Deploying Rails Applications
A Step-by-Step Guide

Ezra Zygmontowicz
and Bruce Tate
with Geoffrey Grosenbach

The Facets of Ruby Series
The book you’re reading is still under development. As an experiment, we’re releasing this copy well before we normally would. That way you’ll be 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. And there’s been no effort spent doing layout, so you’ll find bad page breaks, over-long lines, incorrect hyphenations, and all the other ugly things that you wouldn’t expect to see in a finished book. 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!

Throughout this process you’ll be able to download updated PDFs from http://books.pragprog.com/titles/fr_deploy/reorder. When the book is finally ready, you’ll get the final version (and subsequent updates) from the same address. In the meantime, we’d appreciate you sending us your feedback on this book at http://books.pragprog.com/titles/fr_deploy/errata.

Thank you for taking part in this experiment!

► Andy and Dave
# Contents

1 Introduction .................................................. 8
   1.1 The Lay of the Land .................................... 11
   1.2 Finding a Home ......................................... 14
   1.3 Conventions ............................................. 17
   1.4 Acknowledgements ...................................... 18

2 Refining Applications for Production .......................... 19
   2.1 The Lay of the Land .................................... 19
   2.2 Source Code Management .............................. 20
   2.3 Subversion Tips ........................................ 28
   2.4 Stabilizing your Applications ....................... 29
   2.5 Active Record Migrations ............................ 32
   2.6 Application Issues for Deployment .................. 37

3 Shared Hosts .................................................. 42
   3.1 The Lay of the Land .................................... 42
   3.2 Choosing a Shared Host ............................... 45
   3.3 Setting Up your Domain and DNS .................... 48
   3.4 Configuring your Server ............................. 49
   3.5 Server Setup: Create a Database ................... 50
   3.6 Installing your Application .......................... 52
   3.7 Configuring your web server .......................... 54
   3.8 Application Setup: Rails Config Files .............. 58
   3.9 The Well-Behaved Application ....................... 60
   3.10 Troubleshooting Checklist ........................... 62
   3.11 Conclusion ............................................. 68

4 Virtual and Dedicated Hosts ................................ 70
   4.1 The Lay of the Land .................................... 70
   4.2 Virtual Private Servers ............................... 73
   4.3 Dedicated Servers ..................................... 75
   4.4 Setting up Shop ....................................... 76
## 4.5 Wrapping Up ................................................. 88

## 5 Capistrano .................................................. 90
5.1 The Lay of the Land ........................................... 91
5.2 How It Works ................................................ 93
5.3 Local and Remote Setup for Rails ......................... 95
5.4 Standard Recipes ............................................ 104
5.5 Writing Tasks ............................................... 107
5.6 A Little Extra Flavor ....................................... 117
5.7 Troubleshooting ............................................. 120
5.8 Conclusion ................................................... 121

## 6 Managing Your Mongrels ..................................... 123
6.1 Training your Mongrels ....................................... 123
6.2 Obedience Training for Mongrel with Monit .............. 129
6.3 Keeping FastCGI Under Control ......................... 133
6.4 Building in Error Notification ............................... 135
6.5 Heartbeat ..................................................... 138

## 7 Scaling Out .................................................. 141
7.1 The Lay of the Land .......................................... 141
7.2 Scaling Out ................................................ 142
7.3 Mirror Images ............................................... 146
7.4 Domain Names and Hosts .................................. 148
7.5 Deploying to Multiple Hosts ................................. 150
7.6 Apache ....................................................... 156
7.7 Nginx, from Russia with Love. ............................. 167
7.8 Clustering MySQL ........................................... 174
7.9 Summary ..................................................... 185

## 8 Deploying on Windows ........................................ 187
8.1 Setting up the server ......................................... 187
8.2 Mongrel ....................................................... 191
8.3 Mongrel and Pen ............................................ 196
8.4 Using Apache 2.2 and Mongrel ............................. 199
8.5 IIS Integration ............................................... 203
8.6 Reverse Proxy and URLs .................................... 206
8.7 Strategies for hosting multiple applications .......... 207
8.8 Load-testing your applications. ............................. 214
8.9 Final Thoughts ............................................... 214
8.10 Developing on Windows and deploying somewhere else 216
8.11 Wrapping up ............................................... 218
Building Rails apps brings the joy back into development. But I have a confession to make. There was a brief moment that I didn’t like Rails at all.

I’d just graduated from the five minute tutorial to develop my first real Rails application. The helpers, plugins, and generators reduced the amount of code I needed to write. The logical organization and layout of the files let me painlessly find what I needed, and the domain specific languages in Active Record let me express my ideas with simplicity and power. The framework bowed to my will, and aside from a few trivial mistakes, I finished the app. Pure joy washed over me.

But then, it was time to deploy. Deployment means moving your application from a development environment into a home that your customers can visit. For a web application, that process involves choosing a host, setting up a web server and database, and moving all of your files to the right places with the right permissions.

I quickly discovered after the joy of development, deployment was a real drag. All of those waves of euphoria completely disintegrated against the endless stream of crash logs, Rails error pages, and futile install scripts. I spent hours wading through the Rails wikis, blogs, and books for answers, but each one gave me a mere fragment of what I needed. Much of the information I found was contradictory or flat out wrong.

Deployment also involves making the best possible environment for your customers, once you’ve settled into your new home. There, too, I failed miserably. When I finally made my site work, it was too slow. Stumbling through page caching seemed to make no difference, and my end users watched the spinning (lack of) progress indicator in frus-
tration. I struggled to fix memory leaks, broken database migrations, and worthless server configurations until eventually my site purred in appreciation. Then came success, which means more visitors, followed by more failure. I screamed some choice words that would make a sailor’s dead parrot blush. No, at that moment, I really didn’t like Rails.

I’m not going to sugar coat it. If you don’t know what you’re doing, Rails deployment can stretch the limits of your patience, even endurance. What’s worse, Rails deployment suffers especially in areas where Rails development is easy:

- You can always find plenty of Rails development documentation, but when it’s time to deploy, you can often find only a fraction of what you need. People just seem to write more about development than deployment.

- You can choose your development platform, but you can’t always choose your deployment platform. Most hosts with Rails support run some variant of Linux, some others run FreeBSD or Solaris. And the software stack for different hosts can vary wildly, as can application requirements.

- When your development application breaks, you can find mountains of information through breakpointing, rich development logs, and the console. In production, when things go south, there are fewer sources of information, more users, and more variables. You might encounter a problem with the operating system, your application server, system resources, plug-ins, your database server, or any one of dozens of other areas. And your caching environment works differently than your development environment.

- Rails is an integrated platform that keeps choices down. You’ll probably use Active Record for persistence and Action Pack for your controllers and views. You’ll use Script.aculo.us and Prototype for Ajax. But your deployment environment will require many choices that are not dictated by Rails, including the most basic choice of your web server.

But I’m living proof that you can learn to master this beast. Over time, I’ve come to understand that my approach to deployment was rushed, and a little haphazard. I found that I needed to approach deployment in the same way that I approached development. I had to learn how to do it well, effectively plan each step, and automate as much as possible so
I left little to chance. I needed to plan for problems, so I could anticipate them, and get automatic notification at the first sign of trouble. At my company, Engine Yard, I support some of the largest and most popular Rails sites in the world. I want to help you learn to do the same.

Because Rails is so new, some people question whether anyone can deploy a sophisticated, scalable, and stable Rails application. Based on my experience at Engine Yard, I’d like to first debunk a few myths:

**Myth: The Ruby on Rails development framework is much more advanced than the deployment framework.**

That’s false. Deployment tools for Rails get much less attention, but they are also growing in form and function. If you know where to look, you can find deployment tools that are proven, effective, and free to use. These tools use techniques that are every bit as advanced and functional as those used by the most mature Java or C# development shops. Ruby admins can deploy a typical Rails application with one command, and move back to a previous release should that deployment fail, again with one command. You can deploy Rails to simple single-server setups or multi-server sites with very few changes. And if you now copy PHP files to your server by hand or rsync Perl scripts to multiple machines, your life is about to become a lot easier (and yes, you can use some of these same tools as well). I’ll show you how to do these things in Chapter 5, Capistrano, on page 90.

**Myth: Rails is too new to have any large, sophisticated deployments.**

That, too, is false. Ruby on Rails is in use on very large sites that are spread across multiple machines. Some of those applications require many full servers just to serve their full feature set to their community. And the list of large Rails sites grows daily. Twitter, Base Camp, and 43 Things are all multi-server large Rails sites. Many more enter production every month.

**Myth: The Ruby language is inherently unstructured, and poorly suited for web applications.**

That’s mostly false. Ruby is an interpreted, dynamically typed language that presents real challenges in high-volume production settings, but the Rails framework has features and strategies that mitigate many risks associated with these challenges. The Rails caching model and performance benchmarking tools help developers build high performance sites. The Rails testing frameworks, sometimes in combination...
with other Ruby testing frameworks, help developers catch errors that a compiler might catch in a statically typed language. And the Rails shared-nothing architecture, like many of the highest volume Internet sites in existence, allows Rails sites to scale by adding additional hardware. You’ll learn how to cluster in chapter Chapter 7, Scaling Out, on page 141.

Myth: Rails can get you into trouble if you don’t know what you’re doing.

That one is true. If you want to stay out of serious trouble, you need to know how to wield your chosen tools. No development language is immune to bad design. And a poor deployment strategy will burn you. You must always arm yourself with knowledge to protect yourself. In this book, I hope to help you do exactly that.

Rest assured that the Rails deployment story is a good one. You can learn to predictably and reliably deploy your applications. You can use repeatable techniques to understand what the performance characteristics of your system are likely to be. And you can improve the stability and scalability of your system given knowledge, time, and patience. I’m going to start quickly. I want to walk you through the same deployment road map that every Internet application will need to use.

1.1 The Lay of the Land

Web 2.0, the new buzzword that describes a new class of web applications, sounds like a daunting mix of new technologies that radically change the way you think about the Internet. But when you think about it, from a deployment perspective Web 2.0 doesn’t change too much at all.

- The Internet still uses the same communication protocols and the same type of web servers.
- You still scale Internet applications the same way, by clustering.
- You can even use some of the same servers, and the new ones work mostly like the old ones.
- You still keep your source code in source control.
- The operating system is still usually Unix-based.

For all of the talk about the way your applications may change, deployment remains precisely the same. Think of the Internet as a road map.
The buildings and places are servers, browsers on clients, routers, firewalls, and balancers. The roads are the networks between them and the various communication protocols those networks use. I like the map analogy because when all is said and done, the Internet is all about moving data from one place to another.

When you deploy, you’re using the Internet to move your application from one place to another. You can think of every deployment story as a map. In fact, every deployment story in this book will come with a map. Figure 8.1, on page 194 shows a generic version of the simplest possible deployment story.

Look at the components of that figure. First, you have a host with an environment. I’ll spend much of this book showing you how to build the environment that will host your application. The environment, in this case, includes all of the different components that an Internet application needs. You’ll learn to build each of these pieces yourself, or rely on another vendor to build those pieces for you. Those pieces will include the operating system, the Ruby language, the Rails framework, and various pieces that will tie them together. The host represents where your customers will go to find your application. As you can well imagine, that host image will get much richer as I take you through the various pieces of this book.

You also see a development client. My machine is my trusty MacBook Pro, but I’ve also developed on the Windows platform. You might think that this book is about the road that goes from the application on the client to the server. And I’ll start the book that way. Figure 1.1, on the next page will use plain old FTP to move your application from the client to the server.

Deployments are rarely as simple as the one you see in Figure 1.1, on the following page. You’re going to find that shared hosting is a little limited. And you probably know that plain old FTP may be simple, but it will not handle the demands of effectively managing the site. You will need better deployment tools. You will want to throw a source control repository into the mix. If you’re lucky, one web server may not be enough. You’ll wind up with a more sophisticated map, like the one in Figure 7.1, on page 144.

In the complete map, you see a vastly different story. What may appear as one site to the user has its own environments. The first change is the web site. You can no longer assume a single host. Those environments
Figure 1.1: Basic Deployment Map
might be virtual environments which all reside on a single machine, or each individual environment could have its own hardware. Your deployment strategy will have to install your application into each Rails environment. You will need to configure the pieces to work together. And that’s the subject of this book.

In the first few chapters, you might think that we’re oversimplifying a little bit. Don’t worry. You’re not going to be using FTP or shared hosting by the time you finish this book. I’ll get to the second map. I’ll just build it slowly, one piece at a time. We’ll keep extending the map throughout the book until you get to your eventual goal. In the next section, I’ll walk you through what you can expect in the chapters to follow.

1.2 Finding a Home

You’ve seen that our maps have one goal in mind. They want to get your development code to its eventual home in the best possible way. By now, you also know that the type of map that you need depends on where your application is going to live. You can’t adequately understand deployment unless you understand where you’re going to put your code, but finding a platform for your Rails code is hard. The process feels like finding a home without a real estate agent, the Internet, or any consolidated home buyers guides. Over the course of this book, I’d like to take you into that hidden universe. You will learn how to:

- develop Rails applications with painless deployment in mind;
- choose between shared hosts, virtual private servers, or dedicated servers;
- understand the software stack that the pros use to deploy Rails for high performance;
- build and configure your web servers and other services;
- stress your application before your users do;
- and streamline your application in production using advanced strategies like caching so your site can scale.

Throughout this book, I’ll treat deployment like buying a new home for your application. Through each of the chapters, you’ll learn to pick and prepare your home, streamline your stuff for everyday living, and even move up into wealthier neighborhoods, should you ever need to do so. Let me take you on a guided tour of the book:
Finding a Home

Packing Up: Tending to your Application. Before you can move, you need to pack up. If you want a good experience, you need to organize your stuff to prepare for your move. On Rails, that means minding your application. You will need to prepare source control. You will also need to make some important decisions that will have a tremendous impact on your production application, such as the structure of your migrations and your attention to security. This chapter will add source control to your map.

Finding a Starter Home: Shared Hosting. Not everyone can afford a house. When most of us leave home, we first move into an apartment building or a dorm. Similarly, most Rails developers will choose some kind of shared hosting to house that first application: a blog, or a simple photo log. Shared hosting is the first and cheapest of the hosting alternatives. Setting up shared hosting involves many of the same steps as moving into your first apartment: find a home that meets your requirements, set up your address so that others can find you, and customize your home as much as possible. Like apartment living, shared hosting has its own set of advantages and disadvantages. Shared hosting is cheap, but you need to learn to be a good citizen, and you’ll also likely encounter those who aren’t. In this chapter, you’ll learn to find and make the best use of your first home. The deployment will be simple. You’ll need a shared host, a simple application, and a simple mechanism like FTP to ship your code up there.

Moving Up: Virtual and Dedicated Hosting. After you’ve lived in an apartment for a while, you might decide to move up to your own home or condo. Your virtual world is the same. When shared hosting isn’t enough, you can move up to virtual and dedicated hosts. Moving up to a home carries a whole new set of benefits and responsibilities: you get more freedom to add that extra closet you’ve always wanted, but you also have to fix the toilet and mow the lawn yourself. Dedicated and virtual hosts are like your own home or condo. These plans are typically more robust than shared hosts, but also require much more knowledge and responsibility. When you set up your own host, you take over as landlord. You need to know how to build and configure your basic software stack from your web server to the Rails environment. This chapter will walk you through building out your hosting platform. Your map will get a little more complicated because you’ll have to build your environment, but otherwise, it will be the same.
Moving In: Capistrano. After you’ve chosen and prepared a place, you can move in. Unlike moving in your furnishings, with Rails, you will probably move in more than once. You’ll want to make that move-in process as painless as possible, automating everything you possibly can. Capistrano is the Rails deployment tool of choice. In this chapter, you’ll learn to deploy your application with Capistrano using existing recipes with a single command. You’ll also learn to roll back the deployment to the previous version if you fail. You will also see many of Capistrano’s customization tools. This chapter will change your map by building a better road between your application and the deployment environment.

Adding On: Proxies and Load Balancing. When your place is no longer big enough, you need to add on or move up. Since we’ve already covered moving up, this chapter will cover adding on through clustering. One of the most common and effective ways to remodel a Rails deployment without buying a bigger plan is to separate the service of static content and application-backed dynamic content. In this chapter, you’ll learn to reconfigure your production environments to handle more load. I’ll show you setups with Apache and nginx serving static content and dynamic content with Mongrel. You’ll also learn how to distribute your applications across multiple servers with a rudimentary load balanced cluster. I’ll also walk you through potential database deployments. The host side of your deployment map will get much more sophisticated because you’ll be deploying to a cluster instead of a single host, but with Capistrano already in the bag, you won’t have to change the client side at all.

Planning for the Future: Benchmarking. As you grow older, your family may grow. Without a plan, your house may not be able to accommodate your needs a few years from now. In Rails or any other Internet environment, capacity planning becomes a much larger problem, because your home may need to serve hundreds of times the number of users that it does today. To get the answers you need, you have to benchmark. After you’ve chosen your stack and deployed your application, you’ll want to find out just how far you can push it. In this chapter, you’ll learn to use the base Ruby tools, and a few others, to understand just what your environment can handle. You’ll also learn a few techniques to break through any bottlenecks you do find. The deployment map won’t change at all.
Managing Things: Monitoring. As you live in your new home, you’ll often find that you need help managing your household. You might turn to a watch dog to monitor comings and goings, or hire a service to do it for you. With the many Rails configuration options, you’ll be able to manage some of your installation yourself. You can also use an application called Monit to automatically tell you when a part of your system has failed or is about to fail. You will only make subtle adjustments to your map to allow for the additional monitoring of the system.

Doing Windows: Deploying on Windows. Homeowners hate doing windows. Rails developers often do, too. But sometimes, you don’t have a choice. When you do have to deploy on Windows, this chapter will walk you through the process. We’ll keep it as simple and painless as possible. This chapter will focus on the host side of the map to offer the Windows alternative as you build out your environment on Windows.

When you’ve finished the book, you’ll know how to pick the best platform for you. You’ll understand how to make Capistrano finesse your application from your development box to your target environment. And you’ll be able to configure a variety of deployment scenarios from the inside out. If you’ve built up any resentment for Rails because of deployment problems in the past, this book should get you back on the path to enjoying Rails development again.

1.3 Conventions

Throughout this book, you’ll see several command line terminal sessions that show various deployment, setup, and configuration tasks. You’ll need to make sure you type the right command in the right place. You wouldn’t want to accidentally clobber your local code, or accidentally load your fixtures to your production database (destroying your data in the process!) To be as safe as possible, I will follow a few conventions with the command line prompts to make it easier to follow along.

On most Unix like systems, when the command line prompt is the pound sign #, it is letting you know that you are logged in as root. When the prompt is the dollar sign $, you are logged in as a regular system user. These are the conventions for the Bourne Again Shell (bash). If you are running another shell you might have slightly different indica-
tors in your prompt and you should adjust accordingly. On the Ubuntu system we are about to set up, the default shell is bash.

The following prompts show how you should log in to run the various shell commands we use in the book. When you should be logged in as root, the prompt will look like this:

```
root#
```

When you should be logged in with your regular user account, the prompt will look like this:

```
ezra$
```

When you should be running a command from your local computer and not the server, the prompt will look like this:

```
local$
```

1.4 Acknowledgements
Before you can move into your new house, you need to pack up. With Rails, you need to do the same thing: prepare your Rails application for deployment. You’ll need to organize your code and prepare it for production. You’ll need to think about a few things:

- Making your source code repository work smoothly with your production setup will make your deployments go smoother and more secure.
- Strengthening your brittle migrations can save you from models that change and developers that collide.
- Locking down Ruby, Rails, and Gems code to a single stable version.

Fundamentally, you want to build every application with deployment in mind. The earlier you think about deployment issues, the better off you’ll be. I’m not saying that you need to make early deployment decisions at demand time. You just need to make sure you build intelligent code that is less likely to break in production situations. Your first order of business is to simplify your Subversion setup.

## 2.1 The Lay of the Land

The map shown in Figure 1.1, on page 13 is the first enhancement to the most basic deployment map. The following list shows a list of what you’ll need to accomplish in this chapter.
• Set up source control. If you’ve not already done so, setting up source control will make the rest of your deployment picture much simpler, and will improve your development experience as well.

• Prepare your application configuration and performance. You will make some simple changes to your application, or better yet, build it right the first time.

The three Rails environments–development, test and production–will make it easy for you to isolate configuration for deployment. The rest is just common sense. It all starts with source control.

### 2.2 Source Code Management

Good deployment strategies always start with a good foundation. I want to be able to deploy the same application to my servers with identical results every time. That means that I need to be able to pull a given version of the application from a central source. Anything less won’t
give me the dependable, repeatable results. Luckily, Rails will automate a whole lot of the deployment scripts, but only if you use the common infrastructure that other Rails developers do.

Unless you have a strong reason to use something else, you’ll want to use Subversion for your application’s source code control. The majority of Rails developers use it, and the Rails team uses Subversion for Rails, and the Rails plug-in system also uses Subversion. Version Control with Subversion is a great book about Subversion. You should also check out Pragmatic Version Control for a pragmatic view of source code control in any language. For this chapter, I’m going to assume you already have Subversion installed and running.

**Subversion on Rails**

The keys to using Subversion with Rails are maintaining the right structure. You want to keep the right stuff under source control and keep the wrong stuff out. Setting up your application’s repository right the first time will save you time and frustration. A number of items in a Rails application that do not belong in source control. Many a new Rails developer has clobbered his team’s database.yml, or checked in a 500KB log file. Both of these problems are with the Subversion setup, not Rails or even the Rails developer. In an optimal setup, each developer will have their own database.yml file and the log files. Each development or production instance of your application will have its own version of these files, so they need to stay out of the code repository. You might already have a Subversion repository already, but I’ll assume you don’t, and work you through the entire process from scratch.

**Repository creation**

Start off by creating a new Subversion repository for your Rails project. Log in to the server that will have your Subversion repository. Create a directory for your repository and let Subversion know about it:

```
$ svnadmin create /home/ezra/svn
```

The authors of Subversion recommend creating all repositories with three folders at the root: trunk, tags and branches. This setup works best if you have one project per repository. You don’t have to create the top-level folders for Subversion to work, but I suggest you do so. The

---

1. This is explained in more detail in “Choosing a Repository Layout” in the Subversion book.
better Subversion repositories I have seen adhere to this convention, and if you have only one project in your repository, this approach will let you tag and branch at will. These commands will build your initial directories:

```bash
$ svn mkdir --message="Initial project layout"
   file:///home/ezra/svn/trunk file:///home/ezra/svn/tags
   file:///home/ezra/svn/branches
Committed revision 1.
```

**Importing a Simple Rails Application**

I suggest you practice with an empty Rails project first. Create the Rails application as usual:

```bash
$ rails ~/deployit
create
create app/controllers
... 
```

At this point, you could do an `svn import` and put the whole directory tree in the repository. I recommend against doing so. If you use the ‘in-place import’ procedure instead, you can selectively commit the pieces that you want, not the whole tree. See Subversion's “How can I do an in-place 'import'” FAQ for the full details.

Start your in-place import by checking out `trunk` into the folder you want to import:

```bash
$ svn checkout file:///home/ezra/svn/trunk ~/deployit
Checked out revision 1.
$ cd ~/deployit
```

Next, add the whole tree to the working copy. The results are no different from `svn import` initially, except all changes are local to the working copy, and you can selectively revert the files and folders you do not want in the repository before committing. The end result is more convenient control over what actually becomes part of the repository. Add your Rails project like so:

```bash
$ svn add .
A       app
... 
A       README
```

2. If you have an existing project you want to import in Subversion, simply skip this step. All other steps are identical.
The Rails command helpfully creates most of the tree. Since I will later use migrations in all of my Rails projects, I immediately create the db/migrate folder. Rails 1.1+ also creates a tmp folder when it needs it. For completeness’ sake, I will create both folders immediately.

```
$ svn mkdir db/migrate tmp
```

A `db/migrate`
A `tmp`

Removing the log files from version control

At this point, Subversion would helpfully track all changes to the log files, and some following Friday at 6:30, some poor harried developer would then accidentally check in an obscenely large log file, and the rest of the developers would complain that the checkout was taking way too long. To ease our burden, the easiest thing is to tell Subversion to ignore any log files.

```
$ svn revert log/*
Reverted 'log/development.log'
Reverted 'log/production.log'
Reverted 'log/server.log'
Reverted 'log/test.log'

$ svn propset svn:ignore "*.log" log
property 'svn:ignore' set on 'log'
```

That’s all there is to it. Next stop: database.yml.

Managing the database configuration

Since database.yml might be different for each developer, you do not want to create havoc by accidentally committing database.yml. Instead, you’ll have a sample of the file in the repository so each developer will have their own safely ignored database.yml. These commands do the magic:

```
$ svn revert config/database.yml
Reverted 'config/database.yml'

$ mv config/database.yml config/database.yml.sample
$ svn add config/database.yml.sample
A config/database.yml.sample

$ svn propset svn:ignore "database.yml" config
property 'svn:ignore' set on 'config'
$ cp config/database.yml.sample config/database.yml
$ svn status --non-recursive config/
A config
A config/routes.rb
Joe Asks...  

What about the deployed database.yml?

Using the template file technique means the database.yml is not under version control on your production server. Some solutions to this problem are:

- Use a branch to deploy, and keep database.yml under version control in the branch. See Section 2.2, *Using a stable branch for deployment*, on page 26 for how to do that.

- Have Capistrano copy the file forward on every deployment. I discuss this solution in the (as yet) unwritten *sect.vc.deploy.using-capistrano-for-deploying-database.yml*, and Capistrano itself in Chapter 5, *Capistrano*, on page 90.

- You can leave the database.yml file on the server in the shared directory. You can then create a symlink to that file. It’s best to create this symlink in an after_update_code Capistrano task.

