on a Unix system is assigned a unique pid. Using that pid, information such as the processes memory and cpu usage can be determined.

A typical Monit definition for monitoring a pidfile might look like this:

```
check process nginx with pidfile /var/run/nginx.pid
  start program = "/etc/init.d/nginx start"
  stop program = "/etc/init.d/nginx stop"
  if 15 restarts within 15 cycles then timeout
```

This simple definition tells Monit to check a process called Nginx (this is just a human readable name of the service and can be anything) based on value in the pidfile at /var/run/nginx.pid.

The pidfile model is extremely simple. When a process starts, it will create a file - its pidfile - in a known location containing the pid it was allocated. Other applications which need to interact with the process can simply query that file, to find the pid allocated to it.

Additionally if there is no pidfile, it's an indicator that the process may not be running. This is not however conclusive, since a pidfile is just a standard file, it's entirely possible for the file not to be written due to permission issues or for the file to have been subsequently deleted.

This is a common cause of hard to debug errors; where a process is running without a pidfile. In this scenario our monitoring may well try and start the process in question, assuming it has failed. If for example this is a database server which binds to port 3306, we may then find a large number of failed attempts to start the server with an error that the port is already in use.

Returning to our simple Monit definition above, in addition to the name and pidfile location, we also define a start command and a stop command. On each check, if process is not found to be running, Monit will execute the `start` command.

This is extremely powerful. As long as we can find a pidfile for an application and define a command which can be used to start it, Monit can be used to check that it is running and if not attempt to start it.

As we'll see below, we can also have Monit alert us to changes in such processes so we know when manual intervention is required or likely to be required.

The final line in our simple definition above is this:

```
if 15 restarts within 15 cycles then timeout
```

This line means that if there have been 15 attempts to restart a process in the last 15 cycles, then stop trying to restart it. This is to deal with scenarios where it is clear that the process is not going to start without manual intervention.

This is particularly important if the start process which is failing involves brief periods of intense CPU usage. Were this qualifier not there, our startup script would be called on every cycle indefinitely leading to extremely high CPU usage and potentially causing the rest of the system to fail as well or at least slow down dramatically.

## Finding Pidfiles

Finding the pidfiles for applications can be something of an art form.

The first place to look is the configuration files for the application in question. Often the configuration allows the pids location to be specified and includes a default value. Furthermore not all applications will generate a pidfile by default, for some this will be an option which will have to be enabled.

If there is no mention of the pidfile in the application, the next place to look is `/run` which is the "standard" location for pidfiles in Ubuntu (previously /var/run `which is, as of 11.10 now just a symlink to `/run‘

When specifying pidfile locations, /run is a good bet however see the section below on pidfile permissions.

## Pidfile Permissions

When specifying a pidfile location in a config file, be mindful of permissions. A pidfile is a file like any other, therefore the user who creates it, must have write permissions to the relevant part of the filesystem.

If the application in question runs as a specific user, be sure to double check that the user has write access to the path you're specifying. If this is not the case, the application may fail to start or simply log the error and continue as if nothing has happened.

A common source of this error is a work flow like the following:

- a process is to be run as a none root user and so a sub directory in /var/run is created and the relevant user given write access.

- In initial tests this works fine and the configuration is flagged as working
- The server is restarted and suddenly the pidfile can't be written and Monit can't find the service and sometimes the service itself will not start.

The problem here is that whilst /run is simply a part of the filesystem, it's actually a mounted tmpfs. What this means is that the contents of /run are never persisted to disk - they are stored in RAM. On reboot, the contents is lost.

Therefore after a restart, the folder which was created with appropriate permissions in the initial setup, will no longer exist.

There are various approaches to solving this problem, the simplest is to ensure that our applications startup scripts (such as those in /etc/init.d/) include logic to check for the existence and permissions of pidfile target locations and if these are not presents, creates them.

An example of such logic is the following:

```
# make sure the pid destination is writable
mkdir -p /var/run/an_application/
chown application_user:application_user /run/an_application
```

## Monitoring Ports

In addition to checking the status of processes, Monit can check whether connections are being accepted on particular ports. So we could expand out Nginx definition above to the following:

```
check process nginx with pidfile /var/run/nginx.pid
  start program = "/etc/init.d/nginx start"
  stop program = "/etc/init.d/nginx stop"
  if 15 restarts within 15 cycles then timeout
  if failed host 127.0.0.1 port 80 then restart
```

The additional line `if failed host` means that on each cycle, Monit will attempt to establish a connection 12.0.0.1:80. If a connection cannot be established, then it will attempt to restart the process.

Here we cam see that a restart command is available even though we've only defined start and stop commands. Restart simply calls the stop and start commands sequentially. At time of writing there was no option to specify a separate restart command but its been slated as an upcoming feature for a while so this may change.

## Free Space Monitoring

An often overlooked factor in server health is the amount of available free space. Particularly now that disk space is so cheap, it's easy to forget that it's still entirely possible to run out. Once your production database server has run out of space once, it's unlikely you'll decide against including checks for this again.

A simple Monit check for available disk space might look like this:

```
check filesystem rootfs with path /
  if space usage > 80% then alert
```

This is fairly self explanatory, the filesystem is checked and if the space in use is over 80%, an alert is sent.

## Alerts and avoiding overload

In our original Monit configuration at the start of the chapter we had the following line:

```
set alert <%= email %>
```

which translated to:

```
set alert user@example.com
```

This is what's known as a global alert. By default Monit will send alerts to this address whenever anything that is being monitored (a service) changes, including:

- A service which should exist, does not exist
- A service which didn't exist, starts existing
- The pid of a service changes between cycles
- A port connection fails

A default catch all alert statement like this can generate a lot of email traffic. If, for example you're monitoring 5 unicorn workers, every time you deploy, you'll receive at least 5 notifications from Monit to tell you that the the pids of all 5 unicorn workers have changed.

The danger is that receiving Monit alerts will become so commonplace, that they get a similar treatment to spam, when one is received, the subject is glanced at and then the email archived. This makes it very easy to miss an important alert.

It is therefore worth spending some time tuning our alerts, starting off with a bias towards alerting too much and then regularly reviewing over the first few weeks of operation to tune out alerts for events which we do not need to know about.

Our sample Monit configuration from the beginning of this chapter included the following alert definition:

```
set alert <%= email %> but not on {instance, pid}
```

This line was included when the `minimise alerts` flag is set on the node definition.

This means that globally alerts will be sent for all events except for instance and pid changes. Whilst I strongly recommend you tune your own configuration, in my experience the above is often sufficient to minimise the amount of alert traffic while ensuring critical events are still sent.

The Monit documentation on managing alerts is excellent and well worth reading:

http://mmonit.com/monit/documentation/monit.html#alert_messages

## Serving the web interface with Nginx

### Using the example template

Monit provides a web interface which displays the current status of all monitored processes and allows us to restart each of them as well as to manually start processes for which automatic restart has failed.

In our example configuration at the start of this chapter it is configured to run on port 2812 and be accessible only to localhost.

It's possible to have Monit serve the web interface to other IP's directly however it's preferable to have all web traffic served from NGinx.

To do this we would add an Nginx virtual host similar to the below:

```
server {
  listen 80;
  server_name monit.example.com;
  location / {
    proxy_pass http://127.0.0.1:2812;
    proxy_set_header Host $host;
  }
}
```

This simply means take all requests for `monit.example.com` and send them to `127.0.0.1`on port `2812`.

### Serving multiple Monit interfaces from one nginx interface

An additional benefit of this is that we can use a single Nginx instance to serve the Monit interface from multiple machines. For example we could establish a convention <br/>`machine-name.monit.example.com` always points to the Monit interface for `machine-name`. We could then specify in our Monit config files that the admin interface on 2812 is only accessible to the private (internal) IP address of the Nginx instance we're using to serve Monit interfaces.

If the private IP address of the additional server we're monitoring was `168.1.1.5` then our Nginx virtualhost would look like this:

```
server {
  listen 80;
  server_name machine-name.monit.example.com;
  location / {
    proxy_pass http://127.0.0.1:2812;
    proxy_set_header Host $host;
  }
}
server {
  listen 80;
  server_name machine2-name.monit.example.com;
  location / {
    proxy_pass http://168.1.1.5:2812;
    proxy_set_header Host $host;
  }
}
```

The above definition show that Nginx is serving both the Monit interface for itself (on 127.0.0.1) and for the second server on `168.1.1.5`.

# nginx

## Overview

Nginx is (among other things) a high performance HTTP server and reverse proxy. When a web request comes into the server and has been allowed through the firewall, it first reaches nginx. If the request is for a static asset, such as an image or css file, nginx can serve this very quickly and directly, without ever touching our Rails application.

If on the other hand the request is for a part of our Rails application, Nginx proxies this request back to our application server. In development the application server is often Webrick, in this book we'll be using Unicorn as a production application server. Puma is another powerful alternative to Unicorn.

Nginx is serving a vital role here because many application servers, Unicorn included, are designed to interact with "fast clients" rather than end users.

A fast client is one which has a fast, low latency connection between it and the server. In this case this is a local network or socket connection between nginx and the Unicorn app server.

This fast connection means that Unicorn can spend as much time as possible processing requests, rather than waiting for the client. Nginx is responsible for queuing up and maintaining connections from the outside world and then passes them to Unicorn for very quick processing, as soon as Unicorn signals it is available to process another request.

This is a huge over simplification but serves to illustrate why we need Nginx at all. There's more on the concept of fast clients and reverse proxying here: http://unicorn.bogomips.org/PHILOSOPHY.html

## Why Nginx

There are two key models for web servers, process based and event based.

The best known process based web server is Apache. A process based web server (simplistically) spawns additional processes to handle new incoming connections. This approach is fast under light loads but as the load increases, so does the RAM usage. This makes resource usage hard to predict which is a particular problem in resource constrained environments such as a typical VPS.

The best known event based server is Nginx. An event based server does not create additional processes for each new request. Instead when the server starts, multiple single threaded worker processes are spawned. Each of these worker processes can handle many incoming connections at any one time. Instead of continuously handling one request, each worker process can handle several thousand concurrent connections, in part by working on a part of one while waiting for part of another one to complete.

One of the outcomes of this is that Nginx's memory usage is both very low and more importantly very predictable. This is a significant benefit both for general resource planning and in situations where Nginx is running alongside other important processes.

## The Cookbook

Installation and configuration of NGinx is handled by the nginx community cookbook available here https://github.com/miketheman/nginx which is included by the `nginx-server` role.

This role definition is as follows:

**roles/nginx-server.json**

```json
{
   "name": "nginx-server",
   "description": "Nginx server",
   "default_attributes": {
      "firewall" : {
        "rules" : [
          {"allow http on port 80" : {"port" : 80}}
        ]
      },
      "nginx" : {
        "default_site_enabled" : false
      }
   },
   "json_class": "Chef::Role",
   "run_list": [
      "recipe[nginx::repo]",
      "recipe[nginx::default]",
      "recipe[monit_configs-tlq::nginx]",
      "recipe[ufw::default]"
   ],
   "chef_type": "role"
}
```

It first includes the `nginx::repo` recipe which adds the ppa for the current stable version of nginx. This will provide a significantly more recent version than would be provided by the standard Ubuntu packages.

It then includes `nginx::defualt` which will install nginx using the native package provided by the OS's package manager. Since this is included after the `nginx::repo` recipe, this package will be the current stable one provided by the creators of Nginx rather than the default one in the Ubuntu repositories.

Finally it includes the example nginx monit configuration and the UFW cookbook. This is intentionally included both here and in the `server` role in case we ever use the `nginx-server` role without also using the `server` role.

The default firewall rule `{"allow http on port 80" : {"port" : 80}}` will allow access to port `80` (the default port for web servers) from any server. Bear in mind though that as we saw in the security chapter, if we also define firewall rules at node level, these will override the rule specified here. So in that scenario it would need to be re-specified at node level.

## Virtualhosts

### What They Are

Virtual hosts are used to define what Nginx should do when requests for a particular website are received. This might be to serve some static files from a filesystem location or proxy the request back to something like a Rails application server (such as Unicorn) or another application like Monit.

### What Creates Them

The approach taken throughout this book is that anything specific to a single application being deployed, should be handled by the applications deployment process rather than the server provisioning process. This allows a single server to to be used to host multiple applications without provisioning changes which may be disruptive to all applications on it.

With this in mind, the approach suggested is that only server specific virtual hosts, such as Monit configurations should be created during provisioning. All subsequent virtual hosts should be created and managed by the applications themselves.

This means that all site specific configuration, including how to setup SSL, is covered in the second part of this book.

### The Default Virtualhost

A common problem is to find that having configured a server and deployed an application, the default "It Works" page is still displayed. This is often because the default virtualhost in `/etc/nginx/sites-enabled/default` has not been removed. This is covered again in the second part of the book but is worth mentioning here as well because there's nothing more frustrating than spending hours troubleshooting a problem only to discover it was this.

This line `"default_site_enabled" : false` in `server.json` ensures that this default virtual host is not created.

If however, you run into symptoms like those above, double check that the default entry has been removed. If it hasn't, remove it and then run `nginx -s reload` to reload the configurations.

## Monitoring

As with the other services in the example configurations an example of a simple Monit configuration for Nginx is included in the `monit_configs-tlq` cookbook. This is automatically added by the `nginx-server` role (`recipe[monit_configs-tlq::nginx]`). The generated `monit` configuration looks like this:

```
check process nginx with pidfile /var/run/nginx.pid
  start program = "/etc/init.d/nginx start"
  stop program = "/etc/init.d/nginx stop"
  if failed host 127.0.0.1 port 80 then restart
  if 15 restarts within 15 cycles then timeout
```

Which checks for both the existence of the pidfile and that connections are being accepted on port 80.

# PostgreSQL

## Overview

In this chapter we'll cover getting the PostgreSQL server packages installed, setting up authentication and managing databases with the console.

