The Pragmatic Programmers

Deploying Rails

Automate, Deploy, Scale, Maintain, and Sleep at Night

Anthony Burns
and Tom Copeland

Edited by Brian P. Hogan

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/cbdepra/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

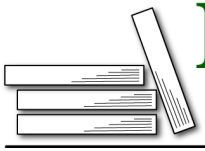
Dave & Andy

Deploying Rails

Automate, Deploy, Scale, Maintain, and Sleep at Night

Anthony Burns
Tom Copeland

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-93435-695-1
Printed on acid-free paper.
Book version: B1.0—January 19, 2012

Contents

Preface

Ruby on Rails has taken the web application development world by storm. Those of us who have been writing web apps for a few years remember the good old days when the leading contenders for web programming languages were PHP and Java, with Perl, Smalltalk, and even C++ as fringe choices. Either PHP or Java could get the job done, but millions of lines of legacy code attest to the difficulty of using either of those languages to deliver solid web applications that are easy to evolve.

But Ruby on Rails changed all that. Now thousands of developers around the world are writing and delivering high quality web applications on a regular basis. Lots of people are programming in Ruby. And there are plenty of books, screencasts, and tutorials for almost every aspect of bringing a Rails application into being.

We say "almost every aspect" because there's one crucial area in which Rails applications are not a joy; that area is deployment. The most elegant Rails application can be brought low by runtime environment issues that make adding new servers an adventure, unexpected downtime a regularity, scaling a difficult task, and frustration a constant. There are good tools out there for deploying, running, monitoring, and measuring Rails applications, but pulling them together into a coherent whole is no small effort.

In a sense, we as Rails developers are spoiled. Since Rails has such excellent conventions and practices, we expect deploying and running a Rails application to be a similarly smooth and easy path. And while there are a few standard components for which most Rails developers will reach when rolling out a new application, there are still plenty of choices to make and decisions which can affect application's stability.

And that's why we've written this book. After several years of full-time consulting for companies who were writing and fielding Rails applications, we've learned a few things about running busy Rails applications in production environments. Throughout this book we'll explore various aspects of deploying

Rails applications, and we'll review and apply the practices and tools that helped us keep our consulting clients happy by making their Rails applications reliable, predictable, and generally speaking, successful. When you finish reading this book you'll have a firm grasp on what's needed to deploy your application and keep it running. You'll also learn pick up valuable techniques and principles on constructing a production environment that watches for impending problems and alerts you before things go wrong.

Who Should Read This Book?

This book is for Rails developers who, while comfortable with coding in Ruby and using Rails conventions and best practices, may be less sure of how to get a completed Rails application deployed and running on a server. Just as you learned the Rails conventions for structuring an application's code using REST and MVC, you'll now learn how to keep your application faithfully serving hits, how to know when your application needs more capacity, and how to add new resources in a repeatable and efficient manner so you can get back to adding features and fixing bugs.

This book is also for system administrators who are running a Rails application in production for the first time, or for those who have a Rails application or two up and running but would like to improve the runtime environment. You probably already have solid monitoring and metrics systems; this book will help you monitor and measure the important parts of your Rails applications. In addition, you may be familiar with Puppet, the open source system provisioning tool. If so, by the end of this book you'll have a firm grasp on using Puppet and you'll have a solid set of Puppet manifests. Even if you're already using Puppet you may pick up a trick or two from the manifests that we've compiled.

Finally, this book is for project managers who are overseeing a project where the primarily deliverable is a Rails application that performs some business functionality. You can use the major sections in this book as a checklist. There's a chapter on monitoring; what kind of monitoring is being done on your application, and what kind of situations might occur where would you like to trigger some sort of alert? There's a chapter on metrics; what kind of charts and graphs would best tell you how the application is meeting the business' needs? If your application has some basic story for each chapter

in this book you'll know that you're covering the fundamentals of a solid Rails application environment.

What is in the Book?

This book is centered around an example social networking application "MassiveApp". While MassiveApp may not have taken the world by storm just yet, we're confident that it's going to be a winner, and we want to build a great environment in which MassiveApp can grow and flourish. This book will take us through that journey.

We'll start with Chapter 2, *Getting Started with Vagrant*, on page 9, where we'll learn how to set up our own virtual server with Vagrant, an open-source tool that makes it easy to configure and manage VirtualBox virtual machines.

In Chapter 3, *Rails on Puppet*, on page 25, we'll introduce what's arguably the most popular open source server provisioning tool, Puppet. We'll explain Puppet's goals, organization, built-in capabilities, and syntax, and we'll build Puppet manifests for MassiveApp. This chapter will provide you with a solid grasp on more in-depth Puppet topics such as writing your own Puppet types, using Puppet defines, and getting better visibility into Puppet status with the Puppet Dashboard.

In Chapter 4, *Basic Capistrano*, on page 51 we'll explore the premier Rails deployment utility, Capistrano. We'll build a deployment file for MassiveApp, and we'll describe how Capistrano features like hooks, roles, and custom tasks can make Capistrano a linchpin in your Rails development strategy.

Chapter 5, *Advanced Capistrano*, on page 69 is a deeper dive into more advanced Capistrano topics. We'll make deployments faster, we'll use the Capistrano multistage extension to ease deploying to multiple environments, we'll explore error handling and recovery, and we'll look at capturing output from remote commands. Along the way we'll explain more of the intricacies of Capistrano variables and roles. This chapter will get you even further down the road to Capistrano mastery.

In Chapter 6, *Monitoring with Nagios*, on page 79 we'll look at monitoring principles and how they apply to Rails applications. We'll build a Nagios Puppet module that monitors system, service, and application-level thresholds. We'll build a custom Nagios check that monitors Passenger memory process size, and we'll build a custom Nagios check that's specifically for checking aspects of MassiveApp's data model.

Chapter 7, *Collecting Metrics with Ganglia*, on page 107 will cover the nuts and bolts of gathering metrics around a Rails application, both from an infrastruc-

ture and an application level. We'lll install and configure Ganglia and we'll explore the Ganglia plugin ecosystem. Then we'll write a new Ganglia metric collection plugin for collecting MassiveApp user activity.

Chapter 8, *Maintaining the Application*, on page 109 discusses the ongoing care and feeding of a production Rails application. We'll talk about backups, recovering from hardware failures, managing log files, and handling downtime both scheduled and unscheduled. This chapter is devoted to items that might not arise during the first few days that an application is deployed, but will definitely arise as the application weathers the storms of user activity.

Chapter 9, *Running Rubies with RVM*, on page 111 covers the Ruby Version Manager (RVM). RVM is becoming more and more common as a Rails development tool, and it has its uses in the Rails deployment arena as well. We'll cover a few common use cases for RVM as well as some tricky issues that arise due to the way that RVM modifies a program's execution environment.

Chapter 10, *Special Topics*, on page 113 discusses a few topics which don't fit nicely into any of the previous chapters but are nonetheless interesting parts of the Rails deployment ecosystem. You don't need to use these tools in every Rails deployment, but they're good items to be familiar with.

How to Read This Book

If you're new to Rails deployment you can read most of this book straight through. The exception would be Chapter 5, *Advanced Capistrano*, on page 69, which you can come back to once you've gotten your application deployed and have used the basic Capistrano functionality for a while.

If you're an advanced Rails developer you may be more interested in the system administration sections of this book. You can get a working knowledge of Puppet by reading Chapter 3, *Rails on Puppet*, on page 25, and you may pick up some useful monitoring and metrics tips from those chapters. The Chapter 8, *Maintaining the Application*, on page 109 chapter is also worth a read to ensure you have your bases covered with applications that you've already deployed.

If you're a system administrator, you may want to read the Rails-specific parts of Chapter 6, *Monitoring with Nagios*, on page 79 and Chapter 7, *Collecting Metrics with Ganglia*, on page 107 to see how to hook monitoring and metrics tools up to a Rails application. You might also be interested in Chapter 8, *Maintaining the Application*, on page 109 to see what a few solutions to problems that come up with a typical Rails application. If you've got a number of Rails applications running on a single server, you will also be interested in Chapter

9, *Running Rubies with RVM*, on page 111 to see how to isolate those applications' runtime environments.

Throughout this book we'll have short "For Future Reference" sections which display a summary of the configuration files and scripts presented in the chapter. You can use these as a quick reference guide to how we recommend using the tools that were discussed; they'll contain the final product without all the explanations.

Tools and Online Resources

Throughout the book we'll be building a set of configuration files. Since we want to keep track of changes to those files, we're storing them in a revision control sytem, and since Git is a popular revision control system we're using that. There's a variety of documentation on Git's home page [1] and there are some excellent practical exercises on http://gitready.com/. Using Git isn't strictly necessary, but you'll want to have it installed to get the most out of the examples.

This book has a companion website [2] where we post articles and interviews and anything else that we think would be interesting to folks who are deploying Rails applications. You can also follow us on Twitter at http://twitter.com/deployingrails/.

1. FIXME
2. http://deployingrails.com/

CHAPTER 1

Introduction

A great Rails application needs a great place to live. There are a variety of choices for hosting Rails applications. In this chapter, we'll take a quick survey of that landscape.

Much of this book is about techniques and tools. Before diving in, though, let's make some more strategic moves. One choice that we need to make is deciding where and how to host our application. We also want to think philosophically about how we'll approach our deployment environment. Since the place where we're deploying to affects how much we need to think about deployment technology, we'll cover that first.

1.1 Where Do I Host my Rails Application?

Alternatives for deploying a Rails application generally break down into either shared hosting, using Heroku, or rolling your own environment at some level. Let's take a look at those options.

Shared Hosting

One option for hosting a Rails application is to rent resources in a shared hosting environment. A shared hosting environment means that your application is running on the same server as many other applications. The advantage of shared hosting is cost; the price is usually quite good. There's also a quick ramp-up time; you can have an application deployed and fielding requests within a few minutes of signing up for an account and configuring your domain name. There's no need for any capital investment (no writing big checks for purchasing servers) and any future hardware failures are someone else's problem.

The downsides are considerable, though. Shared hosting, as the name implies, means that the resources which are hosting the application are, well, *shared*.

The costs may be low for a seemingly large amount of resources, like disk space and bandwidth. However, once you actually start to use more than a small amount of resources, the server your application is hosted on can often encounter bottlenecks. When another user's application on the same server has a resource usage spike, your own application may respond more slowly or even lock up and stop responding, and there isn't much you can do about it. Therefore you can expect considerable variation in application performance depending on what else is happening in the vicinity of your application's server. This can make performance issues come and go randomly; a page that normally takes 100 milliseconds to load can suddenly spike to a load time of 2-3 seconds with no code changes. The support and response time is usually a bit rough since just a few operations engineers will be stretched between many applications. Finally, although hardware issues are indeed someone else's problem, that someone else will also schedule server and network downtime as needed. Those downtime windows may not match up with your plans.

That said, shared hosting does still have a place. It's a good way to get started for an application that's not too busy, and the initial costs are reasonable. If you're using shared hosting, you can skip ahead to the Capistrano-related chapters in this book.

Heroku

Heroku is a popular Rails hosting provider. It deserves a dedicated discussion simply because the Heroku team has taken great pains to ensure that deploying a Rails application to Heroku is virtually painless. Once you have an account in Heroku's system, deploying a Rails application to Heroku is as simple as pushing code to a particular Git remote. Heroku takes care of configuring and managing your entire infrastructure. For a small application or a team with no system administration expertise, Heroku is an excellent choice.

The hands-off nature of deploying on Heroku does have its drawbacks. One downside is that, as with shared hosting, the cost rapidly increases as your application's usage grows. Heroku's prices are reasonable given the excellent quality of the service, but this is nonetheless something to consider. Another downside is the lack of flexibility. Heroku has a strictly defined set of tools that your application can use; for example, your application must run atop the PostgreSQL relational database rather than using MySQL. There are other limitations, some of which can be worked around using Heroku's "add-ons", but generally speaking, by outsourcing your hosting to Heroku, you're choosing to give up some flexibility in exchange for having someone else

worry about maintaining your application environment. Finally, some businesses may have security requirements or concerns that prevent them from putting their data and source code on servers they don't own; for those use cases Heroku is out of the question.

Like shared hosting, Heroku may be the right choice for some. We choose not to focus on this deployment option in this book since the Heroku web site has a large collection of up-to-date documentation devoted to that topic.

Rolling Your Own Environment

For those who have quickly growing applications and want more control, we come to the other option. You can build out your own environment. Rolling your own application's environment means that you're the system manager. You're responsible for configuring the servers and installing the software that will be running the application. There's some flexibility in these terms; what we're addressing here is primarily the ability to do your own operating system administration. Let's look at a few of the possibilities that setting up your own environment encompasses.

In the Cloud

When creating your own environment, one option is to use a cloud-based virtual private server (VPS) service such as Amazon's Elastic Computing Cloud or the Rackspace Cloud. VPS's have rapidly become one of the most popular options for deploying web applications. With a VPS, you don't have a physical server, but you will have a dedicated virtual machine (VM) that needs to be configured. You won't have to worry about stocking spare parts, you won't have any capital expenditures, and any downtime issues will be handled by your hosting provider. On the downside, you are using a VM, so performance issues can pop up unexpectedly. Disk input/output performance is something to consider with a VM, especially with a database-heavy application. Costs may also increase dramatically as the application resource needs grow. But you do have full control over the software above the operating system layer; if you want to run the latest beta of MySQL or deploy an Erlang component, that's a decision you're free to make.

Dedicated Hosting

Getting closer to the metal, another option is to host your application on a dedicated machine with a provider like Rackspace or SoftLayer. With that choice you can be sure you've got physical hardware dedicated to your application. This option eliminates a whole class of performance irregularities. The up-front cost will be higher since you may need to sign a longer-term contract,

but you also may need fewer servers since there's no virtualization layer to slow things down. You'll also need to keep track of your server age and upgrade hardware once every few years, but you'll find that your account representative at your server provider will be happy to prompt you to do that. There's still no need to stock spare parts since your provider takes care of the servers, and your provider will also handle repair of and recovery from hardware failures such as bad disk drives, burned out power supplies, etc.

As with a cloud-based VPS solution, you'll be responsible for all system administration above the basic operation system installation and the network configuration level. You'll have more flexibility with your network setup, too, since your provider can physically group your machines on a dedicated switch or router; that usually won't be an option for a cloud VM solution.

Colocation and building your own servers

A still lower-level option is to buy physical servers from a reseller (or build them yourself from components) and host them in a either in a colocation facility or in a space that you own and operate. An example of this might be the server room in your building. This gives you much more control over hardware and networking, but will also require that your team handles hardware failures and troubleshoots networking issues. Some will find this requirement a good motivation for building better fault tolerance into their application's architecture. The capital cost of buying the servers and power control systems, and the operational cost of hiring staff for your datacenter is a consideration, and there's always the fear of over or under-provisioning your capacity. But if you need more control than hosting with a provider, this is the way to go.

A Good Mix

We feel that a combination of physical servers at a server provider and some cloud servers provides a good mix of performance and price. The physical servers give you a solid base of operations, while the cloud VMs let you bring on temporary capacity as needed.

This book is targeted for those who are rolling their own environment in some regard, whether the environment is all in the cloud or all on dedicated servers or a mix of the two. And if that's your situation, you may already be familiar with the term DevOps which we'll dive into next.

1.2 DevOps

While Ruby on Rails has swept through the web development world, a similar change is taking place in the system administration and operations world under the name "DevOps". This word was coined by Patrick Debois to describe "an emerging set of principles, methods and practices for communication, collaboration and integration between software development (application/software engineering) and IT operations (systems administration/infrastructure) professionals."[1]. In other words, software systems tend to run more smoothly when the people who are writing the code and the people who are running the servers are all working together.

This idea may seem either a trueism or an impossible dream depending on your background. In some organizations, the software is tossed over the wall from development to operations, and after a few disastrous releases the operations team learns to despise the developers, demand meticulous documentation for each release, and require the ability to roll back a release at the first sign of trouble. In other organizations, developers are expected to understand some system administration basics, and operations team members are expected to do enough programming so that a backtrace isn't complete gibberish. Finally, in the ideal DevOps world, developers and operations teams share expertise, consult each other regularly, plan new features together, and generally work side by side to keep the technical side of the company moving forward and responding to the needs of the business.

One of the primary practical outcomes of the DevOps movement is a commitment to automation. Let's see what that means for those of us who are developing and deploying Rails applications.

Automation Means Removing Busywork

Automation means that we're identifying tasks that we do manually on a regular basis and getting a computer to do those tasks instead. Rather than querying a database once a day to see how many users we have, we write a script that queries the database and sends us an email with the current user count. When we have four application servers to configure, we don't type in all the package installation commands and edit all the configuration files on each server. Instead, we write scripts so that once we've got one server perfectly configured we can apply that same configuration to the other servers in no more time than it takes for the packages to be downloaded and installed. Rather than checking a web page occasionally to see if an application is still

1. http://en.wikipedia.org/wiki/DevOps

responding, we write a script to connect to the application and send an email if the page load time exceeds some threshold.

Busywork doesn't mean that the work being done isn't complicated. That server iptables configuration may be the result of hours of skillful tuning and tweaking. It only turns into busywork when we have to type the same series of commands each time we have to roll it out to a new server. We want repeatability; we want to be able to roll a change out to dozens of servers in a few seconds with no fear of making typos or forgetting a step on each server. We'd also like to know that we did roll it out to all the servers at a particular time, and that if need be we can undo that change easily.

Automation Is A Continuum

Automation is like leveling up; we start small and try to keep improving. A shell script that uses curl to check a web site is better than hitting the site manually, and a monitoring system that sends an email when a problem occurs and another when it's fixed is better still. Having a wiki with a list of notes on how to build out a new mail server is an improvement over having no notes and trying to remember what we did last time, and having a system that can build a mail server with no manual intervention is even better. The idea is to make progress towards automation bit by bit; then we can look around after six months of steady progress and be surprised at how far we've come.

We want to automate problems as they manifest themselves. If we have one server and our user base tops out at five people, creating a hands-off server build process may not be our biggest bang for the buck. But if those five users are constantly asking for the status of their nightly batch jobs, building a dashboard that lets them see that information without our intervention may be the automation we need to add. This is the "Dev" in "DevOps"; writing code to solve problems is an option that's on the table.

So we're not saying that everything must be automated. There's a point at which the time invested in automating rarely-performed or particularly delicate tasks is no longer being repaid; as the saying goes, "why do in an hour what you can automate in six months?" But having a goal of automating as much as possible is the right mindset.

Automation - Attitude, not Tools

Notice that in this automation discussion we haven't mentioned any software package names. That's not because we're averse to using existing solutions to automate tasks, but instead that the automation itself, rather than the

particular technological means, is the focus. There are good tools and bad tools, and in later chapters we'll recommend some tools which we use and like. But any automation is better than none.

Since we're talking about attitude and emotions, let's not neglect the fact that automation can simply make us happier. There's nothing more tedious then coming to work and knowing that you have to perform a repetitive, error-prone task for hours on end. If we can take the time to turn that process into a script that gets the job in ten minutes, we can then focus on making other parts of the environment better.

That's the philosophical background. To apply all of these principles in the rest of this book, we'll need an application which we can deploy to the servers that we'll be configuring. Fortunately we have such an application; read on to learn about MassiveApp.

1.3 Learning with MassiveApp

MassiveApp, as befits its name, is a social networking application that we expect to be wildly successful just as soon as we can get the word out to all the popular blogs and startup news sites. The primary innovation that MassiveApp brings is the ability to share bookmarks with other people; we think it's really going to take off. If not, we'll add it to our resumé as a growing experience. In the meantime, we've built ourselves a fine application and we want to deploy it into a solid environment.

For the purposes of illustrating the tools and practices in this book, we're fortunate that MassiveApp is also a fairly conventional Rails 3.1 application. It uses MySQL as its data store, implements some auditing requirements, and has several tables (`accounts`, `shares`, and `bookmarks`) which we expect to grow at a great rate. It's a small application (as you can see in its Git repository[2]), but we still want to start off with the right infrastructure pieces in place.

Now that we've established some of the underlying principles of deployment and devops, let's move on and see how we can set up the first thing needed for successful deployments, our working environment. In the next chapter we'll explore two useful free utilities that will let us practice our deployments and server configuration without ever leaving the comfort of our laptop.

2. https://github.com/deployingrails/massiveapp

CHAPTER 2

Getting Started with Vagrant

In the previous chapter we discussed choosing a hosting location and decided that rolling our own environment would be a good way to go. That being the case, knowing how to construct a home for MassiveApps is now at the top of our priority list. We don't want to leap right into building out our production environment though. Instead, we'd like to practice our buildout procedure first. Also, since MassiveApp will run on several computers in production, it be handy to run it on several computers in a practice environment as well.

You may well have a couple of computers scattered in your basement, all whirring away happily. But these computers probably already have assigned roles, and rebuilding them just for the sake of running this book's exercises is probably not a good use of your time. At work, you may have access to an entire farm of servers, but perhaps you don't want to use those as a testbed; even simple experiments can be a little nerve-racking since an errant network configuration can result in getting locked out of a machine.

Another possibility is to fire up a few server instances "in the cloud". By cloud we mean any of a number of virtual machine providers, like Amazon's Elastic Computing Cloud (EC2), the Rackspace Cloud, Slicehost, Bytemark, and so forth. The downside of using these cloud providers is that you'll be charged for the time that you use. Worse yet, suppose you start up an instance and forget about it? Over the course of a week or two it could burn five, even ten dollars of your hard earned money! And we don't want to be responsible for suggesting that. There's also the need for a good network connection when working with cloud servers, and even a fast connection might not be enough to handle large file transfers. Finally, you (or your employee) may have concerns about putting your company's code and data onto servers over which you have no physical control.

Instead, we'll explore several tools, VirtualBox[1] and Vagrant[2]. These will let us run VM instances on our computer. In the words of the Vagrant web site[3], Vagrant is "a tool for building and distributing virtualized development environments." It lets you manage virtual machines using Oracle's VirtualBox and gives you the ability to experiment freely with a variety of operating systems without fear of damaging a "real" server.

Setting up these tools and configuring a few virtual machines will take a little time, so there is an initial investment here. But using these tools will allow us to set up a local copy of our production environment that we'll use to develop our system configuration and test MassiveApp's deployment. And as MassiveApp grows and we want to add more utilities, we'll have a safe and convenient environment in which to try out new systems.

2.1 Installing VirtualBox and Vagrant

First we need to install both Vagrant and Virtual Box.

Installing VirtualBox

Vagrant is primarily a driver for VirtualBox virtual machines, so VirtualBox is the first thing you'll need to find and install. VirtualBox is an open source project and is licensed under the General Public License (GPL) version 2, which means that the source code is available for your perusal and you don't have to worry too much about it disappearing. More importantly for our immediate purposes, there are installers available for Windows, Linux, Macintosh, and Solaris hosts on the VirtualBox web site.[4] We can download and run the installer to get going. Make sure you're installing at least version 4.1, as recent versions of Vagrant depend on it.

We've got VirtualBox installed. We can run VirtualBox and poke around a bit; it's an extremely useful tool in its own right. In the VirtualBox there are options to create virtual machines and configure them to use a certain amount of memory, to have a specific network configuration, to communicate with serial and USB ports, etc. But now we'll see a better way to manage our VirtualBox VMs.