```ruby
task :after_update_code, :roles => :app, :except => {:no_symlink => true} do
  run <<-CMD
    cd #{release_path} &&
    ln -nfs #{shared_path}/config/database.yml →
    #{release_path}/config/database.yml &&
    ln -nfs #{shared_path}/config/mongrel_cluster.yml →
    #{release_path}/config/mongrel_cluster.yml
  CMD
end
```

If you use this approach, you’ll need to be sure you communicate. Since you’d make any changes to database.yml.sample, other developers might not notice the changes. Most of the time though, the sample file will not change, and leaving it as-is is OK. Alternatively, you can call the sample file database.sample.yml, so your editor can pick up syntax highlighting.
Database structure dumps during testing

When you run the tests, Rails will dump the development database's structure to db/schema.rb.\(^3\) This file should not be under version control, as it will regularly change. Instead, you should keep your migration scripts under version control, and they will enable you to recreate the database schema at will. Because Rails generates schema.rb for each build, you don’t want it under version control. Ignore it with these commands:

```
$ svn propset svn:ignore "schema.rb" db
property 'svn:ignore' set on 'db'
```

Managing tmp, documentation, scripts, and public

Rails 1.1 and above now have a tmp folder. This folder will only hold temporary files such as socket, session and cache files. Ignore anything in it:

```
$ svn propset svn:ignore "*" tmp
property 'svn:ignore' set on 'tmp'
```

The doc folder holds many subfolders: appdoc and apidoc among others. To keep things simple, just ignore any 'doc' suffix:

```
$ svn propset svn:ignore "*doc" doc
property 'svn:ignore' set on 'doc'
```

Subversion also has a property to identify executable files. Set the property on files you might run from the command line. On *nix, you will have to name each file on the command line:

```
$ svn propset svn:executable "*" `find script -type f | grep -v '.svn'`
public/dispatch.*
property 'svn:executable' set on 'script/performance/benchmarker'
...
property 'svn:executable' set on 'public/dispatch.fcgi'
```

On Windows systems, do this instead:

```
C:\deployit> svn propset svn:executable script\performance\* script\process\* script\about script\breaker script\console script\destroy script\generate script\plugin script\runner script\server public/dispatch.*
property 'svn:executable' set on 'script/performance/benchmarker'
...
```

---

3. If the Active Record configuration variable named config.active_record.schema_format is set to :sql, the file will be named development_structure.sql instead. Simply replace schema.rb with development_structure.sql in the commands.
Joe Asks...  

What if I'm using Rails Engines?

Rails Engines copies some files to public on startup. Since you do not want to see those files on svn status, you should ignore them:

```bash
$ svn propset svn:ignore "engine_files" public
property 'svn:ignore' set on 'public'
```

property 'svn:executable' set on 'public/dispatch.fcgi'

Since I will deploy on Unix/Linux machines, it makes sense to have the dispatchers use a proper line-ending. To do so, set svn:eol-style to native to let Subversion manage the line ending according to local conventions:

```bash
$ svn propset svn:eol-style native public/dispatch.*
property 'svn:executable' set on 'public/dispatch.cgi'
```

...  

Last but not least, projects usually have a default home page served by a Rails action. This means building a route and removing public/index.html:

```bash
$ svn revert public/index.html
Reverted 'public/index.html'
```

$ rm public/index.html

Saving your work

After all of these changes, commit your work to the repository:

```bash
$ svn commit --message="Initial project checkin"
Adding README
...  
Adding vendor/plugins
Transmitting file data ...............................  
Committed revision 2.
```

Using a stable branch for deployment

Many simple applications simply run off of the trunk. Others will feel more comfortable deploying from a stable branch. Several great books address this topic better than I possibly could, but I do want you to get a feel for what’s involved. For detailed information on this topic, you should read “Pragmatic Version Control with Subversion”.  

Prepared exclusively for Arthur Pinkney
The changes you do on trunk might not be fully tested, or you could be in the middle of a major refactoring when an urgent bug report comes in. You need to have the ability to deploy a fixed version of the application without having to deploy the full set of changes since the last deployment. In Subversion, you can copy a branch of development to another name, and you can setup Capistrano to deploy from your stable branch instead of your development branch. Developers call this technique stable branch deployment.

First, set the repository in Capistrano’s deploy.rb, and commit:

```
$ svn commit --message "Set Capistrano to deploy from stable branch"
```

Then, create the stable branch, which will be a copy of trunk:

```
$ svn copy --message "Create the stable branch"
```

When you are ready to merge a set of changes to the stable branch, check the last commit message on the branch to know which revisions you need to merge:

```
$ svn log --revision HEAD:1 --limit 1
```

Using the information in the log message, you can now merge all the changes to the branch:

```
$ svn merge --revision 422:436
```

Finally, commit and deploy:

```
$ svn commit --message "Merged r422:436 from trunk/
```

...
Transmitting file data ....
Committed revision 437.

$ cap deploy_with_migrations
...

You now have a good Subversion repository, and you can use it to deploy. You’ve ignored the files that will break your developers will or just your application, and you’ve used common Rails conventions. Still, you should know a few things about developing with Subversion, with successful deployment in mind. I’d like to walk you through some tips you can use when you’re using Subversion with Rails.

2.3 Subversion Tips

Now that your repository is off and running, I’ll quickly give you some tips for using Subversion for your day-to-day coding. I’ll teach you how to link to Edge Rails with an external link, to generate code that’s automatically checked in, and a few other tricks as well.

Running Edge Rails

If you are like me, you enjoy keeping up with the latest changes in the Rails trunk, or Edge Rails. Get the latest and greatest features right as they are added by using 

You can get Edge Rails to automatically update when you update your working copy by setting the vendor directory’s svn:externals property by running this command:

$ svn propedit svn:externals vendor

When your editor opens up to allow you to set the svn:externals property, add this line:

rails http://dev.rubyonrails.org/svn/rails/trunk/

The next time you update\(^4\), Subversion will download the entire Rails trunk to vendor/rails for you.

If you want to negate that option, you can use the following as of Subversion 1.2\(^5\)

\(^4\) If you set svn:externals before the first commit, the update will not fetch the external source code.

\(^5\) http://subversion.tigris.org/svn_1.2_releasenotes.html
Edge Rails has all the greatest features, but is sometimes unstable. Make sure you have a fairly wide set of unit, functional and integration tests to catch any bugs Edge Rails might introduce. Don’t forget to report any breakage to the rails-core mailing list and/or to create a ticket on the Rails Trac: http://dev.rubyonrails.org/. When reporting a bug, you should always report which revision of Rails you were using at the time:

\$ svnversion vendor/rails
4077

### Checking In Generated Code

During normal Rails development, you'll use generators to create many new files. Some generated files should not go into the repository. As a general rule, if Rails generates a file from scratch at runtime (such as schema.rb), you won’t want to check it in. If you will edit a generated file, you'll want to check it in.

Whenever you build a scaffold, you'll want to add the generated files to Subversion. You can save time by adding them as they are created. Rails makes this easy when you use the `script/generate` command to create new files. Just add the `--svn` flag. Rails will generate the files, and then automatically `svn add` them for you, like this:

\$ script/generate scaffold --svn Post
   exists  app/controllers/
   exists  app/helpers/
   create  app/views/posts
   A       app/views/posts
dependency model
   exists  app/models/
   exists  test/unit/
   exists  test/fixtures/
   create  app/models/post.rb
   A       app/models/post.rb
   ...

### 2.4 Stabilizing your Applications

Rails is a fairly forgiving application framework in development mode, with one user. When you push your application up to a production server, it becomes real production software, whether it’s ready or not.
This section will walk you through a few things you can do to stabilize your application.

**Locking Down Plugins and Gems**

You probably install third party gems once on your local machine and forget about them. You don’t need to do anything unless you want a later gem that fixes a bug, or you need features of a new gem. Shared hosts are a different story, because they often upgrade gems without your knowledge, which could hose your application at the most embarrassing moment conceivable. To prevent this unfortunate circumstance from happening to you, copy each dependency to vendor. Unpack each gem to vendor like this:

```
$ cd vendor
$ gem unpack money
Unpacked gem: 'money-1.5.9'
$ ls
money-1.5.9 plugins rails
```

Gems all reside in a `lib` folder. To move your gem to version control, you just need to copy the content’s of that `lib` folder to vendor, like this:

```
$ cp -R money-1.5.9/lib/* .
$ cp money-1.5.9/MIT-LICENSE LICENSE-money
$ rm -Rf money-1.5.9/
$ ls
LICENSE-money bank money money.rb plugins rails support
```

Make sure you abide by your license agreements, too. For example, to comply with the above gem’s license, you also need to copy the license along with the code. Next, add and check in the new files:

```
$ svn add --force *
A LICENSE-money
...
A support/cattr_accessor.rb
```

```
$ svn commit --message="Imported Money library 1.5.9"
Adding LICENSE-money
...
Transmitting file data .......
Committed revision 4.
```

**Upgrading an unpacked gem**

When you are ready to integrate a new version of the gem into your application, you essentially follow the same procedure:
$ gem unpack money
Unpacked gem: 'money-1.7.1'
$ cp -Rf money-1.7.1/lib/* .
$ cp -f money-1.7.1/MIT-LICENSE LICENSE-money
$ rm -Rf money-1.7.1
$ svn status
M money/core_extensions.rb
M money/money.rb
X rails
$ svn commit --message="Upgraded Money to 1.7.1"
Sending money/core_extensions.rb
Sending money/money.rb
Transmitting file data ..
Committed revision 5.

If the library provider deleted or moved files around, you need to do the same thing too. Check the library’s release notes to learn about any requirements for backward compatibility. A great tool that automates importing new releases of a library is `svn_load_dirs.pl` ([http://svn.collab.net/repos/svn/trunk/contrib/client](http://svn.collab.net/repos/svn/trunk/contrib/client)).

**Freeze the Rails Gems**

Even new versions of Rails can break backwards compatibility. Bruce’s shared host once upgraded to Rails 1.1 while he was in Spain to give a Ruby talk at a Java conference. The new version immediately broke his blog which was bad enough. As you can imagine, the broken blog made it nearly impossible to extoll the virtues of Rails.

After several decades of intense therapy he has finally recovered from this incident and is a better person because of it. You can protect yourself against this possibility by freezing a copy of the Rails libraries to your app’s `vendor` directory. Your application will use the exact version of Rails that you:

- *considered when you designed your application*. Some versions of Rails have philosophical differences between other versions, such as the new forms model in 1.2.

- *used to test your application*. If you don’t freeze your Rails gems, you’re fundamentally saying that you don’t need to test how thousands of lines of code will work with your application. If you make such a choice, I wouldn’t recommend any long trips to London.

- *understand*. Rails is an active framework. You need to make sure that you have a good grasp on changes in the framework before you deploy.
When you upgrade to a newer version of Rails, you can integrate your application, test, and then re-freeze it to the vendor directory:

`local$ rake rails:freeze:gems`

If you’ve come from a C, Java, or C# platform, you may be surprised the Ruby gems often break backward compatibility. In truth, this decision is a two-edged sword. If you don’t respect backward compatibility, your applications can break, but there’s a benefit. Breaking backward compatibility allows your framework to evolve much more quickly and cleanly, without the risk of framework bloat. (See Enterprise JavaBeans or XML, for two examples.) Ruby and especially Rails developers value a cleaner code base more than backward compatibility. As more enterprise developers use Rails, you may see a change, but don’t ever rely on a future that lets your application run safely without your own version of Rails. With versioned code and gems in hand, you can move on to organizing your migrations.

### 2.5 Active Record Migrations

Migrations, a Rails feature that lets you express your database tables in Ruby instead of SQL, are a great way to manage your database schema throughout your development process. You can quickly create, change, or delete tables and indexes. If you are already using migrations, I’ll show you how to whip them into shape for your production environment. If you’re deciding whether to use them, you should know the strengths and weaknesses of the approach.

**Migration Strengths and Weaknesses**

On the plus side, migrations generally provide a more comfortable environment, and ease the process of keeping your production schema up to date. More specifically:

- *Migrations let you express database independent code in Ruby instead of SQL.* Because you’re working in Ruby, you can often express your ideas in a cleaner, simpler way.

- *Migrations integrate with Rake (and Capistrano to a lesser extent).* You can call Rake commands to move your migrations up to a precise level, or move your schema back to a point in time. You can also ask Capistrano to run your migrations automatically when you deploy.
• **Migrations deal with data.** Some database schema changes require changes in data. Migrations can handle both, since they are Ruby scripts. Setting the data for new columns, selectively adding or deleting rows, or defining lookup tables are all examples of dealing with data in migrations.

• **Migrations simplify backing up.** Rails developers make just as many mistakes as any others. If your latest build is a stinker that also changes the schema, migrations can allow you to back up quickly.

• **Migrations make it easy to change schemas without losing data.** Since migrations use the `ALTER TABLE` command rather than create and drop table, you can easily make changes to the schema without worrying about losing production data. Also, you can use the same tools to manage your development and production scheme.

Keep in mind that migrations are not a silver bullet. Some teams can make them work, and others can’t. In general, small teams with a simple deployment strategy will work great with migrations. Larger teams, or those that manage multiple releases, or those who refactor model code on a regular basis and simultaneously use data migrations will struggle. These are some of the disadvantages of migrations:

• **Migrations do not integrate with Subversion.** If an older migration depends on a particular model, and that model no longer exists, it will break. The source code history in Subversion has no effective link to the database schema history, which lives in your latest Subversion version.

• **Migrations have some curious defaults.** By default, columns allow null. My experience shows that most developers don’t think about null columns until it’s too late, leading to database integrity problems later.

• **All developers depend on a unified numbering scheme, but have no tools to manage them.** If you create a migration and your friend creates one at the same time, they will both have the same number, and they will fail.

• **Branches are tough to manage.** If you want to add a major branch, perhaps to develop a major new feature without deploying it to the public until it is stable, you will effectively have to write your own migration support to do so, because each part of the application will need its own migrations.
Components have a tough time depending on migrations. Try to integrate an existing blog to an existing application and you'll see what I mean. Migrations don't provide a good default to deal with this problem.

For the most part, I like migrations. They are quick and convenient for most of the time, and if you can make them work with your team's model, you'll usually be glad you did. If you've already committed to migrations, make sure you look at the disadvantages, and you understand them. You will want to solve the problems you're likely to face before a migration blows up in production.

First look at Migrations

Regardless of whether you have a schema defined already or you are starting a new project, you can easily start using migrations. If you already have a schema in place, you'll find Rails has some good tools that will help you convert them.

Assume you have a forums table defined in a MySQL database, and the SQL looks like this:

```sql
CREATE TABLE `forums` (  `id` int(11) NOT NULL auto_increment,  `parent_id` int(11) NOT NULL default '0',  `title` varchar(200) NOT NULL default ' ',  `body` text NOT NULL,  `created_at` datetime default NULL,  `updated_at` datetime default NULL,  `forums_count` int(11) NOT NULL default '0', PRIMARY KEY (`id`) ) TYPE=InnoDB;
```

In order to start using a pure Ruby schema, Rails includes a handy Rake task to kick-start your migration (pun intended). Run this command from your application's root:

```
$ rake db:schema:dump
```

This command will create a schema.rb file that looks like this:

```ruby
# This file is autogenerated. Instead of editing this file, please use the  # migrations feature of ActiveRecord to incrementally modify your database, and
```

---

6. Pre Rails 1.1, you need to change `config.active_record.schema_format` in environment.rb to :ruby. That environment variable tells Rails that you will be using Ruby code to define your db/schema.rb.

---
# then regenerate this schema definition.

ActiveRecord::Schema.define do
  create_table "forums", :force => true do |t|
    t.column "parent_id", :integer, :default => 0, :null => false
    t.column "title", :string, :limit => 200, :default => "", :null => false
    t.column "body", :text, :default => "", :null => false
    t.column "created_at", :datetime
    t.column "updated_at", :datetime
    t.column "forums_count", :integer, :default => 0, :null => false
  end
end

With db/schema.rb in place, you can start writing migrations. Rails will apply each change to your initial schema.rb. You will never need to edit this file directly as Rails generates a fresh one after each migration of your schema. You also shouldn’t check it in to your Subversion repository, because each developer will want to move to his own version of the database. Initially, you will need to copy the initial file to your test and production environments.

db/schema.rb serves as the starting place for each migration. The file holds your entire database schema at any point in time in one place for easy reference. Also, migrations create a table called schema_info. That table holds a single version column, with one row, and a single number: the version number of the last Migration that you ran. Each migration is a Ruby file beginning with a number. The migration has an up() method, and a down() method. Migrating up starts with schema.rb and applies the migrations with higher numbers than the number in schema info, in order. Migrating down will apply the migrations with lower numbers, greatest first.

So now that you have schema.rb, you have everything you need to create migrations at will. Your first migration will create schema_info for you. I don’t want to teach you how to build a Rails application here, but the Rails documentation is fairly complete. I do want to make sure you know enough to stay out of trouble.

**Putting Classes Into Migrations**

Good Rails developers generally don’t depend on domain model objects in migrations. Five weeks from now, that Forum model might not even exist anymore. Still, some data migrations will depend on a model, so you need to create model instances directly inside your migration.
class CleanupForumMessages < ActiveRecord::Migration
  class Forum < ActiveRecord::Base
    has_many :messages, :class_name => 'CleanupForumMessages::Message'
  end

  class Message < ActiveRecord::Base
    def cleanup!
      # cleanup the message
      self.save!
    end
  end

  def self.up
    Forum.find(:all).each do |forum|
      forum.messages.each do |message|
        message.cleanup!
      end
    end
  end

  def self.down
  end
end

Notice that I declare each class that I need in the migration itself, which acts like a Module. Make sure you use the :class_name feature of has_many(), has_one(), belongs_to() and has_and_belongs_to_many() because Rails uses the top level namespace by default, instead of the current scope, to find the associated class. If you do not use :class_name, Rails will raise an AssociationTypeMismatch when you try to use the association.

The solution is not perfect. You’re introducing replication, and some features like single-table inheritance become troublesome, because you need to declare each and every subclass in the migration. And good developers can hear the word DRY—don’t repeat yourself—in their sleep. Still, your goal is not to keep the two versions of your model classes synchronized. You are merely capturing a snapshot of the important features of the class, as they exist today. You don’t necessarily have to copy the whole class. You only need to copy the features you intend to use.⁷

---
⁷ Thanks to Tim Lucas for the original blog post: http://toolmantim.com/article/2006/2/23/migrating_with_models.
More Migrations Tips
You should keep a few other things in mind as you deal with migrations. These tips will hopefully improve your experience with them:

- *Keep migrations short.* You shouldn’t group together many different operations, because if half of your migration succeeds, it will be too hard to unwind. Alternatively, you can include your migrations in a transaction if your database engine supports DDL statements like create and alter table in a transaction. PostgreSQL does; MySQL doesn’t.

- *Make sure you correctly identify nullable columns.* Columns are nullable by default. That’s probably not the behavior you want for all columns. Rails migrations probably have the wrong default.

This list of tips is by no means an exhaustive list, but should give you a good start. Now, it’s time to shift over to look at improving the rest of your application.

2.6 Application Issues for Deployment
Rails is a convenient framework for developers. Sometimes, the convenience can work in your favor. You can build quickly, and Ruby is malleable enough to let you work around the framework. But if you’re not careful, all of that flexibility can bite you. In this section, I’ll walk you through some common security problems, and a few performance problems as well.

Security Problems
Rails has the security characteristics of other web-based frameworks based on dynamic languages. Some elements will work in Rails favor. You can’t secure something that you don’t understand. The framework is pretty simple, and web development experts already understand the core infrastructure pretty well. But Rails has some characteristics you’ll have to watch closely.

Rails is a dynamic, interpreted language. You need to be sure that you don’t evaluate input as code, and that you use the tools that Rails provides that can protect you.
Using view helpers

You likely know how Rails views work. Like most web frameworks, Rails integrates a scripting language into HTML. You can drop code into Rails by using `<%= your_code_here %>`. Rails will faithfully render any string that you may provide, including a name, helpful HTML formatting tags, or malicious HTML code like this:

```html
<img src='http://porn.com/your_favorite_porn_image.jpg'/>
```

You can easily prevent this problem by using the template helpers. If you surround your code with `<%=h your_code_here %>`, Rails will escape any HTML code a malicious user may provide.

Don’t evaluate input

At the same time, you need to be sure not to evaluate any code that any user might type as input. Ruby is a great scripting language, and you should be careful any time you try to evaluate any code, and you should never evaluate user input. For example, consider the following code that assumes you’re picking the name of an attribute from a selection box:

```ruby
def update
  ...
  # Don't do this! Potential string = "The value of the attribute is " + "#{Person.send(param[:attribute])}"  
  ...
end
```

That code would work just fine as long as the user cooperated with you and picked "first_name" or "email" from a selection box. But if a Rails developer wanted to exploit your system, he could send data to your controller by opening a curl session, and posting his own data. Or, if you don’t verify that the command is a post, he could simply key the following into a url:

```
your_url.com/update/4?attribute=destroy_all
```

Assume all user input is tainted. Not all metaprogramming is good. Don’t ever evaluate any data that comes from a user unless you’ve scrubbed it first.

Don’t evaluate SQL

You can make a very similar mistake with SQL. Say you want to look up a user with a user ID and a password. You could issue the following Active Record command:
# Don't do this!
condition = "users.password = #{params[:password]} and
          users.login = #{params[:login]}"
@user = find(:conditions => condition)

And all is well. At least, all is well until someone types the following instead of a password:
up yours'; drop database deployit_production;

The first semicolon ends the first SQL statement. Then, the hacker launches some mischief of his own, dropping the production database.

An alternative would be to try to create a user with enhanced permissions. This type of attack, called SQL injection, is growing in prominence. You can easily prevent the attack by coding your condition like this:

conditions = ["users.login = ? and users.password = ?",
              params[:login], params[:password]]
@user = find(:conditions => conditions)

This form of a finder with conditions allows Rails to do the right thing: properly escape all parameters and input that Active Record will pass through to the database.

**Check permissions**

Rails gives developers plenty of help when it comes to building pretty URLS. The bad news is that others who would attack Rails also know this. Consider the following action, which is commonly created through scaffolding:

def destroy
  Person.find(params[:id]).destroy
  redirect_to :action => 'list'
end

To secure the command, you decide to add before_filter :login_required to the top of your controller, meaning people need to log in before accessing the destroy() method. For an application where only admins can delete, that protection is enough. But if any user can create an account and log in, that protection is not nearly enough. Any user can create an account, and start deleting records by sequentially typing id numbers into the browser:

/people/destroy/1
/people/destroy/2
/people/destroy/3
/people/destroy/4
Worse yet, a bot could log in and delete all of your records. You need to check that the logged in user has permission to delete the file within the controller action. Assume that each Person object is associated with the User who created it. Also, assume current_user returns the current logged-in user. Then, you could protect destroy() like this:

```ruby
def destroy
  person = Person.find(params[:id])
  person.destroy if current_user == person.user
  redirect_to :action => 'list'
end
```

Logging in is not enough. You must scope individual destructive actions to one user. That covers the most common security flaws. There are others like exposing your .svn directories to the web. The easiest way to get around this one is to do an `svn export` instead of an `svn checkout` when deploying your code to production. This will export your code without the subversion meta data and keep prying eyes away.

If you take heed of these various issues then your Rails application should be nice and secure. Do make sure you keep up with the main Rails blog: http://weblog.rubyonrails.org/ for any security updates or warnings.

**Database Performance Problems**

Active Record belongs to a family of database frameworks called *wrapping* frameworks. A wrapping framework starts with a single table, and places a wrapper around it to allow object oriented applications to conveniently access rows in the table. The performance of wrapping systems like Active Record is highly dependent on you, the programmer. The biggest thing you can do is benchmark your application. We’ll discuss benchmarking in the (as yet) unwritten chp.performance. In the mean time, I’ll show you the most common problem you’re likely to see.

**The N + 1 problem**

Active Record makes it easy to retrieve a given object and access its attributes. Bad things happen when those attributes are lists of other Active Record objects. Let’s say you’re building the next great social
networking site. You have a Person that has_many :friends. To populate a list of friends, you write some harmless code that looks like this:

```ruby
friends = Person.find(:all, :conditions => some_friend_conditions)
@friend_addresses = person.friends.collect { |friend| friend.address.street }
```

To be sure, that code will work, but it’s also horribly inefficient, and will get worse as the list of friends grows. You’re actually running an Active Record query for the list of friends, and another for every address you need to fetch. You can fix that problem using *eager associations*, meaning you’ll tell Active Record what to load in advance with the :include option:

```ruby
friends = Person.find(:all, :conditions => some_friend_conditions, :include => :address)
@friend_addresses = person.friends.collect { |friend| friend.address.street }
```

This code works in exactly the same way to you, but the performance will improve dramatically. Active Record will load all people and their addresses instead of just loading people in the first query and address as you touch them the first time.

**Indexes**

Rails lets database developers get pretty far without knowing anything about the database underneath, or even the theory surrounding relational databases. If you trust Active Record to take care of you, it’s likely that you and your users will be disappointed. One of the easiest things to forget when you’re coding Rails is the creation of indexes. For any large tables, make sure you create an index on any column you need to search. And periodically, you should run statistics, so the database optimizer knows when to use indexes. Database administration performance techniques are beyond the scope of this book.
Finding a host for your first Rails app is a lot like finding your first home. When I left home the first time, I wanted to move right into the Taj Mahal, but real life didn’t work out that way. Most people first move into an apartment or dorm room. True, apartments don’t come with their own throne room and servant’s quarters, but they do have their advantages. You’re sharing common resources and infrastructure with many others, so you wind up paying less. You don’t have to mow the lawn or paint the fence. For most people, the first Rails app runs in modest quarters for many of the same reasons: shared infrastructure, lower costs, and help with the maintenance. In this chapter, you’ll learn how to pick and prepare a shared host.

3.1 The Lay of the Land

Many Rails apps start life on a shared server. Figure 3.1, on the next page tells the story. You’ll buy one slice of a larger server that will have the ability to serve your Rails application and static content. You will control only a few directories for your application. You’ll use Subversion to install your application while you set up your initial infrastructure, until you’re ready to automate with Capistrano. For a few dollars a month, you’ll have your own domain name, access to a database server, several email accounts, and maybe even a Subversion repository. For many people, this setup is enough for a blog, a site prototype, or even a bug tracking system.

If you can cache your application, and sometimes even if you can’t, you can serve hundreds of users daily without needing to pay US$100 per month for a dedicated server. Your hosting company will fix intermittent problems, occasionally upgrade your machine, and keep things running
smoothly. When you are ready to move on to a more powerful dedicated machine or a cluster of servers, you can upgrade within your hosting company or transfer to a colocation facility.

At least, that’s the theory. Shared hosting is not all sunshine and roses. You may be sharing a single server with over a thousand other websites. I’ve received more than my share of nasty grams telling me that my app’s memory was out of control, or that my unstable app crashed the whole server. I’ve been on the other end of the equation, too. I’ve been the good citizen, but someone else ran a script that monopolized the server’s resources and slowed my application to a belly crawl. I’ve also had my site become wildly popular and subsequently got a bill for
US$500. (I am much happier now that I am on the sending end of those bills!) A good host will keep tabs on these statistics and will notify you if you are using more than your share. Most will even give you a little grace period and eat small overages for short periods of time.

All things considered, you can do almost anything on a shared server that you can do on a dedicated server, but you'll have more resource constraints. If you are just learning how to develop a database-driven website, you can focus on the mechanics of your application instead of worrying about the details of configuring DNS, daemons, and disk partitions. Even so, I recommend that you treat a shared host as a starting point, not the final destination for your application. If you are earning more than US$30 per month from your site or if your business depends upon it, you should upgrade to a virtual private server or a dedicated server.

When you look for a new home, you can’t do it all at once. You'll have to consider the time it takes to pick a place, change your address, bribe the landlord, and decide whether you'll keep or throw out all of that fine stuff like your old Commodore 64. Moving takes time, and things go more smoothly when you plan. Treat setting up a shared host the same way. You have one goal—making your application run on a shared host—but it's best to define a few discrete steps to get there:

- Find the right place. Pick the plan that works best for your application, and your pocketbook.
- Tell the world where you live. On the web, that means updating your DNS entries.
- Move in your stuff. Install a simple app. For Rails, that means installing a simple app. Later, I’ll help you automate this step.
- Set up your utilities. In the web world, that means configuring your web server and database server to work with your Rails app.

When all is said and done, your setup might not work the first time. That's OK. I'll walk you through the process of pulling it all together. When you're done, you'll have a slice of a common server, a Mongrel web server, and a database-backed Rails application.
3.2 Choosing a Shared Host

Way back at the dawn of Rails history (a.k.a. the fall of 2004) only a few hosts officially supported Ruby on Rails. That number increases every day as Rails becomes better known. There are many capable hosts and I won’t recommend any single provider, but you should look for several critical features in a shared host.

Basic Requirements

At minimum, the host you choose must have the following features:

- A reasonably current version of Ruby. At the time of this writing, version 1.8.6 can run Rails 1.2, but Ruby version 1.8.5 is fine, too.

- Mongrel support. Some shared hosts don’t yet support Mongrel, but there are plenty that do, and Mongrel is rapidly emerging as the de facto standard within the Rails community.

- Ability to specify the web server’s document root directory. There are ways to get around this, but it is much easier if the host provides an interface where you can point the web server to your preferred directory, perhaps one like /home/ezra/brainspl.at/current/public.

- SSH access. This is crucial for troubleshooting your installation, and is required for deploying with Capistrano. Some very inexpensive hosts only allow you to transfer files by FTP, so choose one that has SSH as an option.

- A database server and the required Ruby gems to connect to it. MySQL and PostgreSQL are popular, but you can use file-based database managers like SQLite just as easily.

For the optimal Rails setup, I recommend these features:

- OS-dependent gems, such as RMagick. These gems let you easily generate graphics, make thumbnails of photographs, and do other useful tasks. You can copy pure Ruby gems into your Rails application’s lib directory, but you really want your hosting provider to install gems requiring compilation and C libraries because building and installing these gems takes more authorization than your account will typically have.

- Subversion repositories that are accessible over HTTP (or secure HTTPS). Whether you are a professional programmer or a hobbyist, you should be using source code control. With a source
code control system, you can deploy with Capistrano or publish Rails plugins that other Rails developers can install with the built-in `./script/plugin` mechanism. You can use Subversion in countless other ways, but HTTP access is the most versatile in the context of a Rails application.

**The Core Rails Libraries**

If you were paying attention, you saw that an installation of the core Rails libraries was missing from both lists! In fact, you will experience more stability if you use the built-in Rake `rails:freeze:gems` command to save a specific version of Rails to your application’s vendor directory. You can find more about this in Section 2.4, *Freeze the Rails Gems*, on page 31. If your host decides to upgrade to a blazingly fast new version of Rails in the middle of the night—one that might break your application—your application will still run with the older version residing in your vendor directory. When you have tested your app against the newer version of Rails, you can again call `rake rails:freeze:gems` to upgrade.

The adventurous can use `rake rails:freeze:edge` to use a copy of the development version of Rails (commonly called the trunk version or edge Rails). Even though Rails is almost three years old, development continues at a rapid pace. Before the release of Rails 1.1, I used edge Rails on a personal app in order to take advantage of RJS Javascript templates and polymorphic joins. Both features were in Rails 1.1 and were stable enough to use until Rails 1.1 was released officially.

You can also tie your application to the newest Rails trunk so it is updated every time the development branch of Rails is updated, but I don’t recommend doing so because that strategy adds another unknown element into your deployment process. The Rails core team is very active, and updates to the Rails trunk are made several times a day. Your application might work at 8:05, but an update made at 8:06 could break your app when it is deployed on the server at 8:07. Keeping a consistent version of Rails will let you decide when to upgrade, on your schedule.

If you have installed other third party libraries or plugins, they may use undocumented features of Rails that could change without notice. Well-behaved plugins will be more stable, but no authorized group of developers certifies plugins. By using a consistent release (or edge) version of Rails in conjunction with a thorough test suite, you can guar-
antee that the combination of code libraries in your application passes all your tests.

Whether you choose to live on the edge or use an older version of Rails, freezing them within your application will help you simplify your deployment and maintenance in the long run. Next, I'll walk you through some of the intangible factors you should consider when selecting a shared host.

**Intangible factors**

Shared hosts have variable reputations. You'll want a shared host with a good reputation for support, and one that runs a tight ship. You may not like that nasty email from the overzealous, pimple-ridden admin that threatens the life of your firstborn because your application is taking more than its share of resources, but he’s exactly who you want running your server. He will keep all of the other apps on that shared host in line too. You'll also want a company with enough experience to give you a little grace if your traffic spikes once in a while, and won’t simply kill your Mongrels without any notification when things step out of the common parameters. System maintenance like regular backups and cycling log files is a plus because you won’t have to do them yourself.

The best way to measure intangibles is to ask around. Good Rails programmers will know who the good vendors are. I won’t list any here because I am such a vendor, but be forewarned. The best shop today—one with good deals and good admins—could experience unmanageable growth, or lose that key admin who made everything run like a swiss watch, or just get lazy. Ask around and keep up. You’ll be glad you did. You can start with the excellent Ruby on Rails Wiki¹ or by chatting with Rails developers on IRC².

After you’ve done your homework and picked your host, you’ll want to start setting things up. But first things first. You’ll need to tell the world where your application lives. That means you’ll need a domain name and need to configure DNS.

---

3.3 Setting Up your Domain and DNS

Any home you choose will need an address so people can find you. On the Internet, you’ll have two addresses: the one with numbers and dots is your IP address, and the name with a .com or .org on the end is your domain name. The IP address contains four numbers, each with values from 0 to 255, separated by periods. Since memorizing up to twelve digits is hard, most of your users will refer to your site by domain name instead. You will buy your domain name from a domain name service such as GoDaddy, NetworkSolutions or Enom, and you’ll get your IP address from your hosting service. Under the covers, a domain name service (DNS) will associate your name with your IP. The Internet has several public domain name registries that associate IP addresses with domain names. So to establish your address, you need to do the following:

• Buy a domain name. They will give you a name and a way to configure the IP address.

• Pick a shared host. They will give you your IP address.

• Associate your name with your IP address.

You can easily buy a domain name for as little as $10 USD, which means the hardest part is picking a name to use! I recommend choosing something that is easy to remember and easy to spell. Hopefully your name is simpler than EzraZygmuntowicz.com. You can use a site like Instant Domain Search\(^3\) to help you find a name that’s not already in use.

Most domain name registrars will also give you a web page to configure your DNS settings. If you cannot easily find any place to make DNS changes, contact your providers support system to ask about DNS settings. You want these to point to your shared hosting company’s servers so the domain name server will forward requests for your domain name to your specific server.

DNS stands for Domain Name Service. It is responsible for resolving a domain name and returning an IP address where the service really lives. Your shared hosting provider will have their own name servers. When you sign up, ask them to give you their DNS nameserver addresses. They will be something like ns1.foobar.com and ns2.foobar.com. Take

---

3. [http://instantdomainsearch.com](http://instantdomainsearch.com)
Using Hosting Company Sub-domains

Some shared hosts also provide you with a subdomain to use while you are setting up your site. This might be something like http://ezmobius.myhost.com. I strongly recommend against this approach because it looks less professional and requires that you use a third party service for your email. Also, if you switch to a different host you will have to inform all your visitors that your address has changed. When using your own domain name, you can switch from one host to another with fewer consequences and your customers never need to know.

note of these addresses, go to your domain name registrar where you registered foobar.com, and click on the section for DNS or nameservers for your domain. They will have multiple form fields you can fill out. Most hosts will give you two nameservers but they may have more. Once you enter these details and save the results, it will take 12-72 hours for the nameservers all across the world to propagate your new records. Once DNS is propagated you can type foobar.com in your browser and the domain name service will find the server’s IP address and route your request to your domain.

You may be able to see your new application before the bulk of your customers, depending on where you live and the DNS services near you. Propagation time can range from under an hour to a couple of days, so plan in advance. While you’re waiting for the DNS changes to take hold, you can switch gears and focus on configuring the shared host.

3.4 Configuring your Server

Configuring your server is surprisingly easy when you follow a simple plan. It really amounts to providing access to your server through SSH and your document root, and creating your application’s database. If you do these steps correctly, you’ll have a good foundation for the easy deployment of your web server and Rails app.

First, you’ll want to determine your application’s root directory. If you don’t know exactly where to put your application, ask. Most host accounts will create this directory for you, and set permissions appropriately. In
the rest of this chapter, I’ll refer to that directory as the application root. This is the top level directory for your Rails project. Your document root is the root directory for your web server, and will hold your static files. Usually, you’ll put your Rails project in a directory called current, meaning your document root will be current/public. From here on out, I’ll refer to that directory as the relative root (/current).

**Server Setup: Document Root and SSH**

To prepare your shared host for your application, you’re going to need a way to talk to it securely (SSH). You will also need a new user account on the shared server. Many shared hosts provide only one SSH user per account, and your hosting company may have created one for you already. If not, each hosting company has a different web interface, but the process is usually simple. Just read the documentation they provided.

After you’ve created an account, try it out. If you’re running Windows, you’ll need to set up your SSH client. PuTTY is a good one. There are several free clients, and you’ll find plenty of documentation for them. If you’re running a Unix derivative, you’ll have a much easier time. Usually, you won’t even need to install anything. Open a terminal and type ssh username@hostname, using the information from your hosting provider. If you have any trouble, ask your hosting provider. Make sure you get your SSH connection working, because it’s your secure window into the system for all of your deployment, maintenance, and debugging.

While you are logged in, you will want to change the site’s document root, also called the web root, to /current/public. Check the control panel for your site for the right place to configure your document root. The tool to change it will vary based on your provider. Shortly, you’ll upload your Rails application to the current directory. The public directory has your application’s public directory, which hosts static resources for your application. Capistrano, the Rails utility for deployment, will use the current directory to always hold the most recent version of your site.

### 3.5 Server Setup: Create a Database

The final task to do on the server is to create a database. Again, the means for doing this depends on your hosting provider. As you know, when you created your Rails application, you gave it a name, for exam-
ples, rails ezra. By default, Rails will use three databases for each of the test, development, and production environments. The default database name for each environment is the name of the Rails project, followed by an underscore, followed by the name of the environment. For example, for an application called ezra, Rails would generate ezra_development for the development database. In practice, many developers omit the _production for their production database.

Normally, your hosting provider will require that you share your database with other users. They’ll likely give you a user id and password to access your own database namespace. You’ll use that ID to create your database. Whichever name you choose, make sure to keep the database name, combined with an admin-level username and password, in a safe place for use later. Here’s an example on MySQL that creates a database called ezra and grants all of the appropriate privileges. Your setup may vary:

```
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP, ALTER, INDEX
mysql> ON ezra.* TO 'ezra'@'hostname' IDENTIFIED BY 'password';
mysql> FLUSH PRIVILEGES;
mysql> CREATE DATABASE ezra;
```

If you want to peek ahead, the production section of your Rails application’s config/database.yml will look like this:

```
production:
  adapter: mysql
  database: ezra
  username: ezra
  password: password
  host: brainspl.at # provided by your shared host provider
```

At this point, you’ll want to make sure your database is running and you can access it. Just using the database and showing the list of tables is enough for now:

```
mysql> use ezra;
Database changed
mysql> show tables;
Empty set (0.00 sec)
```

Now, you should have a working domain that’s pointed to your hosting provider, you should be able to reach your server through SSH, and you have created and accessed a database on the server. With all of
that background work out of the way, it’s time to configure your web server.

3.6 Installing your Application

Even before I knew how to write a single line of PHP, I could copy a basic PHP script to a server and run it. Deploying Rails is not that easy. You need to have at least a basic idea of what files and folders are used by Rails when it runs.

Fortunately, Ruby has excellent tools that let you deploy your application easier than PHP. The main one is Capistrano, which you’ll see in Chapter 5, *Capistrano*, on page 90. When we’re done, your Capistrano script will check your application directly out of your Subversion repository, and put it exactly where it needs to be. But you need to walk before you can run. I’m going to take you through the installation process manually. That way, you’ll see where everything goes and you’ll have a greater appreciation of what Capistrano is doing for you. Don’t worry, though. I’ll walk you through automating the works soon enough.

Your first job is to put your Rails application on your shared host. You should name your root project directory `current`. (As we mentioned earlier, that’s the name that Capistrano will use when you automate your deployment.) Since you have SVN installed, you can use `svn export` to copy your application to your server, like this:

```bash
$ svn export your_repository_url ~/webroot/current/
```

You’ve already done some work to prepare your application for deployment. Even so, you’ll often want to build a tiny working application to work out your deployment story before your full application enters the picture. If you decide to take this approach, you can build a tiny Rails app in a couple of minutes. You’ll create a Rails project, generate a model, create your migration, and configure your database. I’ll walk you through that process quickly, so you’ll have a starter application. If you already have a starter application, skip the next section.

Creating a Starter Application

When you are testing a deployment configuration, you’ll often want an application simple enough for your grandmother to build. You’ll want this simple application to do enough with Rails so you can see whether
your production setup works. You don’t want to have to debug your
deployment environment and your application at the same time. With
Rails, you can take five or ten minutes and build a dead simple starter
application. Once you have that working you can move on to your real
application.

You’ll want to do all of the following steps on your development machine,
not your shared host. Your goal is to build a Rails application that
exercises Rails models, views, and controllers. Since the default Rails
project already tests the controller and views by default, you need only
worry about a primitive model.

Run the commands rails ezra, cd ezra, and ruby script/generate model per-
don. (You don’t need the ruby on some platforms). You’ll get results sim-
ilar to the following:

```
~ local$ rails ezra
  create
  create  app/controllers
  create  app/helpers
  ... more stuff ...

~ local$ cd ezra/

~/ezra local$ script/generate model person
  exists  app/models/
  exists  test/unit/
  exists  test/fixtures/
  create  app/models/person.rb
  ... more stuff ...
```

These steps give you a project and a model, but one without database
backing. The last command also generated a migration that you can use
to create your database-backed model. Edit the file db/migrate/001_create_people.rb.
Add a column called name, like this:

```ruby
class CreatePeople < ActiveRecord::Migration
  def self.up
    create_table :people do |t|
      t.column :name, :string
    end
  end

  def self.down
    drop_table :people
  end
end
```

Create a MySQL database called ezra_development, which you can access
with a user called 'root' and no password. (If your database engine, user, or password are different, you'll simply have to edit database.yml to match.) Run the migration with rake db:migrate, and create a scaffold with script/generate scaffold person people. You have everything you need to test your Rails setup.

```bash
~/ezra local$ mysql -u root
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 33 to server version: 5.0.24-standard

mysql> create database ezra_development;
Query OK, 1 row affected (0.02 sec)

mysql> exit
Bye
~/ezra ezra$ rake db:migrate
(in /Users/batate/rails/ezra)

== CreatePeople: migrating ====================================================
-- create_table(:people)
  -> 0.1843s
== CreatePeople: migrated (0.1844s) =================== ========================

~ezra ezra$ script/generate scaffold person people
exists app/controllers/
exists app/helpers/
create app/views/people
exists app/views/layouts/
... more stuff ...
```

If you've done any Rails development at all, you know that these commands create your database, create your initial table for your Person model in a Rails migration, and create a simple scaffolding-based application that you can use to test your simple production setup. Next, you'll move on to the web server configuration. Later, I'll use this starter application to make sure things are working.

3.7 Configuring your web server

On most shared hosts Rails can run with either the Apache or the lighttpd web server with FastCGI, but I recommend Mongrel behind Apache, Nginx or lighttpd instead if your host supports it. We will cover using a proxy in front of a Mongrel cluster later in the book. For this chapter we will cover configuring Mongrel as well as Apache or lighttpd with FastCGI. As you will see, the Mongrel configuration is trivial, and that's exactly the point of using it. If you're on a hosting provider that
forces you to use FastCGI, you can still grin and bear it. I'll help you
get set up regardless of web server.

Configuring Mongrel

Your shared host likely has Mongrel already installed. You'll be amazed
at how simple configuration, startup, and shutdown can be. Mongrel is
a web server written by Zed Shaw in 2500 lines of Ruby and C. Mongrel
is custom tailored to running Ruby web applications like Rails. Since
Mongrel is an HTTP server in its own right, you gain the ability to use it
with a wide variety of pre-existing tools built to work with HTTP. Here
is what Zed has to say about it on the project’s home page4.

“Mongrel is a fast HTTP library and server for Ruby that is intended
for hosting Ruby web applications of any kind using plain HTTP rather
than FastCGI or SCGI.”

Mongrel is truly one of the best weapons in your arsenal when it comes
time to deploy your application. It also makes for a great development
server environment. Mongrel uses the best of both the Ruby and C
worlds. Internal HTTP parsing is done in C and the API for configuration
and application interface is done with Ruby. This C foundation gives it
very good speed, and the clean Ruby wrapper provides a familiar ruby
interface for configuration and extension.

Mongrel offers huge deployment advantages because it breaks away
from opaque protocols like FastCGI or SCGI and uses plain HTTP for
its transfer mechanism. HTTP is a proven, well-tooled, transparent pro-
tocol that all sysadmins know well. Because of the affinity to HTTP, you
will have a lot of options for integrating Mongrel into your production
environment. With Mongrel, you can interrogate individual Rails pro-
cesses with simple command line tools like curl or by using a browser
and adding the individual Mongrel port number to the url. By con-
trast, Apache and lighttpd use FastCGI, so there is no way to commu-
nicate with your Rails process without going through your front end
web server.

If your hosting provider has already installed Mongrel for you, you can
take your new dog for a walk. Fire up one of your Rails applications on
Mongrel. Navigate to your project directory and type the following:

ezra$ cd ~/webroot/current/

ezra$ mongrel_rails start -d

That command will start a Mongrel daemon running in the background on port 3000. That port is fine for your development machine, but your shared host can’t have everyone on port 3000. Find out which port you should use. You’ll have to start Mongrel on your preassigned port with the -p extension:

ezra$ mongrel_rails start -d -p 7011

It is just as simple to restart or stop the Mongrel server.

ezra$ cd ~/webroot/current
ezra$ mongrel_rails restart
Sending USR2 to Mongrel at PID 27033...Done.
ezra$ mongrel_rails stop
Sending TERM to Mongrel at PID 27037...Done.

And that’s it. You can use the -p 8080 option to specify port 8080 and -e production to specify the production environment. In Chapter 4, Virtual and Dedicated Hosts, on page 70, you’ll learn more about configuring Mongrel for more advanced needs. In the mean time, your hosting provider probably has some documentation for their policies for dealing with Mongrel. Look them over and follow them closely. You can point your browser at your domain name and see the starter application you built earlier. You will probably not have to specify your port number, assuming you’re following the port allocation and other instructions that your hosting provider gave you.

If you’re working with Mongrel, you’re done. Whistle a happy tune, and skip ahead to Section 3.8, Application Setup: Rails Config Files, on page 58. You can skip ahead while I appease the poor pitiful sots who must deal with lighttpd or Apache.

Apache + FastCGI

All kidding aside, Apache is a great general-purpose web server. In a Rails environment, Apache works best for serving static content. Serving your Rails application will take a little more time to configure. To use Apache, you’ll have to configure Rails to run using FastCGI. Then you'll tell the server to forward your request to those Rails FastCGI processes. For security sake, most shared hosts control the majority of the configuration options for the Apache web server. However, you can specify some directives that will make your Rails application run more smoothly.
By default, Rails provides a `.htaccess` file for Apache in the public directory of new Rails apps. By default, this will run your application in normal, syrup slow CGI mode. If you chose a quality shared host with FastCGI support, you should turn on FastCGI in `.htaccess`. You’ll do so by editing the `public/.htaccess` file in your Rails project to look like the following:

```
# Make sure the line that specifies normal CGI
# is commented out
# RewriteRule ^(.*)$ dispatch.cgi [QSA,L]
# Make sure this line is uncommented for FastCGI
RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
```

You can include many other directives in the `.htaccess` file. I’ll not walk through all of them right now, but one important one is specifying a few custom error pages. No one wants to see: Application error (Rails).

Even though it is not turned on by default, Rails provides a `404.html` that you can customize to gently inform the visitor that a page could not be found. To use these static error pages, make sure you remove any default error directives and use the following instead:

```
ErrorDocument 404 /404.html
ErrorDocument 500 /500.html
```

You can even write a custom Rails controller that handles 404 (Page Not Found) errors and provides a search box or a list of popular pages:

```
ErrorDocument 404 /search/not_found
```

You will need to verify that you have the correct path to the ruby interpreter for your host’s servers in your `dispatch.fcgi` file. The path to ruby in your `dispatch.fcgi` will be set to the location of ruby on the machine you used to generate your Rails application. The easy way around this problem is to use following line instead:

```
#!/usr/bin/env ruby
```

That line will load the environment to set the $PATH variable and then find the ruby binary by looking at the path. This makes it portable across most unix like operating systems.

All in all, Apache + FastCGI is a pretty decent general-purpose Rails platform, but you’ll need to watch a few quirks. You’ll need to make sure that Apache creates no rogue FastCGI processes. I’ll walk you through that process in Chapter 6, *Managing Your Mongrels*, on page 123. I’ll walk you through these items and a few others in the sections to follow.
That’s pretty much the story for configuring your Rails application with Apache and FastCGI. If you don’t need lighttpd, you can skip the next section.

**lighttpd**

The lighttpd web server runs independently of the other websites on your shared server, so you must include a complete `lighttpd.conf` to start it. If you have installed lighttpd on your development machine, Rails will copy a minimal config file to the `config` directory that you can use as a reference. This file is also located inside the Rails gem in the `rails-1.0.0/configs/lighttpd.conf` directory.

If you are running several domains or subdomains with one instance of lighttpd, you should keep the `lighttpd.conf` in a directory outside of any specific Rails application. A good place might be `~/config` or `~/lighttpd` in your home directory.

I wrote a specialized Capistrano recipe that builds a `lighttpd.conf` customized for running lighttpd on TextDrive. It could easily be customized to use the filepaths of other hosts as well. You can find it at the Shovel page.\(^5\)

### 3.8 Application Setup: Rails Config Files

A freshly generated Rails application only needs a minor amount of customization to run in production mode.

**config/database.yml**

First, make sure that you have edited `config/database.yml` with the appropriate information for the database you created earlier.

```yaml
production:
  adapter: mysql
  database: db_production
  username: db_user
  password: 12345
  host: my_db.brainspl.at
```

In my experience, you can omit the socket attribute for most shared hosts. Some shared servers are configured to use localhost as the host, where others require you to create a separate subdomain for your database.

---

\(^5\) [http://nubyonrails.com/pages/shovel](http://nubyonrails.com/pages/shovel)
Finally, make sure you use the correct username for the database. It may be different from the account you use to SSH to the server.

For the most security, you shouldn’t keep this file under source code control. A common technique is to copy this file to a safe place on your server and add an *after_update_code* task to Capistrano that copies it into the live application after it is deployed. (See Chapter 5, *Capistrano*, on page 90 for more details.)

**RAILS_ENV**

Rails was intelligently designed to run with different settings for development, test, production, or any other environment you define. It will use the ‘development’ environment unless told otherwise. When running on your shared server, you will want ‘production’ mode to be in effect.

There are at least three ways to set the RAILS_ENV for your application, each with different repercussions. The goal is to set it in a way that will take effect on the server, but not on your local development machine.

**Option 1: Set the environment in your .bash_login file on the server**

The best way is to set the actual RAILS_ENV environment variable. Rails and Capistrano work best with the bash shell.