## Installation

In our simple configuration, the only parameter which must be set is the postgres users password. It's important that we keep a record of this as it will be required to gain access to the postgres user which will be needed for creating databases. This is generated using:

**Terminal (remote)**

```
openssl passwd -1 "plaintextpassword"
```

And entered into the node definition:

**nodes/rails_postgres_redis.json.example (excerpt)**

```
    "postgresql" : {
      "password" : {
        "postgres" : "openssl_output"
      }
    },
    ...
```

If, as happends very easily, this password is lost or forgotten, we can reset it with:

**Terminal (remote)**

```
sudo passwd postgres
```

Which will then prompt you to enter and re-enter the plain text password.

The sample configuration contains a simple `postgres-server` role which automatically includes the PostgreSQL server recipe as well as a simple monit configuration:

**roles/postgres-server.json**

```
{
   "name": "postgres-server",
   "description": "Postgres database server",
   "default_attributes": {

   },
   "json_class": "Chef::Role",
   "run_list": [
      "postgresql::server",
      "monit_configs-tlq::postgres"
   ],
   "chef_type": "role"
}
```

This can be included in a node definition by adding:

```
    "role[postgres-server]"
```

to the run list.

## Accessing the psql console

The PostgreSQL console can be accessed when logged into the server with ssh by first switching to the postgres user:

**Terminal**

```
su postgres
```

And entering the password chosen above, then entering:

**Terminal**

```
psql
```

We'll then see something like:

**Terminal output (remote)**

```
postgres@precise64:/etc/postgresql/9.1/main$ psql
psql (9.1.9)
Type "help" for help.

postgres=#
```

This console will be the main tool used for creating databases and managing access to these databases.

You can exit the console by typing:

**Terminal (remote)**

```
\q
```

## Creating Databases

Access the psql console as above and then enter:

**Terminal - PSQL Console**

```
CREATE DATABASE database_name;
```

and press enter.

What this process actually does is create a copy of the default database 'template1' (which is created when PostgreSQL is first installed) with the name database_name. Advanced usage and manipulation of template databases is beyond the scope of this book but is well documented in the official PostgreSQL manual.

If, having created this database, you wanted to execute commands on it using the console, you would enter:

**Terminal**

```
psql database_name
```

while logged in as the postgres user. psql expects the first none option (e.g. not - or --) to be the name of the database to be connected to. You can also specify a database with the -d flag, for example:

**Terminal**

```
psql -d database_name
```

## Adding users to Databases

At the moment only the postgres superuser can access the newly created database. In practice for security we want each database to have its own user which can access only one specific database and cannot create or modify other databases and users.

To create a new user, in the psql console enter:

**Terminal - PSQL Console**

```
CREATE USER my_user WITH PASSWORD 'my_password';
```

and press return.

You can then grant this user all privileges on a specific database with:

**Terminal - PSQL Console**

```
GRANT ALL PRIVILEGES ON DATABASE database_name to my_user;
```

You can now exit the psql console with `\q` and return to the deploy user by entering `exit` in the console.

If we now wanted to start a psql console specifically to execute commands in the database we have just created as the user we just created, there is no need to switch to the postgres user, we can simply use:

**Terminal**

```
psql -h 127.0.0.1 database_name my_user
```

or

**Terminal**

```
psql -h 127.0.0.1 -d database_name -U my_user
```

Note the addition of `-h 127.0.0.1`. This will force the connection to be established via tcp/ip rather than the default unix-domain socket. We'll see later in the Configuring Authentication section that in our default configuration, password based authentication is only enabled for tcp connections.

## Listing all databases and permissions

An extremely useful command when in the psql console is the simple:

**Terminal - PSQL Console**

```
\l
```

Which, when run as the postgres user, will output a list of all databases and who has access to them.

## Configuring Authentication

Postgres supports a variety of authentication methods but we'll consider three key ones here; Peer, Ident, and md5.

### pg_hba.conf

The authentication methods which are allowed on a PostgreSQL server are defined in a file traditionally called `pg_hba.conf`. You can find this in:

```
/etc/postgres/version/pg_hba.conf
```

Where version is the version of postgres you have installed.

Whilst the generation of this file will be taken care of by our Chef recipe, understanding how this file is structured and what it means will make troubleshooting connection issues and creating custom configurations further down the line much easier. This explanation is not exhaustive (the PostgreSQL manual is excellent for more details) but covers the basic structure and options relevant to our configuration.

Our default configuration will look something like this:

**/etc/postgres/version/pg_hba.conf**

```
# This file was automatically generated and dropped off by Chef!

# PostgreSQL Client Authentication Configuration File
# ===================================================
#
# Refer to the "Client Authentication" section in the PostgreSQL
# documentation for a complete description of this file.

# TYPE  DATABASE        USER            ADDRESS                 METHOD

###########
# Other authentication configurations taken from chef node defaults:
###########

local   all             postgres                                ident

local   all             all                                     ident

host    all             all             127.0.0.1/32            md5

host    all             all             ::1/128                 md5

# "local" is for Unix domain socket connections only
local   all             all                                     peer
```

Each none-comment line of the file represents a new record, with columns being separated by spaces or tabs.

The first column (type) specifies the type of connection the record applies to. Host means tcp/ip connections and local means unix-domain sockets.

Database and user refer to the database and user the rule applies to, our configuration will work on the basis that all access methods we define are available to all users on all databases so these will generally be `all`.

For our purposes the address field only applies to entries of type `host` and contains either a single ip address, an ip address range or a hostname.

Method refers to one of the authentication methods supported by PostgreSQL, a selection of which are covered in more detail below.

## Md5

This is the primary authentication method we will use for Rails applications. md5 is a type of password authentication allows us to authenticate a user by sending the md5 hash of the users password where the user is a PostgreSQL database user (created above with `CREATE USER`) as distinct from a system user.

In general the client will take care of creating the md5 hash and passing it to the server so whether entering details in a Rails `database.yml` file or connecting to a psql console instance, we will never need to generate the hash ourselves. We provide the plain text password and the client will generate an md5 hash of it and send that to the server.

The benefit of md5 over simple password authentication is that the plain text password is never sent over the network making packet sniffing less of a concern. That said, md5 is no longer a secure hashing algorithm so minimal weight should be placed on this.

We can now see why:

**Terminal**

```
psql -d database_name -U my_user
```

would fail but:

**Terminal**

```
psql -h 127.0.0.1 -d database_name -U my_user
```

Would prompt for a password and succeed (assuming the correct password is entered).

The entries to allow md5 auth have a type `host` and therefore are only available if a connection is made via tcp/ip. By default psql will attempt to connect via a unix-domain socket (type local) for which there are no password authentication methods available.

It's worth noting that in our default configuration, md5 authentication is only available on the local loopback interface due to the host options being defined as the loopback IPV4 and IPV6 addresses.

## Peer

Peer authentication queries the servers kernel for the username of the user who executed the command. Because it requires direct kernel access it can only be used for connections of type local.

The username of the user executing the command is taken as the database username. Therefore if your Unix username was `deploy` and the user `deploy` had been granted access to the database `my_app_production` then scripts running as this user would be able to access the database without any further authentication as would the psql console.

To grant your `deploy` user access to the the database `database_name` switch to the postgres user and load the psql console and enter:

**Terminal - PSQL Console**

```
CREATE USER deploy;
GRANT ALL PRIVILEGES ON DATABASE database_name to deploy;
\q
```

You can then `exit` back to your deploy user and enter:

**Terminal**

```
psql database_name
```

And you will be able to enter the psql shell for the database database_name as user deploy without entering any password.

Note that PostgreSQL users are completely separate to system users therefore even though the unix user deploy already exists, we have to create a matching user in PostgreSQL.

Peer authentication looks at the current system user and then, if a PostgreSQL user with a matching name exists, authenticates as that user. It is also possible to create mapping tables which set which system user maps to which PostgreSQL user, the details of this are beyond the scope of this book but are covered in depth in the PostgreSQL manual.

### Ident

Ident authentication is covered here only because it occurs often in documentation or tutorials about PostgreSQL and so an understanding of what it is and how it differs to Peer authentication is useful.

It is very similar to peer authentication except instead of querying the kernel for the username of the user executing the command it queries an ident server on tcp port 113.

An ident server answer questions such as "What user initiated the connection that originated your port X and connected to this server on port Y?".

The obvious downside of this is that if the client machine is compromised or malicious, it can be set to return anything when port 113 is queried. As a result this authentication method is only suitable for use on networks where both client and server are entirely trusted.

It's important to note that where the ident method is specified but the type is local, peer authentication will be used instead.

## Allow External Access

By default our chef recipe will create the following line in `postgresql.conf`:

**/etc/postgresql/VERSION/postgresql.conf**

```
listen_addresses = 'localhost'
```

Which means that PostgreSQL will only accept connections from localhost.

To allow connections from other hosts, we can add the following to the `postgres` section of the node definition:

**nodes/SERVERIP.json (extract)**

```
"config" :{
  "listen_addresses" : "*"
}
```

which gives the following line in `postgresql.conf`:

**/etc/postgresql/VERSION/postgresql.conf (extract)**

```
listen_addresses = '*'
```

This will cause postgres to listen on all available network interfaces. I generally work on the basis that it is the job of the Firewall (in our case ufw + iptables) to manage incoming connections and so allowing the database to listen on all interfaces is acceptable.

## Mangaging pg_hba.conf with chef

As with all of our configuration files, we should never modify them directly as changes will be overwritten by Chef. Instead, all changes should be made within our chef recipes.

The PostgreSQL cookbook we're using provides an easy interface for manging entries in `pg_hba/conf`. The `postgres` section of our our node definition (in `nodes/hostname.json` accepts an array like this:

**nodes/HOSTNAME.json (excerpt)**

```
"postgres":
  {"pg_hba" : [
    {"comment" : "# Data Collection",
      "type" : "host",
      "db" : "the_database_name",
      "user" : "the_user",
      "addr" : "the_host",
      "method" : "authentication method"}
  ],
  ....
 }
```

The `"pg_hba"` key should contain an array of hashes, each of which will be converted into an entry in `pg_hba.conf`. So for example the following:

**nodes/HOSTNAME.json (excerpt)**

```
{"comment" : "# Data Collection",
  "type" : "host",
  "db" : "my_database_name",
  "user" : "some_username",
  "addr" : "0.0.0.0/0",
  "method" : "md5"}
```

Would create the following entry in `pg_hba.conf`:

**/etc/postgresql/VERSION/pg_hba.conf**

```
# Data Collection
host    my_database_name  some_username       0.0.0.0/0                md5
```

This would have the effect of allowing the user some_username to connect to my_database_name using md5 auth from any IP address. In practice this is very rarely a good idea and you'd want to specify individual IP's or ranges who should be allowed.

Bare in mind that if you're enabling access from external hosts, you'll also need to modify your configuration as explained in the Configuring Authentication section earlier.

## Importing and Exporting Databases

PostgreSQL includes the utility `pg_dump` for generating an SQL dump of a database. Its options are very similar to the psql command. To generate a dump of the database `test_db_1` authorising as the user `test_user_1` we'd use the following command:

**Terminal (PSQL Console)**

```
pg_dump -h 127.0.0.1 -f test1.sql -U test_user_1 test_db_1
```

Where the `-f` option is for specifying the filename to export to and the first none option argument is the database to be exported.

This file can then be imported using the following command:

**Terminal (PSQL Console**

```
psql -U test_user_1 -d database_to_import_to -f current.sql -h 127.0.0.1
```

This takes the contents of the file specified with the `-f` option and executes the sql contained in it on the database specified with the `-d` option. This can be used for anything from copying databases from production to development servers, migrating between servers or cloning production data to staging.

## Monit

A monit configuration for postgresql is available in the `monit_configs-tlq::postgres recipe`:

**monit_configs-tlq/templates/default/postgres.conf.erb**

```
check process postgresql with pidfile <%= node['monit_configs-tlq']['postgres']['\
pidfile'] %>
  start program = "/etc/init.d/postgresql start"
  stop program = "/etc/init.d/postgresql stop"
  if 15 restarts within 15 cycles then timeout
```

Note that the pid file name will change depending on the version of PostgreSQL being run. The recipe will default to `'/var/run/postgresql/9.3-main.pid'` but this can be overriden in the `postgres-server` role by setting the following:

**roles/postgres-server.json (extract)**

```
"default_attributes": {
   "monit_configs-tlq" : {
     "postgres" : {
       "pidfile" : "/var/run/postgresql/9.4-main.pid"
   }
  }
},
```

Depending on the version of Postgres you've chosen to install.

# Redis and Memcached

## Redis

### Â Overview

Redis is an Open Source key value cache and store. In a Rails environment it is commonly used for:

- View Caching
- Geocoder Request Caching
- API Rate Limit Management
- Counter Caching
- Background Job Queues (e.g. Sidekiq)

Among other things.

In the example configuration we use the Redis cookbook we created in chapter 5. An alternative community cookbook is available here https://github.com/brianbianco/redisio if needed which allows for more advanced configuration such as multiple instance installation and a light weight resource provider for integrating Redis management into your own cookbooks.

## Configuring

Redis is famously simple to setup and run. The `redis-server` role simply includes the `redis-server` default recipe we created in chapter 5 followed by the example monit configuration. This will install Redis, bound to `127.0.0.1` on port `6379`.

### Manage Size

In the default configuration, there is nothing to limit the total size the Redis dataset can grow to. In some situations, such as when using Redis as a data store for a queue such as Sidekiq, this is desirable, we don't want data to be discarded.

If however we're using Redis as a cache, we may want to limit its maximum resource usage. We can add the following to either the `redis-server` role or the node definition:

**roles/redis-server.json**

```
'redis-server' : {
        'additional_configuration_values' : {
          'maxmemory' : 419430400,
          'maxmemory-policy' : 'allkeys-lru',
          'maxmemory-samples' : '10'
        }
}
```

This defines an `lru` maximum memory policy which means "least recently used". Once Redis reaches the maximum size allowed (in the above example 419430400bytes or ~420mb), `maxmemory-samples` number of keys will be selected and the least recently used of these (10 in the case of the example above) will be discarded. Note that this does not mean that the the least recently used key in the entire dataset will be discarded, only the least recently used in the randomly selected sample.