1. http://virtualbox.org
2. http://vagrantup.com
3. http://vagrantup.com
4. http://www.virtualbox.org/wiki/Downloads

Installing Vagrant

The next step is to install Vagrant. It's a RubyGem, so we can do that via a standard `gem install` command:

```
$ sudo gem install vagrant
Successfully installed vagrant-0.8.10
1 gem installed
Installing ri documentation for vagrant-0.8.10...
Installing RDoc documentation for vagrant-0.8.10...
```

Vagrant has a variety of dependencies (eighteen as of this writing) so it may take a minute or two to install. Once it's in place you can verify that all's well by running the command line tool, aptly named `vagrant`, with no arguments. You should see a burst of helpful information:

```
$ vagrant
$ vagrant
Tasks:
vagrant box         # Commands to manage system boxes
vagrant destroy     # Destroy the environment, deleting the created virtual machines
vagrant halt        # Halt the running VMs in the environment
vagrant help [TASK] # Describe available tasks or one specific task
vagrant init [...]  # Initializes the current folder for Vagrant usage
vagrant package     # Package a Vagrant environment for distribution
vagrant provision   # Rerun the provisioning scripts on a running VM
vagrant reload      # Reload the environment, halting it then restarting it.
vagrant resume      # Resume a suspended Vagrant environment.
vagrant ssh         # SSH into the currently running Vagrant environment.
vagrant ssh_config  # outputs .ssh/config valid syntax for connecting to via ssh
vagrant status      # Shows the status of the current Vagrant environment.
vagrant suspend     # Suspend a running Vagrant environment.
vagrant up          # Creates the Vagrant environment
vagrant version     # Prints the Vagrant version information
```

Now that Vagrant is installed we'll use it to create a new VM on which to hone our Rails application deployment skills.

Creating a Virtual Machine with Vagrant

Vagrant has built in support for creating Ubuntu 10.4 virtual machines and we've found Ubuntu to be a solid and well-maintained distribution. Therefore we'll use that operating system for our first VM. The initial step is to add the VM definition; in Vagrant parlance, that's a *box* and we can add one with the `vagrant box add` command. Running this command will download a large file (the Ubuntu 10.4 64 bit image, for example, is 380MB), so wait to run it until you're on a fast network or be prepared for a long break:

```
$ vagrant box add lucid64 http://files.vagrantup.com/lucid64.box
```

```
[vagrant] Downloading with Vagrant::Downloaders::HTTP...
[vagrant] Copying box to temporary location...
«a nice progress bar»
```

The name "lucid64" is derived from the Ubuntu release naming scheme. "lucid" refers to Ubuntu's 10.4 release name "Lucid Lynx", and the "64" refers to the 64 bit installation rather than the 32 bit image.

Once the box is downloaded we can create our first virtual machine. We'll put everything in a new deployingrails directory, and over the course of this book we'll grow quite a farm of subdirectories as we experiment with a variety of virtual machines.

```
$ mkdir -p ~/deployingrails/first_box
```

After creating the directory we'll cd into it and run Vagrant's init command to create a new Vagrant configuration file, which is appropriately named Vagrantfile:

```
$ cd ~/deployingrails/first_box
$ vagrant init
create Vagrantfile
```

Let's open Vagrantfile in a text editor such as Textmate or perhaps Vim. Don't worry about preserving the contents; we can always generate a new file with another vagrant init. Let's change the file to contain the following bit of Ruby code:

Download introduction/Vagrantfile

```
Vagrant::Config.run do |config|
  config.vm.box = "lucid64"
end
```

That's the bare minimum to get a VM running. We'll save the file and then start our new VM with vagrant up:

```
$ vagrant up
[default] Importing base box 'lucid64'...
[default] Matching MAC address for NAT networking...
[default] Clearing any previously set forwarded ports...
[default] Forwarding ports...
[default] -- ssh: 22 => 2222 (adapter 1)
[default] Creating shared folders metadata...
[default] Running any VM customizations...
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Mounting shared folders...
[default] -- v-root: /vagrant
```

Where am I?

One of the tricky bits about using virtual machines and showing scripting examples is displaying the location where the script is running. We're going to use a shell prompt convention to make it clear where each command should be run.

If we use a plain shell prompt (for example, `$`) that will indicate that a command should be run on the host, i.e., on your laptop or desktop machine.

If a command is to be run inside a virtual machine guest, we'll use a shell prompt with the name of the VM (for example, `vm $`). In some situations we'll have more than one VM running at once; for those cases we'll use an appropriate VM guest name (for example, `nagios $` or `app $`).

Despite the warning that "This can take a few minutes", this entire process took just around a minute on my moderately powered laptop. And at the end of that startup interval, we can connect into our new VM using `ssh`:

```
$ vagrant ssh
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known hosts.
Linux vagrantup 2.6.32-24-generic-pae \
#39-Ubuntu SMP Wed Jul 28 07:39:26 UTC 2010 x86_64 GNU/Linux
Ubuntu 10.04.1 LTS

Welcome to Ubuntu!
* Documentation:  https://help.ubuntu.com/
Last login: Wed Aug 11 17:42:46 2010
vm $
```

Take a moment to poke around. You'll find that you've got a fully functional Linux host that can access the network, mount drives, stop and start services, and do all the usual things that you'd do with a Linux box.

We can manage this VM using Vagrant as well. As we saw in the `vagrant` command output earlier in this chapter, we can suspend the virtual machine, resume it, halt it, or outright destroy it. And since creating new instances is straightforward, there's no reason to keep an instance around if we're not using it.

We connected into the VM without typing a password. This seems mysterious, but remember the "Forwarding ports" message that was printed when we ran `vagrant up`? That was reporting that Vagrant had successfully forwarded port 2222 on our computer to port 22 on the virtual machine. To show that there's nothing too mysterious going on, we can bypass Vagrant's `vagrant ssh` port forwarding whizziness and simply `ssh` into the virtual machine like this (enter `vagrant` when prompted for a password):

VirtualBox Guest Additions

When firing up a new VirtualBox instance you may get a warning about the "guest additions" being out of date. The guest additions make working with a VM easier; for example, they allow you to change the guest window size. These guest additions are included with VirtualBox as an ISO image within the installation package, so to install the latest version you can connect into your VM and run these commands:

```
vm $ sudo apt-get install dkms -y
vm $ sudo /dev/cdrom/VBoxLinuxAdditions-x86.run
```

If you're running Mac OSX and you're not able to find the ISO image, look in the Applications directory and copy it over if it's there:

```
$ cp deployingrails/first_box/
$ cp /Applications/VirtualBox.app/Contents/MacOS/VBoxGuestAdditions.iso .
$ vagrant ssh
vm $ mkdir vbox
vm $ sudo mount -o loop VBoxGuestAdditions.iso vbox/
vm $ sudo vbox/VBoxLinuxAdditions.run
Verifying archive integrity... All good.
Uncompressing VirtualBox 4.1.6 Guest Additions for Linux.........
«and more output»
```

This is all the more reason to build a base box; you won't want to fiddle with this too often.

```
$ ssh -p 2222 vagrant@localhost
vagrant@localhost's password:
Linux vagrantup 2.6.32-24-generic-pae \
#39-Ubuntu SMP Wed Jul 28 07:39:26 UTC 2010 x86_64 GNU/Linux
Ubuntu 10.04.1 LTS

Welcome to Ubuntu!
* Documentation:  https://help.ubuntu.com/
Last login: Sat Jan 15 19:06:22 2011 from 10.0.2.2
vm $
```

Vagrant includes a "not so private" private key that lets `vagrant ssh` connect without a password. We can use this outside of the Vagrant context like this (depending on where and which version of the `vagrant` gem is installed):

```
$ ssh -i /Library/Ruby/Gems/1.8/gems/vagrant-0.8.10/keys/vagrant \
-p 2222 vagrant@localhost
Linux vagrantup 2.6.32-24-generic-pae \
#39-Ubuntu SMP Wed Jul 28 07:39:26 UTC 2010 x86_64 GNU/Linux
Ubuntu 10.04.1 LTS

Welcome to Ubuntu!
* Documentation:  https://help.ubuntu.com/
Last login: Sat Jan 15 19:08:33 2011 from 10.0.2.2
```

```
vm $
```

The Vagrant site refers to this as an "insecure" keypair, and since it's distributed with the Vagrant RubyGem we wouldn't recommend using it for anything else. But it does help avoid typing in a password when logging in. We'll use this forwarded port and this insecure private key extensively in our Capistrano exercises.

Now we have a way of creating VMs locally and connecting to them using using ssh. We're getting closer to being able to deploy MassiveApp, but first let's work with Vagrant so that our newly created virtual machines will have all the tools we'll need to run MassiveApp.

Building a custom base box

We used the stock Ubuntu 10.04 64-bit base box to build our first virtual machine. Going forward, though, we'll want the latest version of Ruby to be installed on our virtual machines. So now we'll take that base box, customize it by installing Ruby 1.9.2, and then build a new base box from which we'll create other VMs throughout this book. By creating our own customized base box we'll save a few steps when creating new VMs; we won't have to download and compile Ruby 1.9.2 each time. Notice that for our next VM we won't have to download the base box, so creating a new one will only take a minute or so.

Let's create a new virtual machine using the same Vagrantfile as before. First we'll destroy the old one just to make sure we're starting fresh:

```
$ cd ~/deployingrails/first_box
$ vagrant destroy
[default] Forcing shutdown of VM...
[default] Destroying VM and associated drives...
```

This time we have a basic Vagrantfile so we don't even need to run vagrant init. We'll just create a Vagrantfile with the same contents as before:

Download introduction/Vagrantfile

```
Vagrant::Config.run do |config|
  config.vm.box = "lucid64"
end
```

Now we execute vagrant up to get the VM created and running:

```
$ vagrant up
$ vagrant ssh
```

Our VM is started and we're connected to it, so let's use the Ubuntu package manager, apt-get, to fetch the latest Ubuntu packages and install some devel-

opment libraries. Having these in place will let us compile Ruby from source and install Passenger:

```
vm $ sudo apt-get update -y
vm $ sudo apt-get install zlib1g-dev libssl-dev libreadline-dev \
       git-core curl libcurl4-dev libsqlite3-dev apache2-dev -y
```

Now we'll remove the Ruby installation that comes with Ubuntu 10.4. We'll also remove the vagrant user's .gem directory since we haven't installed any gems yet and since it happens to be owned by root which would cause Bundler installation problems down the road:

```
vm $ sudo apt-get remove ruby ruby1.9.1 ruby-dev \
  ruby1.8 ruby1.8-dev libruby1.8 libruby1.9.1 -y
vm $ sudo rm -rf /home/vagrant/.gem/
```

Next we'll download and install Ruby 1.9.2 using curl. The --remote-name flag tells curl to save the file on the local machine with the same filename as the server provides:

```
vm $ curl --remote-name http://ftp.ruby-lang.org/pub/ruby/1.9/ruby-1.9.2-p290.tar.gz
```

Now we can uncompress the file, move into the expanded directory, and compile the code:

```
vm $ tar -zxf ruby-1.9.2-p290.tar.gz
vm $ cd ruby-1.9.2-p290/
vm $ ./configure
vm $ make
vm $ sudo make install
```

We can verify that Ruby is installed by running a quick version check:

```
vm $ ruby -v
ruby 1.9.2p290 (2011-07-09 revision 32553) [x86_64-linux]
```

We've set up our VM the way we want it, so let's log out and package this VM up as a new base box:

```
vm $ exit
$ vagrant package
[vagrant] Attempting graceful shutdown of linux...
[vagrant] Clearing any previously set forwarded ports...
[vagrant] Cleaning previously set shared folders...
[vagrant] Creating temporary directory for export...
[vagrant] Exporting VM...
```

This created a package.box file in our current directory. We can add that to our box list with the vagrant box add command:

```
$ vagrant box add lucid64_with_ruby192 package.box
```

```
[vagrant] Downloading with Vagrant::Downloaders::File...
[vagrant] Copying box to temporary location...
[vagrant] Extracting box...
[vagrant] Verifying box...
[vagrant] Cleaning up downloaded box...
```

We can verify that Vagrant knows about our base box with the vagrant box list command:

```
$ vagrant box list
lucid32
lucid64
lucid64_with_ruby192
```

Our new box definition is there, so we're in good shape. Let's create a new directory on our host machine and create a Vagrantfile that references our new base box:

```
$ mkdir ~/deployingrails/vagrant_testbox
$ cd ~/deployingrails/vagrant_testbox
$ cat > Vagrantfile
Vagrant::Config.run do |config|
  config.vm.box = "lucid64_with_ruby192"
end
[CTRL-D]
```

Now we'll fire up our new VM, ssh in, and verify that Ruby is there:

```
$ vagrant up
$ vagrant ssh
vm $ ruby -v
ruby 1.9.2p290 (2011-07-09 revision 32553) [x86_64-linux]
```

Looks like a success. We've taken a stock lucid64 installation, customized it with some development libraries and the latest version of Ruby, and built a Vagrant base box so that other VMs we create will have our customizations already in place.

Installing a newer Ruby version isn't the only change we can make as part of a base box. A favorite.vimrc, .inputrc, or shell profile file that always gets copied onto servers are all good candidates for building into a customized base box. Anything that prevents post-VM creation busywork is fair game.

Next we'll build another VM that uses other useful Vagrant configuration options.

2.2 Networking and More

Our previous Vagrantfile was pretty simple; we only needed one line to specify our custom base box. Let's build another VM and see what other customizations Vagrant has to offer. Here's our starter file:

Download vagrant/Vagrantfile

```
Vagrant::Config.run do |config|
  config.vm.box = "lucid64"
end
```

Setting VM Options

Before we add anything else to it, note that the Vagrantfile is written in Ruby. As with many other Ruby-based tools, Vagrant provides a simple *domain specific language* (DSL) for configuring VMs. The outermost construct is the Vagrant::Config.run method call. This method call yields a Vagrant::Config::Top object instance that we can use to set configuration information. Let's set the hostname and the memory limit; we can do this by calling config.vm.customize and passing in a block with those settings:

Download vagrant/Vagrantfile_with_options

```
config.vm.customize do |vm|
  vm.name = "app"
  vm.memory_size = 512
end
```

The config.vm.customize block yields the actual VirtualBox::VM instance that Vagrant uses when setting up the VM. The settings available in this block allow us to set any of the options that are available if we were to create a VM manually via VirtualBox's CLI or GUI. We've found that it's a good practice to always specify this block in our Vagrantfile so that we can at least set the name and the memory limit.

There are a variety of other settings that we can specify in the customize block. Most of those we probably won't need to worry about since they're low-level things such as whether the VM supports 3D acceleration; VirtualBox will detect these values on its own. Both the VirtualBox documentation[5] and the RDoc for the VirtualBox gem[6] are good sources of information on the settings for customizing a VM.

5. https://www.virtualbox.org/manual/UserManual.html
6. http://mitchellh.github.com/virtualbox/VirtualBox/VM.html

Setting up the Network

Our VM will be much more useful if the host machine can access services running on the guest. Vagrant provides the ability to both forward ports to the guest, so we can (for example) browse to localhost:8000 and have that forwarded to port 80 on the guest. We can also set up a network for the VM that makes it accessible to the host via a statically-assigned IP address. We can then assign an alias in the host's /etc/hosts file so that we can browse our VM using a domain name in a browser search bar or via ssh.

First let's add a setting, host_name, which specifies the host name that Vagrant will assign to the VM. This will also be important later for Puppet, which will check the host name when determining which parts of our configuration to run:

Download vagrant/Vagrantfile_with_options

```
config.vm.host_name = "app"
```

We want to be able to connect into the VM via ssh and we'd also like to browse into the VM once we get MassiveApp deployed there. Thus we'll add several instances of another setting, forward_port. This setting specifies a descriptive label, the port on the host to forward, and the port on the VM to forward to. The :auto option tells Vagrant that if the port isn't available it should search for another available port. Whenever we add new forward_port directives we need to restart the VM via vagrant halt and vagrant up. Let's add those directives:

Download vagrant/Vagrantfile_with_options

```
config.vm.forward_port "ssh", 22, 2222, :auto => true
config.vm.forward_port "web", 80, 4567
```

The last setting, network, specifies the IP address for the VM on the host-only network. We want to avoid using common router-assigned subnets such as 127.0.0.2 or 192.168.*, as these may clash with existing or future subnets that a router or DHCP assigns. The Vagrant documentation recommends the 33.33.* subnet, and we've found that works well. Keep in mind that when multiple Vagrant VMs are running, they will be able to talk to each other as long as they are in the same subnet. So a VM with the IP address 33.33.12.34 will be able to talk to one at 33.33.12.56 but not at 33.44.12.34. Let's add a network setting to our Vagrantfile:

Download vagrant/Vagrantfile_with_options

```
config.vm.network "33.33.13.37"
```

As with the port forwarding settings, changing the network requires a VM restart.

Sharing Folders

Vagrant provides access to the contents of a directory on the host system using the share_folder setting. share_folder specifies a descriptive label, the destination path of the directory on the VM, and the path to the source directory (relative to the Vagrantfile). We can have as many of these as we like, and can even use shared folders to deploy our code instantly between the host and the VM, although we'll still do it the old-fashioned way in Chapter 4, *Basic Capistrano*, on page 51. Let's share in the puppet directory under /etc/puppet by adding a new share_folder directive:

Download vagrant/Vagrantfile_with_options

```
config.vm.share_folder "puppet", "etc/puppet", "."
```

This greatly simplifies moving data back and forth between the VM and the host system and we'll use this feature throughout our configuration exercises.

2.3 Running Multiple VMs

Vagrant enables us to run multiple guest VMs on a single host. This is handy for all sorts of things and we'll use this feature extensively throughout our exercises. Let's give it a try now; first we need a new directory to hold our Vagrantfile:

```
$ mkdir ~/deployingrails/multiple_vms
$ cd ~/deployingrails/multiple_vms
```

Now we'll need a Vagrantfile. The syntax to define two separate configurations within our main configuration is to call thedefine method for each with a different name:

```
Vagrant::Config.run do |config|
  config.vm.define :app do |app_config|
  end
  config.vm.define :db do |db_config|
  end
end
```

We'll want to set vm.name and a memory size for each VM:

```
Vagrant::Config.run do |config|
  config.vm.define :app do |app_config|
    app_config.vm.customize do |vm|
      vm.name = "app"
      vm.memory_size = 512
    end
  end
  config.vm.define :db do |db_config|
    db_config.vm.customize do |vm|
```

```
      vm.name = "db"
      vm.memory_size = 512
    end
  end
end
```

We'll use the same box name for each VM, but we don't need to forward port 80 to the db VM, and we need to assign each VM a separate IP address. Let's add these settings to complete our Vagrantfile:

```
Vagrant::Config.run do |config|
  config.vm.define :app do |app_config|
    app_config.vm.customize do |vm|
      vm.name = "app"
      vm.memory_size = 512
    end
    app_config.vm.box = "lucid64_with_ruby192"
    app_config.vm.host_name = "app"
    app_config.vm.forward_port "ssh", 22, 2222, :auto => true
    app_config.vm.forward_port "web", 80, 4567
    app_config.vm.network "33.33.13.37"
  end
  config.vm.define :db do |db_config|
    db_config.vm.customize do |vm|
      vm.name = "db"
      vm.memory_size = 512
    end
    db_config.vm.box = "lucid64_with_ruby192"
    db_config.vm.host_name = "db"
    db_config.vm.forward_port "ssh", 22, 2222, :auto => true
    db_config.vm.network "33.33.13.38"
  end
end
```

Our VMs are defined, so we can start them both with vagrant up and we'll see output as both VMs are started. Note that the output for each VM is prefixed with the VM name:

```
$ vagrant up
[app] Importing base box 'lucid64_with_ruby192'...
[app] Preparing host only network...
[app] Matching MAC address for NAT networking...
[app] Clearing any previously set forwarded ports...
[app] Forwarding ports...
[app] -- ssh: 22 => 2222 (adapter 1)
[app] -- web: 80 => 4567 (adapter 1)
[app] Creating shared folders metadata...
[app] Running any VM customizations...
[app] Booting VM...
[app] Waiting for VM to boot. This can take a few minutes.
```

```
[app] VM booted and ready for use!
[app] Enabling host only network...
[app] Setting host name...
[app] Mounting shared folders...
[app] -- v-root: /vagrant
[db] Fixed port collision 'ssh'. Now on port 2200.
[db] Importing base box 'lucid64_with_ruby192'...
[db] Preparing host only network...
[db] Matching MAC address for NAT networking...
[db] Clearing any previously set forwarded ports...
[db] Forwarding ports...
[db] -- ssh: 22 => 2200 (adapter 1)
[db] Creating shared folders metadata...
[db] Running any VM customizations...
[db] Booting VM...
[db] Waiting for VM to boot. This can take a few minutes.
[db] VM booted and ready for use!
[db] Enabling host only network...
[db] Setting host name...
[db] Mounting shared folders...
[db] -- v-root: /vagrant
```

We can connect into each VM using our usual `vagrant ssh`, but this time we'll also specify the VM name:

```
$ vagrant ssh app
Last login: Wed Dec 21 19:47:36 2011 from 10.0.2.2
vagrant@app:~$ hostname
app
vagrant@app:~$ exit
logout
$ vagrant ssh db
Last login: Thu Dec 22 21:19:54 2011 from 10.0.2.2
vagrant@db:~$ hostname
db
```

Generally, when we want to run any Vagrant command on one VM in a multiple VM set up we need to specify the VM name. For some commands (e.g., `vagrant halt`) this is optional and we can act on both VMs by not specifying a host name.

We can connect from the `db` VM over to the `app` VM via `ssh` as the `vagrant` user with a password of `vagrant`:

```
db $ ssh 33.33.13.37
The authenticity of host '33.33.13.37 (33.33.13.37)' can't be established.
RSA key fingerprint is ed:d8:51:8c:ed:37:b3:37:2a:0f:28:1f:2f:1a:52:8a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '33.33.13.37' (RSA) to the list of known hosts.
vagrant@33.33.13.37's password:
```

```
Last login: Thu Dec 22 21:19:41 2011 from 10.0.2.2
vagrant@app:~$
```

Finally, we can shut down and destroy both VMs with vagrant destroy:

```
$ vagrant destroy
[app] Forcing shutdown of VM...
[app] Destroying VM and associated drives...
[db] Forcing shutdown of VM...
[db] Destroying VM and associated drives...
```

We're using two VMs here, but Vagrant can handle as many VMs as you need up to the resource limits of your computer. So that ten node Hadoop cluster is finally a reality.

2.4 Conclusion

In this chapter, we covered:

- Installing VirtualBox in order to create virtual machines.
- Setting up Vagrant to automate the process of creating and managing VMs with VirtualBox.
- Creating a customized Vagrant base box to make new box setup faster
- Examining a few of the features that Vagrant offers for configuring VMs.

In the next chapter we'll introduce Puppet, a system administration tool which will help us configure the virtual machines that we can now build quickly and safely. We'll learn the syntax of Puppet's DSL and get our Rails stack up and running along the way.