```
ezra$ ~/.bashrc
export RAILS_ENV="production"
```

This is the most comprehensive way since migrations and other scripts will also use that environment. Keep in mind this only works if your SSH user runs your web server. If not, you'll need to use one of the following approaches.

**Option 2: Set the environment in your web server config file**

If you can’t set the Rails environment variable in your shell, you must look for another way to do it. The next best place is in your web server configuration. Most shared hosts won’t let you set environment variables from a local *htaccess* file. You must use one of the other options if you are using Apache.

However, you can set the environment if you are using lighttpd. Edit the bin-environment directives in the FastCGI section of your server’s *lighttpd.conf*:

```
fastcgi.server = ( ".fcgi" =>
    ( "localhost" =>
```


This setup can work well even if you use lighttpd locally for development. When you start `./script/server` for the first time, Rails creates a file in `config/lighttpd.conf` that sets RAILS_ENV to 'development.' When you start your local server with `./script/server`, that script uses the `lighttpd.conf` file to start lighttpd. If you put your shared server's `lighttpd.conf` in a different location, you will have harmony between your local and remote environments.

In practice, most seasoned Rails developers run lighttpd this way, since most people want to run several domains or subdomains with one lighttpd server. Your `lighttpd.conf` is usually located somewhere in your home directory and your Rails applications are located in a subdirectory (such as `sites`). Even though each Rails application may have its own `lighttpd.conf` file, these will be ignored on the production server, exactly as they should be.

**Option 3: Edit environment.rb**

The final way to set the environment for a shared host is to uncomment the following line at the top of `environment.rb`:

```ruby
ENV['RAILS_ENV'] ||= 'production'
```

Normally, Rails defaults to development mode, expecting other environments to specify a different RAILS_ENV if necessary. Uncommenting this line changes the default to production and shifts that responsibility to the other environments. In practice, production is a much better default. If you use lighttpd for development, Rails will make a `lighttpd.conf` file for you that explicitly specifies development mode. WEBrick will use development mode unless explicitly told to do otherwise. Either way, you get the right environment.

### 3.9 The Well-Behaved Application

The shared hosting environment is a jungle of constantly changing elements. One of the otherwise peaceful co-inhabitants of your server may
momentarily take the lion’s share of resources. Your host may upgrade software or hardware for maintenance or out of necessity. Your host may enforce resource limits and kill your application if it loads down the CPU for too long. By following a few simple guidelines, you can make your application behave as well as possible and also protect yourself from other poorly behaved applications.

**One Rails App Per User**

Some shared hosts allow you to create several user accounts and each user account has its own memory allowance. So, you will benefit if you run one Rails application per user account. If you need to run another application, you should create a new user account and run the app under that user.

**Be Miserly With Memory**

A bare Rails application with no other libraries will use 30-50 MB of memory. Adding the RMagick image manipulation libraries can easily push that over 100 MB. Unlike VPS servers, shared servers don’t usually sell plans where a fixed amount of memory is guaranteed to your application. However, there is a fixed amount of total memory available to all applications on the server, and some shared hosts will periodically kill processes that use more than their share, usually about 100 MB.

This practice of killing processes is especially problematic if you are trying to run the lighttpd web server. If the host’s maintenance bot kills your lighttpd daemon, lighttpd will not restart itself automatically. To make matters worse, some hosts restrict the use of automated scripts that restart dead or zombified FastCGI processes. Even though the Apache web server can leak memory when running FastCGI processes, it will automatically restart them if they have been killed. The bottom line is that you need to conserve memory and make sure you don’t have any leaks. Rails doesn’t have any silver bullets for dealing with memory leaks, but I’ll tell you what I know throughout the chapters that follow.

The cruel reality is that a Rails app can outgrow the shared server environment. I wrote an application for a client that used several large libraries including RMagick and PDF::Writer. The overall memory requirements meant that the app was too large to run reliably within the constraints of a shared host. Both are useful libraries, but if they cause your app to use too much memory you must either reconsider your
choices or move to a virtual private server. I’ll walk you through memory in the (as yet) unwritten *chp.performance*.

In all likelihood, you already know that some applications just won’t work in a shared hosting environment. If you’re ramping up the scalability curve on the next Facebook application, you already know that shared hosting is not the ultimate answer. And if you have applications with intense number crunching, your shared hosting provider and anyone on your box will curse you until you give up. As a hosting provider, I’m watching you. Do the right thing.

### 3.10 Troubleshooting Checklist

Rails deployment means more than dropping in a JAR file or a PHP file. Even if you follow the instructions above precisely, your installation may not run smoothly. Here are some common problems and ways to easily fix them.

**Look at the web server error logs (in addition to Rails)**

One of the best places to start troubleshooting are the web server’s error logs, especially when you are initially debugging your configuration. Rails can’t start writing to `production.log` until it has launched, so Rails logging can’t help you if your initial setup has critical problems. File permission problems and other errors will show up in the web server’s `access_log` and may give you clues about what is going wrong.

The `tail` command is often the most useful way to view your logs. Normally, `tail` shows the last ten lines of a file. You can ask for more lines with the `-n` argument (e.g., `tail -n 50 access_log`). For realtime output, the `-f` argument will continue to print new lines as the web server writes them. On Unix-based systems, you see this kind of output when you run Rails `script/server` command during development.

With some versions of the `tail` command, you can even tail several logs simultaneously. If your operating system doesn’t include a capable version of `tail`, you can download a version written in Perl by Jason Fesler at 6.

In any case, find where your host keeps the httpd logs and tail them all:

ezra$ tail -f log/production.log log/fastcgi.crash.log
  httpd/error_log httpd/access_log

Refresh any page of your site and you should see some kind of output in the logs.

[Mon April 17 11:20:20 2007] [error] [client 66.33.219.16] FastCGI:
  server "/home/ezra/brainspl.at/public/dispatch.fcgi"
  stderr: ./../config/../app/helpers/xml_helper.rb:11:
    warning: Object#type is deprecated; use Object#class

A common error is Premature end of script headers which is quite vague and usually signals a problem that needs to be debugged separately. However, file permission errors will show up with the full path to the file or folder that has the problem.

**Do files have the correct permissions?**

Rails has to write to several kinds of logs, so those files and folders need to be writeable by the user that runs the FastCGI processes. Usually this login is the same as the user account used to SSH to the server, but it might be different. If you have other FastCGI processes or Mongrels already running, you can run the top command and see what user is running them:

```
ezra$ top
  PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
25698 ezra 11 2 22072 21m 2288 S 0.0 1.1 0:03.30 dispatch.fcgi
```

Here are a few important files and the permissions they must have:

- The `log` directory and files must be writeable by the user running the FastCGI or Mongrel process. With Capistrano, the `log` directory is a symbolic link to the `shared/logs` directory. Make sure both, and the files inside, are writeable.

- The `public` directory must be writeable by the user running the FastCGI or Mongrel process if you are using page caching. Rails will run if the public directory is not writeable, but will not use caching.

- `public/system` directory: Capistrano makes a system directory inside public. It also makes links from this directory to the shared directory holding the core Rails logs, including `production.log`. The benefit is that you can store uploads and other files there and they will remain between deployments. If you keep the page cache or uploads there, the folder (and the link) must be writeable by the FastCGI user.
• \texttt{dispatch.fcgi} must be executable, but not writeable by others. Properly configured web servers will refuse to execute files if \texttt{dispatch.fcgi} is writeable by the general public. \texttt{dispatch.fcgi} is the main file that handles requests and fires them off to the controllers in your application, so it must be executable.[[Author: Ezra, we don’t care about Mongrel, right? It’s packaged as a gem, right?]]

If you have permission problems with any of these files, you can remedy the problem with the Unix \texttt{chmod} command. The \texttt{chmod} command has many capabilities which are beyond the scope of this book. You can read about it in excruciating detail in the Unix manual page by running the command \texttt{man chmod}.

\begin{verbatim}
ezra$ chmod 755 public/dispatch.fcgi
ezra$ ls -l public/dispatch.fcgi
-rwxr-xr-x 1 ezra group 855 2006-01-15 02:03 dispatch.fcgi
\end{verbatim}

The \texttt{rwx} means that \texttt{ezra} can read, write, and execute the file. The first \texttt{r-x} means that only others in the same group can read and execute the file. The last \texttt{r-x} means anyone can read and execute, but not write the file. This permission set is the proper setup for the \texttt{dispatch.fcgi} file.

\textbf{Did you specify the correct path to the Ruby executables?}

When generating a brand new Rails application, Rails uses the location of your local Ruby executable to generate the dispatch scripts in the public directory and all the scripts in the script directory.

Your shared server may not have a copy of Ruby in the same location. For example, I used MacPorts to build a fresh copy of Ruby on my development machine, so my local copy of Ruby is located at /\texttt{opt/local/bin/ruby}. Even though I made a symbolic link from /\texttt{usr/bin/ruby}, Rails builds all my script files like this:

\begin{verbatim}
#!/opt/local/bin/ruby
# ERROR: Incorrect for most production servers!
\end{verbatim}

If you’ve already generated your application, you need to manually edit your configuration files, including possibly \texttt{dispatch.fcgi} and others, to match the actual location of Ruby on your shared server. You can find this information by connecting to your server via SSH and issuing this command:

\begin{verbatim}
ezra$ which ruby
/usr/local/bin/ruby
\end{verbatim}
If you know this information before you start building your application, you can send it to Rails as you generate your application:

```
ezra$ rails my_rails_app --ruby /usr/local/bin/ruby
```

All the relevant files will then start with the correct location:

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

If your development machine doesn’t have a link to Ruby in that location, you can make one to match your production server.

```
# Link to the actual location of Ruby from an aliased location
ezra$ sudo ln -s /opt/local/bin/ruby /usr/local/bin/ruby
ezra$ sudo ln -s /opt/local/bin/ruby /usr/bin/ruby
ezra$ sudo ln -s /opt/local/bin/ruby /home/ezra/bin/ruby
```

Now I can set the application’s shebang\(^7\) to the location on the remote server, but the application will still run on my development machine.

**Does the sessions table exist in the database?**

Some of the most baffling errors happen when Rails can’t save its session data. I’ve gotten a completely blank page with no errors in any of the logs. Often, the problem is an unwriteable /tmp folder or an absent sessions table. Fortunately, Rails 1.1+ has been enhanced to give a more informative message when this happens.

If you are storing user sessions in the database (as you should be), you may have started out by creating the sessions table in your development database like this:

```
ezra$ rake db:sessions:create
```

This will create a numbered migration file that can be run against the production database to add a sessions table. Then, you’ll have to add the database sessions to your environment by the line config.action_controller.session_store = :active_record_store to environment.rb.

**Are current versions of necessary files present?**

Rails is a complete framework and expects to find files in certain places. It is common to omit core files with the svn:ignore property while developing. However, those files must be included in your build process on the production server.

\(^7\) shebang refers to the #! characters. The #! specify the interpreter that will execute the rest of the script.
A common example is `database.yml`. For security reasons, people don’t want to have their name and password flying all over the internet every time they checkout the code for a project. But if it isn’t on the production server, Rails won’t be able to connect to the database at all.

A solution is to save it to a safe place on the server and make an `after_update_code()` task to copy it into the current live directory. (See Chapter 5, *Capistrano*, on page 90 for a detailed example.)

If you have saved a copy of the core Rails libraries to the `vendor` directory, your application will not run unless they are all there. And like the rest of Rails, file and directory names matter. I once saw an odd situation where the built-in `has_many()` and `belongs_to()` methods were causing errors. The rest of the application ran fine until the programmer asked for data from a related table. We discovered that the actual filenames in ActiveRecord had somehow been truncated and were causing the error.

**Is the RAILS_ENV environment variable set correctly?**

One of the most common problems during deployment is an incorrect RAILS_ENV. For your production server, RAILS_ENV should be set to `production`. There are several ways to set RAILS_ENV and several ways to determine the current setting of RAILS_ENV. The only thing that really matters is the value of RAILS_ENV inside your Rails application, and there is no direct way to test that (apart from a fully running application).

This can also be confusing since some scripts don’t tell you what environment they are using, and others don’t tell you what environment they are using until they are running properly.

However, you can use the following troubleshooting tools to find out what RAILS_ENV might be.

**The about script**

Rails 1.0 and above ship with a script that prints useful information about your configuration. Unfortunately, this will not be accurate if the environment was set in `lighttpd.conf`.

```
ezra$ ./script/about
About your application's environment
Ruby version 1.8.6 (i686-darwin8.9.1)
RubyGems version 0.9.2
Rails version 1.2.3
```
Active Record version 1.15.3
Action Pack version 1.13.3
Action Web Service version 1.2.3
Action Mailer version 1.3.3
Active Support version 1.4.2
Application root /Users/ez/brainspl.at/current
Environment development
Database adapter mysql
Database schema version 3

Echo

If you have set the environment in your shell, you should be able to SSH to your shared server and print it out like this:

ezra$ echo $RAILS_ENV
production

It should also take effect when you start the console.

ezra$ ./script/console
Loading production environment.
>>

**TIP Comprehensive list of script arguments**

For reasons known only to God, each of Rails’ scripts uses a different argument for setting the environment manually. If you want to run under a different environment than the default, you will need to provide the proper argument. All of these scripts can run with no arguments and will default to the value set in the shell or to development.

If your mind is like mine and turns to mush after 10pm, you might find this table useful:

<table>
<thead>
<tr>
<th>Missing: Table of args to scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Is the database alive and present?</strong></td>
</tr>
</tbody>
</table>

Although Rails can run without touching a database, most applications use the database in some way. If the database doesn’t exist, your application will not run and will show errors.

You can test the database independently of the rest of your application. Call up the console in production mode and you should be able to send simple queries to the database.

ezra$ ./script/console production
Loading production environment.
>> User.find 1
=> #<User:0x263662c @attributes=...
• Can you connect directly to the database with a command-line client such as mysql or psql? If not, the database may be down or may not be accessible from your shared server. This method is not always conclusive. I have used hosts where the command-line MySQL client cannot connect to the server, but the Rails application runs without any problems.

• Do you have the proper database, username, password, host, and port specified in the production section of config/database.yml?

Has the database been migrated to the correct version for your application?

You may have run your migrations, but was the proper database affected? If RAILS_ENV is missing from the shell you may have migrated your development or test database instead of the production database.

The easiest way to discover your current schema_version is to use the ./script/about command. Newer versions of Rails will display the schema version on the last line.

```
ezra$ ./script/about RAILS_ENV=production
About your application's environment
Ruby version 1.8.6 (i686-darwin8.9.1)
RubyGems version 0.9.2
Rails version 1.2.3
Active Record version 1.15.3
Action Pack version 1.13.3
Action Web Service version 1.2.3
Action Mailer version 1.3.3
Active Support version 1.4.2
Application root /Users/ez/brainsplat/current
Environment development
Database adapter mysql
Database schema version 47
```

If this doesn't show the version you expect, you may need to run your migration again or check your other database connections to make sure the correct database is being addressed. To re-apply the migration manually, issue this command:

```
ezra$ rake db:migrate RAILS_ENV=production
```

3.11 Conclusion

When possible, I run applications on a VPS or dedicated server. However, I still have several shared-hosted applications that run with an
acceptable degree of reliability. By following the steps mentioned here you can also run small applications reliably on an inexpensive, affordable shared host.

In this chapter, you learned to set up a typical shared hosting Rails environment. You also learned that you can’t push shared hosts too hard, and you can’t rely on them for perfect service. In the next chapter, you’ll see the alternative: virtual and dedicated hosting. If you’re bursting at the seams and hearing your neighbor through paper thin walls, it’s time to move up. Read on.
Chapter 4

Virtual and Dedicated Hosts

Most shared host plans are roughly equivalent to a 7' x 7' apartment with a shared bathroom and no kitchen. After a little time on that shared host, it may start to feel pretty cramped. When you come to a point where you are pushing the limits of your shared host, it’s time for your own virtual private server or dedicated server. You will be able to stretch out and take over the whole environment without worrying about fighting others for your cpu time and memory. You’re probably thinking to yourself, "Ah, the good life."

Not so fast. With your newly-found space and flexibility comes great responsibility. No one else will hold your hand and watch over your server, unless you’re willing to pay big bucks for a fully managed server. You will have to decide whether that extra cost of disk redundancy through RAID is worth it; you will be responsible for backing up your system and restoring the data should something go wrong. For better or worse, you are living the great American dream: full home ownership. This chapter will walk you through the move-in.

4.1 The Lay of the Land

In this chapter, I’ll show you how to build out a server from scratch. You’ll first build and install your operating system. Then, you’ll build some of the tools that you will need to build the Ruby stack. You will move on to build out the Ruby stack, including Ruby, Rails, RMagick, and Mongrel. You will also install a database and your web server, though you won’t integrate your web server until Chapter 7, Scaling Out, on page 141. You’ll have a working Rails installation like the one in Figure 4.1, on the following page.
In practice, the setup will not work much differently from the one you saw in Chapter 3, *Shared Hosts*, on page 42. You will serve each request with a single Mongrel. That architecture will not scale, but your emphasis is on building a workable foundation that you can use as a foundation. In later chapters, you’ll cluster your mongrels and then, if your application requires it, you’ll scale out using one of the options in Chapter 7, *Scaling Out*, on page 141. For now, focus on building all of the pieces that you’ll need through the rest of the book.

**Introducing Your Own Host and Administrator**

As if deploying a Rails application to someone else’s server wasn’t enough fun, now you’re ready to build your own machine. Whether you want the role or not, you’re an administrator. You earned that rank the moment you built your own environment. I recommend you take your new role seriously. As an administrator you have at many responsibil-
ities, topics which other books will do a better job of exhausting. You can group your responsibilities into these categories:

- Security and Stability
- Configuration and Upgrades
- Documentation

Keeping your server secure and stable is a tough business and is a subject I won’t even attempt at tackling in this book. Managing configuration and upgrades can be a weekly occurrence, and patching the system is an important security practice. But the one that I really want to drive home is documentation. Developers have a bad habit of avoiding documentation. Programmers I know use insanely creative rationalizations. Here are a couple of the best. You may not think you have the time or skill to pull it off, but you’ll pay with your time now or later. You might think that documentation is always out of date and unable to keep up with changes in code, but I’m not talking about code. The configuration won’t change as fast as code and even if it does, that’s all the more reason to document it. None of the old arguments against documentation work for infrastructure. As system administrators, you have to document our configurations. The following tips will help with managing documentation:

- Keep a server journal next to the machine or in a common place if there are multiple administrators or remote servers. Treat it like the conch from Lord of the Flies: only the person with the journal can modify the server configuration. In the journal, record and date which changes were made and why.

- Keep a directory containing dated session logs in a well known directory. If at all possible, comment the logs with “why” statements. It’s often easy to see what someone did with the logs, but the "why" needs to be filled in to communicate with others.

- Update formal documentation once per week or anytime a major change is made. Formal documentation includes simple diagrams and organized sections of documentation for key infrastructure components.

- Make sure everyone reads the documentation. Let people know when you update the documentation and walk them through it over coffee. These practices will ensure that they understand the
system and also encourage discussion and questions about the
system.

When you must move to multiple servers, you will need to do your best
to keep the configurations in sync. And by "in sync", I mean preferably
identical. You will want to try to automate differences in configuration
by your application or even capistrano, and your service provider will
manage others for you. To keep your configurations identical, you will
need good organization and up-to-date documentation. I've preached
enough for now. Roll up your sleeves and let's get to work.

4.2 Virtual Private Servers

Even after you've decided that shared hosting is not enough, one size
does not fit all. Before you decide to spring for that dedicated host pack-
age, take a deep breath and look at another attractive alternative first.
The virtual private server, or VPS for short, is the first logical step up
from a shared server plan. Some hosts might call these virtual dedi-
cated servers or VDS. These type of servers run in a virtual machine.
Multiple VM instances run on one physical hardware server. Before you
run away kicking and screaming, you need to know that a VPS is not
a shared server package. You will get complete root access to your VPS
and often you'll even pick your operating system from some Linux dis-
tribution or FreeBSD.

Your host may run one of quite a few different server virtualization soft-
ware packages. Out of all of these that I have tried, Xen is my favorite.
Xen is a relative newcomer to the virtualization scene but the open
source package is built right into certain Linux kernels, so the virtu-
alyzed server processes run a little closer to the metal. Xen also offers
superior disk IO which is a big issue for anything that deals with many
files. And guess what folks? The majority of the time, a web applica-
tion does nothing more than deliver file after file to the user. More and more
web hosts are making Xen based virtual servers available to hosting
clients so it shouldn't be hard to find one.

Once you acquire your own VPS it acts like a dedicated box. You have
full root access to install or remove anything you need. On a Xen-based
VPS you could even recompile your own kernel if you wanted to (but you
should ask your host provider first). Since most virtual servers will run
on high-end hardware, you can get very good performance. Of course,
the more VPS’s that your host tries to squeeze onto one physical box,
the less resources there are to go around. It is a good practice to ask the provider about the hardware setup and exactly how many virtual machines they run per box. Generally speaking, bigger slices on bigger boxes are better. If you need help interpreting the numbers, ask an expert. You might pay one if you have a lot of money riding on the answer, or you might simply post the question on one of the many excellent Ruby on Rails forums.

Memory

Usually, your performance bottleneck will not be processing power, but memory. Some shared hosting providers oversell their hardware in the hopes that not everyone will be running full blast at once. When too many customers need too much, your VPS can easily run out of memory and start swapping out memory pages to disk. To understand the impact on performance, imagine an Olympic sprinter running at full speed in perfect conditions, and then plunging him in water up to his chest to finish the race. But with Xen-based VM’s, the memory allocation you get for your server is your memory only. A Xen-based architecture will not allow a hosting provider to oversell the memory of the physical box, and your application can run in the clean air of memory instead of the quagmire of disk swapping. Products like Virtuozzo and OpenVZ are a few to watch out for.

Depending on what you’re doing, I’d recommend a minimum of 160-256MB of RAM on your VPS. This amount of memory will allow you to run one or two small Rails sites, depending on the application’s resource usage. But you can be more precise. Rather than take a blind guess, you can estimate how much you will need based on one critical question: how many Mongrels or FastCGI listeners will you need?

One or two backend processes is plenty for many Rails applications. A typo or Mephisto blog that gets a medium amount of traffic will usually be fine on one process. A typical Rails process can take anywhere from 35-120MB, but some Rails application may take more. Keep in mind there are always exceptions to the rule and you should test locally to see what your memory consumption is before you order your VPS. I’ll show you how in the (as yet) unwritten chp.performance

Even if you get your initial memory size wrong, a VPS system is very easy to upgrade. If you need more RAM, disk space or other resources, usually all you have to do is request these from your host and reboot your VPS. When it comes back online you will have the new resources
available without the need to change anything in your settings to take advantage of them. Another benefit of the virtual server approach is the ability to easily migrate your entire server to another physical box or host when the time comes. If you choose the right provider, upgrading with your traffic should happen smoothly.

**Using Lightweight Web Servers**

Mongrel is emerging as the defacto method of deploying a Rails application on a VPS. You can run a blog or smaller apps on one Mongrel alone. Should you need another web server in front of Mongrel for static content, using nginx or lighttpd can be a huge win. These servers use fewer resources than Apache and are very fast. If you want an alternative to Mongrel, nginx and lighttpd have FastCGI support that is top notch and stable.

Most hosting providers offer a number of Linux distributions to choose from. Primarily, you want a distribution with a minimal footprint. When you run on a smaller memory system, make sure to install only what you need and no more. From there, you should build only what you need. In the end you will come out with a leaner, faster server.

The instructions for setting up a VPS are basically the same as setting up a dedicated server. You’ll need to know a little more about dedicated hosting before I move into the setup tutorial.

### 4.3 Dedicated Servers

Say you have written the latest popular Rails web 2.0 application and a shared host is no longer enough. Your shared host admin is screaming at you about resource usage, he’s not responding to your requests for support quickly enough, and you’ve decided to either challenge him to a duel or switch to a higher plan. It’s time to move. You’re ready for root, and your customers are ready to see your fabulous content without the dreaded spinning globe, or whatever icon their browser is spinning these days.

With a dedicated server you don’t have to worry about memory constraints or disk space as much. A good starting system would have a modern processor, 512MB to 1024MB RAM minimum, and a spacious hard drive. A system like this will let your application service a lot of traffic and concurrent users. I’m not going to bore you with statistics here because there are too many variable factors to weigh, but once
you’re on a dedicated box, if your application keeps growing to the point where you need to start thinking about a cluster, you will be ready.

Typically dedicated boxes cost more than VPS systems. A starter VPS can run you $25-$60 USD per month, whereas a starter dedicated box is usually closer to $150-$300 USD.

Even if you do get your own dedicated box, you may want to consider using Xen. In the real world, Xen offers acceptable performance and gives you a nice long-term solution for scaling your system out as you grow. Installing and configuring Xen is out of scope for this book, but it is definitely worth your time to investigate this alternative. You trade a small percentage of raw performance for ease of administration and scalability. With Xen you can partition your dedicated server into a number of targeted VPS servers that you can easily move to other boxes as you grow. I’ll tell you how to do exactly this in the Chapter 7, Scaling Out, on page 141.

4.4 Setting up Shop

Building a deployment environment is not for the impatient, but with a good knowledge of the command line and a willingness to google, you should be able to build your own setup for running Rails applications in production. I’ll spell out the rest for you. This is all well-travelled territory, so if things go wrong and we don’t have the answers here, don’t be afraid to go searching for answers. I’ll show you many of the answers that you need and Google can help you find the rest.

Regardless of whether you decide to go with a VPS or dedicated server, your Rails setup will be the same. I’ll use Ubuntu Linux 7.10 Server Edition, but don’t worry if your favorite server platform is OS X, BSD or another Linux distribution. Everything pertaining to web server and Rails configuration will work pretty much the same way on any unix like operating system.

I like to build from scratch using Ubuntu Gutsy Gibbon Server. You can get the download image and instructions online\(^1\). In all likelihood, you won’t wind up building your own system. Almost any virtual or dedicated hosting provider offers this as an option when you set up your account with them. If not just ask them to install the Server version of

Ubuntu for you. They will also set up the basic network interface to work with their network and data center.

From here on out I'll assume you are starting from a working install base with the right network install. If you are installing on your own server at home or work, you can get detailed Ubuntu Server installation instructions online\(^2\).

Any virtual host provider will have some version of SSH installed and configured, but you may need to install OpenSSH2 if you’re building your own host locally. Use the ssh install package by typing `sudo apt-get install ssh` to get things working.

I can’t possibly cover all of the details for securing a Linux server, but I’ll give you a few important tips along the way. Here’s the first. Use `ssh` and `sftp` or `scp`, not `ftp` or `rsh`. If you don’t, all of your communications will be in the clear, directly readable at any of the intermediate hosts between your local machine and server. To use SSH, you will need SSH on both the client and host. I’ve already told you about the host system, so shift your attentions to the client. If you are on OS X, Linux or BSD you undoubtedly have a client installed, but if you run Windows locally, you will want to install PuTTY. The Windows defacto standard, PuTTY is an excellent and free SSH program you can find online\(^3\). Follow the instructions you find there and you’ll be ready for the rest of this book.

Building your production setup is not a trivial exercise. Here is the rundown of the whole list you’ll be installing:

- **Gnu Compiler Collection (GCC for short) and associated tools.** You’ll use it to build several components including RubyGems and the RMagick plugin.

- **Ruby and RubyGems.** You’ll get the latest stable version of Ruby and the RubyGems third party library packing and distribution system.

- **Rails.** After installing the operating system and RubyGems, installing and configuring Rails will be surprisingly easy.

- **MySQL and the mysql-ruby bindings.** You can use a variety of different database servers, but I’m going to go with the most popular Rails database engine.

---

2. [https://help.ubuntu.com/](https://help.ubuntu.com/)
• Subversion. Capistrano will checkout your code right out of your Subversion repository.

• Mongrel and dependencies. Mongrel is rapidly becoming the de facto standard for serving Rails applications.

• ImageMagick and RMagick. Many modern web applications allow uploading of images. RMagick can help by automatically creating thumbnails and cropping images.

• FastCGI Developers Kit and the Ruby FastCGI bindings. If you’re running Apache or lighttpd instead of Mongrel, you’ll want to replace the slower, default CGI right away.

• Nginx. This tiny web server is lightning in a bottle, making a good substitute for Apache with many installations. Installing it is easy if you want to go that route.

With these elements, you will have the basic stack that will serve as the basic foundation for all we do in the remainder of the book. We will cover optimizing the individual web server configurations in other chapters in the book. Once you’ve completed the steps in this chapter, you will be able to run a basic production Rails application.

I’m going to break a cardinal publishing rule here and repeat some details from Chapter 1, Introduction, on page 8. I’m doing so not to pad the book but to protect you from major havoc. You’re going to issue commands against your local box and remote servers, with user permissions and root permissions. You’ll need to understand where you are at all times and how much power (and potential for damage) you have at any given time. Moving files around and installing software can destroy lots of work if you’re not extremely careful, so look for the clues that tell you where and how you’re logged in.

If I’m logged on to my production system, the login will begin with ezra. If I’m logged in locally, the login will begin with local. In the bash shell on *nix systems, the # command line prompt tells you that you are logged in as root, and a $ means you are logged in as a regular system user. I’m going to use bash. If you want to use a different shell, make sure you understand the prompt indicators for your system.

**Configuring the Server**

It’s time to get started. I’m going to name my virtual server «tracklayer». Whenever you see «tracklayer» in the commands, replace it with the IP
address or domain name of your server.

You'll need a root user and a regular user. Admins usually set up Ubuntu with a normal user account instead of a root account only. If you are configuring most other Linux distributions, you will need to make a normal user and add yourself to the /etc/sudoers file.

If you have a normal user account, use it to log in and skip past the next two session listings. If you have only root, you'll need to create a normal user. SSH in to your new account's IP with your root user and password:

```
local$ ssh root@tracklayer
root@tracklayer password: <enter your password>
```

If all is well, create your normal user account:

```
root# adduser ezra
Adding user `ezra'...
Adding new group `ezra' (1001).
Adding new user `ezra' (1001) with group `ezra'.
Creating home directory `/home/ezra'.
Copying files from `/etc/skel'
Enter new UNIX password:
Retype new UNIX password:
# you will be asked a few more questions,
# fill them out however you like.
```

Be careful with root. After you've established your account, always login into your machine as your own user instead of root. Simply use su or sudo to gain root privileges as needed. su stands for super user, and sudo stands for super user do. If you're not logged in as root, become root now:

```
$ sudo -s
Password:
root#
```

Now edit /etc/sudoers. You should use a program called visudo to edit the sudoers file as visudo won't let two people edit at the same time.