## Monitoring

An example monit configuration is included from `monit_configs-tlq` by the `redis-server` role. The example monit definition looks like this:

**monit_configs-tlq/templates/default/redis-server.conf.erb**

```
check process redis with pidfile /var/run/redis/redis-server.pid
  group database
  start program = "/etc/init.d/redis-server start"
  stop program = "/etc/init.d/redis-server stop"
  if failed host 127.0.0.1 port 6379 then restart
  if 15 restarts within 15 cycles then timeout
```

This checks for the existence of the Redis process and that it is accepting connections on port 6379.

# Memcached

## Â Overview

Of all the system components covered in this book, Memcached is by far the simplest to install and maintain. This section will be correspondingly brief. We'll cover installation, a single configuration option and a very simple Monit profile and then we're done.

Installation and configuration of Memcached is provided by the Opscode community cookbook available at https://github.com/opscode-cookbooks/memcached.

## Configuring

The `memcached-server` role installs memcached with a sensible set of defaults which are sufficient for the vast majority of use cases. The following two attributes are defined in the role definition:

**roles/memcached-server.json**

```
"memcached" : {
     "listen" : "127.0.0.1",
     "memory" : 64
  }
```

This binds memcached to localhost, preventing it from being world accessible and sets the maximum memory limit to 64mb.

If we were building a multi host system, we might instead want to bind memcached to `0.0.0.0` (world accessible) or a particular private network interface and then rely on the firewall to limit access to the exposed port.

## Monitoring

An example monit configuration is included from `monit_configs-tlq` by the `memcached-server` role. The example monit definition looks like this:

**monit_configs-tlq/templates/default/memcached.conf.erb**

```
check process memcached
   with pidfile /var/run/memcached.pid
   group memcache
   start program = "/etc/init.d/memcached start"
   stop  program = "/etc/init.d/memcached stop"
   if failed host 127.0.0.1 port 11211 protocol memcache then restart
   if 3 restarts within 6 cycles then timeout
```

This checks for the existence of the Memcached process and that it is accepting connections on port 11211 with the memcache protocol.

# Conclusion

The aim of this section has been twofold.

Firstly to clearly document a example configuration for provisioning a server suitable for a typical Ruby on Rails application. Once you're comfortable with this template, getting a new instance ready for a Rails app should be a trivial task. No more complicated - an eventually as instinctual - as setting up a new Rails application for development.

The real power of Chef, or any configuration management tool, comes when you’re sufficiently comfortable with it that using it for every change or improvement to a server becomes second nature.

Therefore secondly and most importantly, I hope this section has demonstrated how simple it is to use tools like Chef to automate the provisioning process. Please go ahead and fork my recipes, swap out my simple recipes for your own forks or more complex versions until you have a template which is perfectly tailored to the apps you’re deploying.

# Deploying with Capistrano Quickstart

## Overview

In the first part of this book, we covered setting up a VPS ready to run a Rails application. In this part we'll cover the process of deploying one or more apps to this VPS.

In this chapter we'll cover the simplest and quickest configuration for setting up Capistrano to deploy to a server setup using the template from the previous section. To do this we'll use a gem `capistrano-cookbook` which bundles together some common helper tasks and provides a Rails generator for bootstrapping all of the commonn .

In the next chapter we'll delve into the generated configuration in detail, ensuring that we understand exactly what's happening behind the scenes and how to extend and tweak it ourselves.

## Capistrano

Capistrano is a ruby gem which provides a framework for automating tasks related to deploying a ruby based application, in our case a Rails app, to a remote server. These include tasks like checking out the code from a git repository onto the remote server and integrating stage specific configuration files into our app each time we deploy.

## Steps

These steps assume we've already deployed a suitably VPS and configured it as per the instructions in the Chapter 5 Quick Start.

### Adding Gems

We begin by adding the following to the `Gemfile` of the application we are deploying:

**Gemfile (extract)**

```ruby
...
# The uicorn application server
gem 'unicorn', '~> 4.8.3'

group :development do
  # Including capistrano cookbook will automatically includes
  # the correct version of capistrano and other plugins
  gem 'capistrano-cookbook', require: false
end
...
```

and then runnning `bundle install`.

If we already have Capistrano configuration, we'll want to either remove it completely or append `.old` to the following:

```
Capfile
config/deploy.rb
config/deploy/
```

Before continuing.

## Generating The Base Configuration

The `capistrano-cookbook` gem includes a standard Raisl generator which automates the creation of the boilerplate files required for deploying an application. To run it, enter the following command:

```
bundle exec rails generate capistrano:reliably_deploying_rails:bootstrap --produc\
tion_hostname='YOUR_PRODUCTION_HOSTNAME' --production_server_address='PRODUCTION_\
SERVER_PUBLIC_IP' --sidekiq --certbot_enable --certbot_email='YOUR_EMAIL_ADDRESS'
```

Replacing:

- `YOUR_PRODUCTION_HOSTNAME` with the address your application will be accessible on. You should create a suitable DNS entry (generally an A record) which points this domain to the IP address of the server you are deploying to. Note that if you are not creating a DNS entry and instead are creating an entry in your local hosts file then you **must** remove the `--certbot_*` flags as they require a functioning DNS configuration to generate the SSL certificate.
- `PRODUCTION_SERVER_PUBLIC_IP` with the IP address of the VPS you are deploying to. In single server configurations this could also be the same as `YOUR_PRODUCTION_HOSTNAME` but that approach adds some fragility if, in the future, you decide to add additional frontend servers behind a load balancer to handle additional load.

- `YOUR_EMAIL_ADDRESS` with the email email address LetsEncrypt can send certificate expiry notifications to. These notifications are generally for information only as our configuration automatically renews certificates when the expire so you may want to add a suffix, e.g. "youremail+letsencrypt" to make it easy to filter these emails with automated rules.

And optionally keeping or removing this flags:

- `--sidekiq` remove this flag if you are not using Sidekiq for background jobs. If this flag is present the generator will include the required logic and templates to have Sidekiq automatically deployed and restarted alongside the core Rails application
- `--certbot_enable` and `--certbot_email` remove these flags if you do not want to have a free SSL certificate for `YOUR_PRODUCTION_HOSTNAME` generated. You'll definitely want to remove these flags if you're testing locally with something like vagrant or if you don't yet have a domain with DNS setup, e.g. when using local hosts file to map domains to IP's.

Which will generate the following files and folders:

```
Capfile
config
  ├── deploy.rb
  └── deploy
      ├── production.rb
      ├── staging.rb
      └── templates
          ├── nginx_conf.erb
          ├── puma_monit.conf.erb
          ├── puma.rb.erb
          ├── puma.service.erb
          ├── sidekiq_monit.erb
          └── sidekiq.service.capistrano.erb
```

The purpose of each of these files is examined in detail in the next chapter.

## Updating deploy.rb

Capistrano allows us to deploy to multiple "stages," e.g. "staging" and "production" these are represented by the files `config/deploy/staging.rb` and `config/deploy/production.rb` respectively.

The file `config/deploy.rb` contains the parts of our deployment configuration which applies to all stages. Our generated `deploy.rb` file looks like this (assuming both sidekiq and automatic ssl are enabled):

**config/deploy.rb**

```ruby
# config valid for current version and patch releases of Capistrano
lock "~> 3.16.0"

set :application, 'rdra_rails6_example'
set :deploy_user, 'deploy'

# setup repo details
set :repo_url, 'git@github.com:TalkingQuickly/rdra_rails6_example.git'

# setup rbenv.
set :rbenv_type, :system
set :rbenv_ruby, '3.0.0'
set :rbenv_prefix, "RBENV_ROOT=#{fetch(:rbenv_path)} RBENV_VERSION=#{fetch(:rbenv\
_ruby)} #{fetch(:rbenv_path)}/bin/rbenv exec"
set :rbenv_map_bins, %w{rake gem bundle ruby rails}

# setup certbot for SSL via letsencrypt
set :certbot_enable_ssl, true
set :certbot_redirect_to_https, true
set :certbot_email, "ben@talkingquickly.co.uk"
set :certbot_use_acme_staging, false

# setup puma to operate in clustered mode, required for zero downtime deploys
set :puma_preload_app, false
set :puma_init_active_record, true
set :puma_workers, 3
set :puma_systemctl_user, fetch(:deploy_user)
set :puma_enable_lingering, true


set :sidekiq_systemctl_user, fetch(:deploy_user)
set :sidekiq_enable_lingering, true


# how many old releases do we want to keep
set :keep_releases, 5

# Directories that should be linked to the shared folder
append :linked_dirs, 'log', 'tmp/pids', 'tmp/cache', 'tmp/sockets', 'vendor/bundl\
e', '.bundle', 'public/system', 'public/uploads'
append :linked_files, 'config/database.yml', 'config/master.key'

# this:
# http://www.capistranorb.com/documentation/getting-started/flow/
# is worth reading for a quick overview of what tasks are called
# and when for `cap stage deploy`

namespace :deploy do
end
```

The following values need to be replaced with your own:

- `app_name` The name of your application, the `capistrano-cookbook` gem will have tried to guess this from your apps name
- `repo_url` The address of the Git repository to deploy from, by default this will have been set to the remote named `origin`
- `rbenv_ruby` to the Ruby version your app requires. This should match the version you configured to be installed on the server as per Chapter 9.

## Setting Stage Details

Stages refer to the different deployments of your application. So generally you would have at a minimum a production stage (for your live site) and a staging stage for testing.

For now we'll set up a single production stage for the live site. Stages are defined in `config/deploy/stage_-name.rb`. The Capistrano installer will have already generated a sample `production.rb` for us, we'll replace that with the following:

**config/deploy/production.rb**

```
set :stage, :production
set :branch, "master"

# This is used in the Nginx VirtualHost to specify which domains
# the app should appear on. If you don't yet have DNS setup, you'll
# need to create entries in your local Hosts file for testing.
set :nginx_server_name, 'YOUR_PRODUCTION_HOSTNAME'

# used in case we're deploying multiple versions of the same
# app side by side. Also provides quick sanity checks when looking
# at filepaths
set :full_app_name, "#{fetch(:application)}_#{fetch(:stage)}"


# Name sidekiq systemd service after the app and stage name so that
# multiple apps and stages can co-exist on the same machine if needed
set :sidekiq_service_unit_name, "sidekiq_#{fetch(:full_app_name)}"


server 'PRODUCTION_SERVER_PUBLIC_IP', user: 'deploy', roles: %w{web app db}, prim\
ary: true

```

```
set :deploy_to, "/home/#{fetch(:deploy_user)}/apps/#{fetch(:full_app_name)}"

# dont try and infer something as important as environment from
# stage name.
set :rails_env, :production
```

These values should have been set or inferred automatically by the generator but we should check that they are each set correctly:

- `branch` to the branch of the Git repository to be deployed from, for production this will often be `master`
- `server_name` to the hostname which the website will be served on. This is used for generating the nginx virtual host.
- `rails_env` to the RAILS_ENV the application should run in, usually `production` for the live site and `staging` for everything else.
- the `server` line, ensuring this matches the production IP of the VPS we are deploying to

### DNS/ HOSTS File

Nginx will be setup to only serve the site on the domain specified in `server_name`.

To see the site working it is therefore necessary to either:

- Create DNS entries on your nameserver pointing to the server, generally an A Record for the hostname pointing at the servers IP
- Modify your local `/etc/hosts` file to temporarily cause the domains to resolve to the IP of your server.

So for example adding the following:

**/etc/hosts (local)**

```
123.456.789.123 example.com www.example.com
```

To `/etc/hosts` would mean that locally, any requests to `example.com` or `www.example.com`, would go to the IP `123.456.789.12`

Note that when using the local hosts file approach, SSL must be disabled. This can be done by changing `set :certbot_enable_ssl, true` to `set :certbot_enable_ssl, false` in `config/deploy.rb`.

## Copying configuration to the destination server

For details on what's going on behind the scenes here, see `Generating Remote Configuration Files` in the next chapter.

To generate our one time configuration on the destination server, we can run:

**Terminal (local)**

```
bundle exec cap production deploy:setup_config
```

This will create one off configuration items on the remote server, as well as instructing certbot to request an SSL certificate for the domain specified in `server_name` of `config/deploy/production.rb`.

## Creating a database

Assumign we're using the single server PostgreSQL configuration from the first part of this book, the `capistrano-cookbook` gem includes a task to automatically generate the `database.yml` and create the database. We can simply run:

**Terminal (local)**

```
bundle exec cap production database:create
```

Which will take care of database and user creation as well as generating a suitable database.yml and populating it with randomly generated credentials.

## First Deploy

With this completed, we're now ready for our first deploy. We simply enter:

```
bundle exec cap production deploy
```

And wait!

## Troubleshooting

If all goes well, once the deploy completes, we'll visit the URL for our site and it will be up and running. If not the following is a simple structure for honing in on the root cause of the problem:

### Is It Running

We can first SSH into the server and enter:

```
ps aux | grep puma
```

If our Rails application is running, we'll see both entries for both a puma master and one or more worker processes, for example:

```
deploy       677  0.0  0.7  86880 30276 ?        Ss   Mar25   1:58 puma 5.2.2 (un\
ix:///home/deploy/apps/rdra_rails6_example_production/shared/tmp/sockets/puma.soc\
k)
deploy     62270  0.0  2.9 705960 114060 ?       Sl   Mar29   1:06 puma: cluster \
worker 0: 677
deploy     62295  0.0  2.8 705976 113884 ?       Sl   Mar29   1:06 puma: cluster \
worker 1: 677
deploy     62303  0.0  2.9 705972 114304 ?       Sl   Mar29   1:06 puma: cluster \
worker 2: 677
```

If we don't see these entries, it means that the puma master failed to start so we can try the following to understand why:

1. Our puma server is managed via the userspace systemd, we can check the names of the systemd units with `ls ~/.config/systemd/user/`, if our application is called `rdra_rails6_-example` and we're deploying the `production` stage we'll see something like `puma_rdra_-rails6_example_production.service`
2. We can then use `systemctl --user status puma_rdra_rails6_example_production.service` to see whether systemd believes the service has been started correctly
3. If it has not, e.g. the service is showing as failed, then some debug output is shown at the bottom
4. If this debug output is not sufficient we can then use `journalctl --user -u puma_rdra_-rails6_example_production.service` to see the log output from the service
5. If the steps above don't yield any information, we can also check the contents of the various logs in `~/apps/YOUR_APP_NAME/shared/log` using something like `tail -n 100 LOG_FILE_-NAME` to view the last 100 lines of that log
6. It can be particularly interesting to live tail using `tail -f` either the nginx access logs or the nginx error logs in this folder, and then try access the page and seeing if new output is added, for example `tail -f nginx.access.log` and `tail -f nginx.error.log`.