2.5 For Future Reference

Creating a VM

To create a new VM using Vagrant, simply run the following commands:

```
$ mkdir newdir && cd newdir $ vagrant init && vagrant up
```

Creating a custom base box

To cut down on initial box set up, create a customized base box:

```
$ mkdir newdir && cd newdir $ vagrant init && vagrant up
# Now 'vagrant ssh' and make your changes, then log out of the VM
$ vagrant package
$ vagrant box add your_new_box_name package.box
```

A Complete Vagrantfile

Here's a Vagrantfile that uses our custom base box, shares a folder, forwards an additional port, and uses a private host only network:

Download vagrant/Vagrantfile_with_options

```
Vagrant::Config.run do |config|
  config.vm.customize do |vm|
    vm.name = "app"
    vm.memory_size = 512
  end
  config.vm.box = "lucid64_with_ruby192"
  config.vm.host_name = "app"
  config.vm.forward_port "ssh", 22, 2222, :auto => true
  config.vm.forward_port "web", 80, 4567
  config.vm.network "33.33.13.37"

  config.vm.share_folder "puppet", "etc/puppet", "."
end
```

CHAPTER 3

Rails on Puppet

Now that we know how to create a virtual machine, we need to know how to configure it to run MassiveApp. We could start creating directories and installing packages manually, but that would be error-prone and we'd need to repeat all those steps next time we built out another server. Even if we put our build-out process on a Wiki or in a text file, someone else looking at that documentation has no way of knowing if it's the current configuration or even if the notes were accurate to begin with. Besides, manually typing in commands is hardly in keeping with the DevOps philosophy. We need some automation instead.

One software package that enables automated server configuration is Puppet. [1] Puppet is both a language and a set of tools that allow us to configure our servers with the packages, services, and files that they need. Commercial support is available for Puppet and there are also a variety of tutorials, conference presentations, and other learning materials available online. It's a popular tool, it's been around for a few years, and we've found it to be effective, so it's our favorite server automation utility.

3.1 Understanding Puppet

Puppet automates server buildout by formalizing a server's configuration into *manifests*. Puppet's manifests are text files which contain declarations written in Puppet's domain specific language (DSL). When we've defined our configuration using this DSL, Puppet will ensure that the machine's files, packages, and services match the items we've defined.

Note that we still have to know how to configure a server to use Puppet effectively; Puppet will just do what we tell it. So if, for example, our Puppet

1. http://puppetlabs.com

manifests don't contain the appropriate declarations to start a service on system startup, Puppet won't ensure that system is set up that way. In other words, system administration knowledge is still useful. The beauty of Puppet is that the knowledge we have about systems administration can be captured, formalized, and applied over and over. We get to continue doing the interesting parts of system administration, that is, understanding and tuning services. Meanwhile Puppet automates away the boring parts, such as typing in configuration information and manually running package installation command.

In this chapter we'll install and configure Puppet, then write Puppet manifests that we can use to build out virtual machines on which to run MassiveApp. By the end of this chapter we'll not only have a server ready for receiving a MassiveApp deploy, but we'll also have the Puppet configuration necessary to quickly build out more servers. And we'll have general Puppet knowledge for creating Puppet manifests for other tools which we'll encounter in later chapters.

3.2 Setting up Puppet

During the course of developing a server's configuration we'll accumulate not only the Puppet manifests but also other scripts we create for maintenance. We also need a place to keep documentation on more complex operations procedures. That's a bunch of files and data, and we don't want to lose any of it. So as with everything else when it comes to development, putting these things in version control is the first key step in creating a maintainable piece of our system. To start out, let's create a new Git repository that we'll use specifically for our operations scripts and Puppet configuration. You can look back at the notes in Section 4, *Tools and Online Resources*, on page v if you're not familiar with Git.

```
$ cd ~/deployingrails/
$ git init massiveapp_ops
```

We've got an empty Git repository now, but it won't be empty for long. Let's configure the virtual machine on which we'll use it. We want to use the lucid64_with_ruby192 base box which we built in *Building a custom base box*, on page 15, so we'll create a new directory:

```
$ cd ~/deployingrails/
$ mkdir puppetvm
```

And we'll use the same Vagrantfile that we built out in Section 2.2, *Networking and More*, on page 18:

Download puppetrails/Vagrantfile

```
Vagrant::Config.run do |config|
```

Puppet alternatives: Chef

Chef is a popular alternative to Puppet. There are a variety of terminology and design differences, but generally, Chef has the same primary focus as Puppet — automating the configuration of servers. We've found that Puppet has a larger community and supports configuring more service types, and we've used Puppet effectively on a variety of projects, so that's the tool we'll focus on in this book. However, all the same principles apply; automation is good, we need repeatability, etc.

```
  config.vm.customize do |vm|
    vm.name = "app"
    vm.memory_size = 512
  end
  config.vm.box = "lucid64_with_ruby192"
  config.vm.host_name = "app"
  config.vm.forward_port "ssh", 22, 2222, :auto => true
  config.vm.network "33.33.13.37"
  config.vm.share_folder "puppet", "/etc/puppet", "../massiveapp_ops"
end
```

When we start the VM and ssh into it we can see that it has access to our Git repository via the shared folder:

```
$ vagrant up
«lots of output»
$ vagrant ssh
app $ ls -l /etc/puppet/.git/
total 12
drwxr-xr-x 1 vagrant vagrant  68 2011-10-14 13:59 branches
-rw-r--r-- 1 vagrant vagrant 111 2011-10-14 13:59 config
-rw-r--r-- 1 vagrant vagrant  73 2011-10-14 13:59 description
«and more»
```

Like Vagrant, Puppet comes packaged as a gem. As of this writing, the current stable release is in the 2.7.x branch, and supports both Ruby 1.8 and 1.9. However, should you install Puppet with a system package manager, you will most likely install a release from the 0.25.x or 2.6.x branches. This is a major downside since those older versions are missing many features and bugfixes from more recent versions, and 0.25.x only supports Ruby 1.8. So we always install the Puppet gem to ensure we've got the latest code.

There are two steps to installing Puppet as a gem. First we need to create a user and group for Puppet to use. We won't need to log in as the puppet user, so we'll set the shell to /bin/false:

```
app $ sudo useradd --comment "Puppet" --no-create-home \
  --system --shell /bin/false puppet
```

Next we'll install the gem:

```
app $ sudo gem install puppet -v 2.7.9 --no-rdoc --no-ri
Successfully installed facter-1.6.4
Successfully installed puppet-2.7.9
2 gems installed
```

We prefer to specify a particular version number when installing Puppet; that way we know what we're working with. Notice that dependent gem, facter, was also installed. facter is the tool that Puppet uses to gather information about the system in order to populate variables and determine which parts of our configuration to run. Let's see some of the variables that Facter provides:

```
app $ facter
$ facter
architecture => amd64
domain => home
facterversion => 1.6.4
«...»
hostname => app
id => vagrant
interfaces => eth0,eth1,lo
ipaddress => 10.0.2.15
ipaddress_eth0 => 10.0.2.15
ipaddress_eth1 => 33.33.13.37
«...»
macaddress => 08:00:27:1a:b2:ac
macaddress_eth0 => 08:00:27:1a:b2:ac
macaddress_eth1 => 08:00:27:93:de:88
«...»
puppetversion => 2.7.9
rubyversion => 1.9.2
```

Here we can see a multitude of information about our VM, including the network interfaces, MAC addresses, hostname, and the versions of Puppet and Ruby. All the keys printed when facter runs are available as *variables* in Puppet. For now, the important thing to know is that Puppet can access all sorts of interesting information about our virtual machine.

Let's set up a minimal Puppet repository. We need a manifests subdirectory to hold our main Puppet configuration files:

```
app $ cd /etc/puppet
app $ mkdir manifests
```

And let's put a file in that directory called site.pp with the following contents. This ensures that Puppet can find the binaries for our Ruby and Rubygems installations:

Download puppetrails/initial_site.pp

```
Exec {
  path => "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
}
```

We want to commit this to our repository, but before we do a commit we'll want to set up a Git username. If we don't set this up the commits will have vagrant as the author and we'd rather have an actual name attached:

```
app $ git config --global user.name "Your Name"
app $ git config --global user.email your.name@yourdomain.com
```

Now we'll add our current Puppet configuration to our Git repository:

```
app $ git add .
app $ git ci -m "Initial Puppet manifests"
```

Like Vagrant, Puppet has a command line application which is appropriately named puppet. We can run this application with a help option to see the available commands:

```
$ puppet help
Usage: puppet <subcommand> [options] <action> [options]

Available subcommands, from Puppet Faces:
  ca                Local Puppet Certificate Authority management.
  catalog           Compile, save, view, and convert catalogs.
«...»
Available applications, soon to be ported to Faces:
  agent             The puppet agent daemon
  apply             Apply Puppet manifests locally
  cert              Manage certificates and requests
  describe          Display help about resource types
«...»
```

We're ready to run Puppet for the first time. We're going to run puppet apply, which won't apply any actual changes since we haven't defined any. However, Puppet uses a number of configuration directories to store system information, security certificates and other internal files it needs to run, and we'll use sudo so that Puppet will have permissions to create those directories:

```
app $ sudo puppet apply --verbose manifests/site.pp
```

When developing a Puppet configuration it's a good idea to pass the --verbose flag to Puppet so that it displays any extended information that might be available if an error happens. We'll see output like this from puppet apply:

```
info: Applying configuration version '1319427535'
info: Creating state file /var/lib/puppet/state/state.yaml
notice: Finished catalog run in 0.02 seconds
```

We've successfully built a tiny Puppet repository and executed Puppet on our VM. Now we'll build out the VM with a standard Rails application stack. That is, we'll configure Apache, MySQL, Passenger, and put the Rails application directory tree into place. Along the way we'll learn all about how Puppet helps us manage our VM's services and configuration.

3.3 Installing Apache with Puppet

A key part of MassiveApp's infrastructure is Apache, and we'll start our MassiveApp buildout by defining a Puppet manifest that installs and configures Apache. As a side benefit this will lead us through a variety of Puppet language and usage details.

Building an Initial Apache Puppet Manifest

We have this initial set of files in /etc/puppet, with our site.pp containing only a single Exec declaration which places various directories on Puppet's execution path:

```
- /manifests
  |- /site.pp
```

We could easily install Apache with Ubuntu's package manager (e.g., sudo apt-get install apache2 -y) but we want to write the corresponding Puppet directives to install it instead. To get Puppet to install Apache we need to define a *resource* to tells Puppet about what needs to be installed. In this case, Puppet calls this resource type a *package*, so we need to use a package declaration. Let's put this into site.pp:

Download puppetrails/site.pp

```
package {
  "apache2":
    ensure => present
}
```

This package declaration is a fine example of the Puppet DSL. There's a type declaration, package, followed by a block delimited by curly braces. That block contains a *title* for the instance of the type that we're configuring; here that's apache2. That title further contains various *parameters* that affect how Puppet will act on the type. In this case, the package type titled apache2 has an ensure parameter that's set to present. This declaration is our way to tell Puppet that it should ensure there is a package called "apache2" installed on the system.

That's a short summary of the package resource type, but we can get many more details about this resource type via the puppet describe command:

```
$ puppet describe package
```

```
package
=======
Manage packages.  There is a basic dichotomy in package
support right now:  Some package types (e.g., yum and apt) can
retrieve their own package files, while others (e.g., rpm and sun)
cannot.  For those package formats that cannot retrieve their own files,
«and many more details»
```

We've been using the word "package", but what exactly does that mean? In Ubuntu, the standard package manager is dpkg, and the package format is the .deb file. If our VM were running a Redhat-based operating system, the package manager would be RPM and the package format would be the .rpm file. Through the use of generic types (like package), Puppet provides an abstraction layer over these operating system-level differences. In other words, we can write a single Puppet configuration file and use it to provision servers running multiple operating systems.

With that background, let's get the apache2 package in place. To do this we'll run the Puppet client:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1315534357'
notice: /Stage[main]//Package[apache2]/ensure: ensure changed 'purged' to 'present'
notice: Finished catalog run in 10.78 seconds
```

Now if we run dpkg we can see that apache2 is installed:

```
app $ dpkg --list apache2
«some table headers»
ii  apache2   2.2.14-5ubuntu8.7  Apache HTTP Server metapackage
```

If we rerun Puppet, the apache2 package won't be reinstalled. Instead, Puppet will notice that it's already in place and consider that package declaration fulfilled:

```
$ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1325010497'
notice: Finished catalog run in 0.03 seconds
```

This is the Puppet lifecycle. We figure out what resources we need on our server, we build manifests which describe those resources, and we run the puppet client to put those resources in place.

Apache is installed and started, and the default Ubuntu package install scripts will ensure that it starts on boot, but what if someone carelessly modifies the system so that Apache isn't set to start automatically? We can guard against that possibility with another Puppet type, service, which helps us manage any

kind of background process that we want to stay running. Let's add this to site.pp:

Download puppetrails/site.pp

```
service {
  "apache2":
    ensure => true,
    enable => true
}
```

Like the package declaration, this service resource declaration has a title of apache2 and an ensure parameter. Rather than setting ensure to present, though, we're setting it to true. For the service type, setting ensure to true tells Puppet to attempt to start the service if it's not running. We've also added an enable parameter with a value of true that ensure that this service will start on boot. Thus each time we run the Puppet client, Puppet will ensure that Apache is both running and set to start on boot. And as with package, we can get many more details on service by running puppet describe service.

We've seen the package and service Puppet types so far. We'll use a variety of other types over the course of this book, but for a full listing go to the Puppet type reference documentation[2]. A quick scan through the types enumerated there will give you a feel for the depth of Puppet's support: cron jobs, files and directories, mailing lists, users, and more are in Puppet's built-in repertoire.

Managing a Configuration File with Puppet

Next let's modify one of the Apache configuration files and see how to manage that using Puppet. Under certain error conditions Apache will display the server administrator's email address to visitors. The value of this email address is controlled by the ServerAdmin setting. The default Apache configuration sets that address to webmaster@localhost, but let's change that to vagrant@localhost. To do this, let's open up /etc/apache2/apache2.conf and add this line to the end of the file:

```
ServerAdmin vagrant@localhost
```

So now we've customized our Apache configuration. But what if we build out another server? We'll need to make the same change to Apache on that server. And although we might remember this change, if we have a dozen changes to six different services' configuration files, there's no way we'll remember them all. That's where Puppet comes in. We'll capture this configuration

2. http://docs.puppetlabs.com/references/stable/type.html

change, formalize it into our Puppet manifests, and Puppet will manage that for us. To get started, we'll add a file resource to site.pp:

Download puppetrails/site.pp

```
file {
  "/etc/apache2/apache2.conf":
    source => "puppet:///modules/apache2/apache2.conf",
    mode   => 644,
    owner  => root,
    group  => root
}
```

The format of the file resource is something we've seen before; we've got a type name containing a title and a set of parameters. In the case of the file resource, the title is the full path to the file. The parameters are where we're seeing some things for the first time. There are several permissions-related parameters; owner and group tell Puppet who should own the file, and mode contains the standard Unix file system permissions. The source parameter is a little more involved. That parameter tells Puppet that it should look in /etc/puppet/modules/apache2/files/ for the file which should be compared to the existing file located at /etc/apache2/apache2.conf. Since we're telling Puppet that there's a modules directory containing the authoritative file, we need to create that directory and copy the default apache2.conf there:

```
app $ mkdir -p modules/apache2/files/
app $ cp /etc/apache2/apache2.conf modules/apache2/files/
```

Now as an experiment, let's open /etc/apache2/apache2.conf and remove the directive we just added (i.e., the line starting with ServerAdmin). We can see that Puppet knows that the file has been changed by running Puppet with the --noop option; this flag tells Puppet to not actually make any changes. Instead we'll see a helpful diff of the changes and what would have happened if we had run Puppet without the --noop option:

```
app $ sudo puppet apply --verbose --noop manifests/site.pp
info: Applying configuration version '1319427848'
--- /etc/apache2/apache2.conf   2011-10-23 20:44:06.492314742 -0700
+++ /tmp/puppet-file20111023-3337-122xno1       2011-10-23 20:44:08.623293237 -0700
@@ -235,3 +235,4 @@
 # Include the virtual host configurations:
 Include /etc/apache2/sites-enabled/

+ServerName vagrant@localhost
notice: /Stage[main]//File[/etc/apache2/apache2.conf]/content: \
  current_value {md5}c835e0388e5b689bfdf96f2.7.9b25c8, \
  should be {md5}f2088e3a492feace180633822d4f4cb5 (noop)
notice: /etc/apache2/apache2.conf: Would have triggered 'refresh' from 1 events
notice: Class[Main]: Would have triggered 'refresh' from 1 events
```

```
notice: Stage[main]: Would have triggered 'refresh' from 1 events
notice: Finished catalog run in 0.13 seconds
```

We've verified that Puppet will make the change we want. To make that change happen, we'll run Puppet without the `--noop` flag:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1315801207'
info: FileBucket adding {md5}86aaee758e9f530be2ba5cc7a3a7d147
info: /Stage[main]//File[/etc/apache2/apache2.conf]:
  Filebucketed /etc/apache2/apache2.conf to puppet
  with sum 86aaee758e9f530be2ba5cc7a3a7d147
notice: /Stage[main]//File[/etc/apache2/apache2.conf]/content:
  content changed '{md5}86aaee758e9f530be2ba5cc7a3a7d147'
  to '{md5}89c3d8a72d0b760adc1c83f05dce88a3'
notice: Finished catalog run in 0.14 seconds
```

As we expected, Puppet overwrote the target file's contents with our changes. Let's commit our current Puppet repository contents to Git so that our Apache configuration file will have an audit trail:

```
app $ git add manifests/ modules/
app $ git commit -m "Building out Apache resource declarations"
[master (root-commit) e8b82e8] Building out Apache resource declarations
2 files changed, 257 insertions(+), 0 deletions(-)
create mode 100644 manifests/site.pp
create mode 100644 modules/apache2/files/apache2.conf
```

Now we can install and configure a service using Puppet, and we know that each time we run Puppet it will enforce our configuration file changes. This is already a vast improvement over manually installing and configuring packages, but we've got more improvements to make. Next we'll look at making our Puppet repository cleaner and easier to understand.

Organizing Manifests with Modules

So far we've seen how to create Puppet manifests to manage packages, services, and configuration files. We've used these to set up Apache, which is one of the the basic components on which MassiveApp will run.

But adding more MassiveApp components for Puppet to manage will become hard to manage in a single manifest; our `site.pp` is getting pretty cluttered already. It'd be better for us to structure our Puppet manifests so that we can find things easily and so that there's a clear separation of the Puppet configuration for each part of MassiveApp's infrastructure.

Puppet's answer to this problem is to structure the manifests using *modules*. Modules allow us to organize our manifests into logical pieces, much as we

do with modules in Ruby or packages in Java. We've already gotten started on a module structure for our Apache-related Puppet configurations by creating a modules/apache2/files directory holding our customized apache2.conf. To get further down this road, let's create a modules/apache2/manifests directory and copy the existing site.pp file into a init.pp file there:

```
app $ mkdir modules/apache2/manifests
app $ cp manifests/site.pp modules/apache2/manifests/init.pp
```

Our new Apache module's init.pp needs to declare a Puppet *class* as a container for the module's resources. Declaring a class is as simple as deleting the Exec declaration and wrapping the remaining contents of init.pp in a class declaration:

Download puppetrails/apache.init.pp

```
class apache2 {
  package {
    "apache2":
      ensure => present
  }

  service {
    "apache2":
      ensure => true,
      enable => true,
  }

  file {
    "/etc/apache2/apache2.conf":
      owner  => root,
      group  => root,
      mode   => 644,
      source => "puppet:///modules/apache2/apache2.conf"
  }
}
```

We can do some cleanup now by deleting everything except the Exec block from manifests/site.pp. With that done, we've got a functioning Puppet repository again. Let's make the same sort of change as we did before to /etc/apache2/apache2.conf (delete the new ServerAdmin line) and then rerun Puppet to verify that everything's working:

```
$ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1325012538'
notice: Finished catalog run in 0.02 seconds
```

Hm, not much happened. That's because we need to tell Puppet to include our Apache module in the manifests that will be applied when we run the Puppet client. To do that, we *include* the module in our manifests/site.pp:

Download puppetrails/site.pp.include

```
Exec {
  path => "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
}

include apache2
```

Now we can run Puppet and see our changes applied:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1315996920'
info: FileBucket adding {md5}81703bb9023de2287d490f01f7deb3d1
info: /Stage[main]/Apache/File[/etc/apache2/apache2.conf]:
  Filebucketed /etc/apache2/apache2.conf to puppet
  with sum 81703bb9023de2287d490f01f7deb3d1
notice: /Stage[main]/Apache/File[/etc/apache2/apache2.conf]/content:
  content changed '{md5}81703bb9023de2287d490f01f7deb3d1'
  to '{md5}89c3d8a72d0b760adc1c83f05dce88a3'
notice: Finished catalog run in 0.12 seconds
```

Our module is working; Puppet has overwritten the file and we're back to our desired configuration.

We've seen Puppet modules in action now, so let's pause and review a few things about using them:

- Modules are named after the service or resource which they provide.
- A module lives in Puppet's modules directory
- A module needs a modulename/manifests/init.pp file to organize its resource declarations
- Our manifests/site.pp needs to include a module for Puppet to activate it

Our directory structure with one module in place looks like this:

```
- /manifests
  |- /site.pp
- /modules
  |- /apache2
     |- /files
        | - /apache2.conf
     |- /manifests
        | - /init.pp
```

We'll see a lot more of modules in this and other chapters; they're a Puppet mainstay. Now let's continue our MassiveApp server buildout by adding more resources to our Apache module.

Restarting Apache on Configuration Changes

Our Puppet manifests install Apache and ensure the initial configuration is in place. We'll be making changes to the Apache configuration, though, and we'll need to restart Apache when that happens. The Puppet way to do this is to use the "package-service-file" configuration pattern. That is, we want Puppet to ensure that the Apache package is installed, then ensure that the configuration file is in place, and then ensure that the service is running with that file.

To enforce these dependencies we'll use some new Puppet parameters in our existing resource declarations. First let's enhance our Apache package resource declaration by adding a before parameter that ensures the package is installed before our custom configuration file is copied in:

Download puppetrails/apache_package_file_service.pp

```
package {
  "apache2":
    ensure => present,
    before => File["/etc/apache2/apache2.conf"]
}
```

The before parameter declares a Puppet *dependency*; without this dependency Puppet can execute the resources in any sequence it sees fit.

We've got one dependency in place. We really have two dependencies, though, since we don't want to start the Apache service until our customized configuration file is in place, and we want to restart the service any time the configuration file changes. So let's use another Puppet service parameter, subscribe, so that the Puppet notifies the service when configuration changes happen:

Download puppetrails/apache_package_file_service.pp

```
service {
  "apache2":
    ensure    => true,
    enable    => true,
    subscribe => File["/etc/apache2/apache2.conf"]
}
```

Now if we shut down Apache, remove the ServerAdmin directive that we put in /etc/apache2/apache2.conf, and rerun Puppet, the configuration file will be updated before the service is started:

```
app $ sudo puppet apply --verbose manifests/site.pp
```

```
info: Applying configuration version '1316316205'
info: FileBucket got a duplicate file {md5}81703bb9023de2287d490f01f7deb3d1
info: /Stage[main]/Apache/File[/etc/apache2/apache2.conf]:
  Filebucketed /etc/apache2/apache2.conf to puppet
  with sum 81703bb9023de2287d490f01f7deb3d1
notice: /Stage[main]/Apache/File[/etc/apache2/apache2.conf]/content:
  content changed '{md5}81703bb9023de2287d490f01f7deb3d1'
  to '{md5}89c3d8a72d0b760adc1c83f05dce88a3'
info: /etc/apache2/apache2.conf: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Apache/Service[apache2]/ensure:
  ensure changed 'stopped' to 'running'
notice: /Stage[main]/Apache/Service[apache2]: Triggered 'refresh' from 1 events
notice: Finished catalog run in 1.24 seconds
```

The before and subscribe parameters can be used with any resource type; these are therefore known as Puppet "metaparameters". We'll mention various other metaparameters as we continue to build out MassiveApp's Rails stack, and there's excellent documentation on all the metaparameters on the Puppet site[3]. We can also see a good overview of subscribe and other metaparameters using puppet describe with a --meta flag:

```
$ puppet describe --meta service
service
=======
Manage running services.  Service support unfortunately varies
widely by platform --- some platforms have very little if any concept of a
«and much more»
```

We'll also need a virtual host for Apache to serve up MassiveApp. We'll add another file to our files directory that contains the virtual host definition:

Download puppetrails/massiveapp.conf

```
<VirtualHost *:80>
  ServerName massiveapp
  DocumentRoot "/var/massiveapp/current/public/"
  CustomLog /var/log/apache2/massiveapp-access_log combined
  ErrorLog /var/log/apache2/massiveapp-error_log
</VirtualHost>
```

And we'll manage this file with Puppet by adding another file resource to modules/apache2/manifests/init.pp. Like the file resource for apache2.conf, this resource is also dependent on the Apache service resource and notifies Apache when it changes.

Download puppetrails/massiveapp.pp

```
"/etc/apache2/conf.d/massiveapp.conf":
  source  => "puppet:///modules/apache2/massiveapp.conf",
```

3. http://docs.puppetlabs.com/references/stable/metaparameter.html

```
  owner   => root,
  group   => root,
  notify  => Service["apache2"],
  require => Package["apache2"];
```

Another Puppet run and we've got our Apache configuration in place. Before moving on let's get all our changes into our Git repository:

```
app $ git add .
app $ git commit -m "More Apache"
[master 7b96dd8] More Apache
 3 files changed, 261 insertions(+), 0 deletions(-)
 «and more»
```

Next we'll continue building out our Rails stack by installing MySQL.

3.4 Configuring MySQL with Puppet

Now that we've configured one service with Puppet, configuring another is much easier. As far as Puppet is concerned, MySQL and Apache are more or less the same resource but with different package, file, and service names.

Let's build out a new module directory for MySQL. First we'll create the appropriate directories:

```
app $ mkdir -p modules/mysql/files modules/mysql/manifests
```

Now we have a place to put our init.pp, and we'll populate that file with the same package-file-service pattern as we did with our Apache module:

Download puppetrails/mysql.init.packagefileservice.pp

```
class mysql {

  package {
    "mysql-server":
      ensure => installed,
      before => File["/etc/mysql/my.cnf"]
  }

  file {
    "/etc/mysql/my.cnf":
      owner  => root,
      group  => root,
      mode   => 644,
      source => "puppet:///modules/mysql/my.cnf"
  }

  service {
    "mysql":
      ensure    => running,
      subscribe => File["/etc/mysql/my.cnf"]
```

```
  }

}
```

In addition to our standard package-file-service resources we'll also add two exec resources. These will set the MySQL root password to root once the service is started and then create the massiveapp_production database. Each exec needs an unless variable since we don't want to reset the password and attempt to create the database each time we run Puppet:

Download puppetrails/mysql.init.exec.pp

```
exec {
  "mysql_password":
    unless  => "mysqladmin -uroot -p status",
    command => "mysqladmin -uroot password root",
    require => Service[mysql];
  "massiveapp_db":
    unless  => "mysql -uroot -proot massiveapp_production",
    command => "mysql -uroot -proot -e 'create database ${massiveapp_production}'",
    require => Exec["mysql_password"]
}
```

Even though we don't have any configuration changes to make to my.cnf at this point, we'll add the default MySQL configuration file contents to modules/mysql/files/my.cnf so that we're ready for any future additions. Also, to get Puppet to load up our module, we'll add it to our manifests/init.pp:

```
include apache
include mysql
```

Now when we run Puppet it installs the MySQL package (or ensures that it's installed), overwrites the default configuration file with our Puppet-controlled file, and starts the MySQL service if it's not already running. As usual, let's commit our changes before moving on:

```
app $ git add .
app $ git commit -m "Support for MySQL"
```

Next we'll use Puppet to set up a few directories where we'll deploy MassiveApp.

3.5 Creating the MassiveApp Rails Directory Tree

We've got Apache and MySQL in place and it's time to set up the directory tree to which we'll deploy MassiveApp. Let's start by creating a new module, massiveapp, with the usual directories:

```
app $ mkdir -p modules/massiveapp/files modules/massiveapp/manifests
```

Bootstrapping a package

There's a bootstrapping problem here. That is, what does our desired configuration file look like, and how do we get that configuration file without installing the package with Puppet? What we usually do is install the package manually, grab the configuration file, and put it in our Puppet Git repository. In the case of MySQL on Ubuntu, that's `sudo apt-get install mysql-server -y && cp /etc/mysql/my.cnf modules/mysql/files/`. Then we can run Puppet normally, and when we decide we want to change the configuration file we've got our dependencies in place.

This may seem a little like cheating; we're installing a package manually and then backfilling it with Puppet. But it's realistic, too. In the real world, we usually don't roll out new services without making some configuration changes to fit our hardware and our usage needs. Installing the package and then using the service's configuration file as a starting place is a good way to get a new service started, and it allows us to record our configuration file customizations in our Git commit history.

To deploy MassiveApp we need a directory that's owned by the user who's running the deployments, so our `init.pp` needs a `file` resource that creates the Rails subdirectories. We won't create the `current` symlink since Capistrano will take care of that when we actually deploy MassiveApp, but we'll make a root directory and a directory in which to store configuration files. We're also using a new syntax here; we're listing several directory titles in our type declaration. This prevents resource declaration duplication and results in Puppet applying the same parameters to all the directories. Here's our `init.pp` with our `file` resource declaration:

Download puppetrails/massiveapp.justdirectories.init.pp

```
class massiveapp {
  file {
    ["/var/massiveapp/",
     "/var/massiveapp/shared/",
     "/var/massiveapp/shared/config/"]:
      ensure => directory,
      owner  => vagrant,
      group  => vagrant,
      mode   => 775
  }
}
```

As usual, we'll include this module in our `site.pp`:

```
include massiveapp
```

When we run Puppet we get the expected directories:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1316435273'
```

```
notice: /Stage[main]/Massiveapp/File[/var/massiveapp/]/ensure: created
notice: /Stage[main]/Massiveapp/File[/var/massiveapp/shared/]/ensure: created
notice: /Stage[main]/Massiveapp/File[/var/massiveapp/shared/config/]/ensure: created
notice: Finished catalog run in 0.04 seconds
```

MassiveApp can't connect to MySQL without a database.yml file. We like to store our database.yml files on the server (and in Puppet) so that we don't need to keep our production database credentials in our application's Git repository. Storing it on the server also lets us change our database connection information (port, password, etc) without needing to redeploy MassiveApp.

So let's add a database.yml file to the config directory. This is a file resource, but it's not like the earlier directory declarations so we can't just combine it with our first resource declaration. Let's declare a new file resource:

Download puppetrails/massiveapp.init.pp

```
file {
  "/var/massiveapp/shared/config/database.yml":
    ensure  => present,
    owner   => vagrant,
    group   => vagrant,
    mode    => 600,
    source  => "puppet:///modules/massiveapp/database.yml"
}
```

We need to put our database.yml in our modules/massiveapp/files/ directory since that's where we've set up Puppet to expect it:

Download puppetrails/database.yml

```
production:
  adapter: mysql
  encoding: utf8
  reconnect: true
  database: massiveapp_production
  username: root
  password: root
```

Now when we run Puppet it copies our database.yml into place:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1316517863'
notice:
  /Stage[main]/Massiveapp/File[/var/massiveapp/shared/config/database.yml]/ensure:
 defined content as '{md5}05754b51236f528cd82d55526c1e53c7'
notice: Finished catalog run in 0.07 seconds
```

We'll also need Bundler to be installed so that it's ready to install MassiveApp's gems. Let's add a package declaration; this one uses the built in Puppet provider for Rubygems:

Download puppetrails/massiveapp.init.pp

```
package {
  "bundler":
    provider => gem
}
```

When we run Puppet it installs the gem as we'd expect:

```
$ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1325588796'
notice: /Stage[main]/Massiveapp/Package[bundler]/ensure: created
notice: Finished catalog run in 2.27 seconds
```

With all those changes in place we'll add this module to our Git repository.

We've written Puppet manifests for our MassiveApp directory structure and our database connection configuration file. Let's set up Puppet manifests for Passenger and we'll be ready to deploy MassiveApp.

3.6 Passenger

Passenger[4] is a Rails application server that makes it easy to run Rails applications under Apache. We like Passenger because it's reasonably easy to install, and once in place it manages a pool of application instances so that we don't have to manage ports and process IDs. It even handles restarting an application instance after a certain number of requests.

Setting up Passenger consists of installing the Passenger RubyGem, building and installing the Passenger shared object library that will be loaded into Apache, and including several directives in our Apache configuration. Let's start by creating a new Puppet module for Passenger:

```
app $ mkdir -p modules/passenger/files modules/passenger/manifests
```

For our module's init.pp we'll introduce another resource type: the Exec resource. This resource type lets us run arbitrary commands if certain conditions aren't met. For example, we'll install the Passenger gem only if it's not already installed. Here's our Exec resource in our passenger module's init.pp. We'll specify conditions using a Puppet unless parameter that checks to see if the gem is already installed by looking for the gem directory, and we don't want to install Passenger before Apache is in place:

Download puppetrails/passenger.init.pp.gem

```
class passenger {
  exec {
    "/usr/local/bin/gem install passenger -v=3.0.9":
```

4. http://modrails.com/

```
      user    => root,
      group   => root,
      alias   => "install_passenger",
      require => Package["apache2"],
      unless  => "ls /usr/local/lib/ruby/gems/1.9.1/gems/passenger-3.0.9/"
  }
}
```

We can compile the Passenger Apache module with another Exec resource, and we'll add a before parameter to our initial exec resource to reference that one. Now our init.pp looks like this:

Download puppetrails/passenger.init.pp.gemandmodule

```
class passenger {
  exec {
    "/usr/local/bin/gem install passenger -v=3.0.9":
      user    => root,
      group   => root,
      alias   => "install_passenger",
      before  => Exec["passenger_apache_module"],
      unless  => "ls /usr/local/lib/ruby/gems/1.9.1/gems/passenger-3.0.9/"
  }

  exec {
    "/usr/local/bin/passenger-install-apache2-module --auto":
      user    => root,
      group   => root,
      path    => "/bin:/usr/bin:/usr/local/apache2/bin/",
      alias   => "passenger_apache_module",
      unless  => "ls /usr/local/lib/ruby/gems/1.9.1/gems/\
passenger-3.0.9/ext/apache2/mod_passenger.so"
  }
}
```

Passenger has a number of configuration settings that are declared as Apache directives. Here's our configuration file; we'll put this in our Passenger module's file directory (modules/passenger/files/passenger.conf).

Download puppetrails/passenger.conf

```
LoadModule passenger_module \
  /usr/local/lib/ruby/gems/1.9.1/gems/passenger-3.0.9/ext/apache2/mod_passenger.so
PassengerRoot /usr/local/lib/ruby/gems/1.9.1/gems/passenger-3.0.9
PassengerRuby /usr/local/bin/ruby
PassengerUseGlobalQueue on
PassengerMaxPoolSize 5
PassengerPoolIdleTime 900
PassengerMaxRequests 10000
```

These Passenger settings are documented in great detail on the Passenger web site. A few highlights, though:

- PassengerUseGlobalQueue <true or false> enables Passenger to hold pending HTTP requests in a queue and wait until an application instance is available. This prevents one or two slow requests from tying up subsequent requests.
- PassengerMaxPoolSize <n> helps to cap Passenger resource usage. If a server is getting too many requests, Passenger will reject the request rather than spinning up application instances until it runs out of memory. This helps the server recover more quickly.
- PassengerPoolIdleTime <n> tells Passenger to shut down unused application instances after the indicated interval. When the site gets busy, Passenger will start more application instances to serve the traffic, and PassengerPoolIdleTime will let Passenger shut down the instances after things slow down and the instances are idle for a while.
- PassengerMaxRequests <n> provides memory leak protection. With this setting in place, Passenger will restart an application instance after it has served the indicated number of requests. This continual recycling of instances helps to keep memory leaks from becoming an issue. If this is set too low, Passenger will spend unneeded resources restarting application instances, so it's good to monitor an application's memory usage over time to help set this value correctly.

Let's add a file resource that copies those Passenger directives into place, and we'll complete the package-file-service patter by adding another before parameter to our first exec resource. Now we have this init.pp:

Download puppetrails/passenger_init.pp

```
class passenger {

  exec {
    "/usr/local/bin/gem install passenger -v=3.0.9":
      user    => root,
      group   => root,
      alias   => "install_passenger",
      before  => [File["passenger_conf"],Exec["passenger_apache_module"]],
      unless  => "ls /usr/local/lib/ruby/gems/1.9.1/gems/passenger-3.0.9/"
  }

exec {
  "/usr/local/bin/passenger-install-apache2-module --auto":
    user    => root,
    group   => root,
    path    => "/bin:/usr/bin:/usr/local/apache2/bin/",
    alias   => "passenger_apache_module",
    before  => File["passenger_conf"],
    unless  => "ls /usr/local/lib/ruby/gems/1.9.1/gems/\
```

```
passenger-3.0.9/ext/apache2/mod_passenger.so"
}

  file {
    "/etc/apache2/conf.d/passenger.conf":
      mode    => 644,
      owner   => root,
      group   => root,
      alias   => "passenger_conf",
      notify  => Service["apache2"],
      source  => "puppet:///modules/passenger/passenger.conf"
  }
}
```

As usual, we'll also add an include to our manifests/site.pp to get Puppet to use our new module:

```
include passenger
```

Now we can run Puppet and get the Passenger gem and module into place:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1319451008'
notice: /Stage[main]/Passenger/Exec[/usr/local/bin/\
  passenger-install-apache2-module --auto]/returns: executed successfully
info: FileBucket adding {md5}b72cb12ea075df2ca1ae66e824fa083b
info: /Stage[main]/Passenger/File[/etc/apache2/conf.d/passenger.conf]: \
  Filebucketed /etc/apache2/conf.d/passenger.conf to puppet \
  with sum b72cb12ea075df2ca1ae66e824fa083b
notice: /Stage[main]/Passenger/File[/etc/apache2/conf.d/passenger.conf]/content: \
  content changed '{md5}b72cb12ea075df2ca1ae66e824fa083b' \
  to '{md5}47757e49ad1ea230630a935d41459b08'
info: /etc/apache2/conf.d/passenger.conf: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Apache2/Service[apache2]/ensure: \
  ensure changed 'stopped' to 'running'
notice: /Stage[main]/Apache2/Service[apache2]: Triggered 'refresh' from 1 events
notice: Finished catalog run in 35.59 seconds
```

We'll add these scripts to our ever-expanding Git repository.

We've installed Passenger and explored some of the Passenger configuration settings. We've also learned about a few more Puppet capabilities, including the Exec resource type and the unless parameter.

3.7 Managing Multiple Hosts with Puppet

Our manifests/site.pp is getting to be a little messy. We've got a mix of two different things in there; we've got an Exec resource declaration which applies globally and we've got module include directives for our MassiveApp server. And what

if we wanted to manage another server from the same Puppet repository; we'd need some place to put that.

The answer is to create another file and use that to contain our host resource assignments which are called *nodes*. A node has a name and a set of directives, although we usually try to restrict those to module inclusion. We'll need a new manifest/nodes.pp file in which we'll define a node:

Download puppetrails/nodes.pp

```
node "app" {
  include apache2
  include mysql
  include massiveapp
  include passenger
}
```

The node name, which must be unique, is the same as the host name of the server. With our nodes.pp file we can add as many hosts as we like and each can include different modules. We can also trim down our site.pp considerably now:

Download puppetrails/site.withoutnodes.pp

```
Exec {
    path => "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
}

import "nodes.pp"
```

Nodes provide a convenient way to separate hosts. We use them on all but the smallest Puppet installations.

3.8 Updating the Base Box

Clearly Puppet is going to be useful for the rest of this book; we'll always want to install it to manage everything else. That being the case, let's update our Vagrant base box to include a basic Puppet installation. This will save us a few steps every time we build out a new VM.

Since we're enhancing our base box, we need to fire up a new VM that's built from that base box. So we'll create a new working directory:

```
$ mkdir ~/deployingrails/updatedvm/ && cd ~/deployingrails/updatedvm/
```

We need a simple Vagrantfile:

```
Vagrant::Config.run do |config|
  config.vm.box = "lucid64_with_ruby192"
end
```

Now we can start the VM and SSH in:

```
$ vagrant up
«lots of output»
$ vagrant ssh
vm $
```

We need a Puppet user and the Puppet gem, so let's set up both of those:

```
vm $ sudo useradd --comment "Puppet" --no-create-home \
--system --shell /bin/false puppet
vm $ sudo gem install puppet -v 2.7.9 --no-rdoc --no-ri
```

That's all the enhancements we're making, so we'll exit, package up the VM, and replace our old base box with our new Puppet-capable box:

```
vm $ exit
$ vagrant package
«lots of output»
$ vagrant box remove lucid64_with_ruby192
[vagrant] Deleting box 'lucid64_with_ruby192'...
$ vagrant box add lucid64_with_ruby192 package.box
[vagrant] Downloading with Vagrant::Downloaders::File...
[vagrant] Copying box to temporary location...
«lots of output»
```

A quick test is in order, so let's clean up our working directory, recreate it, and fire up a new VM with a minimal Vagrantfile:

```
$ vagrant destroy
$ cd ../
$ rm -rf updatedvm && mkdir updatedvm
$ cat > Vagrantfile
Vagrant::Config.run do |config|
  config.vm.box = "lucid64_with_ruby192"
end
[CTRL-D]
$ vagrant up
«lots of output»
$ vagrant ssh
vm $
```

Finally, the test; is Puppet installed?

```
$ gem list

*** LOCAL GEMS ***

facter (1.6.4)
minitest (1.6.0)
puppet (2.7.9)
rake (0.8.7)
rdoc (2.5.8)
```

That what we want to see. We've updated our base box to include Puppet and now we'll save a few steps on future VMs since Puppet will be ready to go.

3.9 Conclusion

In this chapter we've gotten an introduction to Puppet, the system configuration utility. We've seen how it automates and formalizes the installation and configuration of various system components, and we've built a basic set of Puppet manifests to prepare a VM for the deployment of MassiveApp.

In the next chapter we'll look at deploying MassiveApp onto our newly configured VM using the popular Ruby deployment tool Capistrano.

3.10 For Future Reference

Puppet Organization

Puppet repositories are organized by creating a collection of modules. Each module can be included in the site.pp with an include directive, i.e., include apache. Each module contains at least one manifest in the manifests and, optionally, some files in the files directory. Manifests contain class declarations which contain resource declarations which are composed of types and parameters.

Running Puppet

You can apply Puppet manifest to a system using:

```
$ sudo puppet apply --verbose manifests/site.pp
```

Or, to do a test run, use --noop:

```
$ sudo puppet apply --verbose --noop manifests/site.pp
```

Sample Apache Module

Here's a sample Apache module that ensure Apache is installed, has a custom configuration file in place, and starts on boot:

Download puppetrails/apache_package_file_service.pp

```
class apache2 {
  package {
    "apache2":
      ensure => present,
      before => File["/etc/apache2/apache2.conf"]
  }

  file {
    "/etc/apache2/apache2.conf":
      owner   => root,
```

```
      group   => root,
      mode    => 644,
      source  => "puppet:///modules/apache2/apache2.conf"
  }

  service {
    "apache2":
      ensure    => true,
      enable    => true,
      subscribe => File["/etc/apache2/apache2.conf"]
  }
}
```

Passenger Settings

Here are some basic Passenger settings; you'll want some variation on these in your Apache configuration:

```
PassengerUseGlobalQueue on
PassengerMaxPoolSize 5
PassengerPoolIdleTime 900
PassengerMaxRequests 10000
```

CHAPTER 4

Basic Capistrano

Developing a world-class web application isn't going to do much good if we aren't able to find a way for the world to see it. For other web frameworks, this can often be a tedious and time-consuming process. Application code needs to be updated, database schemas need to be migrated, application servers must be restarted. We could do all of this manually by SSH'ing into the server, pulling the latest code from the repository, running database migration tasks, and restarting the application server. Even better, we could write some shell scripts that would automatically perform these tasks for us. But why do all that when we could be spending time improving MassiveApp instead? The good news is that we don't have to.

Battle tested and used for deploying many of today's Rails apps, Capistrano was originally authored by Jamis Buck, is now ably maintained by Lee Hambley, and is the tool that Rails developers have used for years to get their applications from development to production on a daily basis. It was built from the start for deploying Rails applications, and has all the tools we need to get MassiveApp deployed and to help keep deployments running smoothly afterwards. Capistrano's library of tasks, plugins, and utilities have also evolved over the years with the rest of the Rails ecosystem, and thus Capistrano continues to meet the deployment needs of Rails applications both new and old.

In the last chapter we set up a VM for running MassiveApp app. Now we'll use Capistrano to deploy MassiveApp from our workstation to the VM we created. Capistrano's responsibility is to prepare the server for deployment and then deploy the application as updates are made. It does this by executing *tasks* as Unix shell commands over an SSH connection to the server. Tasks are written in a Ruby DSL, so we can leverage the full power of Ruby and the libraries and gems it has available when writing our deployment scripts.

Let's see what it takes to get Capistrano up and running for MassiveApp, and get MassiveApp deployed to our VM for the first time.

4.1 Setting up Capistrano

The first thing we'll do is install Capistrano on our machine. We'll install Capistrano to the machine we're deploying *from*, not the server we're deploying *to*. All of the things that Capistrano does are invoked as shell commands via an SSH connection so Capistrano doesn't need anything special on the server to do its work aside from a user account.

When Capistrano connects to the server, it can use an SSH key pair to authenticate the connection attempt. For that to work, we need to have an SSH private key loaded in our keyring that is authorized with a public key on the server for the user Capistrano tries to connect as. Otherwise, if there is no key pair being used, or the key fails to authenticate, Capistrano will surface a password prompt every time it runs a command. Having to enter a password multiple times during a deployment is annoying at best, and at worst will prevent us from deploying handsfree when it comes to continuous deployment practices. So, to avoid those password prompts, we're going to have Capistrano connect to our VM via the `vagrant` user account. That account already has a public key in place so we'll be able to connect right in.