```
root# visudo
```

Use your arrow key to move the cursor down and add this line at the end of the file:

```
yourusername    ALL=(ALL) ALL
```

Now press the escape key and type :wq. This means write file and quit.
Securing SSH

Most experienced *nix admins tend to run sshd on high port numbers. Here’s why. Crackers commonly create automated attacking programs called bots that crawl the net, visiting one machine after the next to find machines with a running SSH daemon. Then, the bot uses an automated script with a dictionary to try many combinations of user names and passwords. The attacks are so prevalent that these days, most servers on the internet will experience this kind of attack with some frequency. If you have a weak login/password combination on a live SSH port, you’re toast, especially if sshd is on the standard port 22. Since most of these attack bots will not scan ports higher than 1024, you should always assign sshd to a free port above 1024.

Modify the port for sshd by editing /etc/ssh/sshd_config. Edit the line that looks like this:

`Port 22`

Change it to:

`Port 8888  # or any unused port above 1024`

Then save the file and quit the editor. And don’t forget to restart the SSH server daemon.

For security sake, you don’t want to allow SSH root logins because an unauthorized login would be disastrous. Edit /etc/ssh/sshd_config, replacing this line:

```bash
PermitRootLogin yes
```

with this one:

```bash
PermitRootLogin no
```

Now reload its /etc/ssh/sshd_config to pick up the new settings:

```bash
root# /etc/init.d/ssh reload
```

For security and to make things easier on yourself using SSH keys instead of passwords for logins is a great technique. Let’s create a pair of public/private keys and get them installed on our new server, the key generation is done on your local machine:

```bash
local$ ssh-keygen -t dsa
```
This will prompt you for a secret passphrase. If this is your primary identity key, make sure to use a good passphrase. When this is done you will get two files called id_dsa and id_dsa.pub in your ~/.ssh dir. Note: it is possible to just press the enter key when prompted for a passphrase, which will make a key with no passphrase. This is a Bad Idea ™ for an identity key, so don’t do it! We will learn how to achieve passwordless logins in a secure manner shortly.

Now we need to place your public key on the server. Here is a nice bash function that will do this for us, place this in your ~/.bashrc or ~/.bash_profile depending on the type of computer you are using locally.

```bash
function authme {
    ssh $1 'cat >>.ssh/authorized_keys' <~/.ssh/id_dsa.pub
}
```

Once that is in your shells rc file you will need to start a new shell or source the file:

```
local$ . ~/.bashrc
```

With this all in place we can now use the authme command to place your new keys on the server( replace tracklayer with your IP or hostname):

```
local$ authme tracklayer
```

You will be prompted for the password and your key will be placed on the server. Now you will want to enter a new shell with your ssh keys loaded> This will allow you to start a shell and enter your passphrase for your private key only once and then you will be able to ssh to anywhere your key is placed without entering the passphrase again:

```
local$ ssh-agent sh -c 'ssh-add < /dev/null && bash'
```

Now you can ssh to your new server with no passphrase entry:

```
local$ ssh tracklayer
```

**Install the GCC tool chain**

You will need to install a compiler tool chain to build and install many elements including RubyGems. The build-essential package has everything you need to build the components you’ll need to install later. Install it on Ubuntu like so:

```
root# apt-get install build-essential
```
Install Ruby and RubyGems

By default, Debian and Ubuntu have five package repositories called *main, restricted, universe, multiverse*, and *commercial*. For the setup in this chapter, you will need the universe package repository. By default, it is not enabled. Fix that by editing the /etc/apt/sources.list file and uncomment the following two lines:

```
deb http://us.archive.ubuntu.com/ubuntu/ dapper universe
deb-src http://us.archive.ubuntu.com/ubuntu/ dapper universe
```

Now, update your apt-sources file and install Ruby and friends. You will need at least Ruby-1.8.4 to run Mongrel and Rails 1.2.x. Ubuntu’s package manager will have at least Ruby-1.8.4. If you are using a more recent version of Ubuntu then you will probably get 1.8.5.

```
root# apt-get update
root# apt-get upgrade
root# apt-get install ruby ri rdoc irb ri1.8 ruby1.8-dev libzlib-ruby zlib1g
...
root# ruby -v
ruby 1.8.4 (2005-12-24) [i486-linux]
```

Ruby is live. If you want to verify that fact, run *irb*, but for now, I’ll press onward. Install RubyGems. You really don’t want to install Rails without it. Ubuntu and Debian do not officially package RubyGems, so you will need to build it from source. Go to RubyForge and download the latest stable version (0.9.4 at the time of this writing), and then build and install it like this:

```
root# wget http://rubyforge.org/frs/download.php/17190/rubygems-0.9.2.tgz
...
root# tar xvzf rubygems-0.9.2.tgz
...
root# cd rubygems-0.9.2/
root# ruby setup.rb
    ---> bin
    <--- bin
    ...
    Successfully built RubyGem
    Name: sources
    Version: 0.0.1
    File: sources-0.0.1.gem
    Removing old RubyGems RDoc and ri...
    Installing rubygems-0.9.2 ri...
    Installing rubygems-0.9.2 rdoc...
```

Building the latest Ruby from source.

Ubuntu and Debian releases often do some strange things with Ruby. A given release may break Ruby up into tiny pieces, or install some earlier release of Ruby. If you want Ruby-1.8.6, you will need to build from source.

The first step is getting the latest stable release of Ruby. In your web-browser go to the Ruby home* and download the desired release. I’m going to install Ruby-1.8.6.

```
ezra$, wget ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.6.tar.gz
```

Now, unpack, build and install Ruby.

```
ezra$, tar -xvfz ruby-1.8.6.tar.gz
...
ezra$, cd ruby-1.8.6
  ezra$, ./configure && make && sudo make install
...
```

You will need to make sure your $PATH has /usr/local/bin in it.

```
ezra$, echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

If you don’t have /usr/local/sbin:/usr/local/bin in your $PATH then you will need to add it. Open /etc/profile with your editor and add the following line:

```
export PATH=/usr/local/sbin:/usr/local/bin:$PATH
```

That is the basic steps required to build Ruby from source. Please note that if you choose to build your own Ruby, it will not work with the Ruby packages in Ubuntu. This means you will have to install the mysql and rmagick gems through Ruby Gems and not through apt-get.

As of RubyGems 0.8.0, library stubs are no longer needed.
Searching $LOAD_PATH for stubs to optionally delete (may take a while)...
...done.
No library stubs found.

Remember to cleanup after yourself. Delete the RubyGems source files:

```bash
root# cd ..
root# rm -rf rubygems-0.9.2
```

Now you are ready to move into some more familiar territory. You’ve already installed Rails on your development machine, and now you’ll do the same for your production setup. The component version numbers will probably be higher by the time you read this, but the following command will install the latest stable version of Rails. The include-depencies option will make sure that you have all necessary dependencies:

```bash
root# gem install rails --include-dependencies --no-rdoc --no-ri
```

Successfully installed rails-1.2.2
Successfully installed rake-0.7.1
Successfully installed activesupport-1.4.1
Successfully installed activerecord-1.15.2
Successfully installed actionpack-1.13.2
Successfully installed actionmailer-1.3.2
Successfully installed actionwebservice-1.2.2

Usually, your server-side installation won’t need the documentation, so the --no-rdoc --no-ri flags will skip them and keep your installation lean.

Mongrel is next. If you’ve suffered through building and installing Apache, you’ll really appreciate the following command:

```bash
root# gem install mongrel mongrel_cluster --include-dependencies --no-rdoc --no-ri
```

Select which gem to install for your platform (i486-linux)
1. mongrel 1.0.1 (ruby)
2. mongrel 1.0.1 (mswin32)
3. mongrel 1.0 (mswin32)
4. mongrel 1.0 (ruby)
5. Skip this gem
6. Cancel installation

When multiple versions of a gem are available, RubyGems will prompt you for the version and platform you want. I’m on Ubuntu, so choosing 1 gives me the latest compatible version. Most of the time, Mongrel is enough. If I need more performance, I put a proxy in front of Mongrel to serve static content. I’ll talk more about that in Chapter 7, Scaling Out, on page 141. But now, it’s on to the database.
Install MySQL

Next, you will install MySQL and the MySQL-Ruby bindings. You don’t have to use MySQL—there are many available database engines that work quite well with Rails. PostgreSQL is another popular choice. You will also install the zlib1g-dev package as it is a requirement for Ruby Gems and a few other things you will need along the way.

```
root# apt-get install mysql-server-5.0 mysql-client-5.0
    libmysqlclient15-dev libmysqlclient15off zlib1g-dev
libmysql-ruby1.8
```

You’ve installed the database server, but don’t forget to set the root password:

```
root# mysqladmin -u root password <your password here>
```

You can easily verify that the MySQL-Ruby bindings work correctly with a simple `require` command in irb:

```
root# irb
irb(main):001:0> require 'mysql'
=> true
irb(main):002:0> exit
```

This `require` command tells Ruby to load the `mysql` library that provides basic Ruby support for MySQL. Since Rails uses the same bindings, if the require returns true, Rails will probably work too.

So far, you’ve installed Ruby, Ruby on Rails, RubyGems, Mongrel and MySQL. I’m going to walk you through installing nginx and FastCGI as well. You’ll need nginx if you want to use nginx for load balancing and static content. FastCGI is the best alternative to Mongrel, should you ever need an alternative. If you want, you can skip these steps and pick them up later.

Install nginx and FastCGI

Install libfcgi-dev and libfcgi-ruby1.8 like this:

```
root# apt-get install libfcgi-ruby1.8 libfcgi-dev
```

In order to check that fcgi-ruby works, make sure neither installation returns any errors. You want to be sure that you successfully installed the C extension version of ruby-fcgi and not just the pure ruby version. The C extension is much faster then the pure ruby version. Just as you did with MySQL, you’ll use a `require` statement to make sure you have the right libraries installed:
root# irb
.irb(main):001:0> require 'fcgi.so'
=> true
.irb(main):002:0> require 'fcgi'
=> true
.irb(main):003:0> exit

If both requires returned true, you’re ready to proceed. You need the Perl Compatible Regular Expression Library (or libpcre) for the rewrite module in nginx to work properly. You also need the OpenSSL library and development package for SSL support in nginx:

root# apt-get install libpcre3-dev libpcre3 openssl libssl-dev

Now, for the bad news. nginx does not have an Ubuntu package at this time so you’ll have to build it from scratch. Get the latest release\(^5\) (nginx 0.5.31 at the time of this writing). Get it and build it like so:

root# wget http://sysoev.ru/nginx/nginx-0.5.24.tar.gz
...  
root# tar xzvf nginx-0.5.24.tar.gz
...  
root# cd nginx-0.5.24
root# ./configure --with-http_ssl_module
...  
root# make
...  
root# make install
...

That’s it for nginx. As always, clean up after yourself and add nginx to the $PATH:

root# cd ..
root# rm -rf nginx-0.5.24*

The default nginx installation directory is /usr/local/nginx so you need to add /usr/local/nginx/sbin to the $PATH. Open /etc/profile with your editor and add the following line:

export PATH=/usr/local/nginx/sbin:$PATH

Check that nginx is working and in your $PATH. The version command option should do the trick:

root# nginx -v
nginx version: nginx/0.5.24

\(^5\) [http://www.nginx.net/](http://www.nginx.net/)
Install ImageMagick and RMagick

You’ve installed a pretty good stack. Many Rails applications will also need ImageMagick and RMagick to process image upload and manipulation. This set of libraries gives everyone a little trouble, so pay close attention:

```
root# apt-get install imagemagick libmagick-ruby1.8
     libfreetype6-dev xml-core
```

Check to see that RMagick works. Put an arbitrary image file called `test.jpg` in your current working directory for a test and run the following command:

```
root# irb
irb(main):001:0> require 'RMagick'
  #=> true
irb(main):002:0> include Magick
  #=> Object
irb(main):003:0> img = ImageList.new "test.jpg"
  #=> [test.jpg JPEG 10x11 DirectClass 8-bit 391b]
  scene=0
irb(main):004:0> img.write "test.png"
  #=> [test.jpg=>test.png JPEG 10x11 DirectClass 8-bit]
  scene=0
irb(main):005:0>
```

RMagick should now work fine, but it is notorious for being hard to install. If you run into any issues getting RMagick working you can look at the Install FAQ on the website:


Installing Subversion

To get things ready for Capistrano, you’ll need to install Subversion. Let’s install it now.

```
root# apt-get install subversion subversion-tools
```

When you install subversion-tools, it will pull in the exim SMTP server as a dependency. Configuring your mail server is beyond the scope of this book, but during the install you will be prompted to choose the "General type of mail configuration" you want. Choose "internet site; mail is sent and received directly using SMTP".

You will need to create new Subversion repositories that you can reach from your development machine. If you don’t have Apache installed, the best way to run Subversion over the network is with `svnserve` or
over svn+ssh. See the Chapter 2, *Refining Applications for Production*, on page 19 for instructions on setting up and using Subversion for your Rails projects.

**Test it and tweak it!**

Now you should be in great shape for Rails deployment. Generate a skeleton Rails app and fire it up with Mongrel to make sure everything is working fine. Switch to your normal user now because you don’t need root for Rails development:

```
root# su yourusername
ezra$ cd ~
ezra$ rails test
```

Take it for a test drive!

```
ezra$ cd test
ezra$ ruby script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Starting Mongrel listening at 0.0.0.0:3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready. TERM => stop. USR2 => restart. INT => stop (no restart).
** Rails signals registered. HUP => reload (without restart). It might not wo...
** Mongrel available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Since you are running Rails 1.2.3 release or later, the `script/server` command will start Mongrel instead of webrick. If it starts OK, point your browser at [http://tracklayer:3000](http://tracklayer:3000). Remember to replace tracklayer with the IP or domain of your server.

You should see the “Congratulations, you’ve put Ruby on Rails!” page and we are done with the basics for our sweet Rails server stack! You’re the captain of your own ship now.

### 4.5 Wrapping Up

Running your own Rails server is a rewarding experience. With this basic stack in place you can start to build your empire and tweak it to your every desire. With nginx and Mongrel installed you have many configuration options to try out on your new server. Now that the basic building blocks are in place, you’re ready to use the techniques you’ll
find in the rest of this book to make your deployment scale, and make it screaming fast.

In this chapter, you’ve walked through building your basic installation. You have the components to run Ruby via Mongrel or another web server. In the chapters that follow, you’ll put each of those components through their paces. You’ll start by building some scripts to repeatably and reliably deploy your applications. If you’re excited and ready to move in to this brand new home, read on. Moving in with Capistrano is next.
If you’ve ever rented an apartment or bought a house, you know that the financial transaction is only the first tiny step. Moving in comes next, and the process can be, well, overwhelming. In the last chapter, you used FTP to install your application onto the shared host. You simply found the files you needed to copy, and you used Subversion or FTP to push the whole Rails project directory up to your server, wholesale. But the FTP approach presents several important problems:

- It’s *not scalable*. Once you move beyond a single server, your deployment will get much more complicated.

- You need to schedule downtime. While you’re copying your application, the app is in an inconsistent state, with some of the files from your old application and some from the new.

- If something goes wrong, it’s hard to backtrack. You would need to put the old version of your application back. Doing so means more manual work or more guesswork.

- You need to handle source control manually. Unless you’re living in the dark ages, you’re keeping your code base in a source control system. Most Rails developers use Subversion.¹

- The deployment process is not automated. You have to do it by hand, which leaves room for error. Laziness may be one of the programmer’s greatest virtues, but it doesn’t make for perfect execution when many steps are involved.

¹ If you’re not using Subversion, put this book down and run, don’t walk, to pick up Pragmatic Version Control[?]. Running without version control these days is madness.
You might decide to use an existing tool, such as rsync. The rsync open source utility provides fast, incremental file transfer, meaning it only copies files that change between one invocation and the next.

This method works a little better than plain old FTP because you don’t move all of the files at once, but still has most of the same problems. rsync is usually faster than plain FTP, works better with multiple servers, and works without any additional modifications to your server’s configuration. If you are in a situation where you have little control over the rest of the server, this might be an acceptable solution. But as a programmer who aspires to do great things, you want more than just the acceptable solution don’t you?

5.1 The Lay of the Land

So far, you have an application that’s ready for deployment and served from a common repository and a host. To move in with style, you need some software to manage your move. That’s Capistrano. As you see in Figure 5.1, on the next page, the Capistrano tool sits on the developer’s client. Think of Capistrano as the traffic cop who orchestrates the traffic during your move to make sure your application makes it up to the server in an orderly and repeatable manner. You can direct any deployment from the comfort of your development client, without needing to log onto the deployment servers at all.

For the same reason they wrote Rails, 37signals wrote Capistrano to solve actual business problems. As their Rails deployments became more regular and more complex, the growing company needed an automated solution to handle complex application deployments. This typically entailed deploying code updates to at least two web servers and one database server. Any solution would need to do at least the following:

- Securely update multiple web and database servers from a central source code control repository.
- Be sensitive to errors and rollback any changes if necessary.
- Work with the Rails file layout and allow logging to happen without interruption.
- Operate quickly and reduce the need for downtime.
- Work well with Ruby on Rails.
Jamis Buck wrote an application named SwitchTower to solve this problem. Due to potential trademark issues, they renamed the tool to Capistrano. Like Rails, Capistrano is opinionated software and assumes that you do the following:

- Deploy to a dedicated server.
- Use a Unix-like operating system and file system on the server.
- Have SSH access to that server.
- Use Subversion or some other form of source code control.
- Have the ability to run commands as root with the `sudo` command.
- Deploy to a web server, application server, and a database server (on one or more machines).

Fortunately, you can still configure Capistrano if these assumptions don’t hold in your particular situation. Shared hosting is such a situa-
tion, and I will show you how to setup a rock-solid recipe for Capistrano later in this chapter.

5.2 How It Works

You will need to install the Capistrano gem and its dependencies. The recipes in this chapter use Capistrano-2.0, install it now:

    local$ sudo gem install capistrano -y

Always keep in mind that Capistrano runs on your local machine. It uses the Net::SSH and Net::SFTP to connect to your remote servers and run shell commands on them. This means you do not need to install Capistrano on your remote servers, only on the machines you want to use to trigger a deploy.

Once you’ve configured Capistrano with a few variables, called a recipe, you can deploy with the following simple command (some initial setup is required, which I’ll cover next):

    local$ cap deploy

This unassuming command does everything you need to reliably deploy your application. Rather than tell you what’s happening in English, you can just look at the Ruby deploy task:

    task :deploy do
      transaction do
        update_code
        symlink
      end
      restart
    end

That’s a pretty simple block of Ruby code. Capistrano uses a Rake-like syntax for defining tasks that will execute on the server. Each task has a name, followed by a code block. This task, named deploy, does the following:

- Before either of these lines is executed, Capistrano connects to your server via SSH.
- The code begins a transaction. That means that all of the code will happen in the scope of a transaction. If any part of the transaction fails, the whole transaction will fail, triggering a rollback. I’ll cover rollbacks later in this chapter.
• The first step in the transaction, update_code, does a Subversion checkout to load a full copy of your Rails application into a dated release directory on the remote server. For example, a cap deploy might create a directory called releases/20070416190111. The number 20070416190111 is actually a time stamp for 7:01:11 PM on April 16, 2007.

• The next step in the transaction, symlink, links the log directory for your Rails application to a shared log directory. symlink then links your application’s folder to the current directory, so the web server can find it.

• The task finally restarts any active Mongrel or FastCGI processes. Each of these steps is a critical component to a secure, successful deployment. Using Subversion ensures that Capistrano will always get the right code base. SSH provides the necessary security so your files cannot be compromised in transit, and the symlinks force an instantaneous conversion. The symbolic link also lets you revert back to an older version of the code if your code is bad or capistrano encounters an error. Finally, Capistrano restarts any active Mongrels or FastCGI processes and the new code goes live.

The Capistrano deploy task is certainly better than rsync, but the automation comes at a cost. You’ll need to do a little more setup, but not too much. Before I walk you through the gory details, let’s take a look at Capistrano’s file organization.

**Capistrano’s File Organization**

37signals built Capistrano specifically to deploy Rails applications, but you can configure it to work with other types of applications too. The default recipe assumes that you have a log directory for log files and a public directory for the web server’s public files.

Every time you deploy, Capistrano creates a new folder named with the current date and then checks out your entire Rails app into that folder. Next, it wires the whole thing together with several symbolic links. Figure 5.2, on the following page shows the directory structure. Notice the current directory doesn’t have the physical file structure underneath it. current directory is only a link into a specific dated folder in the releases directory. The result is that current will always hold the current active version of your application. For your convenience, Capistrano also cre-
5.3 **Local and Remote Setup for Rails**

Capistrano configuration isn’t too difficult, but you will need to do a few steps on both your development machine and remote server. I’ll list the steps and then go through each in detail. First, on your local machine you’ll need to do the following:

- Install the Capistrano gem.
- Tell Capistrano about your application so it can add the necessary files to it.
- Customize config/deploy.rb with your server’s information.
- Import your application into Subversion.

Those local changes prepare your development machine to deploy your codebase from Subversion. Next, you’ll need to make the following changes.
Practice with a blank Rails project

If you are using Capistrano for the first time, it might help to make a blank Rails project and practice a simple deployment on your server. You don’t need to write any code or even create a database. Just use a new project with the default index.html page that is generated by Rails. To create the default project, just type rails projectname.

on your server:

- Set your web server’s DocumentRoot to current/public.
- Do a checkout of your app so Subversion will cache your login info.

When you’ve completed these steps, you can run the cap deploy:setup and cap deploy tasks. Next, I’ll show you each independent step in more detail.

Install the Capistrano Gem

You only need to install Capistrano on your development machine, not the server because Capistrano runs commands on the server with a normal SSH session. If you’ve installed Rails, you probably already have RubyGems on your system. To install Capistrano, issue this command on your local machine:

```bash
local$ sudo gem install capistrano --include-dependencies
```

```
Attempting local installation of 'capistrano'
Local gem file not found: capistrano*.gem
Attempting remote installation of 'capistrano'
Successfully installed capistrano-2.0.0
Successfully installed net-ssh-1.1.1
Successfully installed net-sftp-1.1.0
Installing ri documentation for net-ssh-1.1.1...
Installing ri documentation for net-sftp-1.1.0...
```

While you are installing gems, install the termios gem as well. (Sorry, termios is not readily available for Windows.) By default Capistrano echos your password to the screen when you deploy. Installing the
Reinstalling Ruby on Mac OS X

Capistrano relies heavily on Ruby’s ability to communicate over SSH, which does not work properly with the default Ruby interpreter included with Mac OS X for versions before Leopard. The C bindings do not always work correctly. (Leopard includes Capistrano, Ruby version 1.8.6, and Mongrel.) If you have one of these versions of OS X, you can fix this problem in a couple of different ways:

- Install the MacPorts package management system and let it install Ruby for you. You can download MacPorts* and then run this command:
  
  ```bash
  ezra$ sudo port install ruby
  ```

- Install Ruby from source. Dan Benjamin has step-by-step instructions†. You can find a shell script that automates an installation of Ruby using Dan’s instructions online‡.

---

* [http://macports.org/](http://macports.org/)
‡ [http://rubyonrails.com/pages/install](http://rubyonrails.com/pages/install)

---

termios gem keeps your password hidden from wandering eyes. To install termios, type the following:

```bash
local$ sudo gem install termios
```

```
Attempting local installation of 'termios'
Local gem file not found: termios*.gem
Attempting remote installation of 'termios'
Building native extensions. This could take a while...
  ruby extconf.rb install termios
    checking for termios.h... yes
    checking for unistd.h... yes
  ...
Successfully installed termios-0.9.4
```

Generate an Application Deployment File

A deployment file contains all of the Ruby scripts and configuration parameters that Capistrano needs to deploy your application. Just as the Rails script/generate command generates some default application code that you later modify, Capistrano has a special flag to copy a few Rake tasks and a sample deployment file to the proper locations inside
your Rails application. Generate your deployment file now by typing the following command:

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

Let's break down what just happened. `capify` is the Capistrano script. The dot tells it to install in the current directory. Alternatively, you can provide the full or relative path to your Rails app. The command creates a `config/deploy.rb` file, which contains the deployment hosts, and `Capfile`, this Capfile tells Capistrano to load its default deploy recipes and where to look for your `deploy.rb` when you run `cap deploy` from inside your rails application.

### Customize `config/deploy.rb`

Here is a first glance at the default `deploy.rb` recipe file that the `capify` created for you.

```ruby
set :application, "set your application name here"
set :repository, "set your repository location here"

# If you aren't deploying to /u/apps/#{application} on the target
# servers (which is the default), you can specify the actual location
# via the :deploy_to variable:
# set :deploy_to, '/var/www/#{application}'

# If you aren't using Subversion to manage your source code, specify
# your SCM below:
# set :scm, :subversion

to :app, "your app-server here"
to :web, "your web-server here"
to :db, "your db-server here", :primary => true
```

There's no rocket science in `deploy.rb`. The `:application` symbol defines the deployment target application’s name. The `:repository` symbol defines the Subversion repository for the application. The next three roles define the machines that serve as the web, application, and database servers for your application. Most applications will need to set a few variables depending on the installed location of the application, the user deploying the app, and the web servers involved. Here is a slightly more customized `deploy.rb`.

```ruby
# Customized deploy.rb
set :application, "brainspl.at"
```
set :repository, "http://brainspl.at/svn/{application}"
set :scm_username, 'ezra'
set :scm_password, proc{Capistrano::CLI.password_prompt('SVN pass:')}
role :web, "web1.brainspl.at", "web2.brainspl.at"
role :app, "app1.brainspl.at", "app2.brainspl.at"
role :db, "db.brainspl.at", :primary => true

set :user, "ezra"
set :deploy_to, "~/home/#{user}/#{application}"
set :deploy_via, :export

The most obvious differences in the default file and the customized file are the roles and the few lines below that. Roles are groupings of machines that handle different tasks for your application. The key roles are the web server (e.g. Apache), application server (e.g. Mongrel), and database server (MySQL). Now, look at the next few lines below the roles.

Capistrano lets you customize many different elements related to your servers, your application, authentication, and Subversion. The above script customizes the deployment directory, the system user, the scm_username, scm_password, and the command used to access Subversion.

In Capistrano-2.0, Subversion is the default SCM module. The user and password used to connect to your Subversion repo are defined as :scm_username and :scm_password. If you use a different source code repository then you can set the :scm variable in your deploy recipe. For example if you use Darcs instead of Subversion, it would look like this:

set :scm_username, 'ezra'
set :scm_password, proc{Capistrano::CLI.password_prompt('SVN pass:')}
set :scm, 'darcs'

You will notice that we have something different going on for the :scm_password variable. Since we don’t want to hardcode the password in our deploy recipe we have asked Capistrano to prompt us for the password every time we run a deploy.

If you’re running a shared host, you can’t run with root access, and you’ll need to handle restarts a little differently. Below, I use a script called the reaper that does the trick:

set :use_sudo, false
set :restart_via, :run

namespace(:deploy) do
desc "Restart with shared-host reaper"
task :restart do
  run "#{current_path}/script/process/reaper --dispatcher=dispatch.fcgi"
end
end

Notice that the :restart is defined inside of the namespace(:deploy) block. Capostrano-2 has an organization concept of namespaces. Namespaces let you collect related concepts into a central grouping of names. The method namespace takes a single parameter, defining the namespace, and a code block. All of the Capostrano tasks in the code block will be part of that namespace. In this case, I'm adding the :restart task to the default namespace. I'll create all of the deploy-related tasks inside of that namespace. To call this restart task in the deploy namespace you need to specify the namespace, like this:

local$ cap deploy:restart

**Import Your App Into Subversion**

In Chapter 5, *Capistrano*, on page 90, you learned how to place your code under source control, if you were not already doing so. Capistrano works with several source code control systems, but Subversion is the most common. Many shared hosts offer Subversion hosting, or you can install the Subversion server on your own dedicated or VPS server (see Chapter 4, *Virtual and Dedicated Hosts*, on page 70).

You will save configuration time by creating a repository named for your deployment domain. For example, the brainspl.at repository stores the Rails app that powers the Brainspl.at site. To import the project for the very first time, issue the command:

```
ezra$ svn import brainspl.at http://brainspl.at/svn/brainspl.at
```

brainspl.at is the local directory containing the application. The source control server is brainspl.at. The repository on the server is also named brainspl.at.

After importing a project for the first time, you must check out a new copy. The new copy will have all the extra files Capistrano needs to keep everything synchronized. To edit the code for development, you can issue this command:

```
local$ svn checkout http://brainspl.at/svn/brainspl.at
```

---

2. http://brainspl.at
Setup Apache or lighttpd to Use the Maintenance Page

The web:enable task assumes that your server will serve the public/system/maintenance.html page instead of the real site, if the maintenance page exists. Add the following Rewrite directive to your Apache config or your local .htaccess in order to make web:disable work correctly:

```
RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
RewriteCond %{SCRIPT_FILENAME} !maintenance.html
RewriteRule ^.*$ /system/maintenance.html [L]
```

This rewrite rule says that if there is a file called %{DOCUMENT_ROOT}/system/maintenance.html, rewrite all requests to /system/maintenance.html. With that rewrite rule in place, if the maintenance page exists, the web server will deliver it to satisfy any requests to this application regardless of what url they requested.

From this point, you can make changes to the source and synchronize it with the server by issuing the `svn commit` command.