## Is The Request Hitting the Server At All

If it looks like everything is running, but we're still not able to access our application it's possible that the request is never making it as far as the Rails application.

Still ssh'd into the server we deployed to, we can use ufw to check that connections to port 80 and, if we're using SSL, port 443 are allowed:

```
sudo ufw status
```

You should see something like:

```
Status: active

To                           Action      From
--                           ------      ----
22                           ALLOW       Anywhere
80                           ALLOW       Anywhere
443                          ALLOW       Anywhere
22                           ALLOW       Anywhere (v6)
80                           ALLOW       Anywhere (v6)
443                          ALLOW       Anywhere (v6)
```

If we don't see ports 80 and 443 listed there, we should go back to our Chef repository and check that the `nginx-server` role is included and that it contains the following:

```
"default_attributes": {
  "firewall" : {
    "rules" : [
      {"allow http on port 80" : {"port" : 80}},
      {"allow https on port 443" : {"port" : 443}}
    ]
  },
```

## Conclusion

In this chapter we've covered how to get a standard Rails application deployed quickly, with a minimal amount of setup required. It uses the `capistrano-cookbook` gem which is a collection of common tasks and templates useful when deploying Ruby/ Rack applications. In the next chapter we'll look in more depth at how to deploy the same configuration without using an external gem. Although in most scenarios, the `capistrano-cookbook` gem is extremely flexible, understanding exactly what each task it provides is doing makes troubleshooting issues easier and makes building custom deployment tasks for none standard requirements possible.

# Deploying with Capistrano

## Overview

In the previous chapter we looked at a quick way to get deployments working with Capistrano by using a gem which packages up common tasks. In this chapter we'll look at how to do the same thing, but without using a third party gem. Instead we'll manually include the helper files in our project.

## Stages

It's normal for production applications to have several 'stages'. In general, you would have, at a minimum, a staging environment and a production environment.

The staging environment is a copy of the production environment which uses dummy data (often copied from production) and can be used to test changes before they are deployed to production.

In this section we'll assume you have one production and one staging configuration, each using a VPS configured using the instructions in the previous section.

## Adding Capistrano to an application

Add the following to your Gemfile:

**Gemfile**

```ruby
group :development do
  gem 'capistrano', '~> 3.2.1'

  # rails specific capistrano functions
  gem 'capistrano-rails', '~> 1.1.0'

  # integrate bundler with capistrano
  gem 'capistrano-bundler'

  # if you are using Rbenv
  gem 'capistrano-rbenv', "~> 2.0"
end
```

As you can see, Capistrano 3 splits out a lot of app specific functionality into separate gems. This increased focus on modularity is a theme throughout the version 3 rewrite.

Then run `bundle install` if you're adding Capistrano 3 for the first time or `bundle update capistrano` if you're upgrading. You may need to do some of the usual Gemfile juggling if you're updating and there are dependency conflicts.

## Installation

Assuming you've archived off any legacy Capistrano configurations, you can now run:

**Terminal (local)**

```
bundle exec cap install
```

Which generates the following files and directory structure:

```
├── Capfile
├── config
│   ├── deploy
│   │   ├── production.rb
│   │   └── staging.rb
│   └── deploy.rb
└── lib
    └── capistrano
            └── tasks
```

The source for the suggested starting configuration is available at https://github.com/TalkingQuickly/capistrano-3-rails-template. I suggest cloning this repository and copying these files into your project as you work through this section.

## Capistrano 3 is Rake

This book provides a fully working Capistrano configuration which should work out of the box when used with the VPS configuration from the previous section. An understanding of how Capistrano is structured does however make a lot of operations much easier to understand so we'll look at it in brief here.

Capistrano 3 is structured as a Rake Application. This means that in general, working with Capistrano is like working with Rake but with additional functionality specific to deployment work flows.

This is particularly interesting when we realise that the file `Capfile` generated in the root of the project is just a Rakefile. This makes understanding what Capistrano is doing under the hood much easier - and removes a lot of the `magic` feeling which makes many uncomfortable about many Capistrano v2 deploy scripts.

Therefore, when we talk about Capistrano tasks, we know they're just Rake tasks with access to the Capistrano deployment specific DSL. In practice, a lot of this deployment specific DSL is actually made up of wrappers around SSHkit.

## The Capfile

We discussed above that the Capfile is essentially just a Rakefile. The example configuration Capfile looks like this:

**Capfile**

```ruby
# Load DSL and Setup Up Stages
require 'capistrano/setup'

# Includes default deployment tasks
require 'capistrano/deploy'

# Includes tasks from other gems included in your Gemfile
#
# For documentation on these, see for example:
#
#   https://github.com/capistrano/rvm
#   https://github.com/capistrano/rbenv
#   https://github.com/capistrano/chruby
#   https://github.com/capistrano/bundler
#   https://github.com/capistrano/rails/tree/master/assets
#   https://github.com/capistrano/rails/tree/master/migrations
#
# require 'capistrano/rvm'
 require 'capistrano/rbenv'
# require 'capistrano/chruby'
 require 'capistrano/bundler'
# require 'sidekiq/capistrano'
# require 'capistrano/rails/assets'
require 'capistrano/rails/migrations'

# Loads custom tasks from `lib/capistrano/tasks' if you have any defined.
Dir.glob('lib/capistrano/tasks/*.cap').each { |r| import r }
Dir.glob('lib/capistrano/**/*.rb').each { |r| import r }
```

Knowing that this is just a kind of Rakefile we can see that it's simply requiring task definitions, initially from Capistrano itself and then from other gems which are intended to add functionality.

It then goes on to include any application specific tasks defined in `lib/capistrano/tasks`.

The final line:

**Capfile (extract)**

```ruby
Dir.glob('lib/capistrano/**/*.rb').each { |r| import r }
```

is non-standard. This allows us to include arbitrary ruby files in the `lib/capistrano/` directory which can be used to define helper methods for the tasks.

If we were using Sidekiq we could simply uncomment the Sidekiq require entry and this would include the tasks the Sidekiq developers include for starting and stopping workers. This is a common pattern; many gems which require specific actions on deployment will provide pre-defined Capistrano tasks we can simply include in this manner.

## Common configuration

When the `Capfile` requires `capistrano/setup`, this:

- Iterates over the stages defined in `config/deploy/`
- For each stage, loads the configuration defined in `config/deploy.rb`
- For each stage, loads the stage specific configuration defined in `config/deploy/stage_-name.rb`

The approach in this book is to keep as much common configuration in `config/deploy.rb` as possible, with only minimal stage specific configuration in the stage files (e.g. `config/deploy/production.rb`).

The `deploy.rb` from the sample configuration looks like this:

**config/deploy.rb**

```
set :application, 'app_name'
set :deploy_user, 'deploy'

# setup repo details
set :scm, :git
set :repo_url, 'git@github.com:username/repo.git'

# setup rbenv.
set :rbenv_type, :system
set :rbenv_ruby, '2.1.1'
set :rbenv_prefix, "RBENV_ROOT=#{fetch(:rbenv_path)} RBENV_VERSION=#{fetch(:rbenv\
_ruby)} #{fetch(:rbenv_path)}/bin/rbenv exec"
set :rbenv_map_bins, %w{rake gem bundle ruby rails}

# how many old releases do we want to keep
set :keep_releases, 5

# files we want symlinking to specific entries in shared.
set :linked_files, %w{config/database.yml}

# dirs we want symlinking to shared
set :linked_dirs, %w{bin log tmp/pids tmp/cache tmp/sockets vendor/bundle public/\
system}

# what specs should be run before deployment is allowed to
# continue, see lib/capistrano/tasks/run_tests.cap
set :tests, []

# which config files should be copied by deploy:setup_config
# see documentation in lib/capistrano/tasks/setup_config.cap
# for details of operations
```

```
set(:config_files, %w(
  nginx.conf
  database.example.yml
  log_rotation
  monit
  unicorn.rb
  unicorn_init.sh
))

# which config files should be made executable after copying
# by deploy:setup_config
set(:executable_config_files, %w(
  unicorn_init.sh
))

# files which need to be symlinked to other parts of the
# filesystem. For example nginx virtualhosts, log rotation
# init scripts etc.
set(:symlinks, [
  {
    source: "nginx.conf",
    link: "/etc/nginx/sites-enabled/#{full_app_name}"
  },
  {
    source: "unicorn_init.sh",
    link: "/etc/init.d/unicorn_#{full_app_name}"
  },
  {
    source: "log_rotation",
   link: "/etc/logrotate.d/#{full_app_name}"
  },
  {
    source: "monit",
    link: "/etc/monit/conf.d/#{full_app_name}.conf"
  }
])

# this:
# http://www.capistranorb.com/documentation/getting-started/flow/
# is worth reading for a quick overview of what tasks are called
# and when for `cap stage deploy`

namespace :deploy do
  # make sure we're deploying what we think we're deploying
  before :deploy, "deploy:check_revision"
  # only allow a deploy with passing tests to be deployed
  before :deploy, "deploy:run_tests"
  # compile assets locally then rsync
  after 'deploy:symlink:shared', 'deploy:compile_assets_locally'
  after :finishing, 'deploy:cleanup'

  # remove the default nginx configuration as it will tend
  # to conflict with our configs.
  before 'deploy:setup_config', 'nginx:remove_default_vhost'

  # reload nginx to it will pick up any modified vhosts from
  # setup_config
  after 'deploy:setup_config', 'nginx:reload'

  # Restart monit so it will pick up any monit configurations
  # we've added
  after 'deploy:setup_config', 'monit:restart'

  # As of Capistrano 3.1, the `deploy:restart` task is not called
  # automatically.
  after 'deploy:publishing', 'deploy:restart'
end
```

When setting variables which are to be used across Capistrano tasks we use the `set` and `fetch` methods provided by Capistrano. Internally we're setting and retrieving values in a hash maintained by Capistrano but in general we don't need to worry about this, just that we set a configuration value in `deploy.rb` and in our stage files with:

```
set :key_name, "value"
```

And retrieve it with:

```
get :key_name
```

The key variables to set in `deploy.rb` are `application`, `repo_url` and `rbenv_ruby`. The Rbenv Ruby you set must match one installed with Rbenv on the machine you're deploying to, otherwise the deploy will fail.

## Running tests

When making small changes to an application, it's easy to forget to run the test suite prior to deploying, only realising there's a problem when some 'unrelated' feature doesn't work. The below lines in `deploy.rb` allow you to select particular Rspec specs which must pass before a deploy will be allowed to continue.

This is using the task defined in `/lib/capistrano/tasks/run_tests.cap` from the sample code (https://github.com/TalkingQuickly/capistrano-3-rails-template/)

**config/deploy.rb (extract)**

```
# what specs should be run before deployment is allowed to
# continue, see lib/capistrano/tasks/run_tests.cap
set :tests, []
```

So, for example, if we were to add "spec" to the above array:

**config/deploy.rb (extract)**

```
set :tests, ["spec"]
```

The command `rspec spec` would be run before deploying and the deploy would only be allowed to continue if there were no failures.

If you already have a full blown continuous integration system setup (or don't want to run specs at all), this can be left as an empty array.

## Hooks

The final section of `deploy.rb` looks like this:

**config/deploy.rb (extract)**

```
namespace :deploy do
  # make sure we're deploying what we think we're deploying
  before :deploy, "deploy:check_revision"
  # only allow a deploy with passing tests to deployed
  before :deploy, "deploy:run_tests"
  # compile assets locally then rsync
  after 'deploy:symlink:shared', 'deploy:compile_assets_locally'
  after :finishing, 'deploy:cleanup'

  # remove the default nginx configuration as it will tend
  # to conflict with our configs.
  before 'deploy:setup_config', 'nginx:remove_default_vhost'

  # reload nginx to it will pick up any modified vhosts from
  # setup_config
  after 'deploy:setup_config', 'nginx:reload'

  # Restart monit so it will pick up any monit configurations
  # we've added
  after 'deploy:setup_config', 'monit:restart'

  # As of Capistrano 3.1, the `deploy:restart` task is not called
  # automatically.
  after 'deploy:publishing', 'deploy:restart'
end
```

Capistrano works by calling tasks in a particular sequence. These are usually a mixture of internally defined tasks (such as those which checkout the source code from version control) and custom tasks such as the `run tests` task documented above.

If we want our custom tasks to be run automatically as part of a Capistrano work flow such as `deploy` then we use before and after hooks. So, for example, the following:

**config/deploy.rb (extract)**

```
before :deploy, "deploy:run_tests"
```

tells Capistrano that before the task called deploy is invoked, it should invoke the task `deploy:run_tests`. Using this methodology we can completely automate all steps required to deploy our application. We'll cover how to write custom tasks in section 15.1.

It's worth taking a look at http://www.capistranorb.com/documentation/getting-started/flow/ to understand the internal task ordering for a typical deploy.

## Setting up stages

A stage is a single standalone environment that an application runs in. At a minimum, a production application will generally have a staging environment, for testing new changes, in addition to the main production environment.

These map - although not necessarily one to one - to the "environments" which rails provides. In general, the only stage which will have its "environment" set to `production` is the live production configuration. All other remote environments generally use `staging`.

Ideally, the staging server would be an identical copy of the production one in order to minimise the chance of there being an error case which exists in production that does not show up in staging. In practice it's often not cost effective to mirror the production environment completely, instead using a lower spec'd VPS for staging which is provisioned using exactly the same chef configuration as production.

Stages are defined in `config/deploy/`. We invoke Capistrano tasks in the format:

**Terminal**

```
bundle exec cap stage_name task
```

Where stage_name is the name of a `.rb` file in `config/deploy`. This means we are not limited to just a staging and a production stage, we can define as many arbitrarily named stages as needed.

