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 2014-05-25

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1.0 - Intro

The purpose of this book

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

- Setup a VPS from Scratch
- Setup 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 config 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.

About Me

Currently based in Google Campus, London, I've been developing web applications for over eight years. Over the last few years specialising in Ruby on Rails development and deployment. I work both as a consultant - primarily to startups - and on projects of my own.

I'm the technical lead at a health and fitness startup who provide the timetabling for many of the UK's public leisure operators as well as producing an international iOS app (Speedo Fit) for swimmers sponsored by a major global fitness brand.

As part of developing the infrastructure for this 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 dealing with the wrath of timezone switching.

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.

It will be useful but not essential if you already have some familiarity with web server structure, e.g. the place of virtual host files, permissions and the like.

How to read this book

This book is structured around sample code, all of which is available on Github. In general each chapter will examine a section of the sample code, explain how it works and how to use it.

From chapter 6 onwards, instead of saying "now add this code to file x," the code being examined will be reproduced in the text and its location within the sample code indicated.

Key commands to perform particular operations will be reproduced but not at the level of:

```
cd xxxxx
mv yyy zzzzz
etc etc
```

It is therefore important to read this book with the code accessible. While each chapter can be read and will make complete sense without directly referring to the full sample code, it's important to spend time exploring and understanding where each file fits in overall.

At a minimum I'd recommend reading each chapter, then opening the files referenced from the sample repository and making sure their contents and structure make sense before moving onto the next chapter.

It is not necessarily envisaged that you'll continue to use the sample code for your own servers (although please feel free to use it as a starting point). Rather that by the end of the book, you'll be comfortable creating a template of your own, customised to your needs.

Structure of Part 1 (Chef)

Chapters 2 and 3 deal with the type of stack covered in this book and language of chef, servers and provisioning.

Chapter 4 is for any 'read the manual afterwards' types and provides the minimal command set needed to get a fully functional server up and running using the example repository.

Chapter 5 follows a traditional step by step approach to get to grips with the structure and workflow of chef. It then covers the basics of how chef projects are structured using as an example a simple Redis configuration, all the way through to applying this Redis configuration to a VPS.

Chapters 6 - 12 move onto using the rails server template configuration included with this book and available at https://github.com/TalkingQuickly/rails-server-template which can serve as a basis for your production configuration.

Using this template configuration we'll cover:

- How best to install Ruby
- Using Monit to ensure everything runs smoothly without our intervention
- Basic security precautions
- Firewall Management with UFW
- Managing users and public keys
- Creating and populating databases (with your provider of choice)
- Setting up Redis and common gotchas
- Setting up Memcached

Chapter 13 covers how to use Vagrant to test your Chef recipes locally.

Structure of Part 2 (Capistrano)

In Part 2 we'll cover deployment (rather than provisioning), in particular:

- What should be managed by Capistrano v Chef
- Creating simple, modular Capistrano Recipes
- Avoiding falling foul of $PATH
- Dealing with Virtualhost files
- Managing SSL Certificates
- Configuring Unicorn
- Zero Downtime Deployment & Gotchas
- Log Rotation
- Copying databases between environments
- Managing Cron jobs with Whenever
- Managing background jobs with Sidekiq

By the end of the book you will have a solid blueprint you can use whenever you need to provision a new server or deploy a new Rails application.

2.0 - 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 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, how to install the component in question, the techniques covered for automating with Chef can be adapted.

Ubuntu 12.04 LTS

Ubuntu has become an increasingly common distribution in a server environment over the last four or five years. 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 renaming 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, it's 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.

Unicorn

Unicorn is a web (HTTP) server for rack applications (of which Rails is one).

If you'd like to understand the internals of Unicorn better, I strongly recommend reading the Github article

https://github.com/blog/517-unicorn

on why they moved from Mongrel to Unicorn.

Architecturally, when requests come into the server, they are handled by Nginx which then passes them back to Unicorn which runs the rails application and returns the response.

For more about the benefits of Unicorn vs Passenger and why the approaches are different, see this Engineyard post; https://blog.engineyard.com/2012/passenger-vs-unicorn

Postgresql / MongoDB / MySQL

Postgresql and MySQL are traditional relational databases. MySQL is probably the best known and due to its track record, is a common choice in larger organisations.

Postgresql 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.

MongoDB is a non-relational database, extremely popular when storing large amounts of unstructured data.

Having managed the provisioning of applications using all of the above, the example code includes all three. My personal preference for most applications and the one I have most experience with is Postgresql so this will be used in a majority of the examples.

Ruby (rbenv)

This book covers and has been tested with Ruby 1.9.x and Ruby 2.0.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) as opposed to rvm which I find more complex.

Anecdotally having tried to do a fair few installs of both on new server builds, I've had far fewer issues with rbenv than rvm. This absolutely shouldn't be taken as a criticism of rvm which I use exclusively on my local development machines.

Redis

Redis is an extremely fast key value store. It's great and very fast for things like caching and api rate limiting.

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's lots of other great combinations out there and the concepts in this book will apply to most of them.

This, with an appropriately selected database, is my go to setup for a new project so hopefully it will serve as a good starting point for your projects as well.

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 write your own Chef recipes to setup almost anything.

3.0 - Chef Definitions

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 you 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 you need” written down should you ever need to do it again.

This approach has a few key pitfalls;

1) It’s very hard to keep track of what you’ve 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, you still need someone 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 long process which I’d expect to repeat more than once or twice a year in the life-cycle of deploying and managing I try and automate.

Automation relates to an approach more than it relates to any particular tool. If you’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 I automate this?”

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 you think. Once you've got your head around chef for automating provisioning tasks and Capistrano for automating your deployment process 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.

The Tools for automating provisioning

Chef & Chef Solo

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.

With chef you can define the steps required to configure a server to fulfill a "role," for example a rails application server or a database server and then apply combinations of these roles to a particular remote machine.

Chef is often run in a hub and spoke style arrangement. A central chef server "knows" the roles that a large number of other servers should have applied to them. If you update the role, the changes are applied to all of those servers automatically.

While this is an excellent and very powerful configuration when managing 10's or 100's of servers, it's overly complicated if we're just looking at managing at managing 1-10 servers.

Luckily it can also be run in a "solo" configuration (chef-solo). Here we use our local development workstation to define server roles and configurations and then manually apply these configurations to servers as and when we need to.

This is perfect for small projects running all parts of the stack on a single box but I've also used it with great success on setups consisting of up to 10 related servers.

If you're project does grow beyond that, almost all of the work done now to automate with chef solo, will be re-usable with chef server.

Knife & Knife Solo

Knife is the command line tool that provides the interface between a local (on our development machine) chef repository and a remote server.

Traditionally this remote server would be the master "chef server" but an additional tool "knife solo" allows us to use chef in solo mode and interact directly with the server we're trying to provision.

More information about knife solo is available on its Github page

https://github.com/matschaffer/knife-solo

and full usage instructions are included in the next chapter.

Berkshelf

The commands to install an individual component on a system are called a “recipe” in chef terminology. For example you may have a recipe for installing ruby and another recipe for installing common rails gem dependencies. Several recipes relating to a particular piece of functionality (for example MySQL server and MySQL client) will often be bundled together in a Cookbook.

Berkshelf is like Bundler for these recipes. You define the recipes your configuration is dependent on, for example the “rbenv” recipe and the “rails_gem_dependencies” recipe. In the same way “bundle install” goes out and gets your gem dependencies (including specific versions), Berkshelf will retrieve the chef recipes (including specific versions) that your server configuration is dependent on.

4.0 - Quick Start - 5 Minute Server

Overview

This section provides a very brief overview of how to use the sample Chef repository to provision a server. If you're the type of person who opens the box, has a go and then reads the instructions afterwards, this section is for you. Otherwise feel free to start from the next section then use this as a quick reference for provisioning servers in future.

The Stack

- Ruby 1.9.3+ (this can be selected)
- Postgres
- Redis

Steps

Install Tools

Begin by cloning the example repository

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

and then running

```
bundle install
```

To install `chef`, `berkshelf`, `knife-solo` and other supporting gems. All three of these key gems are evolving quickly so it's important to always use the Gemfile and `bundle exec` when working with them to ensure we're using a version which has been tested with this book.

Define the server

Next we need to define users, inside data_bags/users copy the file `deploy.json.example` to `deploy.json`.

Generate a password for your deploy user with the command:

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

And update `deploy.json` accordingly. Also copy your SSH public key (`cat ∼/.ssh/id_rsa.pub`) into the public keys array.

Download Cookbooks

Berkshelf is like Bundler for Chef, this command fetches the cookbooks we need and their dependencies:

```
mkdir cookbooks
bundle exec berks install
```

Setup a VPS

This setup is designed to work on any Ubuntu 12.04 VPS, it has been tested on Linode, Rackspace and Digital Ocean. For initial experimentation I'd recommend Digital Ocean or for critical applications where support is key, Linode.

Once your VPS is up and running, copy your SSH key across:

```
ssh-copy-id root@yourserverip
```

Provision the server

Begin by installing chef on the remote machine:

```
bundle exec knife solo prepare root@yourserverip
```

This will generate a file `nodes/yourserverip.json`. Copy the contents of `nodes/rails_postgres_redis.json.example` to this file and change the username and password for monit.

Use the same command as before `openssl passwd -1 “plaintextpassword` to generate a password for postgresql and add this to the node definition file.

Now run:

```
bundle exec knife solo cook root@yourserverip
```

Sit back, relax and enjoy. This process takes quite a while and once it's completed, you've got a server ready for a Rails + Postgres + Redis app.

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 and will almost always break more than it fixes. Section 6.1 goes into more detail on the function of Berksfile.lock

Next Steps

You've now got a completed single box Rails Server running Nginx, Postgresql & Redis.

Part 2 of this book - Capistrano covers how to deploy to such a server.

The rest of this book explains the workings of this configuration and how to customise it to your own needs or construct your own from scratch.

5.0 - Anatomy of a chef solo project

Overview

In this chapter will cover creating a simple Chef project and look at how the chef terminology relates to the folder structure.

Creating a project

Our first step is to install the tools needed to create and manage a chef project.