```
local$ svn commit --message "Bugs have been fixed!"
```

Setting your Public Document Root

The document root is a directory that your shared host will use to serve all of your static web pages. Rails will manage all of the dynamic content. The local host will use a web server such as apache to serve your static content–images, html pages, JavaScripts and style sheets. In Rails, the `public` directory holds all static content. Since the current directory points to your Rails application, you will need to set your document root to `current/public`. Just how you do so will depend on whether you have a shared or dedicated host, and on which web server you’re using. Your hosting provider will tell you how to set that document root appropriately.

Cache Your Password on the Remote Server

Subversion clients cache login credentials for convenience and performance. But remember, Capistrano runs commands in a shell on the server, not your local host. To make things work smoothly, you need to login to Subversion at least once from the remote server so the remote
server’s Subversion client caches your username and password. When you deploy, the server will use the cached info to do checkouts from the repository. An easy way to invoke Subversion is to request a listing from your repository from any directory on the remote server, like so:

```
ezra$ svn list http://brainspl.at/svn/brainspl.at
Password: ******
```

Rakefile
app/
config/
db/
doc/
lib/
log/
public/
script/
test/
vendor/

After you type the command, Subversion will prompt you for your password and then show a list of the folders in the repository. More importantly, the remote server will be able to cache your username and password for subsequent Capistrano commands.

**Run the setup and deploy Tasks**

You are nearly done! Capistrano needs to create a few directories on the remote server for organization, so run the `setup` task:

```
local$ cap deploy:setup
* executing `staging'
* executing `deploy:setup'
* executing "umask 02 && mkdir -p /home/ezra/brainspl.at/releases /home/ezra/brainspl.at/shared /home/ezra/brainspl.at/shared/system /home/ezra/brainspl.at/shared/log /home/ezra/brainspl.at/shared/pids"

servers: ["brainspl.at"]
Password: ******
[brainspl.at] executing command
command finished
```

setup creates the releases and shared directories on the remote server. No current directory exists yet since you’ve not yet deployed. You’re finally ready to remedy that. Run the `deploy` task for the first time:

```
local$ cap deploy
```
If you have a standard setup, you should have a running application, but you should be aware of a few variations on the plain deploy task. If you’re running FastCGI and no listeners are running now, you may need to run the deploy:cold task instead, so Capistrano knows to start FastCGI listeners. Also, if your application has new migrations and you’ve not run them yet, you should run `cap deploy:migrations` to populate your database with the initial schema.

If all has gone well, you will be able to see your new site in a web browser. If not, see the troubleshooting sections in Chapter 4, *Virtual and Dedicated Hosts*, on page 70 and Chapter 3, *Shared Hosts*, on page 42.

You’ve spent a little time to get your deployment right, but you should already be seeing the benefits. You now have a deploy command that is also secure, integrated with source control, informative to your users, and completely automated. True, I’ve only showed you the most basic setup so far, but in the rest of the chapter I’ll walk you through a few more scenarios. First, let me lay a little foundation for customization.
Under the Hood

By now you should understand the basics of Capistrano. Before you start to customize it, I should provide a little more detail about how Capistrano executes tasks. Take a look under the hood.

Capistrano Runs Locally

Some developers get a little confused with where and how Capistrano works. In principle, Capistrano is a client-side tool that issues remote commands via SSH. From that point the commands run within the bash shell just as if you had logged in to the server and typed them manually. So the remote server doesn’t need to know anything about Capistrano, and you don’t even need to install the Capistrano gem there, but you do need the bash shell.

Code Synchronization Happens From Subversion

Local changes to your copy of the source don’t affect the code on the remote server! Capistrano synchronizes the remote server from the remote repository, so you must commit any code changes that you want deployed.

I just lied a little, but only a little. There is one piece of code that does not come from Subversion: deploy.rb. All changes to deploy.rb will take effect during deployment whether those changes have been checked in or not. You already know why: that’s the script that runs Capistrano, including the code that exports out the current version of your app. It’s still a good idea to keep your deploy script under source code control with the rest of your project.

Now, you’ve seen the basic Capistrano deploy script in action, but only in the default configuration with very few customizations. In the sections that follow, I’ll show you how to create your own Capistrano tasks and customize your existing tasks to handle more demanding scenarios. First, you’ll see some tasks, called recipes, that handle some common tasks.

5.4 Standard Recipes

Developers across the globe continually enhance Capistrano. To see the name and description of the current built-in tasks, use the cap -T command. As an added bonus, cap -T shows any of your own custom tasks, too as long as they have a description. From the root of your Rails application:
```bash
local$ cap -T
```

Here are a few useful tasks:
cap deploy:migrate
Use the power of the Rails migration system to update your database or manipulate data. This task will migrate the currently deployed code, so use deploy first, or use the composite deploy:migrations task to keep everything synchronized.

cap deploy:rollback
Houston, we have a problem! Something went wrong and you need to revert to the previous version of your code. Running this task activates your previous release. You can run this several times in a row to continue rolling back to older and older versions. Remember that this only uses the versions that you’ve previously deployed to the server, not older tagged versions from the repository. This task does not touch the database. If older versions of your code require downward migrations, you will have to revert to a previous migration manually.

cap deploy:cleanup
deploy:cleanup deletes older versions from the releases folder. By default, cleanup leaves the five latest versions, but you can configure this number with the :keep_releases variable in your recipe. You could set an after "deploy", "deploy:cleanup" callback task to run deploy:cleanup automatically, but remember that you won’t be able to deploy:rollback further than the number of releases currently on the server.

cap invoke COMMAND=’uname -a’
Run a single command on the remote server. invoke is useful for doing one-time tasks such as executing Rake tasks.

cap web:disable
This command copies a file to
When you call the `web:disable` task you can pass in two environmental variables, `UNTIL` and `REASON`. Capistrano will render these into the maintenance.html page that your web server displayed while your site is down for maintenance. Here is an example call.

```bash
local$ cap web:disable UNTIL='4:30pm'
    REASON='We are deploying new features, please check back shortly.'
```

### 5.5 Writing Tasks

Capistrano becomes even more powerful when you start writing your own tasks. Jamis Buck used Ruby's metaprogramming capabilities to write Capistrano. If you've used Rake (written by Ruby's Metaprogramming Master, Jim Weirich), you'll understand the general format of a Capistrano task, with a few minor differences that you can find at Section 5.5, *Like Rake, But Not Exactly*, on page 115.

Capistrano has many built-in tasks and capabilities that are always evolving. As with Rails itself, it is being developed at a rapid rate and...
this chapter has been rewritten several times as new features have been added. Even now, Capistrano developers are discussing plans that would drastically change the internal organization of Capistrano, but the external interface and operation of existing tasks should remain the same. For the most current information on Capistrano’s built-in methods and variables, see the online documentation.\(^3\).

### Setting Variables

The standard `deploy` task uses several user-customized variables in order to find the server, repository, and remote directory in which to deploy an application. If you are writing your own recipes you can also create variables with the `set()` method:

```ruby
set :food, "chunky bacon"
```

The `food` variable becomes a local variable for any task, on either side of a standard assignment. I can then use the variable in a task:

```ruby
set :breakfast, "a hot slab of #{food}"

task :serve_breakfast, :roles => :web do
  run <<-CMD
    echo "Are you ready for #{breakfast}?"
  CMD
end
```

This important task prints *Are you ready for a hot slab of chunky bacon?* You can also use the `set()` method with an Array, a Hash, or any other kind of object. You can even assign the variable to the output of any method, like so:

```ruby
set :projects, ['todo_list', 'lib/payment_library']

desc "Update remote folders from the repository"
task :update_projects do
  projects.each do |project|
    run "svn update /home/ezra/#{project}"
  end
end
```

In the above listing, I set the `projects` variable to an array value. I reuse that variable in a task that Capistrano runs on the remote server.

---

Lazy Evaluation of variables

Sometimes you want to define a variable that uses other variables that haven’t been defined yet. No problem. You can enclose your variable within #{} and Capistrano will use lazy evaluation. For example:

```ruby
set(:released_stylesheets_dir) {
  "#{release_path}/public/stylesheets"
}
```

In the above example, Capistrano will evaluate #{release_path} when it uses the string. Lazy evaluation lets you wait to bind a given variable to a value, increasing your flexibility.

Standard Variables and their Default Values

Capistrano has many different predefined variables. You can set them to configure different tasks in different ways. Capistrano creates them for use with your tasks or within other custom variables that you set. These are some of the predefined variables and their associated uses:

- **:application** has no default. This variable has the name of your application. Other variables such as deploy_to use this variable. You will probably want to set this to the domain name of the application you are deploying or to the preferred nickname for your application.

- **:repository** has no default. This variable defines the address of your Subversion repository containing the code you want to deploy.

- **:user** defaults to the currently logged-in user. This variable defines the SSH user Capistrano will use to deploy the application. If your username on your deployment machine is the same as your SSH username, then you can use the default value. This user account will also be used to checkout code from the repository and perform sudo commands during the deployment process.

- **:deploy_to** defaults to /u/apps/#{application}. This variable defines the target deployment directory.

- **:use_sudo** defaults to true. Capistrano often needs to run commands under the root user. You can suppress this behavior by setting use_sudo to false.

- **:restart_via** defaults to :sudo. Capistrano uses this option to determine whether the cap deploy:restart should run under the current user (specify the :run value) or root (specify the :sudo option).
• \texttt{:password} has no default. This parameter defines your password for ssh authentication. If you leave this blank Capistrano will prompt you for the password when you try to deploy.

• \texttt{:deploy\_via} defaults to \texttt{:co}. This command defines which Subversion command Capistrano should use to check out your application from your repository. Most Rails developers now use \texttt{:export} instead in order to avoid displaying \texttt{.svn} directory info via the web.

• \texttt{:shared\_path, :release\_path, and :current\_path} all point to the various directories in your environment as I've described them in this chapter.

\section*{Defining Tasks}

Now that you've seen a few existing Capistrano variables and tasks, it's time to build your own. Tasks are made of a description, a name, and a list of applicable roles. They are very similar to Rake tasks (Section 5.5, \textit{Like Rake, But Not Exactly}, on page 115).

Any custom tasks you write that are used during your deployment process should be put in the \texttt{:deploy} namespace. For the following tasks we will assume they are being defined inside the namespace.

Here is a simple task:

\begin{verbatim}
desc "Delete cached files"
task :sweep\_remote\_cache, :roles => :web do
  run "cd #{release\_path}; rake sweep\_cache RAILS\_ENV=production"
end
\end{verbatim}

The \texttt{sweep\_remote\_cache} command runs the Rake task called \texttt{sweep\_cache}. The benefit of building a Capistrano task to do this job is that I can run the task from my development machine. Take a look in greater detail:

• \texttt{desc} is a short description Capistrano will show when anyone runs the \texttt{cap -T} command. This tag is optional, but it's a good idea to use it since your tasks will not show up when you run \texttt{cap -T} unless they have a description defined.

• \texttt{task} identifies the name of the task.

• \texttt{:roles} limits the task to certain groups of servers. The most common roles are \texttt{:app, :db, and :web}. You can define your own roles and sub-roles should you have the need.
Use the cap shortcut for custom tasks

The easiest way to run custom tasks is to use the cap command-line tool. Capistrano will automatically discover the config/deploy.rb recipe file and will run actions that are passed as arguments. It does this by looking for a Capfile in the current working directory that tells it which deploy.rb to load.

```bash
# With cap command and arguments
local$ cap -f config/deploy.rb my_custom_task

# The same command, but even simpler!
local$ cap my_custom_task
```

The rest of the script contains more conventional Ruby code. Capistrano provides the following methods to make custom task-writing easier:

- **run** and **sudo**: Most tasks will use one of these methods. Each sends shell commands to the remote server, but **sudo** runs the commands as root. See the Capistrano page at the Ruby on Rails site\(^4\) for more details.

- **put**: Put will upload a file to the remote server.

- **delete**: This forcibly deletes a file on the server.

- **on_rollback** will execute whenever you explicitly issue a `cap deploy:rollback`, or when a `cap deploy` fails.

Here is the definition of the `web.disable` task from Capistrano’s default recipes:

```ruby
namespace(:web) do
  task :disable, :roles => :web do
    on_rollback { delete "#{shared_path}/system/maintenance.html" }

    maintenance = render("maintenance", :deadline => ENV['UNTIL'],
      :reason => ENV['REASON'])
    put maintenance, "#{shared_path}/system/maintenance.html", :mode => 0644
  end
end
```

\(^4\) [http://manuals.rubyonrails.com/read/chapter/104](http://manuals.rubyonrails.com/read/chapter/104)
This demonstrates the proper use of the `on_rollback`, `render`, and `put` tasks available to any Capistrano recipe file.

**Using the Built-In Callbacks**

If you want to add extra functionality to the standard deployment process to run a Ruby script when you deploy your application, the best way is to use the built-in callback system. For every task, Capistrano looks for a `before` and `after` task and calls it at the appropriate time if it exists. This was how you would write callbacks in Capistrano 1.x. In Cap2 there is a more event driven callback mechanism with methods called `before()` and `after()`.

I mentioned earlier that the `deploy` task calls three other tasks. This gives you a total of four tasks that you can hook in to. If you write a task named `before_deploy`, Capistrano will execute it in advance of the rest of the deployment process. Similarly, `after_deploy` will run after the deployment task.

Here is a short bit of pseudocode that illustrates how this works:

```ruby
# before_deploy
deploy do
  # before_update_code
  update_code
  # after_update_code
  # before_symlink
  symlink
  # after_symlink
  # before_restart
  restart
  # after_restart
end
# after_deploy
```

In addition, Capistrano automatically creates callbacks for each of your own tasks, opening a world of possibilities. For any task you define, you automatically get a `before` and `after` task.

```ruby
task :global_thermonuclear_war do
  ...
end

task :before_global_thermonuclear_war do
  kiss_your_butt_goodbye
end

task :after_global_thermonuclear_war do
  paint_the_house
end
```
At this point, I’m sure your mind is racing with the possibilities. Need to check out code and run the test suite before every deployment? Check. Want to send email to admins after every deployment? Yup. Give yourself a raise in after_symlink? Possible, but not likely. With before and after tasks, you can extend the right task at exactly the right time.

Now, you can take things too far. Let your mind wander a little bit and you’ll see what I mean. If before_deploy is also a task, you could conceivably write a before_before_deploy task. In fact, you can write an action as convoluted as before_before_after_before_deploy, which looks like something out of a Monty Python skit. I said you could do it, not that you should do it. In fact, _I’d like a deployment without so much before in it!_

I hope you never write such a task. Still, imagine with me a little bit longer. A task without any roles will be executed on all servers and all roles, no matter what the parent task is. It seems that before_wash_dishes should only happen on the kitchen server if wash_dishes was defined as a task that happens on the kitchen server. Not so! You must explicitly specify :role => :kitchen for any task that needs to be restricted to the kitchen server.

```ruby
role :kitchen, "kitchen.ezra.com"
role :home_theater, "theater.ezra.com"

# Executed on all servers!
task :before_wash_dishes do
  ...
end

# Executed only on the servers that have the :kitchen role
task :wash_dishes, :roles => :kitchen do
  ...
end

# Here's the right way to limit this to one group of servers
task :after_wash_dishes, :roles => :kitchen do
  ...
end
```

Having taken that trip down the rabbit hole, it is usually better to refactor your tasks and give them self-documenting names. A make_dinner task makes much more sense than a generic before_wash_dishes task. Here’s the revised code, with a better name:

```ruby
desc "Perform household maintenance."
```
task :maintain_house, :roles => :estate do
  mow_lawn
  prepare_soap_bucket
  wash_car
  wax_car
  make_dinner
  wash_dishes
end

Defining callbacks by making tasks named after your tasks with before_ or after_ prepended is still a valid method of using callbacks in Capistrano. But there is also new syntax for defining callbacks that are a bit cleaner. For example:

desc "Perform household maintenance."

task :maintain_house, :roles => :estate do
  mow_lawn
  prepare_soap_bucket
  wash_car
  wax_car
  make_dinner
  wash_dishes
end

before "deploy", "maintain_house"

As you can see this new notation allows you to name your tasks whatever you desire and still be able to hook them to certain events.

Consider a more practical problem. If your database.yml file is not under source code control—remember, that file has your password—you need to use another method to copy it to your server when you deploy. Writing an after callback task is a perfect way to solve this problem.

When you run cap deploy, Capistrano calls other tasks that you can define without having to override the built-in tasks. To build such a task, you would save the appropriate password information to a file in shared/config/database.yml. The shared folder is made by Capistrano when you run the deploy:setup task, but you will have to make the config folder manually. If you have a cluster of servers, you will have to do this on each of your servers. The following task shows how. Add it to deploy.rb:

desc "Symlink the database config file from shared directory to current release directory."
task :symlink_database_yml do
  run "ln -nsf #{shared_path}/config/database.yml
  #{release_path}/config/database.yml"
end
after 'deploy:update_code', 'symlink_database_yml'

Because we did not specify any roles, Capistrano will run the task on all the servers in the cluster (:app, :web, :db, and any others you define). Anytime you deploy, your script will symlink database.yml to the config folder in the current release directory. It’s just so easy!

Using Roles

By default, Capistrano executes tasks in parallel on all the servers defined with the role command. You can limit the scope of a command by explicitly specifying the roles for that task. It is also important to note that Capistrano also runs tasks in parallel, but not concurrently. Imagine three tasks and three servers:

```ruby
role :web, ['one', 'two', 'three']

task :daily do
  wash_dishes
  mow_lawn
  learn_japanese
end
```

The tasks would be executed like this:

```
local$ cap daily
* wash_dishes on server one
* wash_dishes on server two
* wash_dishes on server three
* mow_lawn on server one
* mow_lawn on server two
* mow_lawn on server three
* learn_japanese on server one
* learn_japanese on server two
* learn_japanese on server three
```

Like Rake, But Not Exactly

Earlier I told you that Capistrano was almost exactly like Rake. I'm sure you noticed the almost. I'll point those differences out now.

Tasks are methods

Unlike Rake, other tasks can call Capistrano tasks directly, just as if they were methods. Rake tasks can only call other Rake tasks as tasks,
but not as methods. Capistrano uses this feature internally, but you can use it in your tasks, too. For example, here is a simple task:

```ruby
desc "Play a war game"
task :play_global_thermonuclear_war do
  ...
end
```

`play_global_thermonuclear_war` is a Capistrano task, but you can call it from another task like a normal method:

```ruby
desc "Play several games"
task :play_games do
  play_global_thermonuclear_war
  play_llor_dot_nu
end
```

This strategy lets you run a task alone or together with other tasks. For example, you could call a `rotate_logs` task from a task called `weekly` or alone.

**You can’t list other tasks as dependencies of a Capistrano task**

With Rake, you can pass the name of a task as a hash where the key is the task name and the values are the other tasks that must be run before the current task. Capistrano doesn’t use this syntax. Instead, you must call other tasks as methods or write `before` and `after` callbacks, as mentioned previously.

**You can override Capistrano tasks**

Rake lets you define tasks in stages, so it is not possible to override an existing Rake task. Capistrano gives you the ability to override tasks. If you don’t like the behavior of a built-in task, you can redefine it. For example, if you are deploying to a shared host, you might need to send a special argument to the reaper script in order to restart your FastCGI processes. To do this, define your own `restart` task as if it had never been written:

```ruby
namespace(:deploy) do
  desc "Shared host restart"
task :restart do
    run "#{current_path}/script/process/reaper --dispatcher=dispatch.fcgi"
  end
end
```

Other built-in tasks such as `deploy` will now use this task instead of the built-in `deploy:restart` task.
Capistrano tasks aren’t automatically available as rake tasks

Even though Capistrano tasks look like Rake tasks, they are part of a separate system. Rake doesn’t know about Capistrano tasks, even though older versions of Capistrano tried to bridge that gap. The approved way to call Capistrano tasks is with the cap command. It will automatically discover recipes in config/deploy.rb or Capfile if either of those files exist.

```
local$ cap deploy
```

5.6 A Little Extra Flavor

In this section, I’ll walk you through the topics that will make your Capistrano experience a little sweeter. You’ll sometimes want to see extra output or speed up your checkouts. These extra touches can really improve your overall experience.

Stream

Capistrano has a built in helper called stream. You can use this helper to stream information like log files and other stats from your remote servers to your local terminal.

You can use a task like this to tail the log files of your server:

```
Download capistrano/recipes/stream.rb
task :tail_log, :roles => :app do
  stream "tail -f #{shared_path}/log/production.log"
end
```

You can also continuously monitor the output of a shell command with Capistrano’s streaming callbacks, for example, to get the output of the rails_stat log parser you would use something like this:

```
Download capistrano/recipes/stream.rb
desc "Watch continuous rails_stat output"
task :rails_stat, :roles => [:app] do
  sudo "rails_stat /var/log/production.log" do |channel, stream, data|
    puts data if stream == :out
    if stream == :err
      puts "[Error: #{channel[:host]}] #{data}"
      break
    end
  end
end
```

Report erratum
this copy is (B2.0 printing, December 2007)
In this task you can see that the `sudo` method takes a block with three parameters, channel, stream, and data. The channel is the raw SSH connection, the stream is equal to either :out or :err, and the data is the output from the server.

This produces the following output on my blog:

```
~ 0.4 req/sec, 2.6 queries/sec, 6.7 lines/sec
~ 0.3 req/sec, 1.4 queries/sec, 4.3 lines/sec
~ 0.6 req/sec, 0.6 queries/sec, 4.2 lines/sec
~ 0.5 req/sec, 0.5 queries/sec, 3.5 lines/sec
~ 0.2 req/sec, 0.2 queries/sec, 1.4 lines/sec
```

**Run Solo**

Capistrano can do any kind of task that can be run over SSH, and it can be used with other technologies such as PHP, Perl, or Python (I’ve used it to deploy a web app written in Perl). I run my blog off the Typo trunk, but use a separate theme that is stored in my own repository. In order to easily update it on the remote server, I use a custom recipe kept in its own `deploy.rb` file within the theme directory:

```ruby
set :application, "example.com"
set :user, "ezra"
role :web, application

desc "Update the theme and delete cached CSS files."
task :theme_update, :roles => :web do
  run "svn update #{application}/themes/nuby"
  run "rm #{application}/public/stylesheets/theme/*.css"
end
```

What’s happening here? I use Capistrano’s built-in capability to connect to a remote server and execute commands. By specifying a :role for the task, it knows that it should connect to all the :web servers and run the `svn update` and `rm` commands. I also used its ability to set local variables like `:application` to simplify the recipe. I keep the files in a folder with the name of the domain, which makes it simple to specify a path to the theme folder and the cached stylesheets.

To deploy, you could call it from the command line like this:

```
local$ cap -f /path/to/deploy.rb theme_update
```

Since this `deploy.rb` is in a non standard location, you have to include its filesystem path, that’s what the `-f` argument does. Then you need to specify the task to run with `theme_update`. This makes it easy to
address deploy recipes anywhere on your computer rather than only those in the standard locations.

Capistrano will prompt you for your password and will execute the actions, showing the output as it happens. It will not do the standard deploy:update_code, deploy:symlink, and other tasks. Those are only part of the deploy task. If you write your own tasks, they will be executed independently.

You could also use Capistrano in a similar fashion to do maintenance tasks built into Rails, including log rotation and session sweeping.

desc "A Capistrano task that runs a remote rake task."
task :clear_sessions, :roles => :db do
  run "cd #{release_path}; rake db:sessions:clear RAILS_ENV=production"
end

Do a push deploy instead of pull with a custom deployment strategy

Capistrano is a great system by default. But some people would rather push a tarball of their application codebase to the servers rather than let the servers pull the application from Subversion. Luckily Cap2 has different deployment strategies and it’s easy to change the deploy to work via push instead of pull.

set :deploy_via, :copy

Just by changing the :deploy_via variable to :copy will alter the behavior of your deploy. Now instead of logging in to your servers and doing an svn export, Capistrano will now do a local Subversion checkout to a temporary location on your local machine. It will then compress and create a gzipped tarball of your application. Once it has the tarball it will upload it to the server and create a new release directory. The rest of the deploy tasks will remain unchanged and all of your symlinks and callbacks will fire like usual. This is extremely useful if you can only
access your Subversion repo from inside your office building but not from your servers. Now you can deploy via push to avoid this issue.

5.7 Troubleshooting

**current Directory Can’t Exist as an actual folder**

Capistrano is a tremendously convenient tool, but it’s part of your infrastructure. As with Rake or other Rails scripts, you might find debugging Capistrano recipes a little intimidating. Take heart, though. It’s all Ruby code.

Shared hosts often give you a directory called current as part of the overall setup process. The recipes that I’ve shown you will create that directory for you. You’ll want to delete the host’s version.

**Migrations out of sync with code base**

Capistrano usually makes it easy to deal with migrations if you follow the precautions I lay out in Chapter 2, *Refining Applications for Production*, on page 19. That chapter laid out what you should do to keep your migrations well behaved. If you’ve gotten yourself into trouble, keep these tricks up your sleeve to get you back out.

One problem can occur with partially completed migrations. If a migration has a bug in the up() or down() method, your migration might leave your database in an inconsistent state, or you may be lucky and only need to set the version number correctly. If your version number is wrong, you need to reset it with a SQL query. You can easily do so in the console or from the Rails script runner. Say Rails crashed in migration 44 before setting the version in schema info, so your migrations are always crashing on number 43. You can set the version column of schema_info to 44 with this command: `ruby script/runner `ActiveRecord::Base.connection.execute "update schema_info set version=44"`

You may also have a situation where Rails is breaking because the version of code your migration needs is inconsistent with an earlier migration. (If you put your models in your migrations, this problem won’t occur.) You can solve the problem by deploying an earlier version by running your migrations(on the server) up to a specific version like this: `rake db:migrate VERSION=42`

---

5. Alternatively, you can use the transactional_migration plugin that you can find at [http://www.redhillonrails.org/#transactional_migrations](http://www.redhillonrails.org/#transactional_migrations).
Then, you can simply run `cap deploy:migrations` to deploy your current code base with the rest of your migrations.

**Only the contents of log and public/system will be kept between deployments**

Each time you deploy, Capistrano makes a timestamped release directory. If you have user generated file uploads that end up in `public/`, they will disappear the next time you deploy. This is because Capistrano made a new release directory and symlinked to it. My favorite way to fix this is to make an `after 'deploy:update_code'` hook task to symlink your own folders into `public` from the Capistrano shared directory.

Assume you have an `public/avatars` directory where you store uploaded avatars. You want this directory to persist between deployments and not get overwritten. You need to create an empty avatars directory in the Capistrano shared directory and then have it get symlinked into the proper place each time you deploy:

```ruby
after 'deploy:update_code', 'deploy:link_images'
namespace(:deploy) do
task :link_images do
  run <<-CMD
    cd #{release_path} &&
    ln -nfs #{shared_path}/avatars #{release_path}/public/avatars
  CMD
end
end
```

**User Permissions**

The user performing the SSH deployment will own all of your files. You need to remember that the web server user must be able to read and write to all of the appropriate files. Most Rails shops use a single deployment id to deploy. If you must change permissions as part of a Capistrano script, use an after task to change permissions if necessary.

Keep in mind that any Cron runner or email receiving task should also have write access to the appropriate log file.

### 5.8 Conclusion

In this chapter, you’ve taken a pretty deep stroll through Capistrano. You can now deploy your application in a repeatable, reliable way. You’ve also learned to extend Capistrano using recipes or callbacks.
In the chapters to come, I'll shift the focus to your application. You now know the basics for Rails deployments. It’s time to read about the finer points. In the next chapter, you will learn to build applications that are friendlier to your production environment. Read on.
After you’ve moved into your home and organized the place, you might want to get a watchdog to help keep your place safe. But our Mongrel is not all grown up yet. He’s a little puppy that can only serve one little request at a time. As you look at your Rails implementation, your initial training will mean building a repeatable configuration and clustering so your site can handle more than one request. Later, you will want to add services and scripts to monitor your application server so your pooch can handle any trouble that comes up.

6.1 Training your Mongrels

You’ve seen how easy it is to use a Mongrel server in its default configuration. In practice, you’re often going to need more flexibility than the default configuration. You will want to cluster your mongrels and probably run them as a service. You’ll also likely Fortunately, configuring Mongrel, and even enabling Mongrel clusters, is surprisingly easy. As you recall, to start Mongrel, you run the following commands:

```
ezra$ cd /path/to/railsapp
ezra$ mongrel_rails start -d
```

That starts a Mongrel daemon running in the background on port 3000. It is just as simple to restart or stop the server. You’d use mongrel_rails restart to restart, or mongrel_rails stop to stop. But these little commands simply take your dog for a walk and you’re ready to teach your dog a few more advanced tricks. You can train your dog with much more control through a variety of command line options and configuration files.

The mongrel_rails command line tool contains explanations for all of its options. To access this embedded documentation, use the -h flag.
ezra$ mongrel_rails start -h
Usage: mongrel_rails <command> [options]
   -e, --environment ENV  Rails environment to run as
   -d, --daemonize        Whether to run in the background or not
   -p, --port PORT        Which port to bind to
   -a, --address ADDR     Address to bind to
   -l, --log FILE         Where to write log messages
   -P, --pid FILE         Where to write the PID
   -n, --num-procs INT    Number of processors active before clients denied
   -t, --timeout TIME     Timeout all requests after 100th seconds time
   -m, --mime PATH       A YAML file that lists additional MIME types
   -c, --chdir PATH      Change to dir before starting (will be expanded)
   -r, --root PATH       Set the document root (default 'public')
   -B, --debug           Enable debugging mode
   -C, --config PATH     Use a config file
   -S, --script PATH     Load the given file as an extra config script.
   -G, --generate CONFIG Generate a config file for -C
       --user USER       User to run as
       --group GROUP     Group to run as
       --prefix PATH    URL prefix for Rails app
   -h, --help            Show this message
   --version             Show version

Keep in mind that this list will doubtlessly change as Mongrel grows and improves. For detailed explanation of every command line option, refer to the great online How-To\(^1\). You can also find excellent documentation at the Mongrel website\(^2\).

You can specify all of the above options on the command line each time you start mongrel_rails, but if you need anything more than the most basic configuration, flags will quickly get tedious. This is where the Mongrel configuration file comes into play. The -G or --generate option will create a config file for a given set of command line flags. Once you have a command line with all the options you desire, you can save them to disk for later use. From the root of your Rails application, run the following command:

```
ezra$ mongrel_rails start -G config/mongrel_7000.yml ←
   -e production -p 7000 -d
```

** Writing config to "config/mongrel_7000.yml".
** Finished. Run "mongrel_rails -C config/mongrel_7000.yml"
** to use the config file.

The previous command generates the following file called mongrel_7000.yml in the config/ directory of your Rails application:

---
ezra$ cat mongrel_7000.yml
---
:config_file: 
:daemon: true
:cwd: /Users/ezra/railsapp
:includes: 
 - mongrel
:environment: production
:log_file: log/mongrel.log
:group: 
:config_script: 
:pid_file: log/mongrel.pid
:num_processors: 1024
:debug: false
:docroot: public
:user: 
:timeout: 0
:mime_map: 
:prefix: 
:port: "7000"
:host: 0.0.0.0

That file has a lot of options. Thankfully you don’t usually need all of these settings so you can trim the file down quite a bit, like so:
---
:daemon: true
:cwd: /Users/ezra/railsapp
:environment: production
:log_file: log/mongrel.log
:pid_file: log/mongrel.pid
:docroot: public
:port: "7000"
:host: 0.0.0.0

Now you can make changes to your Mongrel configuration without typing them out on the command line each time you want to start a Mongrel server. To start Mongrel with your shiny new config file, use the -C flag.

```
ezra$ mongrel_rails start -C config/mongrel.yml
```

If you aren’t sure what options you want yet but you want to generate a config file to start with, you can use the -G option without any other arguments.

```
ezra$ mongrel_rails start -G config/mongrel.yml
```

When you run Mongrel on any Unix-like operating system, you can control it with signals similar to WEBrick or FastCGI. The signals that Mongrel understands include the following:
TERM Stops Mongrel and deletes the PID file.
USR2 Restarts Mongrel (new process) and deletes the PID file.
INT Same as USR2. This is a convenience as Control-C generates an interrupt signal, and Control-C is used in debug mode.
HUP Internal reload—might not work well. (The reason this might not work well is that sometimes doing an internal reload will not reload all the code in the system. So you are safer if you do a real USR2 restart.)

You can send these signals with the kill command:

`ezra$ kill -HUP 27333`

**Configuring a Cluster**

You've seen how to configure a single Mongrel instance, and you've also seen how to install a simple cluster. Your next step is to build a more flexible configuration for a cluster. First, you need to generate your `mongrel_cluster.yml` file. Configure a cluster of 3 Mongrels. Then run the following command from the root of your Rails application directory:

`ezra$ mongrel_rails cluster::configure -p 8000 -e production -a 127.0.0.1 -N 3`

Writing configuration file to config/mongrel_cluster.yml.

`ezra$ cat config/mongrel_cluster.yml`

```yaml
---
port: "8000"
environment: production
address: 127.0.0.1
pid_file: log/mongrel.pid
servers: 3
```

You just built a minimal, but working, `mongrel_cluster.yml` file to run a cluster. The `port` option is a little different from the `port` option you used when you configured a single Mongrel instance. For a cluster, `port` specifies the first port number for your first Mongrel. Each subsequent Mongrel starts on the next port. These Mongrels will start on ports 8000, 8001, and 8002. You also specified the Rails environment for your Rails application. Normally, you'll run a single Mongrel in development mode and a cluster for production. Mongrel will listen on the host name or IP address specified by the `address` option. The `pid_file` option specifies the location for Mongrel's pid files, and `servers` specifies the number of Mongrels you want in the cluster. The previous file configures 3 Mongrels running on port 8000, 8001 and 8002. Next, customize this config file a bit to take advantage of a few more attributes.
It’s a good idea to set `cwd` (current working directory) to the root of your Rails application. I also added the `log_file`, `docroot`, `user` and `group` settings. Configuring the `user` and `group` will make mongrel run under that user and group even if you accidentally start it with `sudo`. It is always a good idea to run web applications as a normal user instead of `root`, just in case your application has a security breach; and we know all applications have security holes.

To start and stop your mongrel_cluster you still use the `mongrel_rails` command, but you gain a set of cluster commands to use with it. Try it now from the root of your Rails app:

```
  ezra$ mongrel_rails cluster::start
  Starting 3 Mongrel servers...
  ezra$ mongrel_rails cluster::restart
  Stopping 3 Mongrel servers...
  Starting 3 Mongrel servers...
  ezra$ mongrel_rails cluster::stop
  Stopping 3 Mongrel servers...
```

You’ve just tidied up your Mongrel configuration. Next, you can work on running mongrel as a service.

**Running Mongrel as a Service**

Before we jump into monitoring, let’s first configure Mongrel as a service. Using the `mongrel_rails` command from your local directory is fine for playing around on your local machine, or for staging environments. But in a production environment it’s nice to configure Mongrel more like Apache and MySqI. It keeps things consistent and helps to include Mongrel when automatically starting services when your server starts or restarts. It’s very simple and not unlike the Mongrel configuration you’re already familiar with.

You’ll need to ensure that you have the `mongrel_cluster` gem installed first. Once it is, you simply need to create a file at `/etc/mongrel/myapp.conf`...
where myapp should be the name of your application (but can be anything). If you're running multiple applications on one server, you can have multiple Mongrel cluster configuration files. In the file you configure your Mongrel cluster with a few simple options. They are documented with inline comments in the example configuration file below.

```
# /etc/mongrel/myapp.conf

# The user and group with which to run Mongrel
user: deploy
group: deploy

# The location of our Rails application
# and the environment to run within
cwd: /home/deploy/apps/myapp/current
environment: production

# The number of servers in the cluster
servers: 4

# The starting port
# e.g. with 3 mongrels would bind ports 8000-8002
port: "8000"

# The IP Addresses allowed to connect to Mongrel
# If your web server proxy is separate from your app server,
# its IP address here instead of the localhost IP address
address: 0.0.0.0

# The location of the process ID files relative to the rails app above
pid_file: log/mongrel.pid
```

With that configuration file in place, you can now start, restart or stop Mongrel using the following simple command from any current working directory:

- `mongrel_cluster_ctlstart` will start a Mongrel cluster from scratch.
- `mongrel_cluster_ctlrestart` will restart a running Mongrel cluster.
- `mongrel_cluster_ctlstop` will stop a Mongrel cluster.

You should be able to get your flavor of Linux to start this service automatically on startup and stop it upon shutdown. You’ll need to refer to the relevant documentation for your Linux distribution if you’d like it to work that way.

Now that you have your cluster of Mongrels happily running as a service, you can turn your attention to managing the Mongrel server. The
Monit tool will let you handle scenarios where your mongrels might run out of memory, or experience any other problems.

6.2 Obedience Training for Mongrel with Monit

Monit is a simple utility used to manage files, processes and directories on Unix. You can configure Monit to split your logs if they get too big, start and stop processes, and also keep tabs on resources. Monit can notify you if your memory use gets out of control and actually do something about it. You may want Monit to restart one of the mongrels in your cluster or restarting your nginx web server if someone changes your configuration file.

For starters, you’re going to use Monit to make sure your Mongrels keep running at peak efficiency. You’ll need to do three things to get the management process running:

• You need to install the right version of mongrel_cluster. The minimum version you want to run is 1.0.1.1. Earlier versions do not support the --clean option. This is important because Mongrel 1.0+ will not start if there is a PID (process identification) file sitting on disk. So if your server crashes and has to be rebooted, Mongrel tries to start up and fails because there was a left over PID file. The --clean option deletes leftover PID files if they exist.

• You need a good mongrel_cluster.yml file. You’ve already built one earlier in this chapter and that one should work fine.

• You need a Monit configuration file, called mongrel.monitrc. This configuration file will tell Monit what to do for each Mongrel on your system.

The first order of business is to install Monit. Most Linux distributions will have a Monit package available in their package managers. On Debian/Ubuntu you can run sudo apt-get install monit and on Gentoo you can run sudo emerge monit. If you cannot locate a package for your preferred Linux don’t sweat it, Monit is easy to build from source. Here are the commands:

```
ezra$ wget http://www.tildeslash.com/monit/dist/monit-4.9.tar.gz
  ...
ezra$ tar xzvf monit-4.9.tar.gz
  ...
ezra$ cd monit-4.9
```
Building Monit on RHEL or CentOS

There are a few dependencies you need to install before you can get Monit to build on Red Hat or CentOS distributions. Use `rpm` or `yum` to search for and install the following packages: `flex`, `bison` and `byacc`. Once you have these prerequisites installed you can build Monit with the same instructions shown for other systems.

```
ezra$ ./configure && make && sudo make install
...```

Next up you need to install the correct version of `mongrel_cluster`. You will want the latest version from Rubyforge. It is important to clean up older versions of `mongrel_cluster` if you had any installed.

```
$ sudo gem install mongrel_cluster
&& sudo gem cleanup mongrel_cluster
```

After you’ve set that up, you are ready to configure Monit. I like to create a separate configuration for each Mongrel Cluster. You’ll add the following configuration to `mongrel.monitrc`, which you’ll keep in Monit’s directory, in our case, `/etc/monit.d`:

```
with pidfile /data/deployit/shared/log/mongrel.5000.pid
start program = "/usr/bin/mongrel_rails cluster::start -C
  /data/deployit/current/config/mongrel_cluster.yml
   --clean --only 5000"
stop program = "/usr/bin/mongrel_rails cluster::stop -C
  /data/deployit/current/config/mongrel_cluster.yml
   --only 5000"
if totalmem is greater than 110.0 MB for 4 cycles then restart
if cpu is greater than 80% for 4 cycles then restart
if 20 restarts within 20 cycles then timeout
group deployit
```

Notice that you will need a block for each process that you want Monit to monitor. The config from above is for one Mongrel only. The first directive, `check_process`, identifies a process to monitor. I’ve skipped that directive in favor of the alternative `with pidfile` option that tells Monit which process file to monitor. Recall that each Mongrel instance has a file stored in the `log/mongrel.port.pid` file. The next two directives tell Monit how to start and stop mongrels. The last three directives tell Monit what to do when certain pathological conditions exist. This con-
figuration will restart Mongrel instances if the memory grows over a threshold (110.0 MB in the above configuration) or the CPU is too busy for a process. These directives also can take more extreme measures, such as timing out and notifying administrators. Keep in mind that all of this is fully automated and requires notification only in extreme circumstances.

A final configuration provides the general setup for Monit, including the configuration for the mail server and alerts. This file is located at /etc/monitrc.

```
$ sudo /etc/init.d/monit restart
set daemon 30
set logfile syslog facility log_daemon
set mailserver smtp.example.com
set mail-format {from:monit@example.com}
set alert sysadmin@example.com only on { timeout, nonexist }
set httpd port 9111
    allow localhost
include /etc/monit.d/*
```

This config is fairly straightforward but there are a few things to note. `set daemon 30` tells Monit how often to check processes, in this case every 30 seconds. I have found that 30 seconds is perfect for this setting. You need to set your own SMTP server and email addresses for alerts. The last two directives turn on Monit's built in HTTP server on port 9111, making it only viewable from the localhost, and sets /etc/monit.d to be the directory to include config files from.

When you're done, you can try a couple of commands. You can actually start and stop mongrel cluster instances through Monit. First you need to make sure Monit has your latest configuration loaded:

```
$ sudo /etc/init.d/monit restart
```

When Monit starts it will automatically boot your Mongrels. Then you can restart the mongrels by their group through Monit:

```
$ sudo monit restart all -g deployit
```

Or restart one single mongrel by its name:

```
$ sudo monit restart mongrel_deployit_5000
```

To see the current status of your Mongrels you can use the status command:

```
$ sudo monit status
```
The monit daemon 4.9 uptime: 4d 2h 27m

Process 'mongrel_deployit_5000'

status running
monitoring status monitored
pid 20467
parent pid 1
uptime 55m
childrens 0
memory kilobytes 50432
memory kilobytes total 50432
memory percent 12.8%
memory percent total 12.8%
cpu percent 0.0%
cpu percent total 0.0%
data collected Sun Jul 1 14:38:26 2007

]]>

You will need some custom Capistrano tasks now that you are using Monit to watch your Mongrels. When you use Monit you do not need to use the mongrel_cluster/recipes in your deploy recipe. Instead you will set the Monit group of the Mongrels you are targeting with this line in your deploy.rb:

```
set :monit_group, 'deployit'
```

Now you need to add the following tasks to your deploy recipe:

```
desc <<-DESC
Restart the Mongrel processes on the app server by calling restart_mongrel_cluster.
DESC
task :restart, :roles => :app do
  restart_mongrel_cluster
end

desc <<-DESC
Start Mongrel processes on the app server.
DESC
task :start_mongrel_cluster, :roles => :app do
  sudo "'/usr/bin/monit start all -g #{monit_group}"'
end

desc <<-DESC
Restart the Mongrel processes on the app server by starting and stopping the cluster.
DESC
task :restart_mongrel_cluster, :roles => :app do
```
Now you know how to use Monit to keep a leash on your Mongrels. Monit can be a lifesaver for your production Rails applications and I highly suggest using it whenever you deploy Mongrels.

### 6.3 Keeping FastCGI Under Control

#### Zombie FastCGI Processes

During the dog days of the summer of 2005, I noticed that one of my Rails apps was running a little slower than expected. Confident in my debugging abilities, I fired up my SSH client and logged into my shared server. Almost immediately, the server kicked me out with an odd Resource unavailable error.

After three more tries with the same result, I emailed the customer support team. It turns out that I had fifty processes running, the maximum allowed for any single user! Every one of those processes was a zombie, aimlessly occupying my process allocation but unable to do anything useful. Like a bad horror sequel, one of my Rails apps on a completely different host had the same problem a few days later.

The Apache web server is famous for producing these zombies when running with FastCGI, causing many developers to favor Mongrels or nginx instead. The good news is that a few simple cron tasks can keep zombies from getting out of hand, making the difference between a smoothly running site and one that dies daily. I’ll discuss them in Section 6.3, *The Reaper*, on the following page.

The conclusion to the story is that the sysadmin at the shared host killed the zombie processes and things began working again. I learned to start a daily cron task that cleans out zombies and gives my server a fresh start. Some people restart their dispatch processes every single hour. You will have to experiment with your specific situation and see what works best.
The Reaper

The reaper is not a black-hooded messenger of doom; he is your best friend. The reaper command reliably prunes back FastCGI processes. Capistrano uses it to restart your Rails app after a fresh deployment. You can also use it to restart processes on a regular schedule.

The reaper is a script you run on the command line. By default it restarts FastCGI dispatch processes for your application only, so you won’t disrupt other applications running under the same user account. You can fire off other actions with the reaper as well:

- restart: Restarts the application by reloading both application and framework code (the default). Send the USR2 signal to each dispatch.fcgi process belonging to the current application.
- reload: Reload only the application, but not the framework (like the development environment). Reload sends the HUP signal.
- graceful: Mark all of the processes for exit after the next request. Graceful sends the TERM signal.
- kill: Forcefully exit all processes regardless of whether they’re currently serving a request. kill sends the -9 signal. Use this only if none of the other signals are successful.

You can run the reaper without any arguments or request one of the above actions such as the following:

```
ezra$ ./script/process/reaper --action=graceful
```

In my experience, the defaults do not work on most shared hosts because their output doesn't match the reaper’s expectations. The good news is that you can send an extra argument to match the specific output of your host.

Let me show you how I fine-tuned this on one of my shared hosting accounts. First, I tried to run the dispatcher normally. Even though I knew that there were several dispatch.fcgi processes running at that very moment, the reaper couldn’t find them:

```
ezra$ ./script/process/reaper
Could't find any process matching: /data/deployit/releases/20060224192655/public/dispatch.fcgi
```

Reading through the reaper code revealed the exact command the reaper used to find the list of running processes. I called that command manually:

```
ezra$ ./script/process/reaper
```

```bash
Could't find any process matching: /data/deployit/releases/20060224192655/public/dispatch.fcgi
```
ezra$ ps axww -o 'pid command'

    PID COMMAND
    4830 /usr/bin/ruby dispatch.fcgi
    18714 /usr/bin/ruby dispatch.fcgi
    2076 /usr/bin/ruby1.8 dispatch.fcgi
    12536 -bash
    5607 ps axww -o pid command

I could then see what was happening. The reaper was looking for the full path to the dispatcher, but the ps command on my server returned a shorter version of the current process list. Consequently, the reaper could not find the full path, so I can’t restart this application independently of the others running under that same user account. As configured, the reaper was all or nothing!

Running the same command on my local Mac OS X machine shows the entire path to the dispatch.fcgi script, as it should. A fact of shared hosting is that you can’t control systemwide settings, so you may have to adjust your scripts to match.

With this information in hand, I could send a more general argument to restart all dispatch processes running under that user account in order to keep things fresh and zombie-free:

    ezra$ ./script/process/reaper --action=restart --dispatcher=dispatch.fcgi

Restarting [4830] /usr/bin/ruby dispatch.fcgi
Restarting [18714] /usr/bin/ruby1.8 dispatch.fcgi
Restarting [2076] /usr/bin/ruby1.8 dispatch.fcgi

6.4 Building in Error Notification

With a Mongrel cluster in place, your setup has greater scalability, and you should be able to sustain minor failures. With Monit in place to manage your Mongrel clusters, you have the capability to take preemptive action when a single Mongrel cluster fails, or when resources get scarce. But most of the time, your failures will come from plain old Human error. If you want a good management story, you’re going to have to deal with your programmer’s mistakes. Usually, Rails errors will generate an application error, the dreaded 500 error page. With Ruby, it’s fairly easy to intercept the default error behavior to, for example, send email notifications. And that’s exactly what the exception_notification plugin does.
You can read about the exception notification plugin at the Rails wiki¹³. To install it, simply run the installation script like this:

```bash
ezra$ ruby script/plugin install exception_notification
```

Next, to build notification into a particular controller, include the error notification module. I like to include error notification into application.rb, so I’ll get email notification when any user of any controller encounters an error that I failed to handle correctly, like so:

```ruby
class ApplicationController < ActionController::Base
  include ExceptionNotifiable
  ...
end
```

Next, configure the email addresses that should get notified of Rails exceptions. Put the notification in config/production/environment.rb:

```ruby
ExceptionNotifier.exception_recipients = %w(you@yourdomain.com another@yourdomain.com)
```

Now, if any error should occur, you’ll get an error notification like the following:

```
A ActionView::TemplateError occurred in drives#edit_comment:

undefined method 'title' for nil:NilClass
On line #5 of app/views/drives/edit_comment.rhtml

2: <%= error_messages_for 'gift' %>
3: <!--[form:drive]-->
4:
5: <h1><%= @drive.title %></h1>
6: <div>
7: 8: <table><tr>

#{RAILS_ROOT}/app/views/drives/edit_comment.rhtml:5:in `_run_rhtml_47app47views47drives47edit_comment46rhtml'
#{RAILS_ROOT}/vendor/rails/actionpack/lib/action_view/base.rb:326:in `compile_and_render_template'
#{RAILS_ROOT}/vendor/rails/actionpack/lib/action_view/base.rb:301:in `render_template'
...

------------------------------
Request:                      
------------------------------
```

¹³ http://wiki.rubyonrails.org/rails/pages/ExceptionNotification
* URL: http://changingthepresent.org/drives/edit_comment/65?donate=true
* Parameters: {"donate":"true", "action":"edit_comment", "id":"65", "controller":"drives"}
* Rails root: /home/deploy/importantgifts/current

Session:

* @write_lock: true
* @session_id: "875ce6f70cb9b8e9348a72147999303c"
* @data: {"flash":[]}
* @new_session: true

Environment:

* GATEWAY_INTERFACE : CGI/1.2
* HTTP_ACCEPT : */*
* HTTP_ACCEPT_ENCODING: gzip
* HTTP_CONNECTION : Keep-alive
* HTTP_FROM : googlebot(at)googlebot.com
* HTTP_HOST : changingthepresent.org
* HTTP_USER_AGENT : Mozilla/5.0 (compatible; googlebot/2.1; +http://www.google.com/bot.html)
* HTTP_VERSION : HTTP/1.1
* HTTP_X_FORWARDED_FOR: 66.249.72.161
* HTTP_X_TEXTDRIVE : BigIP
* PATH_INFO : /drives/edit_comment/65
* QUERY_STRING : donate=true
* REMOTE_ADDR : 66.249.72.161
* REQUEST_METHOD : GET
* REQUEST_PATH : /drives/edit_comment/65
* REQUEST_URI : /drives/edit_comment/65?donate=true
* SCRIPT_NAME : /drives/edit_comment/65
* SERVER_NAME : changingthepresent.org
* SERVER_PORT : 80
* SERVER_PROTOCOL : HTTP/1.1
* SERVER_SOFTWARE : Mongrel 1.0

* Process: 1620
* Server:

Backtrace:

On line #5 of app/views/drives/edit_comment.rhtml
Voila! This email message is an actual email notification that helped solve a production problem in the code at ChangingThePresent\(^4\). That email contains a full compliment of debugging information, including a full trace and back trace, the contents of the session, the offending view code, and the full environment for the HTTP request.

You can configure a few other options as well. Configure the sender with ExceptionNotifier.sender_address, and append a string to the subject line (to help with email filters) with ExceptionNotifier.email_prefix. This plugin will send email notifications only when the address is not local. You can configure which IP addresses should be considered as local with ExceptionNotifier.consider_local.

With this solution, Rails will notify you whenever your application experiences an exception. You can configure it to work well with your email clients, and because it’s plugged directly into Rails, as long as Rails does not fail completely and your network and email keep working, you’ll get notification.

### 6.5 Heartbeat

The exception_notification plugin is a great way to understand when your application has errors, whether the errors are consistent or intermittent. It’s not a complete management solution, though. For larger or
more critical production systems, you also need to verify that the sys-

tem is running at all.