### The Production Stage

In the sample configuration, the production stage (defined in `production.rb`) looks like this:

**config/deploy/production.rb**

```
# this should match the filename. E.g. if this is production.rb,
# this should be :production
set :stage, :production
set :branch, "master"

# This is used in the Nginx VirtualHost to specify which domains
# the app should appear on. If you don't yet have DNS setup, you'll
# need to create entries in your local Hosts file for testing.
set :server_name, "www.example.com example.com"

# used in case we're deploying multiple versions of the same
# app side by side. Also provides quick sanity checks when looking
# at filepaths
set :full_app_name, "#{fetch(:application)}_#{fetch(:stage)}"

server 'example.com', user: 'deploy', roles: %w{web app db}, primary: true

set :deploy_to, "/home/#{fetch(:deploy_user)}/apps/#{fetch(:full_app_name)}"

# don't try and infer something as important as environment from
# stage name.
set :rails_env, :production

# number of unicorn workers, this will be reflected in
# the unicorn.rb and the Monit configurations
set :unicorn_worker_count, 5

# whether we're using SSL or not, used for building Nginx
# config file
set :enable_ssl, false
```

The most important variable to update is the address of the server and the git branch to be deployed from.

We'll look at Unicorn configuration in more detail in chapter 16. To begin with, I suggest setting `unicorn_worker_count` to two and then tuning it to suit your application once deployment is working smoothly.

To start, keep `enable_ssl` to false; this is covered in section 17.

## Generating Remote Configuration Files

Capistrano uses a folder called `shared` to manage files and directories that should persist across releases. The key folder is `shared/config` which should contain configuration files that should persist across deploys.

Let's take, as an example, the traditional `database.yml` file that ActiveRecord uses to determine the database and credentials required for accessing the database for the current environment.

We do not want to keep this file in version control since our production database details would be available to anyone who had access to the repository.

With Capistrano 3 we create a `database.yml` file in `shared/config` and the following:

**config/deploy.rb**

```
# files we want symlinking to specific entries in shared.
set :linked_files, %w{config/database.yml}
```

in `deploy.rb` means that, after every deploy, the files listed in the array (remember `%w{items}` is just shorthand for creating an array of string literals) will be automatically symlinked to corresponding files in shared.

Therefore, after our code is copied to the remote server the file `config/database.yml` will be changed to be a symlink which points to `shared/config/database.yml`.

One approach to creating files like is to manually SSH into the remote machine and create files like `shared/config/database.yml` manually. This, however, seems inefficient, as a lot of the configuration will be the same across all our remote servers and can be automatically generated based on the contents of the stage files. The aim of this book is to avoid these manual, error prone steps.

The sample code includes a custom extension to the standard Capistrano 3 approach to configuration files, which makes the initial creation of these files easier by adding the task `deploy:setup_config`.

This custom task is defined in `lib/capistrano/tasks/setup_config.cap` and configured in this section of `config/deploy.rb`:

**config/deploy.rb**

```
# which config files should be copied by deploy:setup_config
# see documentation in lib/capistrano/tasks/setup_config.cap
# for details of operations
set(:config_files, %w(
  nginx.conf
  database.example.yml
  log_rotation
  monit
  unicorn.rb
  unicorn_init.sh
  secrets.yml
))

# which config files should be made executable after copying
# by deploy:setup_config
set(:executable_config_files, %w(
  unicorn_init.sh
))
```

When this task is run, for each of the files defined in `:config_files` it will first look for a corresponding .erb file (so for nginx.conf it would look for nginx.conf.erb) in `config/deploy/#{application}_#{rails_env}/`. If it were not found there it would look for it in `config/deploy/shared/`. Once it finds the correct source file, it will parse the erb and then copy the result to the `config` directory in your remote shared path.

This allows you to define your common config files, which will be used by all stages (staging & production for example), in `shared` while still allowing for some templates to differ between stages.

Finally this section:

**config/deploy.rb**

```
# remove the default nginx configuration as it will tend
# to conflict with our configs.
before 'deploy:setup_config', 'nginx:remove_default_vhost'

# reload nginx to it will pick up any modified vhosts from
# setup_config
after 'deploy:setup_config', 'nginx:reload'

# Restart monit so it will pick up any monit configurations
# we've added
after 'deploy:setup_config', 'monit:restart'
```

Means that after `deploy:setup_config` is run, we:

- Delete the `default` nginx Virtualhost to stop it over-riding our custom VirtualHost
- Reload Nginx to pickup any changes to the VirtualHost
- Reload Monit to pickup any changes to the Monit configuration

## Managing non-Rails configuration

The target of the approach outlined in this book is to ensure that all configuration relating to the Rails application being deployed is managed by the deployment process. This means that in addition to files such as `database.yml`, which are internal to Rails, the configuration files that need to be managed by the deployment process include:

- Unicorn Monit definitions
- Nginx Virtual hosts entries
- Init scripts for unicorn and any background workers
- Log rotation definitions

Subsequent sections cover the contents of these files in detail. In the context of the deployment process, they differ from files like `database.yml` because they need to sit outside of the Rails directory structure.

If we take, as an example, the nginx virtual host file. Recalling from section 10.0 that these should be placed in `/etc/nginx/sites-enabled/`. One possibility is to create a Capistrano task which copies this file directly to that location.

A more elegant solution, however, is to use Symlinks. This allows us to keep all of the app's configuration within `shared/config` and have Capistrano create Symlinks to the appropriate locations.

The below section outlines these symlinks which are created by the `deploy:setup_config` tasks defined in `lib/capistrano/tasks/setup_config.cap`:

**config/deploy.rb**

```
# files which need to be symlinked to other parts of the
# filesystem. For example nginx virtualhosts, log rotation
# init scripts etc.
set(:symlinks, [
  {
    source: "nginx.conf",
    link: "/etc/nginx/sites-enabled/#{fetch(:full_app_name)}"
  },
  {
    source: "unicorn_init.sh",
    link: "/etc/init.d/unicorn_#{fetch(:full_app_name)}"
  },
  {
    source: "log_rotation",
   link: "/etc/logrotate.d/#{fetch(:full_app_name)}"
  },
  {
    source: "monit",
    link: "/etc/monit/conf.d/#{fetch(:full_app_name)}.conf"
  }
])
```

Once you've made any required changes to `production.rb`, `deploy.rb` and the configuration files, use the below command to copy the configuration files to the remote server.

**Terminal**

```
bundle exec cap production deploy:setup_config
```

## Database Credentials

The database example yml file intentionally doesn't include actual credentials as these should not be stored in version control.

Therefore, you need to SSH into your remote server and cd into `shared/config`, then create a database.yml from the example:

**Terminal**

```
cp database.yml.example database.yml
```

Edit it with a text editor such as vim, e.g:

**Terminal**

```
vim database.yml
```

And enter the details of the database that the app should connect to. You'll need to create this database as per the instructions in the chapter "PostgeSQL".

## Deploying

Now that we've run:

**Terminal**

```
bundle exec cap production deploy:setup_config
```

and created a database.yml file, we're ready to deploy. Once we've committed any changes and pushed them to the remote we've chosen to deploy from, deploying is as simple as entering:

**Terminal**

```
cap production deploy
```

And waiting. The first deploy can take a while as Gems are installed.

## Conclusion

This configuration is based heavily on the vanilla Capistrano configuration, with some extra convenience tasks added in `lib/capistrano/tasks/` to make it quick to setup work flows common to many Rails applications.

I strongly recommend forking my sample configuration and tailoring it to fit the kinds of applications you develop. I usually end up with a few different configurations, each of which is used either for a particular type of personal project or for all of a specific client's applications.

In the following sections we'll cover how to create custom Capistrano tasks and look at each of the configuration files from the sample configuration in detail. The remaining chapters provide a reference for the sample configuration along with the information required to customise it rather than step by step instructions for re building it from scratch.

# Writing Custom Capistrano Tasks

## Overview

As with Chef, a big part of Capistrano's power comes from how easy it is to write custom extensions to automate tasks specific to a particular application.

One of the big draws of Capistrano 3 is that because it's based around Rake, any developer who can write Rake tasks will be able to quickly adapt to writing Capistrano tasks.

In this section we'll cover writing a very simple task which runs a single command on a remote server.

## File Structure

At the end of the Capfile is this line:

**Capfile**

```
Dir.glob('lib/capistrano/tasks/*.cap').each { |r| import r }
```

Which loads any files with the `.cap` extensions stored in `lib/capistrano/tasks`.

It's good practice to name the file after the task name. So for this example the file is called `restart.cap`.

## The Task

Our task does a single, seemingly very simple thing. It connects to the remote server and executes the init.d script to restart unicorn.

The complete source for the task looks is below and should be stored in `lib/capistrano/tasks`. It's worth mentioning that unlike much of Rails, the filename is not used, so although it's good practice to name the file according to the task it defines for clarity, not doing so will not prevent it from loading.

**lib/capistrano/tasks/restart.cap**

```
namespace :deploy do
  desc 'Restart unicorn application'
  task :restart do
    on roles(:app), in: :sequence, wait: 5 do
      sudo "/etc/init.d/unicorn_#{fetch(:full_app_name)} restart"
    end
  end
end
```

The first line:

**lib/capistrano/tasks/restart.cap (extract)**

```
namespace :deploy do
```

Defines the task as being in the deploy namespace. In practice this means that it will be invoked in the following format:

**lib/capistrano/tasks/restart.cap (extract)**

```
cap stage_name deploy:whatever_the_task_is_called
```

Where `deploy` is the namespace defined above.

The next line:

**lib/capistrano/tasks/restart.cap (extract)**

```
desc 'Restart unicorn application'
```

Provides a human readable description of the task. This is optional and used primarily for outputting available task listings. At time of writing this functionality wasn't fully working in Capistrano 3 so description for custom tasks were not included.

The next line:

**lib/capistrano/tasks/restart.cap (extract)**

```
task :restart do
```

What's hopefully clear once again here is that so far, this really is just defining rake tasks. This means that all the tricks we already know for doing clever things in Rake, can be re-used here.

Next we move onto something Capistrano specific:

**lib/capistrano/tasks/restart.cap (extract)**

```
on roles(:app), in: :sequence, wait: 5 do
```

Understanding this line is essential to writing any Capistrano tasks and for understanding how Capistrano can help as the application scales.

When we user `server` to define a server in stage file, we specify an array of roles which that server has. Examples of these might be 'web' for a web frontend or 'assets' for a dedicated asset server. While in our simple example we only have one server, eventually each stage might have many servers representing many different web frontends.

When we define tasks the `on` syntax above allows us to choose which roles the task should be run on. Capistrano will then take care of running it on all servers which have that role.

The `in: sequence` part of the definition means that the task will be run on each server with that role defined, one by one, rather than in parallel. If we wanted it to be run in parallel we'd simply use `in: parallel` instead. The final part - `wait: 5` - means that there will be a 5 second delay in between executing the command on each server.

In this scenario the in sequence execution with a brief delay is to avoid all web frontends restarting at exactly the same moment. While zero downtime deployment largely mitigates the problems associated with all frontends restarting at once, it is still possible to run into problems with database connection pools or any other resource which is utilised heavily during app startup so unless you have a specific requirement that restarts occur at the same time, it's preferable not to.

Finally the following:

**lib/capistrano/tasks/restart.cap (extract)**

```
sudo "/etc/init.d/unicorn_#{fetch(:full_app_name)} restart"
```

Defines the command to be executed. Here we're running a system command which must be run as root so we use the `sudo` method from the SSHKit DSL. If we wanted to run the command as the current user, in our case generally 'deploy' we could simply use `execute`.

## A note on Namespacing

In projects with a large number of tasks, the final line of the Capfile can be modified to read:

**Capfile (extract)lang=ruby**

```
Dir.glob('lib/capistrano/tasks/**/*.cap').each { |r| import r }
```

This means that tasks will also be loaded from subfolders. Subfolders should represent namespaces with filenames reflecting task names.

# Unicorn Configuration and Zero Downtime Deployment (Deprecated)

Note: this chapter is due to be replaced in the fifth edition with a chapter on Puma

## Overview

In this section we'll look briefly at what the Unicorn server is and how it fits into the process of serving a request. We'll then look at the configuration provided in the sample code before moving onto how zero downtime deployment works and how to troubleshoot it when there are problems.

## Unicorn and the request flow

Unicorn is a Ruby HTTP Server, specifically it's a HTTP server designed for Rack applications, Rack is what Rails uses for HTTP handling.

If you look at the Unicorn documentation, you'll see that it states it is designed "to only serve fast clients on low-latency, high-bandwidth connections". This might initially seem unreasonably picky! Surely we have no control over how fast our clients are?!

In practice what is meant is that Unicorn is not designed to communicate with the end user directly. Unicorn expects that something like Nginx will be responsible for dealing with requests from the user and when required, Nginx will request a response from the Rails app via Unicorn and then deal with returning that to the user. This means that Unicorn only needs to be good at serving requests very quickly, over very high bandwidth connections (either locally or a LAN) and need not handle the complexities of dealing with slow requests or queuing up large numbers of requests. This fits with the Ruby philosophy we're used to, each component of the system should do one thing, very well.

So when a request comes into your server, it first goes to Nginx, if the request is for a static asset (e.g. anything in the public folder), Nginx deals with the request directly. If it is for your Rails application, it proxies the request back to Unicorn. This behaviour is not automatic or hidden, it's defined in your Nginx Virtualhost. More information on these is in chapter 18.0.

Unicorn is what's called a preforking server. This means that we we start the Unicorn server, it will create a master process and multiple child "worker" processes". The master process is responsible for receiving requests and then passing them to a worker, in simple terms the master process maintains a queue of requests and passes them to worker processes as they become available (e.g. finish processing their previous request).

If you want to understand more about the benefits of a pre-forking server such as Unicorn, this post on why Github switched to Unicorn is well worth reading https://github.com/blog/517-unicorn. For more about the process of creating and managing worker processes, this post http://tomayko.com/writings/unicorn-is-unix includes some great examples and extracts from the Unicorn source to make it clearer.