Create a new folder to contain your chef project and move into it;

```
mkdir my_chef_projects
cd my_chef_projects
```

Now create a new `Gemfile` within `my_chef_projects` and enter the following:

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

    gem 'knife-solo', '0.3.0'
    gem "chef", "~> 11.10.0"
    gem 'chef-zero', '1.7.2'
    gem "berkshelf", "~> 2.0.14"
```

Then run

```
    bundle install
```

To install Chef, Knife Solo and Berkshelf. It's important to use the Gemfile to ensure that the versions installed are the versions which this book has been tested with. Both the Berkshelf and Chef gems are regularly updated so using an unsupported version may lead to unexpected behaviour.

Terminology

Recipe

Chef definition for installing a single component, e.g. ruby, mysql-server, Monit etc

Cookbook

A selection of recipes, so for example a “mysql” cookbook might include a recipe for a mysql server and another one for a mysql client.

Node

A remote server you’re provisioning

Role

A combination of recipes which when applied to a node, allows it to perform a particular role.

For example a “Postgres Server” role might include recipes for installing postgres-server as well as installing and configuring the firewall and setting up suitable monitoring for the server process.

You can sort of think of a role as an equivalent to a mixin in ruby. It allows you to create a piece of re-usable functionality that can later be applied to many nodes.

Data Bag

A JSON file which contains meta data used by recipes, for example lists of users to be created and the valid public keys for authenticating as them.

Chef Repository

A collection of node and role definitions

Creating a repository

Once the Gem installation completes you’re ready to create an empty chef solo project using knife. Execute the following command:

```
bundle exec knife solo init my_first_chef_repo
```

> ### ⚠ bundle exec
>
> `bundle exec` ensures that the version of the gems used matches the version defined in the Gemfile. It’s important that you always use `bundle exec` when running `knife` commands to ensure you’re using gem versions which have been tested with this book.

You should see output similar to the following:

```
WARNING: No knife configuration file found
Creating kitchen...
Creating knife.rb in kitchen...
Creating cupboards...
Setting up Berkshelf...
```

You can then cd into the newly created my_first_chef_repo where you'll see a directory structure like the following:

```
-my_first_chef_repo/
    -.chef
        - knife.rb
    -cookbooks
    -data_bags
    -nodes
    -roles
    -site_cookbooks
```

The Gemfile

I suggest you copy the Gemfile from `my_chef_projects` into `my_first_chef_repo`. If you don't, `bundle exec` from within `my_first_chef_repo` will still work as it will just use the Gemfile from the parent directory but you lose the flexibility of specifying different gem versions or additional gems per repository.

knife.rb

Knife is the command line utility which allows us to interact with chef on our remote server.

Knife solo adds extra commands for interacting directly with the server we're provisioning from our development workstation (rather than via a central chef server)

knife.rb contains the knife configuration options which are specific to our repository. This is more important when working with a centralised chef server which we aren't so for our purposes we'll treat this as the primary (and only) configuration point for knife.

Documentation for the full range of options available is at

http://docs.opscode.com/config_rb_knife.html

they key ones for us are:

`cookbook_path` - this is an array of paths relative to the root repository directory where the cookbooks referenced in roles and/ or node definitions are located.

`node_path` - path relative to the root of the repository where node definitions are stored

`role_path` - path relative to the root directory where role definitions are stored

`data_bag_path` - path relative to the root directory where data_bags are stored.

In general we can leave these at their default values.

cookbooks vs site_cookbooks

Here we'll use the cookbooks directory to store other peoples cookbooks which we install using Berkshelf (covered later) and site_cookbooks to contain our own custom cookbooks.

It's important not to use cookbooks for none Berkshelf ones for the simple reason that it is wiped every time we run knife solo cook! That means that any changes we make in here, will not be persisted.

Next we'll look at the anatomy of an individual recipe.

5.1 - A Simple Chef Cookbook

Overview

As we've covered, a chef cookbook is made up of a collection of recipes generally relating to a specific package. In this example we'll assemble a chef cookbook containing a single recipe which installs and configures Redis.

Cloning the Redis recipe

Begin by cloning the below repository into the `site_cookbooks` folder of the chef repository we created in the previous chapter.

https://github.com/TalkingQuickly/redis-tlq

Site Cookbooks v Cookbooks

As we saw in the previous chapter, if we're writing custom cookbooks, e.g. ones which are not being installed automatically by Berkshelf (covered in 6.1), these go in `site-cookbooks` not `cookbooks`. This is because the `cookbooks` directly is wiped and replaced with the correct versions by Berkself every time we provision.

Structure of the Redis Recipe

metadata.rb

The metadata file contains details about what the cookbook can do, who it is written by, its dependencies and the version.

metadata.rb for our redis cookbook looks like this:

```
name              "redis-tlq"
maintainer        "Ben Dixon"
maintainer_email  "ben@talkingquickly.co.uk"
description       "Installs redis from rwky's ppa"
version           "0.0.1"

recipe "redis-tlq", "Installs redis"

supports "ubuntu"
```

The `supports` line specifies the operating system the cookbook has been created to work with. This is particularly important as our simple cookbook will only be designed for Ubuntu and so will use some Ubuntu specific commands. More complex cookbooks which may be used across multiple OS's will include detection to ensure suitable commands are used depending on the OS.

The `version` line and the `recipe` line specify the name of the recipe and the version (suprising, I know). These are key when working with Berkshelf as it will look at metadata.rb to confirm whether the cookbook it's looking at contains the correct version of a recipe with a matching name.

The one line we haven't covered here but will encounter a lot in future is `depends`. This allows us to indicate that our cookbook is dependent on other external cookbooks.

For example if we were writing a cookbook which installs ruby by building it from source, we'd need access to the standard build toolchain. There is a very common cookbook called "build-essential" written by Opscode which handles installing the build toolchain across most platforms. In order to indicate a dependency on this cookbook we would include the following line:

```
depends "build-essential"
```

default.rb

A chef cookbook should always include a recipe called "default." This is the default recipe which will be installed if the cookbook is specified in a node or role definition without specifying a particular recipe.

`cookbooks/redis-tlq/recipes/default.rb` begins with:

```
# add the stable redis ppa as the version included in
# the Ubuntu repositories is generally quite out of date
bash 'adding stable redis ppa' do
  user 'root'
  code <<-EOC
    add-apt-repository ppa:rwky/redis
    apt-get update
  EOC
end
```

This is a good example of what makes chef so powerful, you'll recognise the commands to add a repository as the standard shell commands you'd use if you'd ssh'd in directly.

The line beginning "bash" tells chef to execute commands in the bash shell, the text afterwards is displayed when the recipe is run to give the user (us) an indication of what the recipe is currently doing.

This then accepts a ruby block. The first line specifies that the user the commands should be run as is "root."

User Gotcha

When setting the user like this, although commands are executed as that user, it's not quite the same as actually logging into a shell for that user. Environment variables (such as HOME and PATH) will not be automatically setup. Therefore if any of the commands rely on environment variables, these should be set explicitly.

The next line, `code <<-EOC` means that everything after that will be executed in the bash shell, as the user provided, until `EOC` is reached.

This is pure Ruby although something which is often missed, feel free to move to the next paragraph if you're already familiar with the <<- syntax for constructing multi-line strings. There is nothing special about the EOC sequence, <<-SOMETHING in Ruby means that all input following should be treated as a single string literal, until SOMETHING is reached.

We could include `apt-get install redis-server` in that list of shell commands to install the redis-server package but chef includes a better way of doing this. Further down `default.rb` we see the following:

```
# install the redis server package
package 'redis-server'
```

This tells chef to check whether the package redis-server is installed using the distributions package manager and if not, install it.

This covers installing the Redis package but we still need to add any custom configuration we need and an init script to start and stop it. This is where templates come in.

Templates

Further on, `default.rb` contains:

```
# Copy our custom redis configuration file
template "/etc/redis/redis.conf" do
  owner "root"
  group "root"
  mode "0644"
  source "redis.conf.erb"
end
```

This will:

- Look for the file redis.conf.erb in redis-tlq/templates/default/ (where redis-tlq is the name of the cookbook, see below re directory structure)
- parse any erb in this file and then copy it to /etc/redis/redis.conf on our node. We'll cover why erb is so useful here in 5.3 (Node Definitions).
- Make this file owned by root with group root with permissions "0644"

A note on template directories

The directory structure of templates can be confusing because of the use of the `default` name. It's easy to assume that the folder `default` within templates, refers to the recipe `default`. This is not the case.

The subdirectories within templates refer to the distribution the remote system is running. Therefore if we're installing on an Ubuntu 12.04 system, the above template block for `redis.conf.erb` which search in the following locations in the order shown:

```
templates/ubuntu-12.04/redis.conf.erb
templates/ubuntu/redis.conf.erb
templates/default/redis.conf.erb
```

If you move to writing more complex cookbooks which are designed to work across multiple systems, this can be extremely powerful. For now though we'll work exclusively in the `default` directory.

Attributes

A recipe can be customised with attributes. There a multiple possible sources of these attributes including:

- node definitions (see 5.2)
- role definitions (see 5.3)
- cookbook defaults (see below)

Generally a cookbook will specify a default for an attribute and you then have the option to override these defaults if required.

Where our the `node` variable referenced below comes from is covered in 5.2, for now it's enough to understand that `node` will contain attributes which we've chosen to define custom values for as a ruby hash.

In the template referenced above, `redis.config.erb` there is a section which looks like this:

```
<% unless node[:redis] && node[:redis][:dont_bind] %>
  bind 127.0.0.1
<% end %>
```

In our redis config file, we first check whether the node has defined a sub element "redis." This ensures that if the node definition does not specify a value for the redis key, our recipe will not fail with an "undefined method [] for nil class" exception when looking for the `dont_bind` key.

Unless that key exists and is a true value (and remember this is just ruby so anything other than 0 or false is considered truthy) then the config file will include the line `bind 127.0.0.1`

The reason we're careful to default to including the line is that without this line, redis will accept unauthenticated connections from any external host. Without a suitable firewall in place this is a serious security risk.

Cookbook Attribute Defaults

As mentioned above, a cookbook can define default values for attributes. It is generally considered best practice to construct a cookbook so that it can be used with minimal custom attributes by a majority of users.

The default attributes for a cookbook are defined in the file `attributes/default.rb`. As you'll notice in the following section, the format for defining default attributes here is different to that of defining custom attributes in node or role definition files.

Our Redis cookbook only has one attribute so its default file is extremely simple:

```
#
# Author:: Ben Dixon
# Cookbook Name:: redis-tlq
# Attributes:: default
#

# whether to prevent redis from binding to 127.0.0.1
default[:redis][:dont_bind] = false
```

So in the above scenario, if it is not subsequently overwritten, the line:

```
default[:redis][:dont_bind] = x
```

will make x available to the cookbook as

```
node[:redis][:dont_bind]
```

Unless the value is subsequently overridden. For more on attribute precedence it's well documented by OpsCode here:

http://docs.opscode.com/essentials_cookbook_attribute_files.html

Although I'd recommend getting comfortable with the following few chapters before delving into this.

Next will look at what a node definition is for and how to construct one.

5.2 - A Simple Node Definition

Overview

A node refers to a single server (for example a single VPS we're provisioning). In this section we'll look at what goes into a node definition file and create a simple one to be used in our example redis project.

A redis node

Create a file `example-redis-server.json` in the `nodes` directory and enter the following:

```
{
  "redis": {
    "dont_bind" : true
  },
  "run_list":
  [
    "role[redis-server]"
  ]
}
```

We can see this is standard JSON object with two key sections. We'll begin by looking a the "run_list" section.

Run List

In chef a `run_list` is an array of roles or recipes to be applied to (installed on) a node. Chef will work through the recipes and roles in the order they are listed and apply them. As we'll see in the next chapter, a role definition is in practice a "mini node definition" which looks almost identical to the above.

A run list can contain a list of roles and recipes, a role is identified in the following format:

```
role[role_name]
```

and a recipe by:

```
recipe[recipe_name]
```

Although roles and recipes can be specified just by name, it's good practice to always specify whether it's a role or a recipe to avoid unexpected results if there are roles and recipes with the same name.

In this case we only want to apply a single role "redis-server" but in practice there would usually be several roles here.

For example there might be "xyz-company-server" role which applies all of the common attributes a particular companies servers have.

I've often used this for clients who have a standard base package set, security setup and look and feel which they apply to every single one of their servers.

Customising Recipes with Attributes

As per section 5.1, attributes can be used to customise the behavior of cookbooks.

The first section of the node definition looks like this:

```
{
  "redis": {
    "dont_bind" : true
  }
....
```

The above section defines the "attributes" of the node. These attributes are accessible by individual recipes to customise their execution.

So in this scenario, the value of `node[:redis]['dont_bind']` would be overridden from the default (false) defined in the cookbooks `attributes/default.rb` file to our new value, true.

We could expand the functionality of this cookbook to be even more granular, for example we could modify the redis.conf.erb file to look like this:

```
<% if node[:redis] %>
    <% if (interface = node[:redis][:interface])
        bind <%= interface %>
    <% else %>
        bind 127.0.0.1
    <% end %>
<% else %>
    bind 127.0.0.1
<% end %>
```

with the first section of our node definition looking like this:

```
{
  "redis": {
    "interface" : 127.0.0.1
  }
....
```

This would, as before, check that the redis key exists and if so look for the subkey "interface." If the interface key is defined, it would add a line to bind redis to that interface (in the above example 127.0.0.1). Otherwise it will always default to binding to 127.0.0.1 (the default loopback interface).

Now we can really see the power of chef. A recipe can be used to automate any combination of commands and file changes we need to make on a server we set up. Any aspect of the setup which varies between nodes can be customised using attributes.

All attributes relating to a particular node are stored in one place so we can easily see exactly how any single node is set to be configured.

Naming Node Definitions

We'll cover generating and naming node definition files in detail in section 5.4.

In brief node definitions should be stored in the `nodes` folder and named according to the hostname you would use to connect to them. So for example if you would use `ssh root@my_shiny_server.co.uk` to ssh into the server you're provisioning, the node definition file should be called `my_shiny_server.co.uk.json`.

This will make your life significantly easier as the number of node definitions grows both for organisation and because it will allow knife solo to automatically select the file based on hostname.

5.3 - A simple chef role

Overview

Roles define the components required for a server to fulfill a particular role. In this example we'll be defining a simple Redis Server role, which will use the Redis cookbook and recipe from sections 5.1 and 5.2.

Creating a role definition

The below shows the JSON for our Redis role stored in roles/redis-server.json:

```
{
   "name": "redis-server",
   "description": "Redis server",
   "default_attributes": {
      "redis": {
        "dont_bind" : false
     }
   },
   "json_class": "Chef::Role",
   "run_list": [
      "monit-tlq",
      "redis-tlq",
      "monit_configs-tlq::redis-server"
   ],
   "chef_type": "role"
}
```

We can see that this is very similar to our node definition from the previous chapter.

We begin by specifying the name of the recipe (redis-server) and a simple human readable description of what the role does. When chef encounters the line `role[redis-server]` in the nodes run list, it will look for roles with the "name" attribute set to "redis-server." It will look for these role definitions in the json files in the folder specified as role_path in knife.rb.

Default Attributes

Default attributes allow us to specify what value an attribute should have if no value is specified in the node definition file.

For example referring back to our earlier redis-server recipe, if it is our preference that our Redis servers never bind to 127.0.0.1, we could have the following default attributes section in redis-server.json:

```
"default_attributes": {
    redis: {
      "dont_bind" : true
    }
 },
 ....
```

In this scenario if our node definition doesn't specify a value for the `don't_bind` key, it will default to true. I'd strongly recommend against this default configuration as it makes your Redis server insecure by default without proper firewall configuration however it serves to demonstrate the principal.

json_class and chef_type

These specify the type of definition this file provides. This is important to differentiate between roles and recipes, particularly if you have roles and recipes with the same names.

run_list

As in our node definition, this is an array which can contain a list of recipes or indeed other roles which should be applied to a node which has this role.

Here we see the `recipe[recipe_name]` syntax as before, we also see the syntax for defining a particular recipe within a cookbook:

```
"recipe[monit_configs-tlq::redis-server]"
```

This tells us that there is a cookbook called monit_configs which contains a recipe redis-server which we should install. When we specify a recipe without the `::` part the default recipe is installed. So writing:

```
recipe[my_great_recipe]
```

is equivalent to:

```
recipe[my_great_recipe::default]
```

You can see that this role uses two new cookbooks, `monit_configs-tlq` and `monit-tlq`. We'll download and add those to `site-cookbooks` in the next chapter.

Now we've got a reasonable understanding of how to construct a simple chef repository with basic role definitions and recipes, we'll move onto applying our simple node definition to a VPS.

5.4 - Applying a node definition to a VPS

Overview

In this section we'll take our simple Redis cookbook and apply it to a VPS.

An important note. This Redis configuration is intended to be a simple example of using chef, in practice such a system would have several other components including security and user management. For this reason this setup should not be used in production, instead look to Section 6 onwards for a production ready stack.

Download additional cookbooks

In this simple example, we're manually copying the cookbooks we need into `site-cookbooks`. Recall that when we created our simple Redis role in the previous chapter, we used two new cookbooks; `monit_configs-tlq` and `monit-tlq`. In order for our role to work, we'll need to clone these cookbooks into `site-cookbooks`.

Move into the site-cookbooks directory and enter:

```
git@github.com:TalkingQuickly/monit-tlq.git
git@github.com:TalkingQuickly/monit_configs-tlq.git
```

To add these.

Site Cookbooks

In practice, it's very rare to clone cookbooks into site-cookbooks, usually these will be managed by Berkshelf (see 6.1) and site-cookbooks is reserved for custom ones we're writing or modifying ourselves.

Set up the VPS

Begin by creating a VPS on your favorite cloud provider, my preference for testing is Digital Ocean and Linode for production. I suggest using Ubuntu 12.04.

Once your image is ready, copy your public key

```
ssh-copy-id root@yourserverip
```

Install chef on the remote machine

Now move into the root of our cookbook directory (my_first_chef_repo).

It used to be necessary to manually install chef on the remote machine, happily knife solo will now take care of this for us. Use the following command to install chef on the remote machine:

```
bundle exec knife solo prepare root@yourserverip
```

This will:

- Create a file yourserverip.json within the nodes folder
- SSH into the remote machine
- Detect the operating system
- Install the correct version of chef

Now enter the below (from 5.2) into the yourserverip.json file:

```
{
  "redis": {
    "dont_bind" : false
  },
  "run_list":
  [
    "role[redis-server]"
  ]
}
```

Applying the recipe is simple, just enter the following command:

```
bundle exec knife solo cook root@yourserverip
```

The cook command will:

- SSH into the remote machine
- Copy your local recipe files to the remote machine
- Apply the recipes and show the output

If you now ssh into the remote machine you can verify that the redis-server package is installed using:

```
dpkg -l redis-server
```

Additionally if you look at our config file in `cat /etc/redis/redis.conf`

You'll see that it matches the config file we defined in 5.1, in particular that it includes the line:

```
bind 127.0.0.1
```

which correctly reflects the `dont_bind: false` parameter in our node definition file.

Make a change and re-apply

So far we've shown that chef allows us to define the steps we would normally take to provision a new server, define them in an easy to read format and apply them to a fresh VPS.

Chefs real power is in allowing us to manage the process of iterating on a server configuration.

Let's say, for example, that we'd begun as above with the "dont_bind" parameter set to false. Further down the line we discover we need to share the Redis install with other machines and so allow external connections.

Change the yourserverip.json file to have dont_bind set to true:

```
{
  "redis": {
    "dont_bind" : true
  },
  "run_list":
  [
    "role[redis-server]"
  ]
}
```

Save the file and enter the command to apply the node definition to the remote machine again:

```
bundle exec knife solo cook root@yourserverip
```

You'll see from the output that chef detects that there have been no changes to our Redis script in init.d but when it comes to redis.conf it detects the changes and updates it, displaying the diff in the output.

Once the command completes, if you ssh into the server and re-examine out redis.conf file, you'll see that the `bind 127.0.0.1` line has been removed in accordance with our `dont_bind: true` attribute.

Summing Up

Chef allows us to easily define the commands we need to provision a server and the file changes which go with these.

When we make changes to these setups, chef will intelligently detect where we have made changes and apply these accordingly.

This makes it simple to iterate on our infrastructure configuration whilst maintaining a blueprint to easily recreate it either when we need to scale or to recover from server failures.

It should be noted here that there are limits to the type of change chef will automatically detect. Whilst chef will detect additions and subtractions to/ from configuration files (e.g. anything templates are being used for), it will not automatically handle the removal of a recipe or role.

So for example if we initially included the redis-server role in our run list and subsequently removed it, chef would not attempt to uninstall the packages associated with that node or remove the config files.

In the next chapter we'll begin working with the example Rails Configuration template available at:

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

This template forms the basis of a configuration we can use in a production environment using exactly the principals and methods discussed in this section.

6.0 - A Template for Rails Servers

Overview

In the previous section we've covered how to build a Chef cookbook from the ground up. Hopefully from this I've 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 of something like Heroku or pre made images which hide the process from you 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 you rely entirely on applying recipes written by other people, as soon as you 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 you next come to provision a node using your chef definition, it won't work the same way.

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

This provides example a node definition file for basic single box Rails servers using each of the following databases:

- PostgreSQL
- MySQL
- MongoDB
- Redis

Along with the cookbooks and role definitions. The cookbooks are a mixture of simple ones written specifically for this book and complex community ones we'll use only a fraction of.

This is illustrative of how a normal chef project will look, a mixture of mainstream recipes with a few bespoke ones of your own specific to your particular setup.

6.1 - Managing Cookbooks with Berkshelf

Overview

We finished the previous chapter by observing that a majority of chef repositories will contain a mixture of our own bespoke cookbooks and community ones from third parties.

It's also fair to say you'll eventually end up with a fair few different chef repositories relating to different projects. If you've written your recipes well, you'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 in gems by others.

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) installed 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 its own 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.

How Berkshelf works with knife solo

By default Berkshelf installs cookbooks to a folder within ~/`.berkshelf`. The integration with knife solo means that when we run `knife solo cook`, the cookbooks we need are automatically copied into the cookbooks Directory.

It's important to recognise therefore that the cookbooks folder should not be used by us for anything because changes to it will not be persisted when `knife solo cook` is run.

Instead, any repository specific cookbooks should be stored in `site-cookbooks`.

Getting Started

Structure of a Berksfile

Begin by cloning the example chef repository:

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

Then `cd` into the rails-server-template directory, you'll see a folder structure exactly as described in section 5 and like the example configuration we created.

Open Berksfile and you'll see something that looks like this:

```
site :opscode

cookbook 'build-essential', git: 'https://github.com/opscode-cookbooks/build-e\
ssential'
cookbook 'sudo', git: 'https://github.com/opscode-cookbooks/sudo.git'
cookbook 'basic-security-tlq', git: 'git@github.com:TalkingQuickly/basic_secur\
ity-tlq.git'
cookbook 'look-and-feel-tlq', git: 'git@github.com:TalkingQuickly/look_and_fee\
l-tlq.git'
cookbook 'users', git: 'https://github.com/opscode-cookbooks/users.git'
cookbook 'chef-solo-search', git: 'https://github.com/edelight/chef-solo-searc\
h'
```

The first line `site :opscode` 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.

You can also use bundler style version definitions so:

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

You’ll see in our example Berksfile the format is generally:

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

Again, like bundler, this allows you to specify a git repository to download the recipe from. I tend to specify git repositories over using the Opscode index. This is entirely personal preference as I regularly refer back to the repositories and their authors.

Finally 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: “eef7e6580\
6e7ff3bdbe148e27c447ef4a8bc3881”
```

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

Installing from the Berksfile

To install the cookbooks, execute the following command in the root of our example configuration folder:

```
bundle exec berks install
```

As explained above, when `knife solo cook` is run later, Berkshelf will copy the cookbooks automatically into the `cookbooks` directory. Since this won’t happen until the first time we apply the configuration to a node, the cookbooks source isn’t initially available in the cookbooks directory. To get around this we can have Berkshelf ‘vendor’ the cookbooks into the cookbooks folder with the following command:

```
bundle exec berks install --path cookbooks
```

Now if you look in the cookbooks folder you’ll see that for each of the cookbooks listed in the Berksfile, as well as some additional dependencies, there’s a directory containing a cookbook structured as per section 5.1.

Remember though, any changes made to the cookbooks in the `cookbooks` directory directly, will not be saved. The contents of this Directory is refreshed every time `knife solo cook` is run.

Any repository specific custom cookbooks should go in site-cookbooks instead.

The following chapters will cover each of the key system elements in detail.