Since we're going to be installing MassiveApp, let's clone it from the Git repository:

```
$ cd ~/deployingrails
$ git clone git@github.com:deployingrails/massiveapp.git
Cloning into 'massiveapp'...
remote: Counting objects: 288, done.
remote: Compressing objects: 100% (213/213), done.
remote: Total 288 (delta 106), reused 213 (delta 53)
Receiving objects: 100% (288/288), 99.66 KiB, done.
Resolving deltas: 100% (106/106), done.
```

The `master` branch of the repository has all the Capistrano files in place, but we'll build them from scratch here. So let's switch over to a branch that will let us do that:

```
$ git checkout before_capistrano
Branch before_capistrano set up to track\
remote branch before_capistrano from origin.
Switched to a new branch 'before_capistrano'
```

Like most Ruby libraries and applications, Capistrano comes as a gem. We could install Capistrano using `gem install capistrano`, but since we're using Bundler

for MassiveApp, let's ensure Capistrano always gets installed by adding it to our Gemfile:

```
group :development do
  gem 'capistrano'
end
```

We add the capistrano gem to the development group since it's not one of MassiveApp's runtime dependencies, thus there's no need to have it bundled with the application when we deploy. Now we'll go ahead and get Capistrano and its dependencies in place with Bundler:

```
$ bundle
«...»
Installing net-ssh (2.2.1)
Installing net-scp (1.0.4)
Installing net-sftp (2.0.5)
Installing net-ssh-gateway (1.1.0)
Installing capistrano (2.9.0)
«...»
Your bundle is complete! Use `bundle show [gemname]`
 to see where a bundled gem is installed.
```

To ensure that Capistrano is installed, and also get a handy list of Capistrano's options, let's run cap -h:

```
$ $ cap -h
Usage: cap [options] action ...
    -d, --debug           Prompts before each remote command execution.
    -e, --explain TASK    Displays help (if available) for the task.
    -F, --default-config  Always use default config, even with -f.
    -f, --file FILE       A recipe file to load. May be given more than once.
    -H, --long-help       Explain these options and environment variables.
«and many more options»
```

We can also get more detailed help information with cap -H, for example:

```
$ cap -H
------------------------------
Capistrano
------------------------------

Capistrano is a utility for automating the execution of commands across multiple
remote machines. It was originally conceived as an aid to deploy Ruby on Rails
web applications, but has since evolved to become a much more general-purpose
tool.
«and much more »
```

Now we'll set MassiveApp up with Capistrano. The Capistrano deploy process needs some supporting files, and happily Capistrano can generate some example files for us when we run the capify command:

```
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

The capify command created two files for us. The first file, Capfile, lives in our project's root directory, and takes care of loading Capistrano's default tasks, any tasks that might be found in plugins (which Capistrano calls *recipes*), and our primary deployment script in config/deploy.rb. Here's the content of Capfile; just three lines:

Download capistrano/Capfile

```
load 'deploy' if respond_to?(:namespace) # cap2 differentiator
Dir['vendor/plugins/*/recipes/*.rb'].each { |plugin| load(plugin) }
load 'config/deploy' # remove this line to skip loading any of the default tasks
```

The code in Capfile calls the Ruby Kernel#load() method, but that means something different here. It's not just loading up Ruby source code from files. For its own purposes, Capistrano overrides Ruby's default behavior to load its configuration files as well as ones that you specify. Capistrano's redefined load() can also be used to load arbitrary strings containing Ruby code (e.g., load(:string => "set :branch, 'release-42'")), and even blocks to be evaluated.

The other file that was generated is config/deploy.rb; this contains our application's deployment script. It contains the settings for our application, as well as custom tasks that we'll define. Next, we'll take a look at the settings contained in deploy.rb and learn how to configure them for deploying MassiveApp to our VM.

4.2 Making it Work

The capify command adds a few comments and default settings to config/deploy.rb. We're going to change almost everything, though, and we can learn more about Capistrano if we just start with a blank slate. So let's open config/deploy.rb in an editor and delete all the contents.

Now that we've got an empty config/deploy.rb we can build out the Capistrano deployment configuration for MassiveApp. MassiveApp is a Rails 3 application, so we need to include the Bundler Capistrano tasks; the line below takes care of that.

Download capistrano/deploy.rb

```
require 'bundler/capistrano'
```

Next we set a Capistrano *variable*. Capistrano variables are much like Ruby variables, although they're a bit different in that setting a variable makes it available to other parts of the deployment script. To enable this we'll need to use the set() method to assign then a value. Let's go ahead and add a setting for the application variable. This is mostly a convenience variable; we'll use it by plugging it into various other parts of the script so that we'll have directory names that match the application name. We'll usually set this to something "Unixy" (i.e., lowercase, no spaces, no odd characters) like massiveapp.

Download capistrano/deploy.rb

```
set :application, "massiveapp"
```

Deploying MassiveApp means getting the code from the source code repository onto a server, so we need to tell Capistrano where that code lives. Let's add the scm setting with a value of :git. This setting can be a symbol or a string (e.g., "git"), but we find symbols more readable. This is followed by the repository. For MassiveApp this is the local directory since that's where our Git repository currently resides.

Download capistrano/deploy.rb

```
set :scm, :git
set :repository, "git://github.com/deployingrails/massiveapp.git"
```

We're deploying to one server, so we need to tell Capistrano where it is. Capistrano has the concept of *roles* which define what purpose a particular host serves in an environment. For example, some servers may be application servers, some may be database servers, and some may be catch-all utility boxes. The VM we've set up will act as both an application and database server, so we'll configure it with both those roles by setting the server variable to those role names:

Download capistrano/deploy.rb

```
server "localhost", :app, :db, :primary => true
```

Capistrano needs to connect via SSH to the server to which it's deploying code, but there's a wrinkle here since we're using Vagrant. When we set up the VM with Vagrant, we saw some output when we ran our first vagrant up. Here's the relevant part of that output:

```
[default] Forwarding ports...
[default] -- ssh: 22 => 2222 (adapter 1)
```

So the default Vagrant configuration we're using has forwarded port 2222 on our computer to the standard ssh port 22 on the VM. We'll need to tell Capistrano about this, so in the next line we tell Capistrano to connect to the port that Vagrant is forwarding. This is followed by the path to the ssh private key

that comes with Vagrant (this location may well be different on your computer). This will save us from having to type in the virtual machine's password, which, as you recall from *Creating a Virtual Machine with Vagrant*, on page 11, is vagrant.

Download capistrano/deploy.rb

```
ssh_options[:port] = 2222
ssh_options[:keys] = "/Library/Ruby/Gems/1.8/gems/vagrant-0.8.10/keys/vagrant"
```

We need to deploy as an existing user and group, so the next few lines contain those settings. We'll also add a deploy_to setting that tells Capistrano where MassiveApp will live on the virtual machine. Also, since we're deploying to a directory that the vagrant user already owns, we won't need to use sudo, so we'll turn that off.

Download capistrano/deploy.rb

```
set :user, "vagrant"
set :group, "vagrant"
set :deploy_to, "/var/massiveapp"
set :use_sudo, false
```

Since we're going to deploy code, we need a way to move the code from the Git repository to the VM. These next few lines use what Capistrano calls the *copy strategy* to move the code up to the VM. The copy strategy consists of cloning the Git repository to a temporary directory on the local machine and, since we've also set copy_strategy to export, removing the .git directory. The copy strategy then compresses the source code and copies the compressed file to the VM where it is uncompressed into the deployment target directory. This is the most straightforward way of getting the code onto the VM since the code export is done locally; we don't need to depend on the VM being able to connect to the remote Git repository or even having Git installed:

Download capistrano/deploy.rb

```
set :deploy_via, :copy
set :copy_strategy, :export
```

So far we've seen Capistrano variable settings; now it's time to add some tasks. When we deploy new code, we need to tell Passenger to load in our changes. We'll do this with a few task declarations. Notice that this series of tasks starts with a namespace() declaration. This allows us to group the Passenger-related tasks logically, and it lets us declare other start and stop tasks to manage other services without those declarations clashing with the Passenger task names. The stop and start tasks are empty since Passenger will serve up MassiveApp as soon as the code is in place, but the restart task contains a single command that signals Passenger to restart MassiveApp.

Download capistrano/deploy.rb

```
namespace :deploy do
  task :start do ; end
  task :stop do ; end
  desc "Restart the application"
  task :restart, :roles => :app, :except => { :no_release => true } do
    run "#{try_sudo} touch #{File.join(current_path,'tmp','restart.txt')}"
  end
end
```

We'll look at tasks more later, but that wraps up our initial tour of a minimal config/deploy.rb. Now we'll actually use it!

4.3 Setting up the Deploy

Now that the deployment configuration is in place we'll need a VM on which to deploy it. From the previous chapter, let's create a Vagrantfile that has a few basic settings:

Download capistrano/vagrantfile

```
Vagrant::Config.run do |config|
  config.vm.customize do |vm|
    vm.name = "app"
    vm.memory_size = 512
  end
  config.vm.box = "lucid64_with_ruby192"
  config.vm.host_name = "app"
  config.vm.forward_port "ssh", 22, 2222, :auto => true
  config.vm.forward_port "web", 80, 4567
  config.vm.network "33.33.13.37"
  config.vm.share_folder "puppet", "/etc/puppet", "../massiveapp_ops"
end
```

Let's fire up that VM and connect into it:

```
$ vagrant up
«...»
[default] Mounting shared folders...
[default] -- v-root: /vagrant
[default] -- puppet: etc/puppet
«...»
$ vagrant ssh
«...»
app $
```

And now we'll set it up as a MassiveApp server using the Puppet manifests that we built in the last chapter:

```
app $ cd /etc/puppet
app $ sudo puppet apply --verbose manifests/site.pp
```

With that VM in place we're getting closer to a code deploy. We set deploy_to to /var/massiveapp, but that directory isn't quite ready yet. In order for Capistrano to deploy to a server, it needs to have a subdirectory structure in place. This directory structure will be used for the application code releases and for assets that are shared by the application between releases. Let's use Capistrano to create these directories for us by running the deploy:setup task:

```
$ cap deploy:setup
  * executing `deploy:setup'
  * executing "mkdir -p /var/massiveapp /var/massiveapp/releases
  /var/massiveapp/shared /var/massiveapp/shared/system
  /var/massiveapp/shared/log /var/massiveapp/shared/pids &&
  chmod g+w /var/massiveapp /var/massiveapp/releases
  /var/massiveapp/shared /var/massiveapp/shared/system
  /var/massiveapp/shared/log /var/massiveapp/shared/pids"
    servers: ["localhost"]
    [localhost] executing command
    command finished in 4ms
```

Running the deploy:setup task created the following directory structure in the /var/massiveapp directory:

```
- /releases
- /shared
  |- /log
  |- /system
  |- /pids
```

Each time we deploy, Capistrano will create a new subdirectory in the releases and place the source code for MassiveApp there. The shared directory will contain various, well, shared resources. Thus, shared/log will contain MassiveApp's log files, and shared/system will contain any files that need to be kept from deployment to deployment. We also have shared/config which we created with Puppet and which contains MassiveApp's database.yml.

We've built out a basic Capistrano configuration file, we've used Capistrano to create some directories on our VM, and we're getting close to deploying some actual code. That's what's coming up next.

4.4 Pushing a Release

Now that we've got our config/deploy.rb file ready to go and the initial directory structure in place, let's get MassiveApp out on the virtual machine. The task to run is deploy:cold; it's named cold because we're deploying the application for the first time. And remember when we included a require 'bundler/capistrano' in config/deploy.rb? With that in place, the deploy:cold task will run bundle:install to get

our gems into place. It's an after_deploy *lifecycle hook* on deploy:update_code, so it'll be run on subsequent deploys as well.

We'll unpack the deploy flow in this section, but Figure 1, *The deploy:code task flow*, on page 60 is an overview of what happens. The top-level task is deploy:cold and that triggers various other dependent tasks.

Deploying with Capistrano produces a torrent of output, but it's worth reading through it at least once to get familiar with what Capistrano is doing. First, deploy:cold runs a dependent task, deploy:update, which in turn runs deploy:update_code. This task exports MassiveApp's code from our Git repository by cloning the repository and deleting the .git directory:

Download capistrano/deploy_cold_output.txt

```
 * executing `deploy:cold'
 * executing `deploy:update'
** transaction: start
 * executing `deploy:update_code'
   executing locally: "git ls-remote
   git://github.com/deployingrails/massiveapp.git HEAD"
   command finished in 386ms
 * getting (via export) revision
   34c7b2318ea6fc75649d14fd478e309bcc42d710 to
   /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834
   executing locally:
   git clone -q git://github.com/deployingrails/massiveapp.git
   /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834
   && cd /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834
   && git checkout -q -b deploy 34c7b2318ea6fc75649d14fd478e309bcc42d710
   && rm -Rf /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834/.git
   command finished in 200ms
   compressing /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834
   to /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834.tar.gz
```

Next the source code gets packaged, compressed, and moved (using sftp) up to the VM. The filename is based on a timestamp with granularity down to the second. Each releases subdirectory name is based on this timestamp, so this allows us to deploy frequently without directory conflicts:

Download capistrano/deploy_cold_output.txt

```
   executing locally: tar chzf 20120112042834.tar.gz 20120112042834
   command finished in 20ms
   servers: ["localhost"]
** sftp upload
   /var/folders/dE/dEW2lQWVGMeQ5tBgIlc5l++++TU/-Tmp-/20120112042834.tar.gz
   -> /tmp/20120112042834.tar.gz
   [localhost] /tmp/20120112042834.tar.gz
   [localhost] done
 * sftp upload complete
```

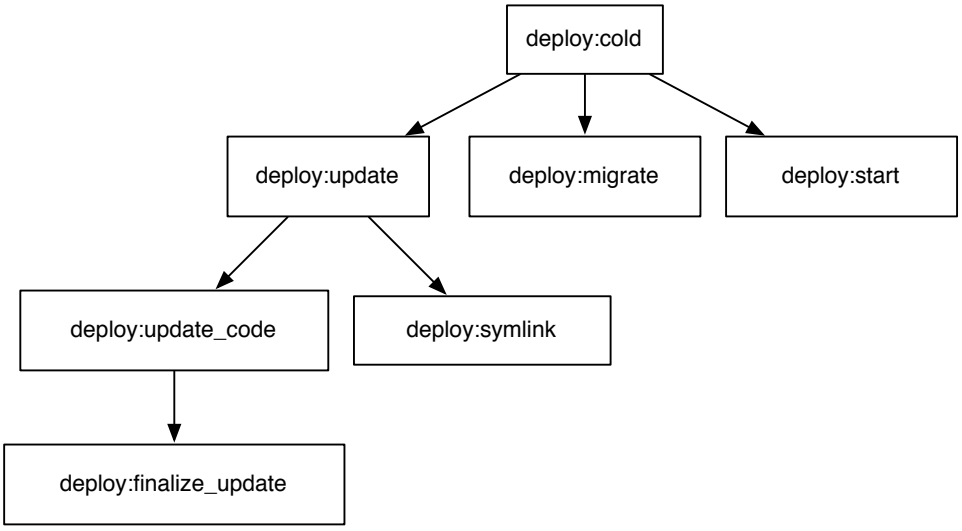


Figure 1—The deploy:code task flow

Next Capistrano uncompresses the code into the releases directory which we created when we ran deploy:setup. After the code is uncompressed Capistrano deletes the tar file. At this point the application code is in the correct location to be served up:

Download capistrano/deploy_cold_output.txt

```
* executing "cd /var/massiveapp/releases
  && tar xzf /tmp/20120112042834.tar.gz && rm /tmp/20120112042834.tar.gz"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 13ms
```

Once that's done, the deploy:finalize_update task sets permissions and creates several symbolic links:

Download capistrano/deploy_cold_output.txt

```
* executing `deploy:finalize_update'
* executing "chmod -R g+w /var/massiveapp/releases/20120112042834"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 5ms
* executing "rm -rf /var/massiveapp/releases/20120112042834/log
  /var/massiveapp/releases/20120112042834/public/system
  /var/massiveapp/releases/20120112042834/tmp/pids
  &&\\\n      mkdir -p /var/massiveapp/releases/20120112042834/public
  &&\\\n      mkdir -p /var/massiveapp/releases/20120112042834/tmp
  &&\\\n      ln -s /var/massiveapp/shared/log
  /var/massiveapp/releases/20120112042834/log
```

```
  &&\\\n      ln -s /var/massiveapp/shared/system
  /var/massiveapp/releases/20120112042834/public/system
  &&\\\n      ln -s /var/massiveapp/shared/pids
  /var/massiveapp/releases/20120112042834/tmp/pids"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 9ms
```

During this step the public/system directory gets symlinked to shared/system. This means that anything in shared/system will be available through MassiveApp's document root. So that's not a good place to put sensitive data, like an environment-specific database.yml file. Not that either of your authors has made that mistake, or, perhaps, will ever make that mistake again.

Next, the task runs touch on static files in various directories. This sets the modification time on those files to be consistent so that Rails' asset timestamping works properly:

Download capistrano/deploy_cold_output.txt

```
* executing "find /var/massiveapp/releases/20120112042834/public/images
  /var/massiveapp/releases/20120112042834/public/stylesheets
  /var/massiveapp/releases/20120112042834/public/javascripts
  -exec touch -t 201201120428.35 {} ';'; true"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 18ms
  triggering after callbacks for `deploy:finalize_update'
* executing `bundle:install'
* executing "ls -x /var/massiveapp/releases"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 6ms
```

We're getting close now. Here's the output of the Bundler command bundle that we discussed earlier; this installs MassiveApp's gems on the VM:

Download capistrano/deploy_cold_output.txt

```
* executing "cd /var/massiveapp/releases/20120112042834
  && bundle install --gemfile /var/massiveapp/releases/20120112042834/Gemfile
  --path /var/massiveapp/shared/bundle
  --deployment --quiet --without development test"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 357ms
```

Then comes the magic moment when the current symbolic link is created. Since the Apache DocumentRoot points to current/public/, this makes the application available:

Download capistrano/deploy_cold_output.txt

```
* executing `deploy:symlink'
* executing "rm -f /var/massiveapp/current
  && ln -s /var/massiveapp/releases/20120112042834 /var/massiveapp/current"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 6ms
```

We defined an after_symlink hook, so Capistrano runs that hook now and copies in database.yml:

Download capistrano/deploy_cold_output.txt

```
  triggering after callbacks for `deploy:symlink'
* executing "cp /var/massiveapp/shared/config/database.yml
  /var/massiveapp/current/config/"
  servers: ["localhost"]
  [localhost] executing command
  command finished in 5ms
```

Since Puppet just created the massiveapp_production database it's empty. So Capistrano runs all the migrations to get our tables into place::

Download capistrano/deploy_cold_output.txt

```
 * executing `deploy:migrate'
 * executing "cd /var/massiveapp/releases/20120112042834
   && bundle exec rake RAILS_ENV=production  db:migrate"
   servers: ["localhost"]
   [localhost] executing command
** [out :: localhost] ==  CreateAccounts: migrating ============
** [out :: localhost] -- create_table(:accounts)
** [out :: localhost] -> 0.0021s
** [out :: localhost] ==  CreateAccounts: migrated (0.0022s) ===
** [out :: localhost]
** [out :: localhost] ==  CreateVestalVersions: migrating ======
** [out :: localhost] -- create_table(:versions)
** [out :: localhost] -> 0.0027s
** [out :: localhost] -- change_table(:versions)
** [out :: localhost] -> 0.0214s
** [out :: localhost] ==  CreateVestalVersions: migrated (0.0241s) =
** [out :: localhost]
** [out :: localhost] ==  CreateBookmarks: migrating ===========
** [out :: localhost] -- create_table(:bookmarks)
** [out :: localhost] -> 0.0023s
** [out :: localhost] ==  CreateBookmarks: migrated (0.0023s) ==
** [out :: localhost]
** [out :: localhost] ==  CreateShares: migrating ==============
** [out :: localhost] -- create_table(:shares)
** [out :: localhost] -> 0.0025s
** [out :: localhost] -- add_index(:shares, :bookmark_id)
** [out :: localhost] -> 0.0032s
** [out :: localhost] -- add_index(:shares, :shared_by)
```

```
** [out :: localhost] -> 0.0036s
** [out :: localhost] -- add_index(:shares, :shared_with)
** [out :: localhost] -> 0.0047s
** [out :: localhost] ==  CreateShares: migrated (0.0141s) =====
** [out :: localhost]
   command finished in 2000ms
```

At last Capistrano calls deploy:start, which we've previously defined to do nothing:

Download capistrano/deploy_cold_output.txt

```
* executing `deploy:start'
```

Done! MassiveApp is deployed to the virtual machine. If you open your browser to localhost:4567 you'll see MassiveApp in action.

Now we've deployed MassiveApp and thoroughly explored the Capistrano deployment process. Next we'll look more closely at some Capistrano concepts that we've only glanced at thus far: tasks, hooks, and roles.

4.5 A Closer Look

Getting Capistrano to deploy MassiveApp was a success. Now we'll look more closely at some areas of the deployment script. We'll dive a little deeper into Capistrano's use of tasks, hooks, and roles.

Capistrano Tasks

We saw a short example of a Capistrano task at the end of config/deploy.rb where we defined a series of tasks to manage MassiveApp's restarts. Here's that section of the file:

Download capistrano/deploy.rb

```
namespace :deploy do
  task :start do ; end
  task :stop do ; end
  desc "Restart the application"
  task :restart, :roles => :app, :except => { :no_release => true } do
    run "#{try_sudo} touch #{File.join(current_path,'tmp','restart.txt')}"
  end
end
```

Working from the top down, we first see a namespace() declaration. A namespace can be any symbol and lets you group tasks by functionality area. For example, the thinking_sphinx RubyGem contains a number of tasks all declared within the thinking_sphinx namespace. On that same line we see that the namespace() method accepts a block. This block can contain more namespace declarations as well as task declarations. We can declare a namespace several times in our script

if we want to add tasks to that namespace at different points. In that respect, redeclaring a namespace is much like reopening a Ruby class.

Inside the passenger namespace you see the actual tasks: stop, start, and restart. stop and start are both "no-ops", that is, they pass in an empty block and thus do nothing. The restart task, on the other hand, has some code and a few interesting features:

- It's prefixed by a desc() method call with a String parameter. With this method call in place, a user running cap -T will see the task listed along with its description. Without a description in place, a user would need to remember to run cap -vT, which displays tasks that don't have descriptions. Who's going to remember that extra flag? Thus, we'll define a description for all but the most trivial tasks.
- There's a :roles => :app parameter. This indicates that this task will only be run on servers that are listed in the :app role. You can see why this makes sense here; there's no need to attempt to restart Passenger unless a server is running MassiveApp. For example, we might have a utility server that needs the MassiveApp code deployed for cron jobs but isn't serving requests over HTTP. There's no need to touch tmp/restart.txt on that server.
- There's also a :except => { :no_release => true } clause. This prevents this task from being executed on servers that are not part of the release process. For example, we may have a server running a message queue that is not affected by releases, but we still may want to execute certain tasks on that server. In that case, we would mark that server as being release-free (e.g., role :message_queue, "my.mq.server", :no_release => true), and deploy:restart wouldn't affect it.
- Finally, there's the body of the task. This contains a call to the run() method, which executes the given String as a shell script on the appropriate servers. Notice that the command contains a reference to the current_path() method. That's the way to interpolate values into the commands that we want to execute on the remote servers.

The Capistrano wiki [1] contains a good deal of information on the various commands a task can call, and that set of commands will vary based on the Capistrano plugins and extensions that are installed. We'll see many more examples of tasks in this book, though, and we'll call out the interesting bits.

1. https://github.com/capistrano/capistrano/wiki

Capistrano Hooks

Most Rails developers are used to seeing ActiveRecord's callback methods. For example, the callback method before_save() provides a way for an ActiveRecord model to execute some code just prior to being saved. These callback methods are sometimes referred to as "hooks" since they allow model code to "hook" into ActiveRecord's object lifecycle management.

Capistrano has a similar feature called a deployment *hook*. This is a piece of code that runs before or after a particular task. You've seen a hint at one of these already; the first few lines of config/deploy.rb back in Section 4.4, *Pushing a Release*, on page 58 contained a require statement that brought in the Bundler Capistrano tasks. To be accurate, it brought in the Bundler Capistrano *task*; as of this writing there's only one. That single task is a deploy:update_code *after* hook that causes the bundle:install task to be run after the deploy:update_code task is completed. It makes sense to run bundle:install at that point since by then the Gemfile is in place. Given that there's an "after" hook, you won't be surprised to hear that *before hooks* also exist for the purpose of, well, running code before a task.

Hooks are as useful with Capistrano as they are with ActiveRecord. For example, if we're feeling cautious we might want our deployment script to prompt for confirmation before deploying the code. To do this, we could register a before_deploy hook for a task that does just that. Here's the code; let's place this in a new file lib/deploy/confirm_deploy.rb:

Download capistrano/confirm_deploy.rb