While there's no need to understand the details of Unicorn's internals in detail, a basic understanding that Unicorn:

- Uses multiple Unix processes, one for the master and one for each worker
- Makes use of Unix signals http://en.wikipedia.org/wiki/Unix_signal to control these processes

Will make troubleshooting issues around zero downtime deployment much easier.

In general, we start the master process and this master process takes care of spawning and managing the worker processes.

## Basic Configuration

The basic Unicorn configuration is copied to the remote server when we run `cap deploy:setup_config`. It is stored in `shared/unicorn.rb`

**config/deploy/shared/unicorn.rb.erb (extract)**

```
root = "<%= current_path %>"
working_directory root
pid "#{root}/tmp/pids/unicorn.pid"
stderr_path "#{root}/log/unicorn.log"
stdout_path "#{root}/log/unicorn.log"

listen "/tmp/unicorn.<%= fetch(:full_app_name) %>.sock"
worker_processes <%= fetch(:unicorn_worker_count) %>
timeout 40
...
```

We set the working directory for Unicorn to be the path of the release, e.g. `/home/deploy/APP_NAME/current`.

We have a pid file written to the `tmp/pids` sub-directory of our app root. Notice that the `tmp/pids` directory is included in `linked_dirs` in our deploy.rb file. This means that our pid is stored in the shared folder and so will persist across deploys. This is particularly important when setting up zero downtime deploys as the contents of current will change but we will still need access to the existing pids.

We then set both errors and standard logging output to be stored in `log/unicorn.log`. If you prefer you can setup separate logfiles for errors and standard logging output.

The `listen` command sets up the unicorn master process to accept connections on a unix socket stored in `/tmp`. A socket is a special type of unix file used for inter process communication. In our case it will be used for allowing Nginx and the Unicorn master process to communicate. The socket will be named `unicorn.OUR_APP_NAME.sock`. The `.sock` is a convention to make it easy to identify the file as a socket and the use of our app name prevents collisions if we decide to run multiple Rails app on the same server.

`worker_processes` sets the number of worker processes as per our stage files.

Finally `timeout` sets the maximum length a request will be allowed to run for before being killed and a 500 error returned. In the sample configuration this is set to 40 seconds. This is generally too generous for a modern web application, typically a value of 15 or below is acceptable. In this case the long timeout is because it's not unusual for apps being put into production for the first time to have some "Rough Edges" with a few requests, often admin ones, taking a long time. A tight timeout value to start with can make getting set up frustrating. Once your app is up and running smoothly, I'd suggest decreasing this based on the longest you'd expect a request to your specific app to take, plus a margin of 25 - 50% for error.

## Unix Signals

We mentioned earlier that Unicorn uses Unix signals to allow for communication with both the master process and the individual worker processes. In general we will only ever communicate directly with the master process which is then responsible for sending appropriate signals onto the work processes.

The Unix command for sending signals to processes is:

```
kill -signal pid
```

Where `signal` is the signal to be sent and `pid` is the process id of the recipient process. This is often confusing because kill is generally associated with terminating processes but we can see from it's `man` page that it's more versatile:

```
The kill utility sends a signal to the processes specified by the pid op-
    erand(s).
```

It's worth getting familiar with the key types of signal which the Unicorn master process responds to here http://unicorn.bogomips.org/SIGNALS.html.

Something to be aware of is that Unicorns use of signals is not entirely standard. Specifically it is standard for the `QUIT` signal to be used to tell a process to exit immediately `TERM` to trigger a graceful shut down, e.g. to allow the processes to go through their normal shut down process and clean up after themselves.

Unicorn however uses `QUIT` to trigger a graceful shut down, which allows any workers to finish processing the current request before shutting down. `TERM` is used to immediately kill all worker processes and then immediately stop the master process.

In general this does not effect us as we will use an `init.d` script for interacting with the master process. Understanding this difference does however make debugging problems with the `init.d` script should they arise, easier.

## Unicorn Init script

This is the primary script for starting, stopping and restarting the unicorn workers.

The unicorn init script is created when we run `cap deploy:setup_config`. It is stored in `config/deploy/shared/unicorn_init.sh.erb` and symlinked to `/etc/init.d/unicorn_YOUR_APP_NAME`.

In a perfect world, we have no direct interaction with this script at all, the `deploy:restart` Capistrano task takes care of calling it after deploys and, like any other Capistrano task, we can invoke it on it's own if needed. As with all other processes, we leave Monit to take care of restarting it if there are problems.

In practice however there will undoubtedly be times when you're SSH'd into a server and need to quickly start or restart the Unicorn workers so it's worth getting familiar with it directly.

Basic usage is simple:

**Terminal**

```
/etc/init.d/unicorn_YOUR_APP_NAME COMMAND
```

Where COMMAND will generally be one of `start`, `stop` and `restart`, `force-stop`.

The `restart` command is covered in detail below in the Zero Downtime Deployment section but first we'll take a brief look at what the `start`, `stop` and `force-stop` command are doing.

Each of these commands makes use of the `sig` function defined in the init script:

**/etc/init.d/unicorn_YOUR_APP_NAME (extract)**

```
  sig () {
    test -s "$PID" && kill -$1 `cat $PID`
  }
```

This tests to see if the Pidfile exists and if so sends the supplied signal to the process specified in it.

### Start

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME start`.

Defined in the init script as:

**config/deploy/shared/unicorn_init.sh.erb (extract)**

```
start)
  sig 0 && echo >&2 "Already running" && exit 0
  run "$CMD"
  ;;
```

This invokes the `sig` function but with a signal of `0`. In practice this just looks to see if there is a Pidfile, which indicates that Unicorn is already running, if so `sig` will return `0` (success) since `kill` with a signal of 0 sends no signal. This means that if a Pidfile already exists, the start command will output "Already Running" and then exit.

If on the other hand no Pidfile exists, `$CMD`, which is the Unicorn start command, is executed and Unicorn is started.

## Stop

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME stop`.

Defined in the init script as:

**config/deploy/shared/unicorn_init.sh.erb (extract)lang=bash**

```
stop)
  sig QUIT && exit 0
  echo >&2 "Not running"
  ;;
```

This sends the "QUIT" signal to the Unicorn master process (if the Pidfile exists) which the Unicorn manual (http://unicorn.bogomips.org/SIGNALS.html) explains signals a:

```
 graceful shutdown, waits for workers to finish their current request before fini\
shing.
```

If the Pidfile does not exist then it outputs the text "Not running" and exits.

## Force-stop

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME force-stop`.

Defined in the init script as:

**config/deploy/shared/unicorn_init.sh.erb (extract)lang=bash**

```bash
force-stop)
  sig TERM && exit 0
  echo >&2 "Not running"
  ;;
```

This operates in the same way as `stop` except it sends the `TERM` signal. which the Unicorn manual explains signals a:

```
quick shutdown, kills all workers immediately
```

## Zero Downtime Deployment

After deploying a new version of your application code, the simplest way to reload the code is to issue the `stop` command followed by the `start` command. The downside of this is that there will be a period while your Rails app is starting up, during which your site is unavailable. For larger Rails applications this startup time can be significant, sometimes several minutes, which makes deploying regularly less attractive.

Zero downtime deployment with Unicorn allows us to do the following:

- Deploy New Code
- Start a new Unicorn master process (and associated workers) with the new code, without stopping the existing master
- Only once the new master (and associated workers) has loaded, stop the old one and start sending requests to the new one

This means that we can deploy without our users experiencing any downtime at all.

In our init script, this is taken care of be the `restart` task:

**config/deploy/shared/unicorn_init.sh.erb (extract)**

```bash
sig USR2 && echo reloaded OK && exit 0
  echo >&2 "Couldn't reload, starting '$CMD' instead"
  run "$CMD"
  ;;
```

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME restart`.

This sends the `USR2` signal which the Unicorn manual states:

```
re-execute the running binary. A separate QUIT should be sent to the original pro\
cess once the child is verified to be up and running.
```

This takes care of starting the new master process, once the new master has started, both the old master and the new master will be running and processing requests. Note that because of the `before_fork` block below, we never actually have the scenario where some requests are being processed by workers running the old code and some running the new.

As described in the above extract from the manual, we must take care of killing the original (old) master once the new one has started. This is handled by the `before_fork` block in our Unicorn config file (`unicorn.rb` on the server, `unicorn.rb.erb` in our `config/deploy/shared` directory locally):

**config/deploy/shared/unicorn.rb.erb (extract)**

```ruby
before_fork do |server, worker|
   defined?(ActiveRecord::Base) and
    ActiveRecord::Base.connection.disconnect!
  # Quit the old unicorn process
  old_pid = "#{server.config[:pid]}.oldbin"
  if File.exists?(old_pid) && server.pid != old_pid
    puts "We've got an old pid and server pid is not the old pid"
    begin
      Process.kill("QUIT", File.read(old_pid).to_i)
      puts "killing master process (good thing tm)"
    rescue Errno::ENOENT, Errno::ESRCH
      puts "unicorn master already killed"
    end
  end
end
```

The `before_fork` block is called when the master process has finished loading, just before it forks the worker processes.

The block defined above begins by gracefully closing any open ActiveRecord connections. It then checks to see if we have a Pidfile with the `.oldbin` extension which is automatically created by Unicorn when handling a `USR2` restart. If so it sends the “QUIT” signal to the old master process to shut it down gracefully. Once this completes, our requests are being handled by just the new master process and its workers, running our updated application code, with no interruption for people using the site.

The final section of our Unicorn config file (`unicorn.rb`) defines an `after_fork` block which is run once by each worker, once the master process finishes forking it:

**config/deploy/shared/unicorn.rb.erb (extract)lang=bash**

```
after_fork do |server, worker|
  port = 5000 + worker.nr
  child_pid = server.config[:pid].sub('.pid', ".#{port}.pid")
  system("echo #{Process.pid} > #{child_pid}")
   defined?(ActiveRecord::Base) and
    ActiveRecord::Base.establish_connection
end
```

This creates Pidfiles for the forked worker process so that we can monitor it individually with Monit. It also establishes an ActiveRecord connection for the new worker process.

## Gemfile Reloading

There are two very common problems which are complained of when using the many example Unicorn zero downtime configurations available:

- New or updated gems aren't loaded, so whenever the Gemfile is changed, the application has to be stopped and started again.
- Zero downtime fails every fifth or so deploy and the application has to be manually started and stopped. It then works again for five or so deploys and the cycle repeats.

Both of these are generally caused by not setting the `BUNDLE_GEMFILE` environment variable when when a `USR2` restart takes place.

If this is not specified then the `Gemfile` path from when the master process was first started will be used. Initially it may seem like this is fine, our Gemfile path is always going to be `/home/deploy/apps/APP_NAME/current/Gemfile` therefore surely this should work correctly?

In practice however this is not the case. When deploying with Capistrano, the code is stored in `/deploy/apps/APP_NAME/releases/DATESTAMP` for example `/deploy/apps/APP_NAME/releases/20140324162017` and the `current` directory is a symlink which points to one of these release directories.

The Gemfile path which will be used by the Unicorn process is the resolved symlink path, e.g. `/deploy/apps/APP_NAME/releases/20140324162017` rather than the `current` directory. This means that if we don't explicitly specify that the `Gemfile` in `current` should be used it will always use the one from the release in which the master process was first started.

In `deploy.rb` we have this:

```
set :keep_releases, 5
```

to have Capistrano delete all releases except the five most recent ones. This means that if we're not setting the Gemfile path back to current on every deploy, we'll eventually delete the release which contains the Gemfile Unicorn is referencing, this will prevent a new master process from starting and you'll see a

```
Gemfile Not Found
```

Exception in `log/unicorn.log` file.

We avoid the above two problems with the following `before_exec` block in our Unicorn configuration file:

**config/deploy/shared/unicorn.rb.erb (extract)lang=bash**

```
# Force unicorn to look at the Gemfile in the current_path
# otherwise once we've first started a master process, it
# will always point to the first one it started.
before_exec do |server|
  ENV['BUNDLE_GEMFILE'] = "<%= current_path %>/Gemfile"
end
```

`before_exec` is run before Unicorn starts the new master process so setting the `BUNDLE_GEMFILE` environment variable here ensures it will always be set to the `current` directory not a release one.

## Troubleshooting Process for Zero Downtime Deployment

Zero downtime deployment can be tricky to debug, the following process is a good starting point for debugging problems with code not reloading or the application appearing not to restart:

- Open one terminal window and ssh into the remote server as the deploy user
- Navigate to `~/apps/YOUR_APP_NAME/shared/logs`
- Execute `tail -f unicorn.log` to show the Unicorn log in real time
- In a second terminal window ssh into the remote server as the deploy user
- In the second terminal execute `sudo /etc/init.d/unicorn_YOUR_APP_NAME restart` (check the naming by looking in the '/etc/init.d/ directory)
- Now watch the log in the first terminal, you should see output which includes `Refreshing Gem List` and eventually (this may take several minutes) `killing master process (good thing tm)`

If this fails with an exception, the exception should give good pointers as to the source of the problem.

If this works but restarts after deploys are still not working then repeat the process but in the second terminal, instead of ssh'ing into the remote server, execute `cap STAGE deploy:restart` and watch the log to see if behaviour is different.

# Nginx Virtualhosts and SSL

## Overview

When a request reaches Nginx, it looks for a virtualhost which defines how and where the request should be routed. Nginx virtualhosts are stored in `/etc/nginx/sites-enabled`.