7.0 - Basic server setup

Overview

In this chapter we'll cover:

- Installing some common packages to make your life easier
- Adding a visual queue when we login to a production environment
- Installing common Rails Gem dependencies

In our example repository, basic server setup is primarily contained in the server role `roles/server.json` with the rails specific elements in `roles/rails-app.json`.

This chapter will primarily focus on the `look_and_feel-tlq` cookbook and briefly touch on `rails_gem_dependencies-tlq`.

Basic Packages

There are some packages it's worth installing on all servers by default because it's highly likely they'll be useful at some point. For this I maintain a simple chef cookbook called `look_and_feel-tlq` the first section of default.rb looks like this:

```
# htop is a prettier (but more resource intensive) alternative
# to top.
package 'htop'

# Vim because we're going to want to edit Rails config files
package 'vim'

# Because not everyone will send us nice  .tar.gz files
package 'unzip'

# Add a banner to ssh login if we're in the production environment
```

This is fairly self explanatory, it simply installs a few admin tools I know I'm likely to need.

Adding a visual queue if you log into a production environment

The next section is more interesting:

```
if node[:environment] == 'production'

  sshd_config = '/etc/ssh/sshd_config'

  seds = []
  echos = []

  banner_path = '/etc/ssh_banner'

  seds << 's/^Banner/#Banner/g'
  echos << "Banner #{banner_path}"

  template banner_path do
    owner 'root'
    group 'root'
    mode '0644'
    source 'production_ssh_banner.erb'
  end

  bash 'Adding visual flags for production environment' do
    user 'root'
    code <<-EOC
      #{seds.map { |rx| "sed -i '#{rx}' #{sshd_config}" }.join("\n")}
      #{echos.map { |e| %Q{echo "#{e}" >> #{sshd_config}} }.join("\n")}
    EOC
  end

  service 'ssh' do
    action :restart
  end
end
```

This section looks to see if our node has:

```
"environment" : "production"
```

In the root node. If so it adds the contents of the production_ssh_banner.erb template as a banner which will be displayed when logging in via SSH. In my example repository this is simply the word "Production" in ASCI art.

The purpose of this is to minimise the possibility of accidentally SSHing into production and thinking we're in a staging environment. This is surprisingly easy to do when flicking between multiple servers, particular if using something like zsh for hostname completion.

I recommend forking this recipe and creating your own version which includes the tools and visual tweaks you're comfortable with.

Rails gem dependencies

A second recipe I install on almost every server is `rails_gem_dependencies-tlq`. This is a very simple recipe which installs packages often needed when installing rails gems, mine currently looks like this:

```
package 'curl'
package 'libcurl3'
package 'libcurl3-dev'
package 'libmagickwand-dev'
package 'imagemagick'
```

You'll probably recognise most of these packages if you've ever tried to installs gems which do image processing or handle accessing web pages. This is my default version of this recipe which I tend to install on all servers intended to run Rails apps.

If an app contains unusual package dependencies I generally fork this gem and put a copy in site_cookbooks and add the app specific packages to that.

We'll now move onto setting up default users and public keys on the remote machine.

7.1 - Users

Overview

In this chapter we'll cover automating adding users and public keys to the remote server.

We'll be looking at the following cookbooks:

- users
- sudo

Which are setup and applied in `roles/server.json`

These are third party cookbooks which deal with the standard requirements of user management, password generation and controlling who has access to sudo.

Users

The `users` cookbook contains a recipe called `sysadmins`. This deals with creating users in a group called sysadmins.

Unlike other elements of the system, user configuration is done outside of the node and role definition files. Instead the configuration is stored in what are called data bags.

Recall in our original chef solo project we had a folder called `data_bags`. Data bags are generally used when a cookbook requires larger chunks of data to be provided to it.

If you look in the `data_bags/users` directory there's a file called `deploy.json.example` which looks like this:

```
{
  "id": "deploy",
  // generate this with: openssl passwd -1 "plaintextpassword"
  "password": "",
  // the below should contain a list of ssh public keys which should
  // be able to login as deploy
  "ssh_keys": [
  ],
  "groups": [ "sysadmin"],
  "shell": "\/bin\/bash"
}
```

ID is the username of the user and should match the name of the file.

The password for the user is not the users plain text password (you don't want this stored in plain text anywhere outside of a password manager). Instead it is a hash of the password which can be generated locally using:

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

which will generate an output like:

```
$1$wNIBg4QQ$/b46CoUAqP8EcmeF9.QaQ/
```

Which is what should be entered for the password value in your data bag.

The `ssh_keys` key should contain an array of public ssh keys which can be used to login as that user. You can usually access your own local public key with:

```
cat ~/.ssh/id_rsa.pub
```

The output of which should then be copied into the array as a string (e.g. surrounded by quotes).

`groups` should be an array of strings, with each string referring to a single group that the user should be a member of.

Finally shell is a string which should be the path to the users default shell. This can be used, for example, for switching from the default bash shell to something like Zsh or Fish.

The users recipe will go through the .json files in `data_bags/users` and create each of the users defined.

The users cookbook allows many other attributes to be set and we've only scratched the surface of what it can do here. The documentation in the cookbooks README is excellent so I strongly recommend spending some time getting familiar with it.

Sudo

In addition to adding the users and setting up public keys, we'll setup who can use sudo and how. The following section in our `server.json` role governs the use of sudo:

```
  "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": "false"
  }
```

The comments should be fairly self explanatory however a quick note on passwordless sudo. This can be very convenient however, if due to a vulnerability in your app, an attacker is able to execute shell commands as your apps user and this user has passwordless sudo enabled, they would be able to execute commands as root, significantly increasing the damage they could do.

If you wish to have an account with passwordless sudo enabled for convenience, I would suggest having a separate devops user which you login as rather than enabling it for your app user.

Next we'll look at some basic steps to lock down the server against unauthorised access.

07.2 - Security

Overview

In this chapter we'll cover some basic steps which can be taken on all servers to help protect against unauthorised access.

A Note on security

One size does not fit all

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

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

Security Gotchas

When we talk about security, it brings to mind firewalls and auto banning brute force attacks. In practice many security breaches are due to not following good practices rather than 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 subscribe 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.

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 which are maintained independently on production and staging environments. The basics of this are covered when we look at deploying with Capistrano.

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.

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.

Basic Measures

We'll primariy be looking at the `basic-security-tlq` cookbook in this section.

The default recipe begin with the following:

```
packages = %w(
  fail2ban
  ufw
  unattended-upgrades
)

packages.each { |name| package name }
```

Which installs three security related packages.

Fail2Ban

Fail2Ban is a tool to prevent brute force attacks. If an IP has too many failed attempts to connect to a configured service, Fail2Ban can be configured to update the firewall to block any further connection attempts from this user.

This is to prevent brute force attacks where an attacker sequentially tries a large number of user and password combinations to a known service.

Its default configuration covers SSH brute forcing (although you'll see below we make this extremely unlikely to occur anyway by disabling password authentication). It's an extremely powerful tool which should be configured for the specific services you're exposing to external access.

A good place to start is:

https://help.ubuntu.com/community/Fail2ban

and

http://www.fail2ban.org/wiki/index.php/Main_Page

SSH Hardening

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 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.

I also disable X11Forwarding simply because there's no graphical interface on the server and so no need for it.

The below section uses the utility `sed` to automatically make changes to the ssh config file:

```
# path to ssh config
sshd_config = '/etc/ssh/sshd_config'

# changes to make to the config file
seds = [
  's/^#PasswordAuthentication yes/PasswordAuthentication no/g',
  's/^X11Forwarding yes/X11Forwarding no/g',
  's/^UsePAM yes/UsePAM no/g'
]

bash 'ssh hardening' do
  user 'root'
  code <<-EOC
    #{seds.map { |rx| "sed -i '#{rx}' #{sshd_config}" }.join("\n")}
  EOC
end
```

Sed is a great utility which allows you to automate the modification of files. It's a little intimidating at first but if you have a spare afternoon well worth spending time getting to know as it's tremendously powerful and can save huge amounts of time. The documentation is universally acknowledged as being terrible but there's a great tutorial available at:

http://www.grymoire.com/Unix/Sed.html

Unattended Upgrades

On this occasion I've saved the most controversial for last. The unattended-upgrades package allows you to set certain updates to be installed automatically at a pre determined interval, without any manual intervention.

It should be noted at this point *this can break things*. If, as I do, you choose to setup unattended upgrades to automatically install security updates when they become available, it's possible that a package update will break an element of your stack and take the server down.

This should be weighed up against the risk of leaving security updates to manual, forgetting to do them and having the server compromised.

In 99% of cases I choose to err on the side of automatic upgrades, the 1% where I don't is when 99.999% uptime is a requirement and I know that there will be ongoing sysadmin experience available to check for, test and install updates.

The section which enables automatic updates is:

```
# Set the system to install security updates automatically each day
file '/etc/apt/apt.conf.d/10periodic' do
  owner 'root'
  group 'root'
  mode '0644'
  content <<-EOF
  APT::Periodic::Update-Package-Lists "1";
  APT::Periodic::Download-Upgradeable-Packages "1";
  APT::Periodic::AutocleanInterval "7";
  APT::Periodic::Unattended-Upgrade "1";
  EOF
end
```

For more on this process and how it can be customised, see the official Ubuntu tutorial:

https://help.ubuntu.com/community/AutomaticSecurityUpdates

Automatically Updating Time

Finally we setup automatic time updating.

```
# updated time from central server every day, particularly important
# when certs are involved.

file '/etc/cron.daily/ntpdate' do
  owner 'root'
  group 'root'
  mode '0755'
  content <<-EOF
    #!/bin/sh

    ntpdate -s ntp.ubuntu.com pool.ntp.org
      EOF
end
```

Here we setup a simple cron job which will update the system time from the Ubuntu time server each day. This is especially important when dealing with SSL certificates where incorrect server times can cause hard to debug errors.

Next will look at configuring the Firewall to keep control of which ports are accessible to the outside world.

7.3 - Firewall

Overview

A firewall acts as a gate keeper layer between the outside world and the services running on your server. You can restrict which remote IP addresses are allowed to collect to which ports on your server.

We'll set up a Firewall which by default blocks everything and then look at how to use chef to manage which ports to open.

UFW (Uncomplicated Firewall)

UFw acts a friendly layer to IPTables, making it simple to add and manage the rules which define what can and cannot connect.

The following section of the default.rb recipe sets up our basic ufw installation:

```
# now allow SSH traffic through the firewall and restart SSH
# unless otherwise specified, block everything
bash "opening ufw for ssh traffic" do
  user "root"
  code <<-EOC
    ufw default deny
    ufw allow 22
    ufw --force enable
  EOC
end
```

You'll see once again, we're just using chef to execute standard terminal commands. The first

```
ufw allow 22
```

Tells ufw to allow all traffic to and from port 22 (the port we're running SSH on). The second:

```
ufw --force enable
```

Enables ufw. It's important to make sure you've allowed ssh traffic through the firewall before enabling it otherwise you can end up unable to SSH in.

You can use the following command on your remote server to see whether ufw is enabled and its current rules

```
sudo ufw status
```

which will show output something like:

```
Status: active

To                         Action      From
--                         ------      ----
22                         ALLOW       Anywhere
```

By default our firewall will block everything unless we specify otherwise. This is desirable because it means that by default when we install a new service, unless we manually specify otherwise, it won't be accessible to the ourside world.

The next section is a little more advanced:

```
# if we've specified firewall rules in the node definition
# then apply them here. These should be in the format:
# {"port": "x", "ip": "xxx.xxx.xxx.xxx"}
if node['firewall_allow']
  node['firewall_allow'].each do |rule|
    bash "open ufw from #{rule['ip']} on port #{rule['port']}" do
      user "root"
      code "ufw allow from #{rule['ip']} to any port #{rule['port']}"
    end
  end
end
```

You'll recall in the default attributes of our `server.json` role, we had a line which looked like this:

```
"firewall_allow":[]
```

This array can contain hashes in the form:

```
{"port": "x", "ip": "xxx.xxx.xxx.xxx"}
```

The basic_security-tlq recipe will then iterate over each of these hashes, opening the specified port to the specified IP.