```
namespace :deploy do
  desc "Make sure the user really wants to deploy"
  task :confirm do
    if Capistrano::CLI.ui.ask("Are you sure you want to deploy?") == "yes"
      puts "OK, here goes!"
    else
      puts "Exiting"
      exit
    end
  end
end
```

In this example the task uses Capistrano's Capistrano::CLI utility to prompt the user for input and exit if the user doesn't type yes. To get Capistrano to see this task, though, we need to modify the Capfile to load it. So let's open the Capfile and add one more line so that Capistrano will load everything in the lib/deploy directory. Now the Capfile looks like this:

Download capistrano/cap2/Capfile
```
load 'deploy' if respond_to?(:namespace) # cap2 differentiator
Dir['vendor/plugins/*/recipes/*.rb'].each { |plugin| load(plugin) }
load 'config/deploy' # remove this line to skip loading any of the default tasks
Dir.glob("lib/deploy/*.rb").each {|f| load f }
```

Now we'll declare a hook to call this task from our config/deploy.rb. We'll add this as the last line of that file:

```
before :deploy, "deploy:confirm"
```

Now we can see this task in action. Let's run cap deploy task and wait for the prompt:

```
$ cap deploy
  * executing `deploy'
    triggering before callbacks for `deploy'
  * executing `deploy:confirm'
Are you sure you want to deploy?
no
Exiting
```

This task definition and the hook consisted of only a few lines of code, but, they needed to go somewhere. We could have put it right into config/deploy.rb, and for such a short bit of code that wouldn't be a terrible solution. But these sorts of things build up over time, and to keep our config/deploy.rb from growing, we prefer to put them in the lib/deploy directory. Then we can give the file an intention-revealing name like confirm_deploy.rb, and someone browsing that directory will be able to get a quick idea of what's in there. We put the task definition in that file and put the actual hook (i.e., before :deploy, "deploy:confirm") into the config/deploy.rb file. That way all the logic is contained in config/deploy.rb and the lib/deploy directory acts as a set of libraries that don't do anything unless explicitly invoked.

The downside of putting things in a separate directory like this is that now all our tasks aren't defined in one place. But there are enough useful plugins and utilities in the Capistrano ecosystem that we've found that we're loading Capistrano code from other directories anyhow. We've found that the organizational benefits outweigh the initial loss of visibility, and putting all the hooks in config/deploy.rb makes a big difference.

Capistrano roles

Back in code, on page 55 we saw a server() declaration that configured localhost to act as both the application and the database for MassiveApp. We can use Capistrano's role feature whenever we want a task to run on one set of servers but not on another. One way to write this is:

```
server "servername", :some_role_name, :another_role_name
```

This is a nice shortcut we only have one host to work with. And an alternate way which we can use when we have a variety of servers in different roles:

```
role :some_role_name, "servername"
role :another_role_name, "servername"
```

Once we declare a role we can use it in task declarations. For example, suppose we wanted the deploy:confirm task to run only on servers configured in the app role. We could use this declaration to accomplish that:

```
task :confirm, :roles => :app do
```

We can also use environment variables to take advantage of role declarations. For example, to make a workers:refresh task run only on servers that are configured with the worker role we could do this:

```
$ ROLES=worker cap workers:refresh
```

The role name can be any valid Ruby symbol. The built-in Capistrano tasks center around three predefined roles:

- app is for tasks that run on application servers. deploy:cold, deploy:update_code, and deploy:restart are all tasks that run on servers in this role.
- db is for tasks that run on database servers. An example of this is deploy:migrate.
- web is for tasks that run on web server, that is, servers dedicated to serving static content. deploy:web:enable and deploy:web:disable are the only standard tasks that use this role.

We've written a Capistrano script and deployed MassiveApp to our virtual machine. Better still, we've got a basic understanding of the elements that make up a Capistrano deployment script. Next we'll see some pointers on keeping your application running.

CHAPTER 5

Advanced Capistrano

Now that we've configured Capistrano to do the heavy lifting when it comes to deploying MassiveApp, we can sit back and enjoy the show, right? Well, as you can probably guess, not quite. Deployment, like any other area of software development, benefits from constantly iterating and improving the code and configuration. When refactoring our Capistrano configuration, there are a number of facilities Capistrano provides that will aid us in both making deployments faster and maintaining MassiveApp once it's deployed. In this chapter, we'll take a look at some of these advanced features, techniques, and best practices with Capistrano to round out MassiveApp's deployment process.

Not all these Capistrano techniques will be useful in every deployment scenario. For example, we'll discuss multistage deployments, which are only needed when deploying an application to more than one environment. But we'll discuss the situations in which each feature would come in handy, and these are all good tools to have at the ready.

5.1 Faster Symlinks

Sometimes we'll see variables defined in block form; here's an example:

```
set(:deploy_to) { "/var/massiveapp/deploy/#{name}" }
```

Blocks are used in Capistrano for variables whose values must be lazy-loaded. They become especially useful when you must interpolate the value of another Capistrano variable, and as a rule of thumb should always be used whenever a reference to another Capistrano variable is made.

Let's look at a more complex use of block variables and speed up our deploys at the same time. As part of the default deployment process, Capistrano symlinks certain directories in the `#{deploy_to}/shared` directory after updating

the application code. By default, each of these symlinks is created using a separate call to run, which in turn creates a separate SSH connection to make each symlink. Establishing and tearing down these SSH connections can take some time. Using Capistrano's block variables, though, we can replace the symlink task with one that gives us more speed, as well as additional flexibility.

First, let's redefine the deploy:symlink task. We'll include a description and we'll run this task with the same restrictions as the builtin tasks; only on app servers:

Download capistrano2/deploy_symlink_override.rb

```
namespace :deploy do
  desc "Create symlinks to stage-specific configuration files and shared resources"
  task :symlink, :roles => :app, :except => { :no_release => true } do
  end
end
```

Now we can fill in the body of the task. First we declare a cleanup_targets variable, which is expected to be an array, and create a shell command to remove each of the target files/directories:

```
symlink_command = cleanup_targets.map { |target| \
  "rm -fr #{current_path}/#{target}" }
```

Next we take an array of directories that should be recreated on each deploy from the release_directories variable:

```
symlink_command += release_directories.map { |directory| "mkdir -p #{directory} }
```

Then we gather a hash of from-to pairs from the release_symlinks variable that should be created from items in the release directory. These will typically be stage-specific configuration files that you don't mind checking in to your repository. The -s flag tells the ln utility to create a symlink, and the -f flag tells ln that if the symlink exists to remove it and recreate it.

```
symlink_command += release_symlinks.map { |from, to| \
  "rm -fr #{current_path}/#{to} && \
  ln -sf #{current_path}/#{from} #{current_path}/#{to}" }
```

The shared_symlinks variable is a hash of from-to pairs where is relative to shared_path and is relative to the current_path:

```
symlink_command += shared_symlinks.map { |from, to| \
  "rm -fr #{current_path}/#{to} && \
  ln -sf #{shared_path}/#{from} #{current_path}/#{to}" }
```

Finally, we concatenate all of these into a single shell command that runs all of the directory and symlink commands at once. This is an easy win for

bringing your deployment times down, as well as giving you more flexibility when configuring your application in different deploy stages.

```
run "cd #{current_path} && #{symlink_command.join(' && ')}"
```

Here's the entire task that we've built up line by line:

Download capistrano2/deploy_task.rb

```
namespace :deploy do
  desc "Create symlinks to stage-specific configuration files and shared resources"
  task :symlink, :roles => :app, :except => { :no_release => true } do
    symlink_command = cleanup_targets.map \
      { |target| "rm -fr #{current_path}/#{target}" }
    symlink_command += release_directories.map \
      { |directory| "mkdir -p #{directory} }
    symlink_command += release_symlinks.map \
      { |from, to| "rm -fr #{current_path}/#{to} && \
        ln -sf #{current_path}/#{from} #{current_path}/#{to}" }
    symlink_command += shared_symlinks.map \
      { |from, to| "rm -fr #{current_path}/#{to} && \
        ln -sf #{shared_path}/#{from} #{current_path}/#{to}" }
    run "cd #{current_path} && #{symlink_command.join(' && ')}"
  end
end
```

And here's a typical usage of the variables that we would define when using this task:

Download capistrano2/sample_variables.rb

```
set :cleanup_targets, %w(log public/system tmp)
set :release_directories, %w(log tmp)

set :release_symlinks do
  {
    "config/settings/#{stage}.yml" => 'config/settings.yml',
    "config/database/#{stage}.yml" => 'config/database.yml',
  }
end

set :shared_symlinks, {
  'log'     => 'log',
  'pids'    => 'tmp/pids',
  'sockets' => 'tmp/sockets',
  'system'  => 'public/system'
}
```

Notice how this technique also separates the configuration data values from the code that processes them, making the deployment configuration more readable. It's a win on both performance and clarity.

5.2 Uploading and Downloading Files

Capistrano has a handy set of functions available for transferring files to and from remote machines. The simplest of these is get, which we can use to fetch a file from a remote server:

Download capistrano2/get.rb

```
desc "Download the production log file"
task :get_log do
  get "#{current_path}/log/production.log", \
    "#{Time.now.strftime("%Y%m%d%H%M")}.production.log"
end
```

We can also fetch an entire directory tree with the recursive option:

Download capistrano2/get_recursive.rb

```
desc "Download the entire log directory"
task :get_log_directory do
  get "#{current_path}/log/", "tmp/", :recursive => true
end
```

get will only connect to one server, but for connecting to multiple servers we can use download. However, if we download files with the same name from multiple servers, they'll just overwrite each other. Fortunately, Capistrano supports a simple macro; it replaces the string $CAPISTRANO:HOST$ with the name of the host to which Capistrano is connecting. So we can write our download task with this macro in the destination file name, and we'll get a series of files all prefixed with the appropriate host name:

Download capistrano2/download.rb

```
desc "Download the production log file"
task :download_log do
  download "#{current_path}/log/production.log",\
    "$CAPISTRANO:HOST$.production.log"
end
```

Capistrano also gives us an upload command for transferring files up to remote servers. The $CAPISTRANO:HOST$ string works with upload, so we can give each server a specific maintenance page by placing a few files in a local directory:

```
$ ls tmp/maintenance_pages/
maintenance.html.server1.com maintenance.html.server2.com
maintenance.html.server3.com
```

Then we'll reference that file in a task that uses upload and each server will receive the appropriate file:

Download capistrano2/upload.rb

```
desc "Upload the host-specific maintenance pages to each server"
task :upload_maintenance_page do
```

```
  upload "tmp/maintenance.html.$CAPISTRANO:HOST$", "#{deploy_to}/maintenance.html"
end
```

Sometimes we don't need to upload a file but instead just the contents of a string. In those cases we can use the Capistrano put() method. To demonstrate this, let's write a small task to copy the current Git revision to the remote servers. We can find the Git revision with rev-parse:

```
$ git rev-parse HEAD
956fe36157c57060414e4f9804bb79fc6f1cae90 # Or some such SHA
```

And we'll upload it to our VM using put():

Download capistrano2/put.rb

```
desc "Put the current revision onto the server"
task :put_revision do
  put `git rev-parse HEAD`.strip, "#{current_path}/public/REVISION"
end
```

With these handy Capistrano features there's no reason to resort to shelling out to scp to move files or data around.

5.3 Speeding Things Up with Git Reset and Command Concatenation

As you have seen, Capistrano provides a great set of defaults for deploying your application in a variety of ways, and most commonly you will be using remote_cache. While the default behavior of remote_cache might seem like all you need, eventually you might notice that things start slowing down a bit. The repository must be fetched and copied, and a growing number of symlinksget created, each within their own separate SSH connection. Copying the repository each time and opening up multiple connections just to create symlinks can become a major bottleneck to the speed of your deploys.

5.4 Roles

We've seen the default app, web, and db Capistrano roles and some examples of how they're used by various built in Capistrano tasks. However, sometimes we have jobs that don't fit into those three categories, and for those situations we can define and use a new server role. Let's set up some roles roles for using Redis[1]:

A basic Redis configuration has an instance running on one server. We'll defining a new redis role by naming it and supplying the hosts to which it applies:

```
role :redis, "redismaster.mydomain"
```

1. An excellent open source key-value store; read all about it at http://redis.io

Now we'll use this new role in a task declaration:

```
task :rewrite_aof, :roles => :redis do |t|
  # send in the bgrewriteaof command
end
```

This means that the rewrite_aof task will only be run on our redismaster.mydomain host. We could also specify several host names on one line if we had multiple instances:

```
role :redis, "redis1.mydomain", "redis2.mydomain"
```

Here's another scenario. Suppose we have several hosts running Redis for gathering statistics and an additional host running an instance which we use for Resque. To accomodate this setup we can attach attributes to the role declarations to differentiate those hosts:

```
role :redis, "resque.mydomain", {:resque => true}
role :redis, "stats1.mydomain", "stats2.mydomain", {:stats => true}
```

To get a weekly summary for the stats hosts, let's declare a task that further restricts the hosts on which it's run:

```
task :weekly_summary, :roles => :redis, :only => {:stats => true} do |t|
  # calculate weekly statistics
end
```

For the ultimate in flexibility, we'll can pass a block to a role declaration and that block will be evaluated when the script is running. This is handy when the list of servers to which a role applies needs to be calculated when the Capistrano script is run. For example, we'll want to run bgrewriteaof on all our Redis servers, and the list of those servers might be stored in a separate YAML file:

```
servers:
  redis:
    - 192.168.10.10
    - 192.168.10.11
    - 192.168.10.14
```

We'll read those servers in with a block associated with our role declaration. In this case we need to wrap the role name in parentheses so the Ruby parser can tell where the parameters to role() end and where the block starts:

```
role(:redis) {YAML.load(File.read("config/servers.yml"))["servers"]["redis"] }
```

Now when we run a task that is declared for that role we'll be sure to get the latest server list.

Roles also give us the ability to run a general task on a restricted set of servers using the ROLES environment variable. For example, we can run the deploy:setup task only on servers in the utility role like this:

```
$ cap deploy:setup ROLES=utility
  * executing `deploy:setup'
  [ ... some boilerplate output ... ]
    servers: ["localhost"]
    [localhost] executing command
    command finished
```

Roles provide a convenient and standard way to restrict a task to a subset of servers. If you find a Capistrano deploy file that contains if statements, those can probably be replaced with roles.

5.5 Multistage

For a small project we usually only have to deploy to one environment, namely, production. For larger projects, though, we'll have a quality assurance environment, and sometimes we'll have a performance testing environment, and maybe a sales demonstration environment, and... generally, we'll be deploying for multiple environments.

We like to solve this problem with a techique called Capistrano *multistage*. Multistage lets us specify a different Capistrano configuration for each environment we're deploying to while continuing to share the common settings in config/deploy.rb.

Multistage comes as part of the capistrano-ext gem, so let's add that dependency to our Gemfile. We'll put it in the development group since this isn't an application dependency runtime:

```
group :development do
  gem 'capistrano-ext'
end
```

After installing the gem with bundle, we'll modify our config/deploy.rb to use multistage. Let's set two variables and require multistage's code. The first variable, stages, is an Array of all the environments that we'll be deploying to. The second variable, default_stage, is the stage that Capistrano will deploy to if we just run cap deploy without specifying a stage. Lastly we require the multistage extension. Here's how that looks at the top of config/deploy.rb with two stages, beta and production:

```
set :stages, %w(beta production)
set :default_stage, "beta"
require 'capistrano/ext/multistage'
```

We've told Capistrano that we have several stages, so now we need to define them. Let's create a new directory to hold our stage configuration; multistage will look for these in config/deploy/ by default so we'll create that directory[2]:

```
$ mkdir config/deploy/
```

Now we can add a new file to that directory with the custom settings for our beta environment. In this case the only thing different about beta is that it's on a non-production server, so let's put that setting into config/deploy/beta.rb:

```
server "beta.mydomain.com", :web, :app, :db, :primary => true
```

And we'll need the same variable in our production stage configuration, config/deploy/production.rb:

```
server "mydomain.com", :web, :app, :db, :primary => true
```

We're setting the server name in both our environment files, so we can remove that setting from config/deploy.rb.

Now we can deploy to the beta environment:

```
$ cap deploy
«output»
```

Or we can explicitly specify the beta environment, although that is the default:

```
$ cap beta deploy
«output»
```

And to deploy to production we can use:

```
$ cap production deploy
«output»
```

We can expand our list of stages as needed, and we can move any of the Capistrano variable settings into our individual stage deployment files. Those files get loaded after config/deploy.rb, so any settings in those files will override the settings in config/deploy.rb.

Multistage provides a simple and effective way to deploy to multiple environments; it's something we find useful in all but the smallest projects.

5.6 Capturing and Streaming Remote Command Output

Everything's going great with MassiveApp and new user registrations are flooding in. However, MassiveApp still has its share of hiccups in production,

2. We could use a different directory with set :stage_dir, "some_directory"

and you've grown tired of opening up a new SSH session, tailing logs, and performing other simple server maintenance tasks.

Sometimes we need to get a bit closer to the metal when diagnosing problems in production. While logging in to our server via SSH is usually an option, you will find that it's often more convenient to be able to run small tasks directly from your working directory. Capistrano provides the capture and stream methods for just this purpose. capture lets you run a remote command on one or many of your servers and get back the output as a string, while stream allows you to keep the session open and get an ongoing dump of the command's standard output. Both methods take the same options as run

Using Capture

First, let's take a look at capture. Let's say we'd like to see the amount of free memory on our production server. Typically, we would log in and run something like the following:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:           998        931         67          0         70        494
«More output»
```

We can make use of this to create a diagnostic task in Capistrano that prints out the information in a nicer form.

Download capistrano2/diagnostics.rb

```
namespace :diagnostics do
  desc "Display free memory"
  task :memory_usage do
    mem = capture("free -m | grep Mem").squeeze.split(' ')[1..-1]
    total, used, free, shared, buffers, cached = mem

    puts "Memory on #{role} (in MBs)"
    puts "Total: #{total}"
    puts "Used: #{used}"
    puts "Free: #{free}"
    puts "Shared: #{shared}"
    puts "Buffers: #{buffers}"
    puts "Cached: #{cached}"
  end
end
```

Capturing a stream

stream lets us get a constantly updated output of the command we want to run. A simple use case for this is obviously tailing our production logs! We

can take advantage of some variable interpolation here to make a task that will tail the correct log no matter what environment we're in:

Download capistrano2/stream.rb

```
namespace :logs do

  desc "Tail the Rails log for the current stage"
  task :tail, :roles => :app do
    stream "tail -f #{current_path}/log/#{stage}.log"
  end

end
```

As you can see, capture and stream are two simple and extremely useful commands. There are an endless number of use cases for these two methods. A good rule of thumb to keep in mind is that each command you find yourself running more than once on the server is a good candidate for one of these methods, especially if you need to run the command on more than one box at once.

CHAPTER 6

Monitoring with Nagios

MassiveApp is deployed and serving the needs of its user base. At least, we're pretty sure that it's serving the needs of the user base, because we're so excited about it that we're manually watching log files, refreshing pages to see new user counts, and generally hovering over the system to make sure that all's well. But we can only do this for so long; eventually we'll get wrapped up in coding the next big feature and forget to check the server load. And while we're not watching things, that's when a suge of activity is likely to bring MassiveApp to its knees.

What we need is someone to check on the server and let us know if anything strange is happening. But since we don't have the budget for a person to do that, we'll need an automated solution instead; we need monitoring. Effective monitoring will provide us with feedback when MassiveApp's hardware, system services, or application code behaves in an unexpected manner. In this context, "unexpected" may mean that a service has crashed or is using too much memory, or it may mean something "good", such as a huge influx of new users. Either way, if something out of the ordinary is happening, a good monitoring solution will let us know about it.

We're going to use Nagios[1] for our monitoring system; here's why:

- It has flexible notifications and scheduling, lots of plugins to check the status of all sorts of services, a web interface to let us examine the state of the system, and a variety of reporting graphs and tables. All these features will come in handy as our monitoring needs evolve.
- It has a solid security model. We can configure it to tunnel remote status checks over `ssh`, and the web interface has a variety of security-related directives that let us lock things down.

1. http://nagios.org

- It scales. If it works well for a large site like Wikipedia, [2] it will work for us. Nagios has various features (like passive checks) that are built specifically to make large installations effective and efficient.
- Commercial support is available. We won't need it to start with, but as MassiveApp expands, it may be nice to have someone on call who does Nagios configuration and tuning for a living.
- Finally, it's open source and free. A popular application like MassiveApp is going to need a lot of servers; the last thing we need is a per-CPU licensing scheme that gobbles up our profits.

That's our justification; now let's get Nagios running.

6.1 A MassiveApp to Monitor

We want to monitor MassiveApp, and MassiveApp (being massive and all) will run on a number of servers. To simulate this with VMs, we'll set up one VM that's running MassiveApp and one VM that's running Nagios and monitoring the other VM. We'll need a Vagrantfile that starts two VMs that use host only networking to communicate. Let's start by creating a new directory to hold the VMs:

```
$ mkdir ~/deployingrails/nagios && cd ~/deployingrails/nagios
```

Here's our Vagrantfile with our two VM definitions. Notice that we forward port 4568 to the nagios VM so that we'll be able to browse to either box. We'll also use the custom base box that we built in *Building a custom base box*, on page 15 for both VMs:

Download monitoring/dual_vm_vagrantfile

```
Vagrant::Config.run do |config|
  config.vm.define :app do |app_config|
    app_config.vm.customize do |vm|
      vm.name = "app"
      vm.memory_size = 512
    end
    app_config.vm.box = "lucid64_with_ruby192"
    app_config.vm.host_name = "app"
    app_config.vm.forward_port "ssh", 22, 2222, :auto => true
    app_config.vm.forward_port "web", 80, 4567
    app_config.vm.network "33.33.13.37"
    app_config.vm.share_folder "puppet", "/etc/puppet", "../massiveapp_ops"
  end
  config.vm.define :nagios do |nagios_config|
    nagios_config.vm.customize do |vm|
```

2. http://nagios.wikimedia.org/

```
      vm.name = "nagios"
      vm.memory_size = 512
    end
    nagios_config.vm.box = "lucid64_with_ruby192"
    nagios_config.vm.host_name = "nagios"
    nagios_config.vm.forward_port "ssh", 22, 2222, :auto => true
    nagios_config.vm.forward_port "web", 80, 4568
    nagios_config.vm.network "33.33.13.38"
    nagios_config.vm.share_folder "puppet", "/etc/puppet", "../massiveapp_ops"
  end
end
```

With that file in our ~/deployingrails/nagios directory we can start the VMs:

```
$ vagrant up
«lots of output as both VMs are started»
```

We can verify that the VMs are up by connecting into the nagios VM:

```
$ vagrant ssh
`vagrant ssh` requires a specific VM name to target in a multi-VM environment.
```

What happened there? Well, since we have two VMs in our Vagrantfile, we need to specify the host name of the VM. Let's try again:

```
$ vagrant ssh nagios
nagios $ ssh 33.33.13.37
The authenticity of host '33.33.13.37 (33.33.13.37)' can't be established.
RSA key fingerprint is ed:d8:51:8c:ed:37:b3:37:2a:0f:28:1f:2f:1a:52:8a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '33.33.13.37' (RSA) to the list of known hosts.
vagrant@33.33.13.37's password:
Last login: Wed Nov  9 03:48:36 2011 from 10.0.2.2
app $
```

That's more like it. Next we need to set up MassiveApp so that Nagios will have something to monitor. From Chapter 3, *Rails on Puppet*, on page 25 we have a Git repository containing manifests to get MassiveApp's services running. Our VMs are both sharing in that repository, so we just need to run Puppet on the app VM:

```
$ vagrant ssh app
app $ cd /etc/puppet
app $ sudo puppet apply manifests/site.pp
«lots of output as services are installed»
```

The nagios VM already has Puppet installed and we haven't built our Puppet module yet, so there's nothing to do on that end.