## A Basic Virtualhost

The below virtualhost is the one included in the sample Capistrano configuration. It includes the rules necessary to route requests to a specific domain to our Rails app, including correctly handling compiled assets:

**config/deploy/shared/nginx.conf.erb (extract)**

```
upstream unicorn_<%= fetch(:full_app_name) %> {
  server unix:/tmp/unicorn.<%= fetch(:full_app_name) %>.sock fail_timeout=0;
}

server {
  server_name <%= fetch(:server_name) %>;
  listen 80;
  root <%= fetch(:deploy_to) %>/current/public;

  location ^~ /assets/ {
    gzip_static on;
    expires max;
    add_header Cache-Control public;
  }

  try_files $uri/index.html $uri @unicorn;

  location @unicorn {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://unicorn_<%= fetch(:full_app_name) %>;
  }

  error_page 500 502 503 504 /500.html;
  client_max_body_size 4G;
  keepalive_timeout 10;
}
```

The `upstream` block allows us to define a server or group of servers which we can later refer to when using `proxy_pass`:

**extract from: config/deploy/shared/nginx.conf.erb**

```
upstream unicorn_<%= fetch(:full_app_name) %> {
  server unix:/tmp/unicorn.<%= fetch(:full_app_name) %>.sock fail_timeout=0;
}
```

Here we define a single server as `unicorn_APP_NAME` which points to the unix socket we've defined for our unicorn server in `config/deploy/shared/unicorn.rb`.

The server block begins by defining the port, root and server name:

**config/deploy/shared/nginx.conf.erb (extract)**

```
server {
  server_name <%= fetch(:server_name) %>;
  listen 80;
  root <%= fetch(:deploy_to) %>/current/public;

  ...
  }
```

The `listen` directive defines the port this virtualhost applies to. In this case any request on port 80, the standard port for web http requests.

The `server_name` directive defines the hostname that the virtualhost represents. This will usually be a domain/ subdomain such as `server_name www.example.com`. You can also use wildcards here such as `server_name *.example.com` and list multiple entries on one line, separated by spaces such as `server_name example.com *.example.com`.

In the example Capistrano configuration, this is set in our stage file, for example:

**config/deploy/production.rb (extract)**

```
set :server_name, "www.example.com example.com"
```

The `root` directive defines the root directory for requests. So if the root directory was `/home/deploy/your_-app_production/current/public/` then a request for `http://www.example.com/a_file.jpg` would route to `/home/deploy/your_app_production/current/public/a_file.jpg`.

The next section defines behaviour specific to the folder containing our compiled assets:

**config/deploy/shared/nginx.conf.erb (extract)**

```
location ^~ /assets/ {
  gzip_static on;
  expires max;
  add_header Cache-Control public;
}
```

The `location` directive allows us to define configuration parameters which are specific to a particular URL pattern defined by a regular expression. In this case to the folder which contains our static assets.

The `gzip_static on` directive specifies that any files in this directory can be served in a compressed form.

The `expires max` directive enables the setting of the `Expires` and `Cache-Control` headers to values which will cause compliant browsers to cache the returned files as long as possible.

The `add_header Cache-Control public` directive adds a flag which is used by proxies to determine whether the file in question is safe to cache.

The next section defines the order in which files should be looked for:

**config/deploy/shared/nginx.conf.erb (extract)**

```
try_files $uri/index.html $uri @unicorn;
```

This means that it will first look for the existence of the file provided by the url with `/index.html` appended. This allows for urls like `/blog/` to display the `/blog/index.html` page automatically. If that is not found then it will look for the file specified by the URL, if that is not found that in will pass the request onto the `@unicorn` location (explained below). This means that any valid requests for static assets will never be passed to our Rails app server.

The `location @unicorn` block defines the `@unicorn` location used to pass request back to our Rails app (which we looked at in the previous lines):

**config/deploy/shared/nginx.conf.erb (extract)**

```
location @unicorn {
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header Host $http_host;
  proxy_redirect off;
  proxy_pass http://unicorn_<%= fetch(:full_app_name) %>;
}
```

This sets headers which provide additional information about the request itself and then uses `proxy_pass` to forward the request onto our Rails application using the `unicorn_full_app_name` location we defined at the top of this file.

Notice that the `proxy_pass` destination starts with `http://` and looks like any other web address. It is, and this is really powerful. In this case we're using it to proxy requests back to our unicorn app server but we could also use it to proxy all or a subset of requests back to any other destination. This can for example be used to make a wordpress blog appear to be on the same domain as your rails app(!).

**config/deploy/shared/nginx.conf.erb (extract)**

```
error_page 500 502 503 504 /500.html;
client_max_body_size 4G;
keepalive_timeout 10;
```

The `error_page` directive indicates that all of the error codes listed should lead to the page at `$root/500.html` being displayed. Since `root` is the public folder of our Rails application, if we haven't defined a custom 500 page, this will be the "We're sorry, but something went wrong (500)" in red text page we all know and love.

Finally `client_max_body_size` defines the maximum size of the request body which will be accepted without an error being generated and `keepalive_timeout` defines how long keepalive connections will be allowed to remain open without activity before being automatically closed.

## DNS Overview

### A Records

It's beyond the scope of this book to go into setting up the DNS for your domain in detail. In general, if your `server_name` is `www.example.com` then you would need to have an A Record setup pointing `www` at the the IP of your server.

### Testing with /etc/hosts

If, for testing purposes, you need to test whether your server will work before adding the DNS entry, you can modify your local hosts file.

On both OSX and most *nix variants, you'll find this in `/etc/hosts`. Editing this will require root access. You can add lines such as the following:

```
89.137.174.152 www.example.com
```

To make `www.example.com` resolve to `89.137.174.152` locally.

## Forcing HTTPS

For some sites it can be desirable to only serve requests over https even when the user request http. To achieve this, replace the server block for port 80 with:

```
rewrite     ^   https://$server_name$request_uri? permanent;
```

To automatically redirect all `http://` requests to the `https://` equivilent.

## Adding SSL

To add an SSL Certificate you will need:

- Your SSL Certificate
- Any intermediate Certificates (see Chaining SSL Certificates below)
- Your SSL Private Key

To enable SSL in Nginx we have a section in `config/deploy/shared/nginx.conf.erb` which is included if we set `:enable_ssl` to `true` in a stage file (for example `config/deploy/production.rb`).

A majority of this file is the same as the none SSL definition, with the following differences:

This line;

**extract from: config/deploy/shared/nginx.conf.erb**

```
listen 443;
```

Means that the virtualhost applies to connections on port 443, the standard port for HTTPS connections.

These lines:

**extract from: config/deploy/shared/nginx.conf.erb**

```
ssl on;
ssl_certificate <%= fetch(:deploy_to) %>/shared/ssl_cert.crt;
ssl_certificate_key <%= fetch(:deploy_to) %>/shared/ssl_private_key.key;
```

This enables SSL and sets the location of the SSL certificate and corresponding private Key. These should not be included in source control so you'll need to SSH into your server and create the relevant files. Both should be provided by your certificate supplier but see the note below on chaining SSL certificates.

Once you've added your certificates and run `deploy:setup_config` again to copy the updated NGinx vitualhost across, you'll need to restart or reload nginx for the changes to be picked up.

## Chaining SSL Certificates

Often when purchasing SSL certificates, you'll be provided with your own certificate, an intermediate certificate and a root certificate. These should all be combined into a single file, for the example configuration this should be called `ssl_cert.crt`.

You'll begin with three files:

- Your Certificate
- Primary Intermediate CA
- Secondary Intermediate CA

These should be combined into a single file in the order:

```
YOUR CERTIFICATE
SECONDARY INTERMEDIATE CA
PRIMARY INTERMEDIATE CA
```

You can do this with the cat command:

```
cat my.crt secondary.crt primary.crt > ssl_cert.crt
```

In the final section of this chapter, "Updating SSL Certificates", there's a simple bash script for automating this concatenation as part of the process of switching out old certificates for new ones.

For the purposes of setting up SSL on NGinx, it's not necessary to understand why chaining is used however for reference, there's a good explanation of it in this superuser answer http://superuser.com/questions/347588/how-do-ssl-chains-work.

## Updating SSL Certificates

SSL Certificates will, after a pre agreed time period (often one year), expire and need to be replaced with new ones. It's worth having reminders in your calendar for the expirations dates of any production certs as there's nothing more embarrassing than your site throwing security warnings to all of your users because nobody remembered to renew the certificate.

Updating them is quite a simple process but one it's easy to get wrong. For this reason I use a simple bash script to automated switching the old cert for the new one and rolling back in case it all goes wrong.

This script assumes your issuer provides you with a certificate, a secondary certificate and a primary certificate (based on certificates from DNSimple). You may need to modify the script if, for example, there's no intermediate certificate.

The script assumes you're starting with 4 files in `/home/deploy/YOUR_APP_DIRECTORY/shared`:

- `my.crt` - Your SSL Certificate
- `secondary.crt` - A secondary intermediate certificate
- `primary.crt` - A root certificate
- `ssl_private_key.key.new` - The private key used to generate the certificate

The following bash script provides a simple interface for switching in new certs and rolling back in the case that something goes wrong. The script should be stored in the same directory as the target for the certificates. In the case of our sample configuration, this is `/home/deploy/your_-app_environment/shared/`.

```
#!/bin/bash

if [ $# -lt 1 ]
then
        echo "Usage : $0 command"
        echo "Expects: my.crt, secondary.crt, primary.crt, ssl_private_key.key.ne\
w"
        echo "Commands:"
        echo "load_new_certs"
        echo "rollback_certs"
        echo "cleanup_certs"
        exit
fi

case "$1" in

load_new_certs)  echo "Copying New Certs"
    cat my.crt secondary.crt primary.crt > ssl_cert.crt.new

    mv ssl_cert.crt ssl_cert.crt.old
    mv ssl_cert.crt.new ssl_cert.crt

    mv ssl_private_key.key ssl_private_key.key.old
    mv ssl_private_key.key.new ssl_private_key.key

    sudo service nginx reload
    ;;
rollback_certs)  echo  "Rolling Back to Old Certs"
    mv ssl_cert.crt ssl_cert.crt.new
    mv ssl_cert.crt.old ssl_cert.crt

    mv ssl_private_key.key ssl_private_key.key.new
    mv ssl_private_key.key.old ssl_private_key.key

    sudo service nginx reload
    ;;
cleanup_certs)  echo  "Cleaning Up Temporary Files"
    rm ssl_cert.crt.old
    rm ssl_private_key.key.old
    rm my.crt
    rm secondary.crt
    rm primary.crt
    ;;
*) echo "Command not known"
   ;;
esac
```

Don't forget to make the script executable with `chmod +x script_name.sh`.

You can then simply run:

```
./script_name load_new_certs
```

to swap in the new certificates and reload nginx. If, after testing the site, something isn't right, you can execute:

```
./script_name rollback_certs
```

To revert to the previous ones. And then repeat `load_new_certs` once you've resolved the issue.

Once you have the new certificates working as intended, you can use:

```
./script_name cleanup_certs
```

To remove the temporary and legacy files created.

# Sidekiq

## Overview

Sidekiq is a simple and extremely memory efficient background job processing library for Ruby. It's not Rails specific but has very tight Rails integration available. It is a drop in replacement for Resque and offers huge memory savings. In this chapter we'll look at how to automatically start and restart Sidekiq workers when we deploy and use Monit to ensure it continues running.

## Sidekiq Version 3

This chapter has been written primarily for Sidekiq 3.x but should also be compatible with Sidekiq 2.x

There's more about the changes in version 3 here http://www.mikeperham.com/2014/03/28/sidekiq-3-0/.

## Capistrano Integration

In Sidekiq 3 Capistrano integration has been factored out into a gem `capistrano-sidekiq`. To include this functionality simply add the following to the development group of your `Gemfile`:

**Gemfile (extract)**

```
gem 'capistrano-sidekiq'
```

Then add the following lines to your `Capfile`:

**Capfile (extract)**

```
require 'capistrano/sidekiq'
require 'capistrano/sidekiq/monit'
```

This will automatically add the following hooks:

```
after 'deploy:starting', 'sidekiq:quiet'
after 'deploy:updated', 'sidekiq:stop'
after 'deploy:published', 'sidekiq:start'
```

This means that when the deploy starts (`deploy:starting`), the following will be automatically issued to instruct the Sidekiq worker process not to start processing any new jobs:

```
sidekiqctl quiet SIDEKIQ_PID
```

Once our application code has been updated, the following will be issued to stop the worker process:

```
sidekiqctl quiet SIDEKIQ_PID
```

And finally once the new version of the app has been published, a worker process using the new codebase will be started with:

```
bundle exec sidekiq ...(options)
```

This is all completely transparent to us and requires no additions to the deploy code other than requiring the Capistrano tasks in our `Capfile`.

The reason for doing this in three steps rather than one is to allow Sidekiq as much time as possible to gracefully finish processing any existing jobs.

Before we deploy again, we need to setup the sidekiq monit configuration with:

**Terminal**

```
bundle exec cap production sidekiq:monit:config
bundle exec cap production monit:reload
```

This assumes that we're either using `capistrano-cookbook` which includes the tasks for restarting monit or have included something like https://github.com/TalkingQuickly/capistrano-3-rails-template/blob/master/lib/capistrano/tasks/monit.cap in `lib/capistrano/tasks`. If not then after executing `bundle exec cap production sidekiq:monit:config` we'd need to SSH into the server and execute 'sudo monit reload'.

We can then run `bundle exec cap production deploy` and once complete, sidekiq will be up and running and monitored by `monit`.

# Backups

## Overview

It is unfortunately true that eventually, our production database server will fail. Having regular backups will be the difference between this being a bad day and a really bad day.

In this chapter we'll look out how to use the `backup` gem to configure regular database backups to an off server location, such as Amazon S3.

These are regular backups, not real-time backups. If you setup hourly backups and have only a single database server, then it's possible for you to lose up to an hour of data. For many small applications, this risk is considered acceptable. If, for your application, this is not acceptable, then it's important to look into database redundancy, as a minimum setting up some form of master, slave replication.

## Structure

One of the most common issues when setting up the backup gem is that if included in the Gemfile of the Rails app, it will tend to cause dependency conflicts. Including the gem in the Rails Gemfile is a mistake. The backup script should be treated as a separate app, with its own Gemfile, which is kept in a sub-folder of the main Rails application.

This means that special care should be taken that as part of your deploy process, `bundle install` is run separately within the backup folder to ensure its dependencies are installed. A sample Capistrano task hooked into `setup_config` is included towards the end of this chapter.

The only gem which should be included in the Rails application is the Whenever gem which is responsible for generating the cron jobs which run the backups.