So, for example, if the server you're configuring was running a MySQL server on port 3306 and you needed that server to be accessible to another one of your servers (168.154.2.3) you would add the following entry:

```
{"port": "3306", "ip": "168.154.2.3"}
```

This would mean that the server with IP address 168.154.2.3 would be allowed to connect to port 3306, but requests on all other IP's to that port will still be dropped.

Remember you can always check your current rules using `sudo ufw status`.

You can also watch what's being blocked by ufw by looking at its log file, by default this is located at `/var/log/ufw.log`. So a simple:

```
sudo tail -f /var/log/ufw.log
```

Will show you a live tail of what's being added to the log. You can then attempt to connect to various ports on the server from different hosts and see if ufw blocks them or not.

This is useful firstly for checking that ports you know you don't want to be externally accessible are being properly blocked and secondly for troubleshooting connection issues between machines that should be able to connect to one another but can't.

7.4 - Applying the sample template to a VPS

Overview

We now have the bare bones of a server configuration consisting of:

- Look and feel
- Localisation
- User structure
- Basic Firewall
- Locked down SSH

In our sample template, these minimal requirements for a server are contained in the `server` role found in `roles/server.json`.

Therefore a simple node definition for just applying these basic elements would look like this:

```
{
  "environment":"production",
  "run_list":
  [
    "role[server]"
  ]
}
```

In this section we'll apply this minimal node definition to a VPS and then continue to build on it in further chapters.

Pre-requisites

This chapter assumes you've followed along from the beginning of section 6, in particular that you have:

- cloned the example repository (https://github.com/TalkingQuickly/rails-server-template)
- Run `berks install --path cookbooks` to install the cookbooks locally

Step by step

The process is exactly the same one we followed for applying the simple redis configuration in 5.4

1) Copy your public key to your VPS:

```
ssh-copy-id root@yourserverip
```

2) Install chef on the remote machine

```
knife solo prepare root@yourserverip
```

3) Populate your new node definition `nodes/yourserverip.json` with the simple node definition:

```
  {
    "environment":"production",
    "run_list":
    [
      "role[server]"
    ]
  }
```

4) Make sure you've set up your user data bags as per 7.1

5) Apply the node definition to the VPS

```
knife solo cook root@yourserverip
```

Hopefully the above really shows how simple Chef makes provisioning a new server. Once you've assembled a collection of roles and cookbooks you're comfortable with, provisioning a new server is just four simple steps.

In fact if you have a node definition you're regularly re-using you can save it as something like `nodes/side_project_server.json` and then use:

```
knife solo bootstrap root@yourserverip nodes/side_project_server.json
```

This will install chef on the VPS at `yourserverip` and apply the node definition `nodes/yourserverip` to it.

Showing that its worked

You should now be able to ssh into the server as the `deploy` user, authenticating with the public key you provided, for example:

```
ssh deploy@yourserverip
```

Once authenticated you can test that packages such as `htop` are available (just type `htop` for an easy on the eyes view of system resource usage). You can also use `sudo ufw status` to verify that the firewall is enabled and the rules we configured have been applied.

8.0 - Installing Ruby

Overview

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-p286-falcon@global/bin:/Users/ben/.rvm/rubies/ruby-1.9.3-p286-falcon/bin:/Us\
ers/ben/.rvm/bin:/Users/ben/.rvm/gems/ruby-1.9.3-p286-falcon/bin:/Users/ben/.r\
vm/gems/ruby-1.9.3-p286-falcon@global/bin:/Users/ben/.rvm/rubies/ruby-1.9.3-p2\
86-falcon/bin:/Users/ben/.rvm/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/bin\
:/usr/sbin:/sbin:/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:/us\
r/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11:/usr/games
```

When you enter a command, 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 installing system wide.
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.

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 and contains the following default attributes:

```
"rbenv":{
  "rubies": [
    "1.9.2-p320"
  ],
  "global" : "1.9.2-p320",
  "gems": {
    "1.9.2-p320" : [
      {"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).

Next we'll look at how to use Monit to have the server automatically recover from many types of failure and alert us if there's a failure it cannot recover from.

9.0 - Monit

Overview

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.rb`.

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:

```
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.

```
set logfile syslog facility log_daemon
```

This line tells Monit to log error and status messages to `/var/log/syslog`.

```
<% if node[:monit][:enable_emails] %>
  set mailserver <%= node[:monit][:mailserver][:host] %> port <%= node[:monit]\
[:mailserver][:port] %>
    username <%= node[:monit][:mailserver][:username] %>
    password <%= node[:monit][:mailserver][:password] %>
    using tlsv1
    with timeout 30 seconds
    using hostname <%= node[:monit][:mailserver][:hostname] %>
    <% node[:monit][:notify_emails].each do |email| %>
        <% if node[:monit][:minimise_alerts] %>
            set alert <%= email %> but not on {instance, pid}
        <% else %>
            set alert <%= email %>
        <% end %>
    <% 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.

```
set httpd port 2812 and
    use address localhost
    allow localhost
    allow <%= "#{node[:monit][:web_interface][:allow][0]}:#{node[:monit][:web_\
interface][: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.

```
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, your monitoring requirements are likely to vary based on your 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 parameters.

Take as an example the below `system` definition from `monit_configs-tlq`.

```
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 uprating it. A 1.0 load averaging on a single core server needs investigating urgently and anything above 1.0 means you have a problem.

If you have more than one core, then you 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.

A Monit line to reflect this rule would look like this:

```
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 5 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 out the current 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 your 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
```

For an example of this, see the redis-tlq recipe, specifically the init.d script template in `templates/redis-server.erb`.

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 your alerts, starting off with a bias towards alerting you too much and then regularly reviewing over the first few weeks of operation to tune out alerts for events which you 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}
```

Used 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 administrator restart of each as well as allowing for manual starting of 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 I prefer to have all web traffic served from Nginx.

To do this you 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`.

The example template provides a handy shortcut for this in the nginx-tlq recipe. In the default recipe you'll see the following:

```
# Monit pass through
if @node[:monit_address]
  template "/etc/nginx/sites-enabled/monit" do
    owner "deploy"
    group "deploy"
    mode "0644"
    source "monit_interface.erb"
  end
end
```

Where `monit_interface.erb` simply contains:

```
server {
  listen 80;
  server_name <%= @node[:monit_address] %>;
  location / {
    proxy_pass http://127.0.0.1:2812;
    proxy_set_header Host $host;
  }
}
```

This means that if you're using the `monit-tlq` recipe and include the "monit_address" attribute in your node definition, for example:

```
"monit_address" : "monit.example.com"
```

Then a virtualhost entry to forward traffic requests for `monit.example.com` to the Monit interface on `127.0.0.1` will be included.

Serving multiple Monit interfaces from one nginx interface

An additional benefit of this is that you can use a single Nginx instance to serve the Monit interface from multiple machines. For example you could establish a convention that `machine-name.monit.example.c` always points to the Monit interface for `machine-name`. You could then specify in your Monit config files that the admin interface on 2812 is only accessible to the private (internal) IP address of the Nginx instance you're using to serve Monit interfaces.

If the private IP address of the additional server you're monitoring was `168.1.1.5` then your 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`.

In the next chapter we'll look at how to ensure Monit keeps running.

Upstart

Overview

Upstart is the primary utility in Ubuntu for managing startup processes and ensuring key system processes remain started. In this chapter we'll look at both using Upstart to ensure Monit is always running as well as how we manage the starting and stopping of services in general on an Ubuntu system.

What monitors Monit?

A common query at this stage is what monitors Monit? If we're using Monit to make sure everything starts up correctly when the system starts and reloads on failure, how do we make sure Monit starts and reloads?

The current tool for doing this on Ubuntu is called Upstart. Upstart takes care of processes which we need to run on boot and allows for them to be re-spawned in the event that they fail.

From this simple description it could be forgiven to think that Upstart could be used instead of Monit completely. This is not however the case. Upstart provides simple pid monitoring, e.g. if the pid for a process it monitors no longer exists, it will try and re-spawn it.

Therefore if the process still exists but is stuck, unresponsive or consuming unreasonable resource requirements, Upstart will not intervene. Additionally it doesn't provide the alerting functionality so integral to a good monitoring configuration.

Upstart Services

Processes which can be managed by scripts located in `/etc/init` are referred to as services, if we look at the `monit-tlq` recipe, we can see it adds the contents of the `monit-upstart.conf.erb` to `/etc/init/monit.conf`:

```
# after adding this file run
#   initctl reload-configuration
#
# You can manually start and stop Monit like this:
#
# start monit
# stop monit
#

```

```
description "Monit service manager"

limit core unlimited unlimited

start on runlevel [2345]
stop on runlevel [!2345]

expect daemon
respawn

exec /usr/bin/monit -c /etc/monit/monitrc

pre-stop exec /usr/bin/monit -c /etc/monit/monitrc quit
```

This creates an entry for Monit in `/etc/init` which defines:

- when the process should be automatically started (run levels 2, 3, 4 or 5)
- when it should be automatically stopped (run level no longer 2, 3, 4 or 5)
- That the type of process is expected to be a daemon (background) process
- That the process should be respawened (e.g. started again) if it's found to no longer exist.

It then goes on to define the command for starting Monit:

```
exec /usr/bin/monit -c /etc/monit/monitrc
```

Which is the part to the Monit executable along with the `-c` flag which allows us to specify the config file it should use.

It then defines commands which should be run on stop, e.g. before killing the process, in this case:

```
exec /usr/bin/monit -c /etc/monit/monitrc quit
```

Which is the command to gracefully shutdown Monit.

In the next chapter we'll look at the steps required to fork my basic Monit configurations and use them in your template configuration.

Forking My Monit Configurations

Overview

In the previous chapter we saw how powerful Monit can be for monitoring services and looked at some basic monitoring configurations. In this chapter we'll look further at why it's important to create custom Monit configurations and examine the step by step process for forking my basic Monit configurations on Github and applying the forked version to an existing VPS.

Even if you don't plan on forking the Monit configurations, this chapter will serve as a simple reference for forking any cookbook or updating a third party cookbook to a newer version.

Why is creating your own Monit configurations so important?

We've already touched on the dangers of Monit alerts being too frequent and consequently treated as spam. The key take away should be that there is no one size fits all monitoring configuration. If the system in question provides business critical services, something as small as a pid change might be important, if on the other hand you're providing a free, advertising based service to a small user base, some level of downtime may be acceptable.

Your monitoring configurations must be defined based on the balance of downtime to personal inconvenience which is acceptable for your service.

I have systems which have now been running for several years with minimal manual intervention. This was achieved by carefully tuning the Monit configurations over the first months to ensure that sources of predictable failure were eliminated or carefully handled by Monit and that suitable alerts would be properly delivered for anything Monit would not be able to deal with.

For these reasons the sample configurations provided are best used as a starting point rather than directly.

Step by Step

For the purposes of this guide I'll assume that you've already run `berks install` once to pull my simple Monit configs and potentially already applied this recipe to your VPS. Therefore we'll cover replacing my Monit configurations with your new ones as well as how to make changes to your configurations and then re-apply them.

To begin with, fork and clone the repository `https://github.com/TalkingQuickly/monit_configs-tlq`. Or, if you prefer starting from scratch, you can create a new repository with the standard chef cookbook structure (see 5.2) and commit this ti a fresh git repository.