We haven't deployed MassiveApp yet, but we've got plenty of things to monitor already — disk space, memory usage, and so forth. Next we'll get Nagios installed and monitoring those basic resources.

6.2 A Nagios Puppet Module

Nagios' architecture is based around a single server and multiple agents. The server is a single host that's the center of the monitoring system; it receives and reacts to monitoring data collected by agents running on each monitored machine. In our MassiveApp setup, the Nagios server is on nagios and we'll have agents monitoring app. Nagios also monitors the system it's running on (i.e., nagios) out of the box, so we get that for free.

We want to set up Nagios on our nagios VM, but we also want to automate the installation and configuration so that we'll know exactly how we set it up. So let's build a new Puppet module that does all the tedious work for us. Let's connect into the nagios VM and move to the /etc/puppet directory to get started. Now that we're in our Puppet root directory we'll create a new module:

```
$ vagrant ssh nagios
nagios $ cd /etc/puppet/
nagios $ mkdir -p modules/nagios/manifests
```

And we'll create a nagios class in modules/nagios/manifests/init.pp to hold our resource definitions:

Download monitoring/init.pp

```
class nagios {
}
```

We're setting up the Nagios server, so we won't put any resources in init.pp. Instead, we'll create a new server class and use that to provision our VM. This starts as an empty class declaration in modules/nagios/manifests/server.pp:

Download monitoring/server-empty.pp

```
class nagios::server {
}
```

Ubuntu's package repository has a Nagios packages, so let's add a package resource declaration to our server class. This will bring in other Nagios packages as dependencies:

Download monitoring/server-with-package.pp

```
package {
  "nagios3":
    ensure => present
}
```

Since we're adding a new module we also need to add this to our manifests/site.pp. We've got two nodes now (app and nagios) so we'll wrap the app includes in one node and the nagios nodes in another. We'll install Apache (as well as Nagios) on the nagios node since Nagios surfaces an HTML interface which we'll want to use:

Download monitoring/dual_site_nodes.pp

```
node app {
  include apache2
  include passenger
  include mysql
  include massiveapp
  include nagios
}

node nagios {
  include apache2
  include nagios::server
}
```

Let's go ahead and run Puppet to get the Nagios packages in place:

```
nagios $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1302145355'
notice: /Stage[main]/Nagios::Server/\
  Package[nagios3]/ensure: created
```

The packages are installed, but we want to ensure that the Nagios server will start when our VM reboots. So let's open modules/nagios/manifests/server.pp and add a service resource that's enabled:

Download monitoring/nagios-service.pp

```
service {
  "nagios3":
    ensure      => running,
    hasrestart  => true,
    enable      => true,
    hasstatus   => true,
    restart     => "/etc/init.d/nagios3 reload",
    require     => Package["nagios3"]
}
```

Let's go ahead and run Puppet get those settings in place.

A *check* is the term that Nagios uses for the process of examining something's status. The Nagios web interface is our window into the checks and their results, so we need to ensure that Apache can serve that up. Nagios comes with an Apache virtual host configuration file which does the job, but it allows Nagios to be accessed using the URL /nagios on any domain that's hosted by

that server. We prefer to run Nagios on its own domain (nagios.somehost.com) so that it won't interfere with anything else that we put on that host. To do that, we'll need a virtual host definition. This will be a static file in our Puppet module, so we need a directory to put it in. Let's create that directory and copy in the default Apache configuration file:

```
nagios $ mkdir -p modules/nagios/files
nagios $ cp /etc/nagios3/apache2.conf modules/nagios/files/
```

Now we'll make some changes to modules/nagios/files/apache2.conf. We'll add a <VirtualHost> element, delete all the commented out lines, and once that's done we've got a relatively small file:

Download monitoring/apache2.conf

```
NameVirtualHost On
<VirtualHost *:80>
  ServerName nagios.localhost
  ScriptAlias /cgi-bin/nagios3 /usr/lib/cgi-bin/nagios3
  ScriptAlias /nagios3/cgi-bin /usr/lib/cgi-bin/nagios3
  DocumentRoot /usr/share/nagios3/htdocs/
  Alias /nagios3/stylesheets /etc/nagios3/stylesheets
  Alias /nagios3 /usr/share/nagios3/htdocs
  <DirectoryMatch (/usr/share/nagios3/htdocs|\
    /usr/lib/cgi-bin/nagios3|/etc/nagios3/stylesheets)>
    Options FollowSymLinks
    DirectoryIndex index.html
    AllowOverride AuthConfig
    Order Allow,Deny
    Allow From All
    AuthName "Nagios Access"
    AuthType Basic
    AuthUserFile /etc/nagios3/htpasswd.users
    require valid-user
  </DirectoryMatch>
</VirtualHost>
```

The Apache configuration includes an AuthType Basic directive. This sets up HTTP basic authentication, which is a good idea since it protects the Nagios installation from prying eyes. In order for that to work, though, we need to create a text file with a username and password. We'll use the Apache utility htpasswd for this, and we'll store that file in our Nagios module's files directory. Let's create a password file with one user, nagiosadmin, with a password of nagios:

```
nagios $ htpasswd -cmb modules/nagios/files/htpasswd.users nagiosadmin nagios
```

We need a Puppet file resource to get the Apache configuration file in place, and another file resource to handle the htpasswd.users file. Let's add those entries to modules/nagios/manifests/server.pp. We've got a notify attribute for the Apache

service on the apache2.conf entry, so when that file changes Apache will be restarted:

Download monitoring/apache2.conf.pp

```
file {
  "/etc/nagios3/apache2.conf":
    source  => "puppet:///modules/nagios/apache2.conf",
    owner   => root,
    group   => root,
    mode    => 644,
    notify  => Service["apache2"];
  "/etc/nagios3/htpasswd.users":
    source  => "puppet:///modules/nagios/htpasswd.users",
    owner   => www-data,
    group   => nagios,
    mode    => 640,
    require => [Package["apache2"],Package["nagios3"]];
}
```

After running Puppet again to get those in place, Nagios is running on the VM and we need to configure our workstation so we can browse into it. We'll do this by adding an entry to our /etc/hosts file, so let's open that file and add this line:

```
127.0.0.1 nagios.localhost
```

Now we can open our browser to http://nagios.localhost:4568/ and we'll see a basic authentication dialog for a username and password. We'll enter the same values as we put in the htpasswd file on the VM (nagiosadmin for the username and nagios for the password) and we can see the Nagios web interface.

The front page of the Nagios web interface has two frames. The one on the right is more or less an advertisement for a Nagios support provider, so the one on the left is the one that we're more interested in. The left frame has links to pages that display various aspects of the systems being monitoring. Nagios refers to servers it's monitoring as *hosts*, so when we click the "Host Details" link we see a list of servers. Somewhat unexpectedly, there are two servers listed on that page: localhost and gateway. That's because Nagios is monitoring the VM that it's running on and it's also pinging the IP address that's configured as the default route on the VM. Both servers are listed with a green background, which means that all's well. We can click on either of the host names and see details about when Nagios last checked that host and how long the check took. We can also see a variety of links deeper into the data that Nagios gathers about the host.

Let's explore a little more. Nagios refers to the items that it is monitoring as *services*, so if we click on the "Service Detail" link in the left panel we can see

a list of services such as disk space, the ssh daemon, the count of logged in users, and a few others. Clicking on an individual service, such as "Disk Space", takes us to another details page showing the time the service was last checked and the results of that check. As with the host detail page, we can go digging a lot deeper, but since we just started Nagios there's really not much data available yet.

But there is one useful option we can try at this point. On the right side of the page for an individual service there's a "Re-schedule the next check of this service" link. This link let's us do an immediate check of a service without waiting for Nagios' next scheduled check to run. If we click that link, we see another page that tells us that we're about to force a service check. Let's click the "Commit" button to force a check, and... hm, no check. Instead we get an error message "Sorry, but Nagios is currently not checking for external commands, so your command will not be committed". Let's fix that.

The issue here is that Nagios locks down some options by default. In this case, there's a check_external_commands variable in the Nagios configuration file that we need to enable. So let's get back into the Puppet mindset and make that happen. First, we'll copy the default Nagios configuration file into our Nagios module:

```
nagios $ cp /etc/nagios3/nagios.cfg modules/nagios/files/
```

Now we can make our change. Let's open up nagios.cfg, find the check_external_commands variable, change the value from 0 to 1, and save the file. We're partway there, but we need to manage this file with Puppet as well. So let's open modules/nagios/manifests/server.pp and add a new file resource. We're using a notify attribute so that Nagios will be restarted when the configuration file changes.

Download monitoring/nagios.cfg.pp

```
"/etc/nagios3/nagios.cfg":
   source  => "puppet:///modules/nagios/nagios.cfg",
   owner   => nagios,
   group   => nagios,
   mode    => 644,
   notify  => Service["nagios3"],
```

Now we can run Puppet and try our "Commit" button again. We see some progress; at least we're getting a different error message now: Error: Could not stat() command file '/var/lib/nagios3/rw/nagios.cmd'! In other words, Nagios uses a file as a signal that commands need to be run, and this file doesn't have the proper permissions. Back into modules/nagios/manifests/server.pp we go, and this time we're adding a file resource that's just setting the permissions on the nagios.cmd file.

Download monitoring/nagios.cmd.pp

```
"/var/lib/nagios3/rw":
  ensure  => directory,
  owner   => nagios,
  group   => www-data,
  mode    => 710,
  require => Package["nagios3"];
```

We run Puppet, click “Commit” one more time, and now we have success. We see a message along the lines of “Your command request was successfully submitted to Nagios for processing.” We can verify that the check ran by clicking on “Service Detail”, then on “Disk Space”, and the “Last Check Time” field will show a very recent timestamp. So now we can force a check at any time. This is especially handy when we’re adding new service checks and want to ensure that things are working.

We’ve got Nagios running on a server, it’s monitoring itself, and we can browse the web interface and see what’s going on. Next we need to get Nagios to monitor what’s happening on our `app` VM. But first we’ll catch our breath by looking at some general monitoring concepts that we’re using so far.

6.3 Monitoring Concepts in Nagios

Let’s think about monitoring scenarios in general; doing this will give us an introduction to a few more Nagios terms. It will also set up our expectations for what concepts our monitoring framework will support. We want to keep track of various servers; as we’ve seen these are Nagios *hosts*. We’ll need to partition these hosts; for example, once our infrastructure gets above a certain size we’ll have some servers that are less critical than others. Nagios provides a way to group hosts via the aptly named *hostgroups*. With hostgroups we can configure Nagio to send an SMS for a production database server server problem, but an email for an issue with the standby performance testing server.

As we continue to think through the monitoring domain, we know our hosts are running various programs (which, as we’ve seen, Nagios calls “services”) that we’re keeping an eye on. Nagios checks services using *commands*. If something goes wrong, someone needs to be notified; that lucky person is a *contact*. Perhaps we’ve got someone else on the team who also needs to know when things break; in that case both people would be part of a *contact group*.

Monitoring disk space is different than monitoring a CPU, and both of those are different than monitoring a network service like MySQL. Nagios addresses the need to monitor different types of services by providing *plugins* and an open plugin architecture. The default Nagios installation comes with over 60

plugins including check_tcp, check_mailq, check_file_age and many more. And that's just the tip of the iceberg. When putting a new service in place, a quick search for "nagios some_service_name" usually finds a variety of approaches to monitoring that service. As an example, there are currently 32 different MySQL monitoring plugins that check everything from replication lag to connection counts to temporary table creation. So before rolling your own Nagios plugin, take a quick look to see what's out there.

Any monitoring system you choose (or write) will involve these general domain concepts (hosts, services, commands, contacts) in some form or another. Even a simple cron job that pings a server once an minute and emails on failure falls under these general categories; there's a host that's receiving a service check and a failure results in a contact receiving a notification. If you inherit a simple monitoring setup that consists of a few shell scripts, take heart; you've got a place to start and you at least know what your predecessor considered important enough to monitor.

This isn't an exhaustive list of monitoring concepts or Nagios concepts. If we flip through a Nagios reference manual we'll see chapters on timeperiods, macros, host dependencies, and much more. The takeaway is that we need to monitor our existing services and make a practice of thinking about monitoring for any new services. We need to keep the vision in mind; we're setting things up so that we'll know when our systems are reaching their limits and we'll constantly improve the monitoring system. At the very least we'll have a nice system to hand off when our inevitable promotion occurs.

Now that we've covered some monitoring concepts and Nagios terms we'll enhance the monitoring for our nagios VM.

6.4 Monitoring Local Resources

A basic form of monitoring is a simple check to ensure that a server is up. The Nagios setup that we've configured for nagios has this in place for any other servers that we add. We don't ping the nagios VM; the Nagios server being down means that notifications wouldn't go out anyway. But it's a good place to start with other servers. Some firewalls will drop pings or, to be more specific, all Internet Control Message Protocol (ICMP) packets. If that was the case, we'd pick some other network service to check. ssh is one good candidate; if the ssh service is not responding, the server is probably having other issues as well.

Once we know the system is up, the next step is to see if we're is approaching any operating system-level resource limits. By "operating system-level" we

Watching the Watcher

Nagios will faithfully monitor MassiveApp, but if the Nagios server goes down, how will we find out? Rather than depending on our customers, we'll use a server in another data center or a service like Pingdom [a]http://www.pingdom.com/ to check up on Nagios. There will be some additional setup cost and time, but it's worth it to be the first to know.

a. http://www.pingdom.com/

mean things like disk space, process count, and CPU load average. The default setup includes these checks, and we really don't want to run a system without them. Sometimes a load average alert will be our first indicator that a flood of traffic has come our way.

Our Nagios setup has default values at which it alerts, and for most use cases those values are pretty reasonable. For example, the disk space check sends a "warning" alert when only 20% of disk space remains and sends a "critical" alert when 10% remains. But maybe we'd like to be a bit more prudent? Let's try it out by editing the Nagios configuration and changing the alert value.

Nagios creates a /etc/nagios3/conf.d directory that contains a few default configuration files. One of these files, localhost_nagios2.cfg (despite its name it works fine with Nagios 3) has the setting we need to modify to change the disk space alert threshold. Let's get this file into our Nagios Puppet module so we can track our changes. First we'll create a new directory in modules/nagios/files/. We're creating a parent conf.d directory that contains a hosts directory; this will let us put all our host configuration in one location:

```
nagios $ mkdir -p modules/nagios/files/conf.d/hosts
```

Now we can copy localhost_nagios2.cfg into our new directory, although we'll drop the Nagios version:

```
nagios $ cp /etc/nagios3/conf.d/localhost_nagios2.cfg \
  modules/nagios/files/conf.d/hosts/localhost.cfg
```

Let's open modules/nagios/files/conf.d/hosts/localhost.cfg and locate the following section:

```
define service {
 use                  local-service
 service_description  Disk Space
 check_command        check_all_disks!20%!10%
}
```

This is one of several service definitions that we'll see in this file. The check_command *variable* has a *value* that specifies the alerting thresholds. Let's change the 20% to something very high, like 99%, and save the file. Now the change is in place, and it's time to add this file to our Puppet module. Let's open modules/nagios/manifests/server.pp and add a new section to our list of file resources. We'll have other files to manage in the conf.d directory tree, so we'll set the recurse attribute to true. We'll also remove the old localhost_nagios2.cfg file using another file resource with the ensure attribute set to absent:

Download monitoring/server-confd-snippet.pp

```
"/etc/nagios3/conf.d":
  source  => "puppet:///modules/nagios/conf.d/",
  ensure  => directory,
  owner   => nagios,
  group   => nagios,
  mode    => 0644,
  recurse => true,
  notify  => Service["nagios3"],
  require => Package["nagios3"];
"/etc/nagios3/conf.d/localhost_nagios2.cfg":
  ensure  => absent;
```

We've made our changes, so let's run Puppet to get those in place and restart Nagios. Now, let's force Nagios to check this service right away rather than waiting for Nagios to run the next scheduled check. We can do this through the web interface since earlier we set up Nagios to allow external commands. We'll use the web interface just like before. First we'll click on the "Service Detail" link in the left pane, then a click on "Disk Space" in the right pane gets us to the disk space check details. Then we'll click on the "Re-schedule the next check of this service" and submit the form on the next page via the "Commit" button. Now when we go back to the "Service Detail" page again we see that the "Disk Space" status is highlighted in yellow and the "Status" column value is "Warning".

Changing the disk space warning threshold to 99% was a good way to ensure we could make that change, but alerting at 99% of free space remaining is probably just a bit too high for normal operation. But we can easily back out that change by editing localhost.cfg, changing the service definition, and forcing another check through the web interface. Then we'll see the "Disk Space" status line go back to a more placid green.

Now we've seen that Nagios is checking the basic system status and we've successfully modified a monitoring threshold to suit our preferences. This is the virtuous cycle of monitoring. We're aware of what we're monitoring, we're

getting alerted when something exceeds a threshold, and then we're resolving the alert in some way.

Now we'll go a little deeper and set up monitoring for services on both our VMs.

6.5 Monitoring Services

The MassiveApp server itself may be up and running, but that doesn't mean that everything's working. MassiveApp uses a variety of network services (Apache, MySQL, memcached) and if any of those are offline, MassiveApp will be either slow or go completely down. By setting up separate monitors for each service, we'll get a specific alert if one component shuts down, and we'll know what needs to be restarted or tuned.

Some services can be monitored by a simple port check. We can get a good comfort level that the service is available by ensuring that it is responding to attempts to connect to it on its configured TCP port. One example of a service that fits this profile is ssh. There's a service check for ssh in the default configuration file for localhost (which we've renamed localhost.cfg); here's what it looks like:

```
service {
  use                 local-service
  service_description ssh
  check_command       check_ssh
}
```

This is a Nagios *object definition*, and it consists of an object name, service, that contains *variables* and their *values*. The use variable is Nagios' way of implementing inheritance; it causes this service check to inherit variables from the local-service *template* that's defined earlier in /etc/nagios3/conf.d/hosts/localhost.cfg. That template in turn inherits from another generic-service template (defined in /etc/nagios3/conf.d/generic-service_nagios2.cfg) which sets default variable values such as a 24x7 notification period. Any of those variables can be overridden at any level, and using templates in this manner can greatly reduce duplication in Nagios object definitions.

The check_command variable is set to check_ssh; that's one of Nagios' built in plugins. By default, this plugin connects to the ssh daemon and puts the response in the "Status Information" field of the Nagios web interface.

Monitoring ssh

Let's take a closer look at using the check_ssh plugin. It provides a reasonable default strategy, but it can also ensure that the ssh daemon provides a partic-

ular response. Nagios provides a convenient way to experiment with a plugin's options; plugins are just files stored in /usr/lib/nagios/plugins, so we can test check_ssh from the shell without tweaking a configuration file. Let's do that now; we'll run the check_ssh plugin without any options:

```
nagios $ /usr/lib/nagios/plugins/check_ssh
check_ssh: Could not parse arguments
Usage:check_ssh [-46] [-t <timeout>] [-r <remote version>]
 [-p <port>] <host>
```

We got an error message since we didn't pass in any arguments, but we also see the parameters that this plugin accepts. It requires a host parameter; let's try that:

```
nagios $ /usr/lib/nagios/plugins/check_ssh localhost
SSH OK - OpenSSH_5.3p1 Debian-3ubuntu4 (protocol 2.0)
```

That's better; the plugin connected to the ssh daemon and received a version string in response. This is the standard ssh daemon response, so we could get this same version string response if we used telnet to connect to an ssh daemon. That's also the output we see in the Nagios web interface for the status of this check. Let's see how it looks when we send in some bad data; the -r option checks for a particular ssh version, so we can get a warning out of that easily:

```
nagios $ /usr/lib/nagios/plugins/check_ssh -r 0.1 localhost
SSH WARNING - OpenSSH_5.3p1 Debian-3ubuntu4 (protocol 2.0)
 version mismatch, expected '0.1'
```

And for an outright failure, let's try to connect to port 23. This is the standard telnet port, which is insecure in exciting ways. But even if it were running it wouldn't respond in a way that would satisfy the check_ssh plugin:

```
nagios $ /usr/lib/nagios/plugins/check_ssh -p 23 localhost
Connection refused
```

This script failure won't affect the Nagios dashboard since we're running these checks out of band; so we can experiment without fear of causing notifications to be sent or ugly colors to appear in the web interface.

Let's also ensure that check_ssh can check the status of the ssh daemon running on our app VM. To make this easier, we'll add a host name entry to our nagios VM's /etc/hosts; that way we won't have to type out app's IP address repeatedly:

```
33.33.13.37 app
```

With that in place we can run the check_ssh plugin against app:

```
nagios $ /usr/lib/nagios/plugins/check_ssh app
```

```
SSH OK - OpenSSH_5.3p1 Debian-3ubuntu4 (protocol 2.0)
```

That's as expected; the app VM is from the same base box as the nagios VM, so it should be the same version. To get Nagios to do that same check automatically, we'll add a new hosts file for our app box. Of course we'll manage that with Puppet, so let's create modules/nagios/files/conf.d/hosts/app.cfg with a new *host* object definition. We'll inherit from another built-in Nagios template, generic-host, and we'll give our new host the name app:

```
define host{
  use       generic-host
  host_name app
  alias     app
  address   33.33.13.37
}
```

We need to check ssh, so we'll add a new service definition to that file. The only thing that changes from our localhost service definition is the host name that we're checking:

```
define service {
  use generic-service
  host_name app
  service_description SSH
  check_command check_ssh
}
```

Now we'll run Puppet, which will deploy these new files and restart Nagios. Once that's done we can look at the web interface and there's our new app host with our ssh check.

We know how to run a Nagios script from the command line, we know what it looks like when it succeeds, and we've seen the failure modes for one particular plugin. We've also added our first remote monitor. Next we'll set up monitoring for more MassiveApp services.

Monitoring Remote Services with NRPE

Our app VM is running some services that aren't listening on an external interface; they only listening on 127.0.0.1 since MassiveApp is the only thing that needs to access them. That means that our nagios VM can't use access them directly to see if they're running; something like check_ssh just won't work. What we need is a way to tunnel from nagios over to app and check those services locally.