## Getting Started

The example configuration is available here: https://github.com/TalkingQuickly/rails_backup_template.

The `backup` folder from this repository should be copied into the root of your Rails project. The `lib/capistrano/tasks` directory should be merged with the existing one in your Rails application.

### The Master Configuration File

The `backup` folder contains the following master configuration file:

**backup/config.rb**

```ruby
##
# Backup v4.x Configuration
#
# Documentation: http://meskyanichi.github.io/backup
# Issue Tracker: https://github.com/meskyanichi/backup/issues

require 'yaml'
require 'dotenv'

# Get our environment variables
Dotenv.load

# Get the current Rails Environment, otherwise default to development
RAILS_ENV = ENV['RAILS_ENV'] || 'development'

# Load database.yml, including parsing any ERB it might
# contain.
DB_CONFIG = YAML.load(ERB.new(File.read(File.expand_path('../../config/database.y\
ml',  __FILE__))).result)[RAILS_ENV]

# Set defaults for S3 which can be shared across multiple backup models
Storage::S3.defaults do |s3|
  s3.access_key_id     = ENV['AWS_ACCESS_KEY_ID']
  s3.secret_access_key = ENV['AWS_SECRET_ACCESS_KEY']
  s3.region            = "us-east-1"
end
```

A common issue when dealing with the `config.rb` file is that the lines:

**extract from backup/config.rb**

```ruby
##
##
# Backup v4.x Configuration
```

Are required for the file to be recognised as a `backup` v4 configuration file. Without these lines, when we try and execute the backup, an error will be thrown that `config.rb` is not a valid `backup` v4 configuration file.

The configuration begins by initializing the dotenv gem which allows us to load environment variables from a configuration file. In this case `backup/.env`. In development, my `.env` looks something like this:

**backup/.env**

```
AWS_ACCESS_KEY_ID=access_key_id
AWS_SECRET_ACCESS_KEY=secret_access_key_id
S3_BUCKET=tlq_backup_test
```

Where `access_key_id` and `secret_access_key` correspond to the credentials for an Amazon AWS user with an IAM policy allowing only read/ write access to a test bucket. This `.env` should be added to your `.gitignore` and new one created on your server with credentials of a user who has access only to the production backups bucket.

An example of a suitable IAM Policy for the `tlq_backup_test` bucket would be:

**Amazon S3 IAM Polciy**

```
{
    "Statement": [
        {
            "Effect": "Allow",
            "Action": ["s3:ListBucket" ],
            "Resource": [ "arn:aws:s3:::tlq_backup_test"]
        },
        {
            "Effect": "Allow",
            "Action": [ "s3:PutObject", "s3:GetObject", "s3:DeleteObject"],
            "Resource": [ "arn:aws:s3:::tlq_backup_test/*"]
        }
    ]
}
```

We saw earlier that to avoid dependency conflicts, the `backup` configuration should be treated as a completely separate app which lives in the Rails directory structure. This means that in order to load details about the database to be backed up from the standard Rails `config/database.yml`, we need to manually load the file. If the database in use is Mongo, this should be `mongoid.yml` instead.

When rails loads `database.yml`, it automatically parses any erb found. This allows us to do things such as loading database details and credentials from environment variables. This is replicated, manually in `backup/config.rb` so that the database config for the provided RAILS_ENV, with any erb parsed will be available in the variable `DB_CONFIG`. This variable is available to any of the backup models we then create (see "Backup Models" below).

If you define your database access credentials in a `.env` file already, then you will need to ensure that the values are replicated in the backup gems `.env` file. Alternatively you can choose to have the environment variables automatically hard coded in to the Cron job when it is generated by the whenever Gem (see "Setting Environment Variables in Cron Jobs" below.)

Finally `backup/config.rb` sets some default values for the S3 storage engine which will be automatically accessible when we use the S3 engine in any backup model.

## Backup Models

Each backup job is referred to as a `model`, not to be confused with Rails models, and is stored in the backup/models directory. Our sample backup for a PostgreSQL backup look like this:

**backup/models/rails_database.rb**

```ruby
# encoding: utf-8

##
# Backup Generated: rails_database
# Once configured, you can run the backup with the following command:
#
# $ backup perform -t rails_database [-c <path_to_configuration_file>]
#
# For more information about Backup's components, see the documentation at:
# http://meskyanichi.github.io/backup
#
Model.new(:rails_database, 'Backups of the Rails Database') do

  ##
  # PostgreSQL [Database]
  #
  database PostgreSQL do |db|
    db.name               = DB_CONFIG["database"]
    db.username           = DB_CONFIG["username"]
    db.password           = DB_CONFIG["password"]
    db.host               = DB_CONFIG["host"]
    db.skip_tables        = []
    db.socket             = DB_CONFIG["socket"]
    db.additional_options = ["-xc", "-E=utf8"]
  end

  ##
  # Amazon Simple Storage Service [Storage]
  #
  store_with S3 do |s3|
    s3.path              = "/#{RAILS_ENV}/"
    s3.bucket            = ENV['S3_BUCKET']
  end

  ##
  # Gzip [Compressor]
  #
  compress_with Gzip

end
```

The `database` method requires an adapter type (see: http://meskyanichi.github.io/backup/v4/databases/ for a full list) and supports all major databases. It then takes a block in which we specify the access credentials for that database as well as whether we want to exclude any tables.

An example of where it might be beneficial to skip tables is if we have large cache tables which, in the event of a failure, we would expect to regenerate rather than restore from a backup. For most standard Rails applications, `skip_tables` will be left empty.

In this case we are taking the database access credentials from the variable we defined in `backup/config.rb` which contains the details from our standard `database.yml` or `mongoid.yml`.

The `store_with` requires a storage location type and takes a block in which we specify the access credentials and paths. Notice that in `backup/rails_database.rb`, we do not have to specify our S3 credentials as we've already defined defaults in `backup/config.rb` which are automatically available to all of our backup models.

The backup gem accepts a large range of storage locations including S3, Dropbox and SFTP (full list: http://meskyanichi.github.io/backup/v4/storages). The important thing is to ensure that your backup location(s) are isolated from your production environment. For example backing up to another node in the same datacentre, while providing some redundancy, doesn't protect you in the case of the entire datacenter being inaccessible. In the same way backing up to S3 does not protect you against your AWS credentials being compromised if the rest of your infrastrucre is on EC2.

The `compress_with` method is fairly self explanatory, it accepts the name of a compressor (full list: http://meskyanichi.github.io/backup/v4/compressors) to keep the size of backups down.

## Manually Triggering the Backup

Later in this chapter we'll setup Capistrano tasks for when we want to manually trigger backups as well as a cron job to automatically trigger them at regular intervals. It is however useful to first look briefly at how the backups can be triggered from the command line of the server.

Once you have created your backup configuration, based either on the sample configuration or by generating a new one using the built in generator (see: http://meskyanichi.github.io/backup/v4/generator/) commit all of the files to your repository and deploy your application.

Now SSH into the server you've deployed to.

Firstly, if you've added your `backup/.env` file to your `.gitignore` as advised, you'll need to re-create this file on the server with suitable credentials.

Our backup app is entirely separate from our Rails app and has it's own Gemfile. Therefore for now, we'll need to move into `APP_DIRECTORY/backup` and run `bundle install` to install the relevant gems. We'll automate this process with Capistrano later.

Now, still in the same folder, we can trigger our backup:

```
bundle exec backup perform --trigger rails_database --config-file ./config.rb
```

If all is well, we'll now see the progress of our backup and a message confirming that it was succesful. It's extremely important to double check the expected destination after doing this to ensure that the files were actually created. This is also a good point to do a "test restore" to make sure the backups we're creating, are actually usable.

## Capistrano Tasks to Setup and Trigger Backups

Because the backup task is an entirely separate application which simply lives in a subfolder of the Rails application, it requires gems to be installed separately.

In the previous step we did this manually but in practice, we want this to be handled automatically when we setup a new server. The easiest way to do this is to write a simple capistrano task to run `bundle install` in the correct folder and then add a hook in our `deploy.rb` file to run this task automatically when we run our `deploy:setup_config` task.

Generally our backups will be triggered automatically by a cron job but, from time to time, we may want to trigger one manually. For example if we're deploying a new feature which we have some concerns about or about to begin a migration to a new database server. The easiest way to do this is with another Capistrano task.

The below is an example of such a task, it can also be found in `lib/capistrano/tasks` of the backup example code.

**lib/capistrano/tasks.backup.rb (backup sample app)**

```ruby
namespace :backup do
  desc 'sets up backups (install gems)'
  task :setup do
    on primary :web do
      backup_dir = "#{current_path}/#{fetch(:backup_path, 'backup')}"
      within backup_dir do
        with fetch(:bundle_env_variables, {}) do
          execute :bundle
        end
      end
    end
  end

  desc 'triggers the backup job'
  task :trigger do
    on primary :web do
      backup_dir = "#{current_path}/#{fetch(:backup_path, 'backup')}"
      within backup_dir do
        with fetch(:bundle_env_variables, {}).merge(rails_env: fetch(:rails_env))\
 do
          execute :bundle, "exec backup perform --trigger #{fetch(:backup_job, 'r\
ails_database')} --config-file ./config.rb", raise_on_non_zero_exit: false
        end
      end
     end
  end
end
```

We can then add the following to `config/deploy.rb`

**config/deploy.rb (extract)**

```
...
after 'deploy:setup_config', 'backup:setup'
...
```

Which will automatically trigger the setup task to install the backup gems after we run `cap deploy:setup_config`.

We can then run `cap backup:trigger` when we want to manually start a backup.

## Triggering Backups with Whenever

The whenever Gem provides a simple Ruby DSL for creating cron jobs on our target system. More information is available here https://github.com/javan/whenever. Our aim in this section is to have our backup job triggered automatically, every hour.

To get started, add the gem `whenever` to your main applications Gemfile (not the standalone backup application):

**Gemfile (extract)**

```
...
gem 'whenever', '~> 0.9.2'
...
```

Then, after running `bundle`, run:

**Terminal**

```
bundle exec wheneverize .
```

To generate the basic whenever configuration `config/schedule.rb`. We'll now take a look at each part of the `schedule.rb` we're creating in detail.

### Logging

Debugging Cron jobs can be a frustrating experience. A huge number of problems come back to not explicitly setting `PATH` to a known or expected value which is covered below. If however you do need to debug a cron job, having consistent logging setup is essential. The following line in `schedule.rb` will ensure that all output from cron jobs is written to `cron.log` in the usual log directory of the Rails application.

**config/scedule.rb (extract)**

```
...
set :output, "#{path}/log/cron.log"
...
```

## $PATH in Cron Jobs

If we were to inspect `$PATH` by executing `echo $PATH` in a terminal having ssh'd into our remote server, we'd probably see something like:

```
/usr/local/rbenv/shims:/usr/local/rbenv/bin:/usr/local/sbin:/usr/local/bin:/usr/s\
bin:/usr/bin:/sbin:/bin:/usr/bin/X11
```

These are the locations which will be searched for an executable when we enter a command. As we saw earlier in the chapter on rbenv, this means that when `ruby` or `bundle` is executed, the shim from rbenv is found and the version of Ruby we've setup rbenv to use is executed.

If however we were to setup a cronjob to echo `$PATH` to a file, we'd probably see something like:

```
/usr/bin:/bin
```

Notice that none of the rbenv paths are here and so will be be searched when we execute `bundle`. Consequently the system default ruby (if there is one) will be found and none of our gems will be available.

Whenever provides a simple way to ensure that the `$PATH` for cronjobs is the same as for our login shell. We simply add the following to our `schedule.rb` file:

**config/scedule.rb (extract)**

```
...
env :PATH, ENV['PATH']
...
```

When we setup Capistrano integration (later in this chapter), the task to generate the crontab will be executed directly on the remote server over a regular login shell, consequently `PATH` will be identical to said shell.

## Triggering the Backup

Whenevers syntax for the cron jobs themselves is very simple, we use the method `every` followed by an interval. For example `60.minutes` or `1.weeks` and then provide a block with the details of the task to be run.

Whenever is extremely powerful, allowing for multiple different ways of defining the time period and with a more advanced DSL for triggering rake tasks or similar. We're only going to touch the surface of what it can do here for our backup job.

The section to trigger the backup looks like this:

**config/scedule.rb (extract)**

```
....
job_type :backup, "cd :path/:backup_path && :environment_variable=:environment bu\
ndle exec backup perform -t :task --config_file ./config.rb :output"

every 1.minutes do
  backup 'rails_database', backup_path: 'backup'
end
```

The `job_type` method defines a custom backup job type. `:task` is always replaced with the first argument passed in (in this case `rails_database`, `:environment_variable` defaults to `RAILS_ENV` and `:environment` will be set later by the capistrano integration to match `rails_env`, otherwise it defaults to production. The remaining `:variable` options are replaced with the corresponding values in the hash passed in when the job is called.

By default the Whenever Capistrano integration (below) will only setup tasks on nodes with the db role, therefore as long as we only have one node with the db node, our backup task will only get run once, rather than running on every node.

So our final `schedule.rb` looks like this:

**config/scedule.rb (extract)**

```
set :output, "#{path}/log/cron.log"
env :PATH, ENV['PATH']

job_type :backup, "cd :path/:backup_path && :environment_variable=:environment bu\
ndle exec backup perform -t :task --config_file ./config.rb :output"

every 1.hours do
  backup 'rails_database', backup_path: 'backup'
end
```

## Capistrano Integration

To have the crontab automatically generated when we deploy, we simply add the following to our `Capfile`.

**Capfile (extract)**

```
....
require "whenever/capistrano"
....
```

If you want to get fancier with your Capistrano integration, it's worth reading the Capistrano V3 section of the Readme at https://github.com/javan/whenever and looking at the available options in https://github.com/javan/whenever/blob/master/lib/whenever/capistrano/v3/tasks/whenever.rake

## Conclusion

In this chapter we've seen how to setup simple automatic backups of our database to one or more remote locations. One of the most common mistakes, but one most people only make once, is to leave it at that. The most important step is to check, regularly, that the backups can be succesfully restored.