If you've forked my repository, your `metadata.rb` will look something like this:

```
name               "monit_configs-tlq"
maintainer         "Ben Dixon"
maintainer_email   "ben@hillsbede.co.uk"
description        "Monit configs for server components"
version            "0.0.1"

recipe "monit_configs-tlq::memcached", "Monit config for memcached"
recipe "monit_configs-tlq::mongo", "Monit config for mongodb"
recipe "monit_configs-tlq::mysql-server", "Monit config for mysql server"
recipe "monit_configs-tlq::nginx", "Monit config for nginx"
recipe "monit_configs-tlq::redis-server", "Monit config for redis server"

supports "ubuntu"
```

Begin by updating the basic metadata about the recipe, in particular the name. Remember that when we add this new cookbook to our Berksfile, the name we specify here will be the name which we refer to it by.

The second section of the file contains a list of all the recipes we will define in this cookbook.

For example:

```
recipe "monit_configs-tlq::memcached", "Monit config for memcached"
```

Defines a recipe called monit_configs-tlq::memcached which will therefore expect a file to exist within `cookbook_root/recipes` called `memcached.rb` which will define the recipe.

Any templates associated with the recipe will be expected to be in `templates/default` - remember the subdirectories within templates refer to distributions, not specific recipes.

You can now modify the existing cookbooks or add your own, this might be as simple as tweaking the scenarios in which alerts are or are not sent for a particular service or adding specific monitoring for file system changes.

Finally you'll want to bump the version number in metadata.rb so that Berkshelf recognises that a new version is available.

Once you've completed your new cookbook, you can modify your Berksfile to include your new cookbook. So if, for example, I'd forked my own repository to a new cookbook called monit_configs_strict-tlq with a Github repository called `monit_configs_strict-tlq` available at:

```
 git@github.com:TalkingQuickly/monit_configs_strict-tlq.git
```

I would update my Berksfile to include:

```
cookbook 'monit_configs_strict-tlq', git: 'git@github.com:TalkingQuickly/monit\
_configs_strict-tlq.git'
```

We can then run:

```
berks install
```

If it's a newly added cookbook or:

```
berks update monit_configs_strict-tlq
```

If I'd already installed the cookbook but had since made changes to it and pushed them to the git repository.

Both will update Berksfile.lock and download our new cookbook to our chef repository.

We can then add the new recipes to a node or roles run_list and when `knife solo cook` is next run the updated cookbooks will be uploaded and applied to the server.

In the next chapter we'll look at installing Nginx as our web server.

10.0 - Nginx

Overview

In this chapter we'll cover the basic configuration of Nginx. We will not cover the details of configuring it to deploy Rails applications here, instead this is covered in part 2 on Capistrano (see 'The Place of Virtualhost files below').

The Nginx Recipe

Nginx is famously quick and simple to configure and the included Nginx recipe is correspondingly simple.

It begins by adding the official Nginx ppa so we can install a more up to date version than is available from the Ubuntu package manager by default:

```
bash 'adding stable nginx ppa' do
  user 'root'
  code <<-EOC
    add-apt-repository ppa:nginx/stable
    apt-get update
  EOC
end

# install nginx
package "nginx"
```

It then adds a simple Nginx configuration file `templates/nginx.conf.erb` which defines some simple generic configuration options. As explained below, most configuration will be defined by the virtualhosts for individual apps.

As discussed in chapter 9, if the attribute `node[:monit_address]` is included, this recipe will include a simple virtual host mapping for the Monit web interface.

The Nginx configuration file this recipe uses is reproduced along with documentation below:

```
# run as the user www-data rather than root for security
user www-data;

# try and automatically determine the optimal number of
# worker processes for the system. Auto will, by default,
# set this to the number of cores available
worker_processes auto;

# store a pid file, this will be used to monitor the process
# with Monit
pid /var/run/nginx.pid;

events {
  # number of worker connections which can be established.
  # this combined with 'auto' for worker processes gives
  # you a maximum of cpu_cores * 8000 concurrent connections
  worker_connections 8000;
}


http {

  ##
  # Basic Settings
  ##

  # speed up file uplaods
  sendfile on;

  # don't send partial frames (increases throughput)
  tcp_nopush on;

  # don't collate small packets together to save bandwidth
  # as more concerned about latency
  tcp_nodelay on;

  # how long a connection can remain idle before being closed
  # in seconds.
  keepalive_timeout 65;

  # increase size of mime types hash (should be a power of 2)
  # to allow for additional mime types
  types_hash_max_size 2048;

  # avoid problems with too many vhosts/ long domains preventing
  # the server names hash from being built on config reload
  server_names_hash_bucket_size  64;

  # define mime types in separate config files (DRY) and set a
  # fallback
  include /etc/nginx/mime.types;
  default_type application/octet-stream;

  # default log locations (if they aren't overriden in the vhost
  # files)
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;

  ##
  # Gzip Settings
  ##

  # enable gzip compression
  gzip on;

  # disable gzip compression for ie6. Strictly speaking only the
  # first version which doesn't support it but ie6 users aren't
  # always the best known for being up to date.
  gzip_disable "msie6";

  # include any additional config files
  include /etc/nginx/conf.d/*.conf;

  # include virtual hosts for specific sites
  include /etc/nginx/sites-enabled/*;
}
```

It is worth noting that at the end of the nginx config file there is the following:

```
include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
```

This allows us to load additional config files in a modular fashion similarly to how we load Monit configurations as opposed to relying on a single monolithic file. This is particularly important in the light of the suggested approach to virtualhost definition.

The place of Virtualhost files

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 interface like Monit.

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 residing there.

With this in mind, the approach suggested is that only server specific virtual hosts, such as Monit configurations if you're choosing to serve the Monit interface through nginx 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` 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.

If you run into such problems, 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.

In the following sections we'll move on to configuring data stores. Feel free to skip to the chapters relevant to the stores you'll be using. If you're stuck deciding on which data store to use, my personal preference is PostgreSQL for its combination of out of the box simplicity with a powerful advanced feature set if needed.

11.1 - 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 you note this as it will be required to gain access to the postgres user which will be needed for creating databases. This is generated using:

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

And entered into the node definition:

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

If, as happends very easily, this password is lost or forgotten, you can reset it with:

```
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:

```
{
   "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:

```
su postgres
```

And entering the password chosen above, then entering:

```
psql
```

You should then see something like:

```
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:

```
\q
```

Creating Databases

Access the psql console as above and then enter:

```
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:

```
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:

```
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:

```
CREATE USER my_user WITH PASSWORD 'my_password';
```

and press return.

You can then grant this user all privileges on a specific database with:

```
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:

```
psql -h 127.0.0.1 database_name my_user
```

or

```
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:

```
\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 postgres 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:

```
# 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 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 host name.

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.

We can now see why:

```
psql -d database_name -U my_user
```

would fail but:

```
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:

```
CREATE USER deploy;
GRANT ALL PRIVILEGES ON DATABASE database_name to deploy;
\q
```

You can then `exit` back to your deploy user and enter:

```
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`:

```
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:

```
"config" :{
  "listen_addresses" : "*"
}
```

which gives the following line in `postgresql.conf`:

```
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:

```
"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:

```
{"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`:

```
# 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 add the configuring defined in "Allowing External Access" above.

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:

```
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:

```
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:

```
check process postgresql with pidfile /var/run/postgresql/9.1-main.pid
  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.

11.2 - MySQL

Overview

In this chapter we'll cover getting the MySQL server packages installed, setting up authentication and managing databases with the console.

Installation

For a simple installation using the 'mysql-server' role, the only parameters it is necessary to set in the node definition are:

```
"mysql": {
    "server_root_password":"your_password",
    "server_debian_password":"your_password",
    "server_repl_password":"your_password"
  }
```

Which are the plaintext passwords for the three roles. The most important one to remember is root which you'll need for creating databases, users and setting up permissions.

It's important to note that the chef recipe can only set the passwords when it is installing MySQL. This cannot be used to change the passwords. This also means that if, for any reason, the mysql-server package has already been installed when the mysql::server recipe is run, it is likely to fail. If therefore you run into strange permissions errors when chef reaches MySQL, check carefully that no other recipe is installing the mysql-server package as a dependency.

Some guides will suggest adding the following to the mysql-server role:

```
client": {
  "packages": ["mysql-client", "libmysqlclient-dev","ruby-mysql", "mysql-serve\
r"]
}
```

As some Rails gems will required the server package to be installed even when the machine they are running on is not the server.

The problem with this is that if your run list looked like this:

```
mysql::client
mysql::server
```

Then the mysql-server package would be installed by the client recipe and so the server recipe would not be able to apply configuration to it and so would fail.

Creating Databases

SSH into the remote server and then start a root mysql console by entering:

```
mysql -u root -p
```

This will then prompt you to enter a password. Enter the password selected as the server root password in your chef configuration and press enter. You'll then be taken to the MySQL console.

Here you can enter arbitrary SQL to create databases, users and assign permissions.

To create a database enter:

```
CREATE DATABASE database_name;
```

You can then type `exit` to return to the shell prompt.

Configuring Authentication

Return to the mysql shell as above and enter the following to create a user:

```
CREATE USER 'user_name'@'localhost' IDENTIFIED BY 'plaintext_password';
```

This will crate a user with the username 'user_name', password 'plaintext_password' who can only connect from localhost.

To give this user access to a database created as above, enter:

```
 GRANT ALL PRIVILEGES on database_name.* TO 'username'@'localhost';
```

You can verify that the user has access to this database by exiting back to the shell and then using the new user to connect to the new database:

```
mysql -u user_name -p database_name
```

and then entering the password selected for that user.

If, at any time, you want to check which database you currently have selected, you can use `SELECT DATABASE();` which will result in output something like:

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| test1      |
+------------+
1 row in set (0.01 sec)
```

Where `test1` is the database you currently have selected. You can view a list of available databases by entering:

```
SHOW DATABASES;
```

Which will result in output similar to:

```
    mysql> SHOW DATABASES;
    +--------------------+
    | Database           |
    +--------------------+
    | information_schema |
    | mysql              |
    | performance_schema |
    | test               |
    | test1              |
    | test2              |
    +--------------------+
    6 rows in set (0.01 sec)
```

If the user access to multiple databases, you can change which database is selected by entering:

```
USE database_name;
```

Importing and Exporting Databases

Like PostgreSQL, MySQL provides a simple utility for exporting data from a database to an SQL file which can then be imported into another MySQL database or kept for backup purposes.

To create a dump of the database `my_database` which can be accessed by the user `user1` with pasword `user1spassword` you would enter:

```
mysqldump -u user1 -p'user1spassword' my_database > ouput_file.sql
```

The above would create an SQL file which contains the commands needed to create both the structure of the database (the tables, columns etc) and the data within it.

Sometimes you may want to just export the data. For example if you want to have Rails create the database and use `rake db:migrate` to create the table structure. This is often desirable as it ensures that the tables are created such that they can be accessed by the rails database user, rather than the user from the machine it was exported from.

To export just the content, you add the `--no-create-info` flag. So the command would be:

```
mysqldump -u user1 -p'user1spassword' --no-create-info my_database > ouput_fil\
e.sql
```

Importing data from an SQL file is even simpler, if you have a file called `a_database_backup.sql` generated with mysqldump, simple copy it to the remote server and then enter:

```
mysql -u username -p -h localhost target_database < a_database_backup.sql
```

To execute the contents of the SQL file on the database target_database. The username and password should be the username and password of a user who already have access to the target database on the target machine.

If you're using this to copy production data to a local development machine, you'll want to run:

```
rake db:drop db:create
```

To remove any existing data first. If you're restoring from a content only backup, this would instead be:

```
rake db:drop db:create db:migrate
```

Monit

A monit configuration for mysql is available in the monit_configs-tlq::mysql recipe:

```
check process mysql with pidfile /var/run/mysqld/mysqld.pid
  group database
  start program = "/etc/init.d/mysql start"
  stop program = "/etc/init.d/mysql stop"
  if failed host 127.0.0.1 port 3306 then restart
  if 15 restarts within 15 cycles then timeout
```

This will check for both the existence of the process and that the service is responding on port 3306. Bear in mind that this will need to be changed if you ever change the port MySQL is available on or remove access on localhost.

Server Admin

Whilst it's always important to have a good grasp of the command line when managing a database server, if you're on OSX, MySQL does have an ace up its sleeve for day to day work; Sequel Pro.

This is a graphical user interface for MySQL servers through which you can manage everything from database and user creation to managing indexes and checking data integrity. As someone who generally leans towards Postgresql when looking for a relational database provider, Sequel Pro is the one thing I consistently miss.

You can get Sequel Pro from:

http://www.sequelpro.com/

11.3 - MongoDB

Overview

In this chapter we'll cover the basics of getting a MongoDB server installed and ready for use with a Rails application. First however, a note of caution. MongoDB is excellent for a very specific subset of applications however deploying it is, in my experience, easier to get wrong than MySQL or PostgreSQL.

The configuration covered here is the simplest possible required to setup a MongoDB application with Rails. It does not include user authentication or replica sets, both of which should be carefully considered if deploying MongoDB at production scale.

For more on configuring user authentication see the excellent documentation at:

http://docs.mongodb.org/manual/administration/security-access-control/

and for more on Replica sets:

http://docs.mongodb.org/manual/core/replication-introduction/

Installation

In the simple sample configuration simple include the role `mongo-server` in a nodes run list, for example:

"run_list": ["role[server]", "role[nginx]", "role[mongo-server]", "role[rails-app]", "role[redis-server]"]

This will include the `mongo-tlq` recipe which installs mongo from the official 10gen (makers of Mongo) repository and adds a simple monit recipe.

Accessing the Mongo Shell

For a simple configuration like this, we need far less interaction with the mongo shell, our rails app will take care of creating the database when we run:

```
rake db:create
```

and as there is no enforced schema, there's no need for `rake db:migrate`.

It is however useful to have a basic familiarity with the shell so some simple commands are covered here. To access the shell, first ssh into your server and enter:

```
mongodb
```

Since we have no authentication at this stage (we rely on the firewall to prevent unwanted external access) this will immediately result in a prompt similar to this:

```
MongoDB shell version: 2.4.3
connecting to: test
>
```

To view a list of available databases:

```
show dbs
```

Which will result in output similar to:

```
local 0.078125GB
another_database_staging 5.951171875GB
test  (empty)
```

Which shows the name of each database followed by its size.

To select a particular database to execute queries on:

```
use database_name
```

For more on the Mongo shell see:

http://docs.mongodb.org/manual/tutorial/getting-started-with-the-mongo-shell/

Importing and Exporting Databases

To create a dump of a Mongo database with name `my_database` in the folder current (which will be created automatically) enter:

```
mongodump --db my_database --out current
```

In general if you're looking to copy this database to a different server (including a local development machine) you'll then want to compress this folder:

```
tar jcvf current.tar.bz2 current
```

And scp it to the target machine. You can then uncompress it with:

```
tar jxvf current.tar.bz2
```

And restore it with:

```
mongorestore -d target_database current/source_db_name --drop
```

the `--drop` option will drop each of the collections it finds (in the provided folder) before restoring the data from the dump.

11.4 - Redis

Overview

Redis is, in general, extremely simple to install and manage for small to medium sized projects. Here we'll cover the basics of installing it using chef as well as key aspects of security and a common gotcha with dataset sizes.

Installation

Simply include the role `redis-server` which includes the `redis-tlq` cookbook and an appropriate Monit config from monit_configs-tlq.

Redis-tlq installs redis from a ppa rather than the default from the Ubuntu package manager, you can see more about the ppa along with the current version it has built here: https://launchpad.net/∼chris-lea/+archive/redis-server

This repository is generally up to date and always more up to date than the standard Ubuntu repository.

The recipe only has one attribute to be set:

```
"redis": {
   "dont_bind" : false
}
```

If this is set to true then Redis will accept connections from all hosts (assuming they are allowed through the firewall) rather than the default which limits connections to just those from localhost.

Security

Redis does not include any sort of authentication. By default it binds to 127.0.0.1 which means that only connections originating from the machine it is installed on can access it. If we've disabled this binding then any incoming connection which is allowed through the firewall will be allowed complete access to anything store in Redis.

It's therefore extremely important that if local binding is disabled, for example when Redis is being run on a server of its own, that the Firewall is carefully configured to only allow connections through from trusted servers.

Managing Size

The redis-tlq recipe assumes that Redis is being used primarily as a cache store. Because of this it has the following none-standard configuration at the bottom of redis.conf:

```
# prevent redis from ever using 400MB of memory
maxmemory 419430400

# when max memory is reached, go through all keys and delete the
# "least recently used" ones. NOTE: this doesn't do quite what it
# sounds like, in practice Redis takes a random sample of keys
# (see below) and of the sample it takes, deletes the one which was
# least recently used.
maxmemory-policy allkeys-lru

# when looking for a key to expire on maxmem, randomly pick 10
# and delete the one with the highest idle time (least recently
# accessed)
maxmemory-samples 10
```

This configuration prevents Redis from ever using more than 400mb of memory. If 400Mb is reached and a new key is inserted, Redis will pick a random sample of 10 (maxmemory-samples) and delete the one which has been least recently accessed.

With this option set, Redis behaves a lot like memcached. However it means that no data is safe from deletion. Therefore if you are planning to use Redis as a permanent data store, you should modify the configuration accordingly.

The simple redis-tlq recipe is setup in this way because in my experience the vast majority of Rails applications which use Redis are in fact using it as a cache. Without this behavior configured, it's surprisingly easy for for the in memory data set to grow hugely in size.

Take for example a common use case; caching geocoder results. If the results are not set to expire automatically (which many of the popular geocoding gems don't do) then the cache will grow indefinitely whenever a new unique query is performed. If the precision of queries is not limited (a good idea for 'search in this area' type queries) this can be a huge volume of new entries every day.

On a single box configuration such as our simple one, where the web server, database server and Redis server all run on the same instance, this can cause significant problems as the Redis set will be kept in memory, gradually causing the system to start swapping and eventually to run out of memory completely.

Monit

monit_configs-tlq::redis-server creates the following simple monit definition for watching the redis process:

```
#Monitoring redis
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 15 restarts within 15 cycles then timeout
```

12.0 - Memcached

Overview

Of all the system components covered in this book, Memcached is by far the simplest to install and maintain. This chapter will be correspondingly brief. We'll cover installation, a single configuration option and a very simple Monit profile and then we're done.

Installation

The memcached-server role simply includes the memcached-tlq and monit_configs-tlq::memcached recipes.

The memcached-tlq recipe installs the memcached package from the standard Ubuntu package repository and creates a config file which sets the following options:

```
# run as a daemon (in the background)
-d

# Log memcached's output to /var/log/memcached
logfile /var/log/memcached.log

# Start with a cap of 64 megs of memory.
# Note that the daemon will grow to this size but does not
start out holding this much memory
-m 64

# Default connection port is 11211
-p 11211

# Run the daemon as root. The start-memcached
# will default to running as root if not specified here
-u memcache

# This should only be excluded using the dont_bind
# attribute if memcached is suitably
# firewalled
#
<% unless node['memcached'] && node['memcached']['dont_bind'] %>
  -l 127.0.0.1
<% end %>
```

As you can see the final option is similar to that of Redis in the previous chapter. If the `dont_bind` attribute is set to true in the node definition like follows:

```
"memcached": {
  "dont_bind" : true
}
```

Then Memcached will accept connections from any IP addresses, otherwise it will only accept connections from 127.0.0.1 (localhost).

Security

Like Redis, Memcached is entirely unauthenticated. As above it can operate either bound to (only accept connections from) localhost or unbound (accept connections from anywhere).

Therefore when running in unbound mode, it's essential that the Firewall is carefully configured to only allow trusted machines to access the Memcached port (11211 by default).

Monit

monit_configs-tlq::memcached includes the following simple configuration for monitoring Memcached:

```
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 both that the process exists and whether port 11211 on localhost is responding to the memcache protocol.

Bear in mind that if the Memcached port is changed, the Monit configuration will need updating accordingly.

13.0 Testing with Vagrant

Overview

Vagrant makes it easy to manage and distribute virtual machines. Vagrant is an extremely powerful tool in itself with a particular strength of making it easy to distribute local testing environments to developers which (almost) perfectly mirror your production configuration.

This section does not cover how to use Vagrant to create these re-usable environments. Instead it covers a very specific work flow I use for testing Chef Recipes.

This can be especially useful when you're testing changes to recipes and want to see how it will interact with your existing production configuration.

For more on this work flow, read on, for a general introduction to the power of Vagrant, start here:

http://docs.vagrantup.com/v2/getting-started/index.html

Getting Setup

Go to http://downloads.vagrantup.com/ and download the most recent version of Vagrant (I used 1.3.5)

Make sure Vagrant is available in terminal by typing `vagrant` and ensuring you get output similar to the following:

âžœ ~ vagrant Usage: vagrant [-v] [-h] command [

```
  -v, --version                    Print the version and exit.
  -h, --help                       Print this help.
```

â€¦..

For help on any individual command run `vagrant COMMAND -h`

Create a new directory then:

vagrant init precise64 http://files.vagrantup.com/precise64.box

This will generate a Vagrantfile

vagrant up

Will then download the pre created image of Ubuntu 12.04, don't worry about the initial note saying that precise64 doesn't yet exist

You can then ssh into this virtual machine using:

vagrant ssh

Which will log you in as the vagrant user. Passwordless sudo is enabled so you don't have to worry about default passwords

SSHing in directly

When testing chef cookbooks, roles and node definitions, I like to be able to apply a definition to my vagrant virtual machine in exactly the same way I will my production server. A prerequisite for this is that I can interact with it without using any vagrant specific commands. Happily SSHing in the old fashion way is very simple. Begin by running:

vagrant ssh-config

This will give output something like the following:

```
HostName 127.0.0.1
User vagrant
Port 2222
UserKnownHostsFile /dev/null
StrictHostKeyChecking no
PasswordAuthentication no
IdentityFile /Users/ben/.vagrant.d/insecure_private_key
IdentitiesOnly yes
LogLevel FATAL
```

Which shows the config which is being used by the `vagrant ssh` command.

From the above we can construct the following ssh command:

```
ssh vagrant@127.0.0.1 -p 2222 -i /Users/ben/.vagrant.d/insecure_private_key
```

Which means establish a connection to 127.0.0.1 on port 2222 using the vagrant generated private key file to authorise the user vagrant

Why not just use Vagrants built in chef and chef-solo support?

Vagrant has excellent chef-solo support built in. Rather than using our existing node definition files, we can effectively include the data from a node definition file, in our vagrantfile. When vagrant up is run for the first time, the vagrant image will be automatically provisioned. This is very powerful when we're using vagrant to make it easy for developers to provision environments which closely match production. However if we're developing testing Chef cookbooks, role definitions and node definitions, I prefer the process of provisioning the Vagrant VM to be identical in as many ways as possible to the process of provisioning production and staging VM's.

For me this means using exactly the same commands (knife solo prepare, knife solo cook etc). I strongly recommend reading http://docs.vagrantup.com/v2/provisioning/chef_solo.html to understand how chef provisioning can be built into a Vagrantfile if you do start using Vagrant for local development environments.

Testing chef cookbooks

Now we have a vagrant virtual machine which we can ssh into as we would any VPS, we can test our chef configuration on it.