Fortunately Nagios provides a means to do this via an *addon* named the Nagios Remote Plugins Executor (NRPE). Architecturally, NRPE consists of a *monitor-*

ing host (i.e., our nagios VM) which communicates with a NRPE daemon process on a *remote host* that we want to monitor. The NRPE daemon handles running the Nagios checks and returns the results to the monitoring host.

We can start getting those pieces in place by adding another package, nagios-nrpe-plugin to our nagios::server class and running Puppet:

```
package {
  ["nagios3", "nagios-nrpe-plugin"]:
    ensure => present
}
```

That puts the monitoring host plugin in place. Now over on app we can set up the other end by defining a nagios::client class in modules/nagios/manifests/client.pp and requiring that the NRPE daemon and the Nagios plugins be installed. We need the NRPE daemon so that the monitoring host will have something to contact, and we'll need the Nagios plugins since those provide the logic for checking all the resources.

```
class nagios::client {
  package {
    ["nagios-nrpe-server","nagios-plugins"]:
      ensure => present
  }
}
```

We also need to include this class in our node definition, so we'll open manifests/nodes.pp on app and add an include directive:

```
include nagios::client
```

We can run Puppet on app and the NRPE daemon will be installed and started. That's not quite enough though; NRPE doesn't know which machine we're running our Nagios server on. We need to modify the NRPE configuration file so that it knows which machine should be allowed to connect and request resource status information. Let's copy /etc/nagios/nrpe.cfg into modules/nagios/files/nrpe.cfg and change the allowed_hosts variable to 33.33.13.38:

```
allowed_hosts=33.33.13.38
```

We want the NRPE daemon to restart when we make configuration file changes like this one, so let's set up the NRPE configuration file in Puppet the same way we did with Apache in *Restarting Apache on Configuration Changes*, on page 37. Here's that same package-file-service pattern for the NRPE configuration file and the NRPE daemon:

```
package {
  ["nagios-nrpe-server","nagios-plugins"]:
```

```
    ensure => present,
    before => File["/etc/nagios/nrpe.cfg"]
}

file {
  "/etc/nagios/nrpe.cfg":
    owner   => root,
    group   => root,
    mode    => 644,
    source  => "puppet:///modules/nagios/nrpe.cfg"
}

service {
  "nagios-nrpe-server":
    ensure    => true,
    enable    => true,
    subscribe => File["/etc/nagios/nrpe.cfg"]
}
```

A Puppet run will move over our modified nrpe.cfg and restart the NRPE daemon:

```
app $ sudo puppet apply --verbose manifests/site.pp
info: Applying configuration version '1321765051'
info: FileBucket adding \
  {md5}a82e7fc5d321a0dd0830c2981e5e5911
info: /Stage[main]/Nagios::Client/File\
  [/etc/nagios/nrpe.cfg]: Filebucketed /etc/nagios/nrpe.cfg \
  to puppet with sum a82e7fc5d321a0dd0830c2981e5e5911
notice: /Stage[main]/Nagios::Client/File\
  [/etc/nagios/nrpe.cfg]/content: content changed \
  '{md5}a82e7fc5d321a0dd0830c2981e5e5911' to '{md5}c49d23685c60ca537e750349ae26e599'
info: /etc/nagios/nrpe.cfg: Scheduling refresh of \
  Service[nagios-nrpe-server]
notice: /Stage[main]/Nagios::Client/Service\
  [nagios-nrpe-server]: Triggered 'refresh' from 1 events
notice: Finished catalog run in 0.40 seconds
```

We can verify that the NRPE daemon on app is accessible from nagios by running the NRPE plugin from a shell. We'll invoke this plugin just like we did with check_ssh; we need to pass in at least a host parameter:

```
nagios $ /usr/lib/nagios/plugins/check_nrpe -H app
NRPE v2.12
```

That looks promising. We can see all the plugins that NRPE is configured run out of the box by looking at our nrpe.cfg on app:

```
app $ grep "^command\[" modules/nagios/files/nrpe.cfg
command[check_users]=/usr/lib/nagios/plugins/check_users -w 5 -c 10
command[check_load]=/usr/lib/nagios/plugins/check_load -w 15,10,5 -c 30,25,20
command[check_hda1]=/usr/lib/nagios/plugins/check_disk -w 20% -c 10% -p /dev/hda1
```

```
command[check_zombie_procs]=/usr/lib/nagios/plugins/check_procs -w 5 -c 10 -s Z
command[check_total_procs]=/usr/lib/nagios/plugins/check_procs -w 150 -c 200
```

Back on nagios let's give that first plugin, check_users, a whirl from the shell. check_users will monitor the number of logged in users on a host and alert if there are too many, so it shouldn't be alerting yet:

```
nagios $ /usr/lib/nagios/plugins/check_nrpe -H app -c check_users
USERS OK - 1 users currently logged in |users=1;5;10;0
```

The check is able to run on app when we run it manually; now we'll configure Nagios to run that check automatically. Let's edit modules/nagios/files/hosts/app.cfg and add a new service check. Normally the command name is the same as the plugin name. In this case, though, the check_nrpe plugin defines two commands. We can see these two command definitions in /etc/nagios-plugins/config/check_nrpe.cfg; there's one command defined for when we pass arguments to the plugin and one when we don't. In this case we're calling check_users and not passing any arguments, so we'll use check_nrpe_1arg. It's "1 arg" because the host name to check is always passed to the check command:

Download monitoring/check_nrpe_commands.cfg

```
# this command runs a program $ARG1$ with arguments $ARG2$
define command {
  command_name  check_nrpe
  command_line  /usr/lib/nagios/plugins/check_nrpe \
    -H $HOSTADDRESS$ -c $ARG1$ -a $ARG2$
}

# this command runs a program $ARG1$ with no arguments
define command {
  command_name  check_nrpe_1arg
  command_line  /usr/lib/nagios/plugins/check_nrpe \
    -H $HOSTADDRESS$ -c $ARG1$
}
```

With that background we can add our service check in modules/nagios/files/conf.d/hosts/app.cfg:

```
define service {
  use generic-service
  host_name app
  service_description Current Users
  check_command check_nrpe_1arg!check_users
}
```

We'll rerun Puppet to get our configuration files in place; now we can look back at the HTML interface and there's our "Current Users" service check showing green for app.

We can run local and remote checks, so our Nagios coverage is improving quickly. Next we'll define a new service check for memcached that uses the NRPE daemon that we just configured.

Monitoring Memcached Remotely

We need a monitoring check for memcached that uses NRPE, so over on gangliaapp let's open up modules/nagios/files/nrpe.cfg and add the following command definition:

```
command[check_memcached]=/usr/lib/nagios/plugins/check_tcp \
  -p 11211 -e "VERSION" -E -s "version\n" -w2 -c5
```

This service check uses the check_tcp plugin to check the memcached status. check_tcp is a flexible plugin that we can use to check many different TCP-based services. In this case, we're sending in quite a few options:

- The -p 11211 parameter is a mandatory port argument; check_tcp needs to know what port we're checking.
- -e "VERSION" tells the plugin what to expect in return
- -E tells check_tcp to send a carriage return after sending quit to close the connection.
- -s "version\n" indicates what text to send once the plugin connects to the port
- -w2 -c5 triggers a warning status if memcached doesn't respond in 2 seconds and a critical state if it doesn't respond in 5 seconds

Let's run Puppet on app to get that new command definition into NRPE's command set. Once that's done, over on nagios we can add this check into our app configuration in modules/nagios/files/conf.d/hosts/app.cfg:

```
define service {
  use generic-service
  host_name app
  service_description memcached
  check_command check_nrpe_1arg!check_memcached
}
```

Now we can run Puppet on nagios and we can see our new app memcached check in place in the HTML interface.

We've gotten two things out of crafting this check. First, we can see that Nagios can check not only that a service is up but also that it's properly answering queries. Second, we can see that we won't always need to download (or write) a plugin for a new service; in many cases we can build one using Nagios' existing built-in plugins.

Next we'll look at a more Rails-specific check that will tell us if Passenger memory usage is getting out of hand.

Monitoring Passenger

We're running MassiveApp with Passenger, and though of course we've written MassiveApp's code perfectly, there may be memory leaks in the libraries that we're using. In order to catch any problems we'll write a Nagios plugin that checks the memory size of each Passenger process.

This is the first time we're adding a new plugin rather than using an existing plugin. But the testing process is the same. We'll write our plugin, run it a few times to ensure it's working, and then integrate it into our Puppet configuration. Passenger is only running on app, so let's start there and put our plugin in /usr/lib/nagios/plugins/check_passenger. The first version will just require the appropriate libraries and display a dump of the Passenger process list. Passenger comes with a utility class that supplies us with the process information, so we'll use that here:

Download monitoring/check_passenger_print

```
#!/usr/local/bin/ruby

require 'rubygems'
gem 'passenger'
require 'phusion_passenger'
require 'phusion_passenger/platform_info'
require 'phusion_passenger/admin_tools/memory_stats'
require 'optparse'

include PhusionPassenger

puts AdminTools::MemoryStats.new.passenger_processes.inspect
```

When we run this on our VM we get a jumbled list of Object#inspect() output. It's not pretty, but now we know we're able to find the Passenger processes:

```
app $ ruby /usr/lib/nagios/plugins/check_passenger
[#<PhusionPassenger::AdminTools::MemoryStats::Process:0x000000025ba200 @pid=22218,
«many more lines»
```

We want to only flag processes with high memory usage, so we'll need to tell the script what constitutes "high" memory usage. We can do this with arguments which we'll parse using the Ruby standard library's optparse capabilities:

Download monitoring/check_passenger

```
options = {:warning_threshold => 80, :critical_threshold => 100}
OptionParser.new do |opts|
  opts.banner = "Usage: check_passenger [options]"
```

```
  opts.on("-w N", "--warning", "Warning threshold in MB") do |w|
    options[:warning_threshold] = w.to_i
  end
  opts.on("-c N", "--critical", "Critical threshold in MB") do |c|
    options[:critical_threshold] = c.to_i
  end
end.parse!
```

We'll also need some code to search the Passenger process list for large processes. Each object in that list is an instance of the Ruby standard library Process class and has an rss() accessor which tells us how much memory it's using. This code also includes the Nagios-specific bits, that is, the output and the exit code. The output is important because Nagios will display it in the web interface. The exit code is important since Nagios uses that to determine the monitored item's state — ok, warning, or critical.

Download monitoring/check_passenger

```
msg, exit_code = if procs.find { |p| p.rss > options[:critical_threshold]*1000 }
  count = procs.count {|p|p.rss > options[:critical_threshold]*1000 }
  ["CRITICAL - #{count} #{pluralize('instance', count)} \
    #{singularize('exceed', count)} #{options[:critical_threshold]} MB", 2]
elsif procs.find {|p| p.rss > options[:warning_threshold]*1000 }
  count = procs.count {|p|p.rss > options[:warning_threshold]*1000 }
  ["WARNING - #{count} #{pluralize('instance', count)} \
    #{singularize('exceed', count)} #{options[:warning_threshold]} MB", 1]
else
  ["OK - No processes exceed #{options[:warning_threshold]} MB", 0]
end
```

Our complete check will conform to Nagios' requirements by printing a message and exiting with the appropriate status code:

Download monitoring/check_passenger

```
#!/usr/local/bin/ruby

require 'rubygems'
gem 'passenger'
require 'phusion_passenger'
require 'phusion_passenger/platform_info'
require 'phusion_passenger/admin_tools/memory_stats'
require 'optparse'

include PhusionPassenger

def pluralize(str, count)
  if count > 1
    "#{str}s"
  else
    str
  end
```

```
end

def singularize(str, count)
  if count > 1
    str
  else
    "#{str}s"
  end
end

options = {:warning_threshold => 80, :critical_threshold => 100}
OptionParser.new do |opts|
  opts.banner = "Usage: check_passenger [options]"
  opts.on("-w N", "--warning", "Warning threshold in MB") do |w|
    options[:warning_threshold] = w.to_i
  end
  opts.on("-c N", "--critical", "Critical threshold in MB") do |c|
    options[:critical_threshold] = c.to_i
  end
end.parse!

procs = AdminTools::MemoryStats.new.passenger_processes

msg, exit_code = if procs.find { |p| p.rss > options[:critical_threshold]*1000 }
  count = procs.count {|p|p.rss > options[:critical_threshold]*1000 }
  ["CRITICAL - #{count} #{pluralize('instance', count)} \
    #{singularize('exceed', count)} #{options[:critical_threshold]} MB", 2]
elsif procs.find {|p| p.rss > options[:warning_threshold]*1000 }
  count = procs.count {|p|p.rss > options[:warning_threshold]*1000 }
  ["WARNING - #{count} #{pluralize('instance', count)} \
    #{singularize('exceed', count)} #{options[:warning_threshold]} MB", 1]
else
  ["OK - No processes exceed #{options[:warning_threshold]} MB", 0]
end

puts "PASSENGER #{msg}"
exit exit_code
```

And here's a test run with a low warning threshold to ensure we'll see some output:

```
app $ ruby /usr/lib/nagios/plugins/check_passenger -w 10
PASSENGER WARNING - 1 instance     exceeds 10 MB
```

Our check works, so the next step is to add it to our Puppet Nagios module. We want to invoke the plugin via NRPE, so as with our memcached check we'll need to make changes on both app and nagios. We'll put check_passenger in a new modules/nagios/files/plugins/ directory in our module. Then we'll add another file resource to modules/nagios/manifests/client.pp that will move our script into place:

Download monitoring/check_passenger.pp

```
"/usr/lib/nagios/plugins/check_passenger":
   source  => "puppet:///modules/nagios/plugins/check_passenger",
   owner   => nagios,
   group   => nagios,
   mode    => 755;
```

Now we'll define a new Nagios command that points to our script. We'll put the command definition in our Nagios Puppet module in modules/nagios/files/conf.d/commands.cfg, and for the command definition itself we just need to name it and give Nagios the command's path:

Download monitoring/commands.cfg

```
define command {
  command_name check_passenger
  command_line /usr/lib/nagios/plugins/check_passenger
}
```

We also need to tell NRPE to make the new check available, so we'll add a new command to modules/nagios/files/nrpe.cfg:

```
command[check_passenger]=/usr/lib/nagios/plugins/check_passenger -w 10
```

The last piece of the puzzle is to configure nagios to run this new command as part of the app checks. This is another service definition like the memcached definition in *Monitoring Remote Services with NRPE*, on page 93, and we'll put this in app.cfg:

Download monitoring/nagios-service-definitions

```
define service {
  use generic-service
  host_name app
  service_description Passenger
  check_command check_nrpe_1arg!check_passenger
}
```

Remember how we used the Puppet recurse attribute for managing the conf.d file resource? Since that's in place we can rerun Puppet, the new Nagios configuration file will be moved into place, and Nagios will be restarted.

And that's it. With those elements in place, a new "Passenger" service shows up in the web interface, and Nagios will check Passenger every few minutes and alert us if any Passenger processes are exceeding our memory thresholds. Currently it's warning since we set the threshold to a low value, but we can easily bump that up in modules/nagios/files/nrpe.cfg to a more reasonable level.

We've seen how to monitor servers and services with Nagios. Next, we'll look at monitoring MassiveApp itself.

6.6 Monitoring Applications

The monitoring techniques we've seen have all been useful with any application; check_disk is helpful for a Ruby on Rails application, a Java application, or a static web site. But Nagios can also monitor an application's specific data thresholds. Let's see how that's done.

We've seen how Nagios uses plugins to bring in new functionality and we've written our own plugin to monitor Passenger memory usage. Now we'll write another small plugin to check MassiveApp's activity. Actually, most of the logic will be in MassiveApp itself; we'll write just enough of a plugin to connect to MassiveApp and report a result.

For the specifics of this check, consider MassiveApp's daily growth rate in terms of accounts. In any 24 hour span we get around a dozen new accounts. If we get many more than that, we want to get an alert so we can think about firing up more servers.

We could do this check in a few different ways. We could query the MySQL database directly, but that would mean we couldn't use our ActiveRecord models with their handy named scopes and such. We could use HTTP to hit a controller action, but then we'd want to ensure that the action could only be accessed by Nagios. So we'll keep it simple by using a Rake task. First we'll declare the task; we can put this in lib/tasks/monitor.rake. We're namespacing the task inside nagios; this keeps all our Nagios-related tasks in one place:

Download monitoring/monitor1.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
  end
end
```

Next let's count the number of "recently" created accounts; in this case "recently" means "in the past 24 hours". We can do this with a straightforward ActiveRecord query:

Download monitoring/monitor2.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
    recent = Account.where("created_at > ?", 1.day.ago).count
  end
end
```

In this plugin we'll ask for a warning if we get more than 50 new accounts in a day and we'll consider it critical if we get more than 90 in a day.

Download monitoring/monitor.rake

```
namespace :nagios do
  desc "Nagios monitor for recent accounts"
  task :accounts => :environment do
    recent = Account.where("created_at > ?", 1.day.ago).count
    msg, exit_code = if recent > 90
      ["CRITICAL", 2]
    elsif recent > 50
      ["WARNING", 1]
    else
      ["OK", 0]
    end
    puts "ACCOUNTS #{msg} - #{recent} accounts created in the past day"
    exit exit_code
  end
end
```

We've got the Rake task in MassiveApp's codebase now; next up, we need Nagios to be able to run it. We can do this with a simple Bash script. We'll name this script check_recent_accounts and put in our nagios Puppet module in modules/nagios/files/plugins/ alongside our check_passenger plugin. That script needs to run our Rake task using the --silent flag to prevent the usual "(in /path/to/the/app)" output message since that would confuse Nagios. It also needs to relay the exit code from the Rake task on to Nagios. We can do that using the Bash special parameter $? which holds the exit code of the last command executed:

Download monitoring/check_recent_accounts

```
#!/bin/bash
cd /home/vagrant/massiveapp/current/
RAILS_ENV=production /usr/bin/rake --silent nagios:accounts
exit $?
```

Switching back into Puppet mindset, we'll add another file resource to our nagios::client class that will move our script into place:

Download monitoring/check_recent_accounts.pp

```
"/usr/lib/nagios/plugins/check_recent_accounts":
   source  => "puppet:///modules/nagios/plugins/check_recent_accounts",
   owner   => nagios,
   group   => nagios,
   mode    => 755;
```

We'll also need to add the new command to nrpe.cfg:

```
command[check_recent_accounts]=/usr/lib/nagios/plugins/check_recent_accounts
```

Let's run Puppet to get the script and the new nrpe.cfg in place. Then we can execute a trial run of this plugin in the same way that we've exercised other plugins; we'll just run it from the shell:

```
app $ /usr/lib/nagios/plugins/check_recent_accounts
ACCOUNTS WARNING - 70 accounts created in the past day
```

We need to tell Nagios about our new check, so we'll add it to commands.cfg:

Download monitoring/commands.cfg

```
define command {
  command_name check_recent_accounts
  command_line /usr/lib/nagios/plugins/check_recent_accounts
}
```

And we'll add this to our app checks:

Download monitoring/nagios-service-definitions

```
define service {
  use generic-service
  service_description Recent Accounts
  host_name app
  check_command check_nrpe_1arg!check_recent_accounts
}
```

Another Puppet run and everything's in place; now Nagios will let us know if (or when) we get a mad rush of new users.

This is the sort of check that only needs to be run once for MassiveApp. That is, when MassiveApp grows to encompass a few servers, we won't run this on each server as we'd do with check_ssh and check_passenger. Instead, we'd designate one host to run this check and that would alert us if the thresholds were exceeded.

6.7 Conclusion

In this chapter we discussed what monitoring is and why it's important. We've talked about a particular open source monitoring tool, Nagios, and why we like to use it to monitor our applications. Then we looked at using Nagios to monitor servers, services, and MassiveApp itself.

Nagios is a large system with a lot of functionality, and there are several good books on it; *Nagios: System and Network Monitoring* [Bar08] is excellent, as is *Learning Nagios 3.0* [Koc08]. We've already seen the Nagios Puppet module and have a functioning monitoring system, so between a book or two, the Internet, and the system we have in place we can get a fairly capable monitoring setup.

Here are a few closing thoughts on monitoring in general. Maintaining and enhancing a monitoring infrastructure may not initially appear to be an exciting task. As the monitoring people, most of the time we're tweaking thresholds, adding monitors that may alert us once every six months, or navigating the somewhat less than impressive web interfaces that most monitoring tools seem to have. There's not a lot of sizzle in this kind of work.

But that said, keeping a monitoring system up to date is an important part of deploying our applications. If we're familiar with our monitoring tools we'll be able to respond to outages by thinking about what checks we can add that will prevent that type of downtime. When our application gets new functionality or rolls out a new component, we'll be in a good frame of mind to add monitors preemptively to ensure the usage patterns are what we expect. Being on the team that keeps the monitoring infrastructure running gives us many windows into the entire system, and that's a great perspective to have.

6.8 For Future Reference

Organizing Nagios Files

Nagios operates in a host/agent model, with a central host monitoring other hosts which are running Nagios' command execution agent, NRPE. Nagios provides a web interface to view system status, but also sends alerts in a variety of ways. Nagios monitors hosts, which are organized into hostgroups, using commands which check services. New commands can be implemented as plugins, which are simply programs that can be written in any language as long as they conform to the Nagios output format.

Defining Nagios Objects

Nagios is configured via a directory tree of configuration files in which templates declare a variety of objects such as hosts, commands, and services. A sample host definition includes the host name and IP address and specifies a parent template which provides default values:

```
define host{
  use       generic-host
  host_name app
  alias     app
  address   33.33.13.37
}
```

We use a service object to define a service check:

```
define service {
  use generic-service
```

```
  host_name app
  service_description SSH
  check_command check_ssh
}
```

Checking a remote service which can only be accessed locally requires NRPE. Here's a check parameter which uses NRPE:

```
check_command check_nrpe_1arg!check_users
```

Adding a new check to NRPE also requires that the command be defined in NRPE's configuration file, nrpe.cfg. Here's an example:

```
command[check_passenger]=/usr/lib/nagios/plugins/check_passenger -w 10
```

Checking Rails with Nagios

For checks specific to Rails applications, plugins can be written as Rake tasks that print a message such as ThingToCheck WARNING - 42 somethings occurred. This output message will be displayed in the Nagios interface. The task should also exit with a status code of 0 for an OK status, 1 for a WARNING status, and 2 for a CRITICAL status. Such a task can then be invoked with a Bash script as follows:

```
#!/bin/bash
cd /path/to/apps/current/
RAILS_ENV=production /usr/bin/rake --silent task:name
exit $?
```

This script can be invoke via NRPE by defining a new command in nrpe.cfg.

CHAPTER 7

Collecting Metrics with Ganglia

Coming soon.

CHAPTER 8

Maintaining the Application

Coming soon.

CHAPTER 9

Running Rubies with RVM

Coming soon.

CHAPTER 10

Special Topics

Coming soon.

APPENDIX 1

Bibliography

[Bar08] Wolfgang Barth. *Nagios: System and Network Monitoring.* No Starch Press, San Francisco, CA, 2nd, 2008.

[Koc08] Wojciech Kocjan. *Learning Nagios 3.0.* Packt Publishing, Birmingham, UK, 2008.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/titles/cbdepra
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/titles/cbdepra*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764