Open a terminal in your chef repository, so in my case

```
cd ~/proj/dev_ops/rails_server_template
```

then install chef on your Vagrant VM:

```
knife solo prepare vagrant@127.0.0.1 -p 2222 -i /Users/ben/.vagrant.d/insecure\
_private_key
```

Where the port, ip and path to private key match the results of vagrant ssh-config before.

The output should be something like:

```
Bootstrapping Chef...
--2013-10-20 15:27:50--  https://www.opscode.com/chef/install.sh
Resolving www.opscode.com (www.opscode.com)... 184.106.28.83
Connecting to www.opscode.com (www.opscode.com)|184.106.28.83|:443... connecte\
d.
HTTP request sent, awaiting response... 200 OK
Length: 6790 (6.6K) [application/x-sh]
Saving to: `install.sh'

100%[======================================>] 6,790       --.-K/s   in 0s

2013-10-20 15:27:56 (799 MB/s) - `install.sh' saved [6790/6790]

Downloading Chef 11.6.2 for ubuntu...
Installing Chef 11.6.2
Selecting previously unselected package chef.
(Reading database ... 51095 files and directories currently installed.)
Unpacking chef (from .../chef_11.6.2_amd64.deb) ...
Setting up chef (11.6.2-1.ubuntu.12.04) ...
Thank you for installing Chef!
```

This will have created an empty node definition file in `nodes/127.0.0.1`. To use this auto generated node definition file (`nodes/ip_address.json`) simply populate it and then enter:

```
knife solo cook vagrant@127.0.0.1 -p 2222 -i /Users/ben/.vagrant.d/insecure_pr\
ivate_key
```

Otherwise use

```
knife solo cook vagrant@127.0.0.1 nodes/my_node_definition.json -p 2222 -i /Us\
ers/ben/.vagrant.d/insecure_private_key
```

Where nodes/my_node_definition.json is the path to an existing node definition.

The output from this will begin by installing the chef recipes from your Berksfile and then show the output from each individual recipe.

Once this process completes, the Vagrant VM should now be configured as per your chef definition.

Users, Sudo and Root

By default Vagrant sets up the `vagrant` user with the password `vagrant`. This user has passwordless sudo enabled so you never have to worry about default root passwords and the like.

When you run `cook` for the first time using the vagrant user, chef will automatically try and use `sudo` where root access is required. The first time this will work correctly because the vagrant user has passwordless sudo enabled. If however the configuration you're applying with chef includes defining who can sudo and how (such as my example rails server template), the next time you try and run `cook` for example to test another change to your repository, you're likely to see something like the following:

```
Running Chef on 127.0.0.1...
Checking Chef version...
Enter the password for vagrant@127.0.0.1:
```

The default password for the vagrant user is vagrant however after entering, the process will still fail with the message:

```
ERROR: RuntimeError: Couldn't find Chef >=0.10.4 on 127.0.0.1. Please run `kni\
fe solo prepare vagrant@127.0.0.1 -i /Users/ben/.vagrant.d/insecure_private_ke\
y -p 2222` to ensure Chef is installed and up to date.
```

This is because the vagrant user is no longer in the sudoers file and so chefs attempt to run commands with sudo fails. You can verify that this is the case by running `vagrant ssh` which starts a shell with the vagrant user and then running `sudo ls` which will give output like:

```
vagrant@precise64:~$ sudo ls
[sudo] password for vagrant:
vagrant is not in the sudoers file.  This incident will be reported.
```

Confirming that our vagrant user can no longer sudo.

There are various ways around this:

The first, and simplest, is to simply add the vagrant user to the sudoers group in your node definition file. This has the added bonus that it allows commands like `vagrant halt` to continue working. This is generally the approach I use in day to day testing.

Some argue however that we should not modify the code we are testing in order to work with the tool we are testing it with. To avoid this we can modify our `cook` command going forward to execute using a user we know we have given sudo rights to, in the case of the rails_example_template this user would be deploy so going forward the command would be:

```
knife solo cook deploy@127.0.0.1 nodes/my_node_definition.json -p 2222 -i /Use\
rs/ben/.vagrant.d/insecure_private_key
```

Personally I prefer a hybrid. I generally add the vagrant user to the sudoers list in the node definition so that the vagrant interface can continue to function as usual however I use the above format when I want to test that provisioning as the deploy user works as expected.

Port Forwarding

If we intend to use this Vagrant box for testing a Rails Web App, we'll need to set up port forwarding to allow access from our local web browser to the VM.

Move back into the original vagrant directory and open `Vagrantfile`. The documentation within this file is excellent so it's worth reading through, the section we're interesting in is this:

```
  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # config.vm.network :forwarded_port, guest: 80, host: 8080
```

Uncommenting or adding the final line:

```
config.vm.network :forwarded_port, guest: 80, host: 8080
```

Will have the effect of making port 80 on the VM available in your local web browser on port 8080 e.g. `localhost:8080`

14.0 End of Part 1

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.

15.0 Deploying with Capistrano

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.

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.

Capistrano 2 or 3

Choosing between them

Capistrano is the defacto standard for deploying Rails applications. The majority of tutorials and existing documents focus on Capistrano 2, in particular there are several excellent Railscasts on using Capistrano 2 to deploy to a VPS which many apps deployment processes are based on.

Capistrano 3 was recently released and, having recently migrated the deployment process for several large client applications from version 2 to version 3, I'm confident that the version 3 rewrite introduces some substantial improvements.

As a result, this book will focus on the use of Capistrano V3. If for any reason you need to use V2, the approach outlined in this blog post of mine: http://www.talkingquickly.co.uk/2013/11/deploying-multiple-rails-apps-to-a-single-vps/ serves as a starting point with sample code.

What's new in V3

For full details see the release announcement[1], but the key bits I think make the upgrade worthwhile are:

- It uses the Rake DSL instead of a specialised Capistrano one; this makes writing Capistrano tasks exactly like writing rake tasks, something most Rails developers have some familiarity with.
- It uses SSHkit for lower level functions around connecting and interacting with remote machines. This makes writing convenience tasks, which do things like streaming logs or checking processes, much easier.

[1] http://www.capistranorb.com/2013/06/01/release-announcement.html

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.

Upgrading from V2

If you already have a Capistrano 2 configuration for the application to be deployed, I suggest you archive all of this off and start from scratch with Capistrano 3. In general this will mean renaming (for example by appending .old) all of the following:

```
Capfile
config/deploy.rb
config/deploy/
```

Adding Capistrano to an application

```
gem 'capistrano', '~> 3.1.0'

# 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"
```

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:

```
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 me uncomfortable about many 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:

```
# 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:

```
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:

```
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(:rb\
env_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 publ\
ic/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.

```
# 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:

```
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:

```
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:

```
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:

```
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:

```ruby
# 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 section 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:

```ruby
# 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.

To address this, this section:

```ruby
# 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
))
```

is 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`.

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:

```
# 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 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`:

```
# 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.

```
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:

```
cp database.yml.example database.yml
```

Edit it with a text editor such as vim, e.g:

```
vim database.yml
```

And enter the details of the database that the app should connect to. If you're using Postgres or MySQL you'll need to create this database using the instructions from Chapter 11.

Deploying

Now that we've run:

```
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:

```
cap production deploy
```

And waiting. The first deploy can take a while as Gems are installed, so be patient.

Failed doesn't mean Failed

Don't panic if you see lots of lines with `(failed)` in brackets. This is generally because Capistrano V3 uses NetSSH which will output (failed) if any command produces a none 0 return code. Not all of the commands follow the "0 = success" convention and so we see some of these (failed) messages. If a command has actually failed, in that the deploy will then fail, Capistrano will halt with an exception.

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 I've found to be efficient for big production configurations.

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.

16.0 - Unicorn Configuration and Zero Downtime Deployment

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 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 its 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 from 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) rather than 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 17.0.

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`

```
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 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 its 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.

Init script

This is the primary script for starting, stopping and restarting the unicorn workers.

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.

The unicorn init script is created when we run `cap deploy:setup_config`. It is stored in `shared/unicorn_init.sh` and symlinked to `/etc/init.d/unicorn_YOUR_APP_NAME`.

Basic usage is simple:

```
/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:

```
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

Defined in the init script as:

```
start)
  sig 0 && echo >&2 "Already running" && exit 0
  run "$CMD"
  ;;
```

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME start`.

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

Defined in the init script as:

```
stop)
  sig QUIT && exit 0
  echo >&2 "Not running"
  ;;
```

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME stop`.

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 f\
inishing.
```

If the Pidfile does not exist then it outputs the text "Not running" and exits.

Force-stop

Defined in the init script as:

```
force-stop)
  sig TERM && exit 0
  echo >&2 "Not running"
  ;;
```

Invoked with `/etc/init.d/unicorn_YOUR_APP_NAME force-stop`.

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 it's 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:

```
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 \
process 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):

```
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, this 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:

```
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/2014032416201` 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 use 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 your `unicorn.log` file.

We avoid the above two problems with the following `before_exec` block in our Unicorn configuration file:

```
# 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.

17.0 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

```
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` 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:

extract from: config/deploy/shared/nginx.conf.erb

```
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:

extract from: config/deploy/production.rb

```
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:

extract from: config/deploy/shared/nginx.conf.erb

```
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 by appending `.gz` to the file.

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:

extract from: config/deploy/shared/nginx.conf.erb

```
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):

extract from: config/deploy/shared/nginx.conf.erb

```
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(!).

extract from: config/deploy/shared/nginx.conf.erb

```
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\
.new"
        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.

18.0 - 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 for Sidekiq 2.x. Sidekiq 3 has been recently released with some significant improvements. This chapter will be updated for Sidekiq 3, until then this process should work as long as the `capistrano-sidekiq` gem is included.

There's more about the changes in version 3 here http://www.mikeperham.com/2014/03/28/sidekiq-3-0/.

Capistrano Integration

Sidekiq 2 provides in built Capistrano integration. In Sidekiq 3 this has been factored out into a gem `capistrano-sidekiq`. To include this functionality simply add the following line to your `Capfile`:

```
require 'capistrano/sidekiq'
```

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 issued to instruct the Sidekiq worker process not to start processing any new jobs:

```
bundle exec sidekiqctl quiet SIDEKIQ_PID
```

Once the code has been updated, the following will be issued to stop the worker process:

```
bundle exec 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 requireing 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.

We want to ensure that Sidekiq stores a Pidfile with a known name so that we can use Monit to keep track of its status. Furthermore we want to ensure that it follows our directory structure convention of storing all pid files in APP_ROOT/tmp/pids. Therefore we add the following to our `deploy.rb`:

```
set :sidekiq_pid, "#{current_path}/tmp/pids/sidekiq.pid"
```

Init Script

Although the Sidekiq process will be automatically started when we deploy, we want an Init Script to allow us to manage it with Monit.

To enable the init.d script provided in the sample configuration, first add `sidekiq_init.sh` to the `:config_files` array in `deploy.rb`.

Then add `sidekiq_init.sh` to the `:executable_config_files` array in `deploy.rb`.

Finally add

```
{
  source: "sidekiq_init.sh",
  link: "/etc/init.d/sidekiq_{{full_app_name}}"
}
```

to the `:symlinks` array in `deploy.rb`.

You'll then need to run `cap STAGE deploy:setup_config` to update the remote server with the init script.

Monit Config

Add the followed to then end of your `config/deploy/shared/monit.erb`

```
check process sidekiq_worker
  with pidfile <%= current_path %>/tmp/pids/sidekiq.pid
  start program = "/etc/init.d/sidekiq_<%=application%>_<%= fetch(:rails_env)%\
› start"
  stop program = "/etc/init.d/sidekiq_<%=application%>_<%= fetch(:rails_env)%>\
  stop"
```

You'll then need to run `cap STAGE deploy:setup_config` to update the remote server with the new monit configuration.