A heartbeat service will tell you when your application fails. I find that
a simple script, running on a separate host, works better than cus-
tom solutions because it’s easy, infinitely customizable, and deployable
on any host with your scripting language. The following script detects
when one of four pages are down at ChangingThePresent:

```ruby
#!/usr/local/bin/ruby
require 'net/smtp'
require 'net/http'
require 'net/https'
require 'uri'

urls = %w{
   http://www.changingthepresent.org/
   http://www.changingthepresent.org/nonprofits/show/23/
   http://www.changingthepresent.org/causes/list/
   https://www.changingthepresent.org/
}

from = 'system@importantgifts.org'

recipients = %w{development@changingthepresent.org}

errors = []

urls.each do |url|
   begin
      uri = URI.parse(url)
      http = Net::HTTP.new(uri.host, uri.scheme == "https" ? 443 : nil)
      http.use_ssl = (uri.scheme == "https" ? true : false)
      http.start do |http|
         request = Net::HTTP::Get.new(uri.path)
         response = http.request(request)
         case response
            when Net::HTTPSuccess, Net::HTTPRedirection
            else
               raise "requesting #{url} returned code #{response.code}"
            end
      end
   rescue
      error = "#{url}: #{@!}"
      errors << error
      puts error
   end
end
```

Report erratum this copy is (B2.0 printing, December 2007)
unless errors.empty?
  msg = "From: #{from}\n"
  msg += "Subject: ChangingThePresent.org is down!\n\n"
  msg += errors.join("\n")
  puts "sending email to #{recipients.join(', ')}"
  Net::SMTP.start('localhost', 25, 'localhost') do |smtp|
    smtp.send_message(msg, from, recipients)
  end
end

The four urls are not haphazard. They represent a secure page, a page cached page, a fragment cached page, and a standard dynamic page. The admin team executes this script once every five minutes via a cron job. The script notifies all of the developers on the project via an email address which is forwarded to all developers whenever the site is down.

The script counts redirects and success as a successful contact. Anything else is a failure. Timeouts will also trigger a notification.

The management strategies in this chapter don’t cost anything, but they are surprisingly robust. By configuring your mongrels in a cluster, you get good performance and some fail over. By using Monit, you get a watchdog that will automatically kill and restart any rogue mongrels. By using the various email notification features, the scripts will notify the recipients of your choice when the server is down, or when anyone encounters a Rails error. Still, our error recovery solutions are not yet complete. You will need a better handle on monitoring resources, and on performance before you have a complete strategy. Read on.
You will often want your web site to grow. When you can’t fit into your existing home any more, you have to find some way to move up or to add on. The Rails model for scaling will take you beyond the single home owner and into the realm of a real estate developer or community planner. This chapter will examine scaling out.

7.1 The Lay of the Land

I’m going to put down my real estate agent hat for a little while and put on the hat of a community planner. If you own a house near a congested city and work in its busy downtown core, you’re all too familiar with multi-lane highway traffic that travels at times as fast as a cheetah and other times as slow as a statue of a cheetah. Presented with this problem, you might start with the following two solutions to the problem:

- Increase the speed limit, or
- Increase the number of lanes

These solutions sound obvious and you’ve probably heard similar analogies before, but there’s a lot more to it. Natural and political laws place a limit on how fast cars can travel safely, and you can only add so many lanes to a highway. These constraints effectively limit how effective either solution can ultimately be.

Sometimes, these solutions are not even attacking the right problem. The obstacles to effectively moving people aren’t always speed limits or lanes. You need to consider interfaces – on ramps and merges – that can slow traffic down. Entrances force lane changes and slowdowns,
exits double the problem, and accidents or construction projects force lane closures. The biggest bottleneck of all though – and the one that’s responsible for the others – is the destination. Not only are all of the cars on the same road, but they’re all heading to the same place. The problem is the \textit{city center itself}. You can only hope that there are enough parking spaces and office space available once you finally reach it!

Your computer infrastructure isn’t much different. Each lane of the highway is a network connection to some service provided by the application. The city center is the resource pool. Every ramp is a client node. Each car is a user request headed downtown to do some business. The idea of adding lanes and increasing speed limits is effectively a way of ”scaling up”. You can scale up by upgrading hardware such as CPUs, memory, disks and network bandwidth. That strategy works sometimes, but upward scaling has its limits. There are only so many CPUs, so much memory, so many disks and so much bandwidth that you can jam into a single box. These limits will likely never allow your application to meet the demands of the global crowd.

The low ceiling isn’t the only problem with scaling up. As your business grows, the cost of failure becomes greater too. You’ll need redundant systems and hot backups to handle failure and even the occasional hardware upgrade. Ultimately, scaling up is harder, with a lower ceiling.

Most successful web businesses scale up, not by chance even necessity, but by preference. I won’t completely write off scaling up. I’ll touch upon it when I address the database because scaling up has some real advantages in that space.

\subsection{Scaling Out}

Scaling out means adding more servers, complete with their own dedicated CPUs, disks, memory and network bandwidth. Think back to the traffic analogy for a moment. The ultimate problem was that everyone was heading to the same city center. Scaling out adds a second city center so half of the travelers that day would suddenly be on a different road and heading toward a different city center. Imagine what your daily commute would be like tomorrow if half the city’s population was simply not on your road! Scaling out certainly yields greater rewards. But what about cost, complexity and maintainability?

Scaling out to hundreds of servers can cost you plenty in dollars and complexity, but you don’t have to pay all of the price at once. Conve-
niently, scaling out lets the complexity scale with you. In the beginning, designing to scale outward costs little, but your preparations will position you for bigger challenges to come. You can deal with new performance demands later by doing a little prep work today. To get ready, you’ll prepare a few key elements of infrastructure. The trick is to get your scaling right early so you can avoid surprises later.

Keep in mind that you’ll still have to get your performance right. Even if you plan to scale by throwing money and hardware at the problem, you’ll want to save enough time to address performance. I’ll talk more about performance in the (as yet) unwritten chp.performance.

From a deployment perspective, you’re looking at a map something like Figure 7.1, on the next page. You’ll have two or three different virtual hosts that may or may not reside on the same servers. Each Mongrel cluster is a separate city center. One server will use Apache or ngnix as a static proxy and load balancer. The other two will serve Rails applications through Mongrel clusters. You’ll use Capistrano to deploy to each of them. The next few sections will help you set up a simple architecture that can easily grow from a single dedicated server to around five servers with minimal changes to your application. In the sections that follow, I’ll walk you through:

• Adding multiple virtual machines to your environment
• Setup of subdomains for your cluster using CNAMEs with your DNS provider
• Ensuring your Mongrel servers are deployed as clusters, and as services
• Setup of a load balancing proxy web server with Apache or NGinx
• Configuration of multi-master and master/slave MySQL clusters

I’ll start with the simplest rails deployment and slowly grow the server into a scaled out model ready to handle your angry mob of Web 2.0 users. When I’m done, you will have a web server that serves as a static proxy and a load balancer that serves content to one or more Mongrel clusters. Figure 7.2, on page 145 shows how this system will serve a typical request. The user makes a request to a gateway server. If it’s a static request, the gateway server simply serves up the static content and you’re done. If not, the gateway server will forward the request to one of several Mongrel clusters. The Mongrel clusters forward the request to an individual Mongrel server.
Figure 7.1: Deployment Map for Scaling Out

Prerequisites

If you don’t have access to a hosted Virtual Private Server (VPS), you can set a couple of virtual machines up on your desktop. For Windows users, both VMWare and Microsoft Virtual PC are free for non-commercial use. Linux users have commercial and free options including VMWare, Xen or OpenVZ. Mac users can use Parallels or VMWare, neither of which is free. Check for free trials if you’re not sure which one to buy. I prefer VMWare, but any one of them will work.

These virtual machine technologies work in a similar way to what a VPS host will provide. They will allow you to install an entirely separate operating system in a contained environment that acts like a computer within your computer. In fact, if you’re considering a VPS hosting service and aren’t sure how much memory or disk space you need, you can test out various configurations with these "home versions" to artificially constrain system resources. Try running your application in a 256MB VMWare virtual machine before committing to 12 months of a 256MB managed Xen VPS.
You can use any Linux distribution, but I've always found the Red Hat derived ones like CentOS and Fedora to be the most friendly. They are similar to what many VPS hosting providers will offer (RHEL and CentOS are very popular). There are a lot of Linux package dependencies to get your Rails application up and running. At a minimum you will need: C/C++ compiler, Ruby and Gems related to your app, Subversion, Apache httpd, MySQL Server, and your favorite text editor, be it vi or emacs. Sometimes you can build out a system by just installing your Linux distribution with its "web server" and "developer" options. However, it's not terribly hard to install everything from a minimal installation, especially given the documentation in Chapter 4, *Virtual and Dedicated Hosts*, on page 70.

You will need two virtual machine instances. With most of the software technologies mentioned above, you can build one and simply copy the virtual machine files, changing only some identification information such as the IP address and the host name.
Joe Asks...  

Why bother with virtualization for scaling out?

Virtualization has many advantages for scaling out. Primarily, you can configure your application to scale out sooner. You can probably get at least two or more virtual private servers for the same price as a single dedicated server. You gain most of the advantages of redundancy and increased performance sooner and you can always switch to a dedicated box later, with a reduced impact to your configuration and less downtime. Personally, I choose to run virtualized environments even on a dedicated server. I find them far easier to manage and configure. Advances in hardware and software have made virtualization technology fast and the benefits far outweigh any marginal performance cost.

Once you have your virtual machines up and running, create the database and deploy your application with Capistrano to one of them using the techniques you learned in the earlier chapters. You can deploy your application in its simplest form so you can at least start up Mongrel and access your application via a web browser by hitting the mongrel server directly.

Any application will do, but it should be more than an empty rails app. If you need a application to work with, try using the example from Agile Web Development with Rails found at: http://www.pragprog.com/titles/rails2/source_code

You can find entire books about administration of Linux, Apache, MySQL. I may skip some steps here and there to keep our focus on topics specific to Rails deployment. When I do, I'll try to direct you to other resources when necessary to further tighten security or tweak performance.

7.3 Mirror Images

Your server farm will have a number of servers across many roles. Try hard to keep servers in the same role configured identically. Your life will be easier, you will experience fewer surprises, and you’ll have less documentation to write. If identical configurations sound like too much
work, you’re going to love virtualization.

Virtual servers are like files that sit on your hard drive. You can move them, copy them, delete them, and back them up. When it comes time to keep configurations the same, the easiest thing to do is just make a copy. Here are a few strategies for ensuring configuration consistency across your VMs, depending on the size and timing of the change.

**Cold copy it**

When you’re first setting up your environment, just configure one virtual server. Get that server to the point where all of the software, patches, dependencies and configurations are in place so you can run a single instance of your app. Then, make one copy for each server role, including application servers, web servers, and database servers.

**Automate it**

You can often automate a change in configuration with Capistrano. You have seen that Capistrano scripts do a good job, use Ruby code, and can distinguish between server roles for different configurations.

**Hot copy it**

Larger, more serious configuration changes may require you to shut down the server. The cool thing about VMs in this case is that you can use the copy strategy. Simply pull one server out of your cluster, make your changes to it and then copy it to replicate all of the other servers in the farm. This hot copy approach lets you introduce a new server into the farm to handle additional load. Be careful. You will want to always be able to uniquely identify each server. Whenever you make a copy you have to be sure to properly set the IP address, hostname and any other unique information for each server. If possible, automate such a thing with a script common to each of the servers. Remember to document the process!

**Just do it**

If you can’t justify automating a change or don’t want to shut down servers, you might just have to walk through the servers and make the change. Remember to keep organized and document your steps though – it will make a world of difference.
Keep offline master copies

A master copy is an offline configuration that has never been deployed or exposed as a server to the public Internet. You will use it the first time for your cold copy. After you build your initial VM, you can use that copy as an online master copy. Whenever you need to make configuration changes, you can change this master copy. Most importantly, this is the copy you can safely deploy should your servers become compromised due to a security breach. Once a server has been compromised, it is hard to ever trust it again. You’ll have enough to worry about regarding potential threats to your data, let alone the nightmare of trying to clean out your server farm. Having offline master copies allows you to patch the security hole in the offline copy that has never been exposed to the network, and redeploy each of the server roles to the farm.

Use third-party tools

So far I’ve only discussed custom brute force approaches to managing these configurations. Third-party tools that can do the same might even be included with your virtual machine software or by your VPS provider. Shop around and talk to your service and software providers.

7.4 Domain Names and Hosts

With multiple servers in your environment, managing their names becomes important. You’ll manage domain names through the web-based user interfaces you used in Chapter 3, Shared Hosts, on page 42 and Chapter 4, Virtual and Dedicated Hosts, on page 70. Remember, changes to the domain name configuration can take hours or even days to propagate fully throughout the Internet. Managing these details for a cluster is a little tougher than managing a single web server, but you can usually get by knowing only a few more concepts:

A Records The default configuration for most domain name services is to support brainspl.at in addition to www.brainspl.at. The primary domain (brainspl.at) is an "A", or "address", record in DNS terms. This record maps brainspl.at directly to your IP and is the lowest level mapping.

CNAME Records CNAME records are like aliases for an A record, be it one of your own, or someone else’s. Aliases usually indicate the kind of service that a server provides. For example, www.brainspl.at is an alias to brainspl.at that indicates interest in the HTTP server,
Joe Asks…

So do I use an A record or a CNAME?

When I researched the A-record versus CNAME issue for myself, I remember coming across debates over various uses and abuses of CNAME records and advantages and disadvantages of different configurations. My conservative nature led me to avoid playing games and so I generally follow the rules:

- Use A records for mappings of names to IP addresses.
- Use CNAME records to alias A records.

and ftp.brainspl.at indicates interest in the file transfer protocol. Keep in mind that longstanding conventions exist, and you should follow them where you can.¹

What names do I need?

Armed with your new knowledge of domain name configuration, you now need to decide what names you will need. Try to name all of your site’s publicly available nodes, including the load balancer and the nodes that it balances. You’ll probably have a firewall installed too. Getting your names and visibility right in a real production environment can require some advanced firewall setup, which is beyond the scope of this book. For simplicity, I’ll let the web server act as the load balancer and remind you later that a hardware load balancing solution is definitely a must have for the most serious websites.

Imagine that you have a configuration involving a load balancer, two web servers and a database server. You will expose the load balancer to the network, but not the database. So you’ll need a name for the load balancer. The database server may have a name on your local intranet, but it will not have a name in the domain name service (so there will be no db.brainspl.at). The load balancer will distribute requests between the two web servers. They will also need names because you’ll eventually need to access individual web servers directly, or redirect a request to a specific server—for debugging, configuration or testing, if nothing

else. I see three A records and one CNAME because the load balancer will map to the load balancer IP address, and each of the web servers will also have an IP address. I’ll use a CNAME for the www alias that visitors use. I’ll throw in an extra CNAME for a potential caching service. The final table looks like this:

A Records

- brainspl.at => 999.999.999.100 - load balancer
- www1.brainspl.at => 999.999.999.101 - first web server
- www2.brainspl.at => 999.999.999.102 - second web server

CNAME Records

- www.brainspl.at => brainspl.at - alias to the load balancer
- content.brainspl.at => content.contentcache.com - alias to the content caching service

That last CNAME is to a third-party A record that provides caching of large content such as music, images and videos.

When setting up both A records and CNAMEs, you need to set the TTL or Time To Live parameter. This value will determine how often name servers return to your configuration to check for changes. Setting this value a low value, like 30 minutes, will give you a little more flexibility to change your configuration with minimal impact to your site. Setting it to a higher value, like 7 days, will provide better performance of your site because client software like browsers will have to do fewer name lookups. I recommend setting it low to start, so that you can make a few mistakes with minimal impact. Then once you’re comfortable with your configuration, return and increase the values for all A and CNAME records to at least 24 hours.

Now that you have named servers, you can start deploying your application to them.

### 7.5 Deploying to Multiple Hosts

With a Rails and Capistrano, you have a lot of deployment options. Capistrano supports three server roles right out of the box:

- The web role points to servers responsible for static content. Apache or NGinx lives here.
The `app` role points to the server that will run your rails application. Mongrel lives here.

The `db` role points to your database server. MySQL lives here.

Capistrano deploys the right files to each server. If you need to, you can override its behavior. With only one server, you generally deploy all roles to that one server. The following Capistrano script is an example of a single server configuration. I'll reference parts of it for the remaining examples in this section.

```ruby
# Customized deploy.rb
set :application, "brainsplat"
set :user, "clinton"
set :repository, "http://brainsplat.at/svn/#{application}"
set :deploy_to, "/home/#{user}/#{application}"

role :web, "www1.brainsplat.at"
role :app, "www1.brainsplat.at"
role :db, "www1.brainsplat.at", :primary => true
```

### Options for Clustering

Given two servers, you now have a decision to make. The three following options are the most reasonable.

**Isolate the Database.** This first option is very simple and may offer the best performance for an individual request depending on the application. If your application is transaction heavy and depends on a lot of dynamic data, then this option might be the right choice. With this configuration the database alone is separate so it has fully dedicated access to the server resources, which should include of lots of fast disks. You will also have the added security of keeping the database further away from the public network interface. You would configure the isolated database option like this:

```ruby
# Relevant lines of deploy.rb
# ...
role :web, "www1.brainsplat.at"
role :app, "www1.brainsplat.at"
role :db, "internal.brainsplat.at", :primary => true
```

**Isolate the Web Server.** The second option is to isolate the web server so it can concentrate on caching and serving static pages with lots of memory and a fast network connection. If your application is heavy on static content and wants a chance at surviving the Digg effect, you might want to choose a dedicated web server. The application and database are isolated on a second internal server. Conveniently, you still have the
security advantage of keeping the database away from the web server. However, deploying the web server and application server to a separate machines loses the benefit of directly serving cached pages that rails creates on the fly. Clustered or shared filesystems, or another web server on the app server may solve the problem. In any case, the isolated web server roles look like this in a capistrano configuration:

```ruby
# Relevant lines of deploy.rb
# ...
role :web, "www1.brainspl.at"
role :app, "internal.brainspl.at"
role :db, "internal.brainspl.at", :primary => true
```

It’s a subtle difference that has huge consequences. The type of application you’re running and the hardware available to you will dictate the right option. If you plan on quickly growing beyond the capacity supported by either of these options, then you might want to consider a third option, because you’ll likely end up there anyway.

Cluster This third option has both servers running all of the roles: web server, application and database. It offers the benefit of full redundancy. You can lose one entire server and the site will keep running with data intact. The site may well perform better under certain kinds of load. Given two users, each will have a full application stack and dedicated hardware ready to serve the individual request. However, the option is somewhat less secure because the database lives in the same environment as the web server, fully exposed to web related bugs and security risks.

This configuration can be a bear to set up, especially with regard to the database. Separating reads and writes to different databases with a single master/slave configuration won’t do you much good if either one crashes.

If you want full redundancy, you need to implement a synchronous database cluster that supports reading and writing to either database instance. I’ve dedicated a full section to it below Section 7.8, Clustering MySQL, on page 174. For now, I’ll assume it’s already set up. The following Capistrano configuration makes use of it:

```ruby
# Relevant lines of deploy.rb
# ...
role :web, "www1.brainspl.at"
role :app, "www1.brainspl.at"
role :db, "www1.brainspl.at", :primary => true
```
Joe Asks:... Why not just "scale up" the database?

You absolutely could "scale up" the database. The database server is a good candidate for scaling upwards, as it is often a bottleneck and can benefit from ultra-fast disks and lots of memory. You can also put redundant disks in a RAID-1, RAID-5 or RAID-10 configuration that would offer a similar level of redundancy as a software cluster, and would probably perform better. Do not underestimate the costs of such hardware and remember the limitations and consequences of a scale-up approach. They still apply!

role :web, "www2.brainspl.at"
role :app, "www2.brainspl.at"
role :db, "www2.brainspl.at"

Combining Approaches with More Servers

Capistrano and Rails are flexible enough to support a great number of server configuration options. As soon as you scale beyond two servers, you have many, many more options. Here are a few of the more popular options:

4 Servers: Clustered database

This configuration has a clustered database with separate web and app servers. This configuration emphasizes transactions and dynamic data. If your database is the bottleneck or you need to add failover at the database level, this configuration is a good place to start:

# Relevant lines of deploy.rb
# ...
role :web, "www.brainspl.at"
role :app, "app.brainspl.at"
role :db, "db1.brainspl.at", :primary => true
role :db, "db2.brainspl.at"

5 Servers: Clustered web servers

This configuration uses clustered web servers with a separate app server and a scaled up database. The emphasis for this solution is on static content, but has extra muscle in the database server to handle load.
10 Servers

This configuration supports a full cluster with no specific emphasis. Two large database servers support the application as clustering a database to more than two servers starts to yield fewer benefits with each server. If you want to cluster the database server more broadly, consider sharding, which I discuss in more detail in Section 7.8, Challenge 3: Clustering vs. Sharding, on page 175.

Web Servers vs. Application Servers: What’s the difference?

At this point you’ve heard a lot about web and app roles. I’ve also hinted that mongrels may not make the best web servers. It’s time to drill down a little deeper.

The Web Server

The web server is good at quickly routing requests, serving static file content and caching, so it makes an excellent proxy that sits in front of the application server. After briefly flirting with LightHTTPD, the Rails community seems to have settled on one of two servers that are preferred in the web role: Apache and NGinx. Apache is a scalable, full-featured, and extremely reliable web server. But with Apache you may get more than you want and may suffer with having to configure it
regardless. Enter nglix. This Russian creation is very capable and well suited to sitting in front of a mongrel cluster. However, NGinx has a simpler configuration, is very fast and uses minimal system resources. It makes a great candidate for virtual hosting services where memory is limited and software efficiency is key.

**The Application Server**

The application server is an application container that is focused on securely executing code and managing the runtime environment. For Rails, a single runtime environment is not enough because Rails uses a single-threaded architecture. One mongrel can only serve one request at a time. Due to its share-nothing architecture, each concurrent user request to a Rails application requires an isolated runtime environment. The Rails application server architecture has two challenges:

1. Mongrel is not optimized for serving static content and doing so puts unnecessary load on the Rails controller.

2. In a production environment you'll have multiple application servers on different ports, so you need a router of sorts to distribute requests among them.

You'll need a web server optimized for routing requests, serving static content and caching. You can also see the need for an application server optimized for serving Rails requests. Mongrel is by far the most preferred server for Rails content due to it’s performance, stability, and security. FastCGI is also an option, and is marginally faster than Mongrel. But due to stability problems and configuration headaches, few would choose FastCGI over Mongrel.

At this point, make sure your Mongrel instance is working as a cluster, and running as a service. In the sections to come, I'll lay out the other side of this equation: the static web server.
7.6 Apache

Apache may be the most successful open source project ever created. Apache powers some of the biggest Internet sites in the world, has a huge community. Apache also has and an official 501(c)(3) nonprofit behind it – The Apache Software Foundation. Apache is a safe bet to remain the de facto standard web server for a long time to come.

Despite its success, Apache is not always easy to install, configure or manage. Like anything, it’s a matter of perspective and opinion. Some people believe Apache is complex and hard to set up. Others believe it is one of the easiest due to it’s incredible flexibility. Regardless of which one is true, if you choose to run Apache, you want to set it up properly now. You don’t want to hammer in a simple configuration and then be a slave to that configuration later. Before you roll up your sleeves, you should have a more detailed picture of what Apache will do for you.

Separation of Concerns: Web Server Style

I’m going to take a little bit of a different approach to configuring Apache here. I’m going to use more virtual host proxies than usual. I’m going to do this for a few reasons:

• To better isolate various configuration elements to make it easier for you to read and follow,

• To separate the concerns and responsibilities into different virtual servers, and

• To mock the behavior of infrastructure components that would be hard to demonstrate while you follow through these examples on your own – unless you happen to have a hardware load balancer on hand to distribute requests between your virtual machines. I don’t, so I’ll use Apache.

That last point is especially important. You normally would have a hardware load balancer distributing requests to the web servers on each of your virtual machines, which would then either serve up some static content or forward the request to one of the mongrels in your cluster. You can already begin to imagine the layering of responsibilities. Initially, Apache serves as a web server, taking all incoming HTTP requests. Then, Apache serves as a static proxy, serving static pages as requests come in. Then, when dynamic requests come in, Apache serves as a load balancer which forwards requests to the appropriate web server.
However, in the absence of a hardware load balancer, you’ll set up a couple of Apache virtual servers to handle the following tasks:

- Load balancing between the virtual machines,
- Forwarding HTTP requests on port 80 to the web server on one of those virtual machines, and
- Forwarding secure HTTPS/SSL requests on port 443, possibly just to a regular non-secure HTTP web server on port 80 within our intranet.

**Prerequisites**

Before we start, you’ll need to ensure that Apache is installed on your server. If you are hosting with a managed VPS host, your provider may have installed Apache by default. I recommend using Apache 2.2 and that’s what I’m going to configure here. You’ll also need mod_ssl installed to handle HTTPS requests on port 443. If you don’t have these installed, it’s usually a simple matter. On CentOS or Fedora use `yum install httpd` and `yum install mod_ssl`. On Debian or Ubuntu, use `apt-get install apache` and `apt-get install libapache-mod-ssl`.

To fully simulate a load balanced environment on your local machine, you’ll need three (yes 3) virtual machines. Now would be a good time to make copies your virtual machine image and configure their hostnames and IP addresses so that you have three uniquely identifiable machines.

**Apache as a Load Balancer**

Using Apache in place of a hardware load balancer may not yield the best performance, but it will give you an opportunity to explore Apache’s proxy balancer features in a simpler context. In essence, you have two virtual machines and you want to distribute evenly between them. How evenly depends on the load balancer implementation, and Apache’s is fairly simplistic, but it will do for now.

Apache’s configuration starts with one file found at: `/etc/httpd/conf/httpd.conf`

I’ve noticed that some VPS service providers like to break this file up into pieces and customize it quite a bit. You should leave their configuration as intact as possible because it’s likely optimized for the limitations of your VPS. The smart service providers make it easy for you to do so by providing your own `apps` directory to hold all of your configurations for your applications. For your own testing at home or for the
odd VPS with a default Apache install, you simply need to add one line to the end of the httpd.conf file:

```
Include conf/apps/*.conf
```

This line simply includes your private configuration file, giving you a place to work. It can be any directory you like. You can now add your own *.conf files to the /etc/httpd/conf/apps/ subdirectory, which act like extensions to the master configuration file. The other 990 lines of httpd.conf are beyond the scope of this book to discuss in detail. The good news is that the vast majority of those lines are actually inline documentation! The Apache developers do a good job of documenting the configuration file, or at least the basics of it. Take a look and read it over. While you’re there, as an exercise, find some of the following pieces of information: the document root, the user and group to run the processes as, the number of worker processes, listening port, log file locations and the virtual host example. Here’s a bonus question: if you have mod_ssl installed, how is the ssl configuration loaded? I’ll touch upon that in a bit.

The Apache configuration file looks kind of like XML, but it uses pound # signs to comment lines. Once you are a little more familiar with the apache configuration files, you can start to create your own. You’re going to be configuring virtual hosts, which will in some cases override some of the configurations you saw in httpd.conf. Most notably, the root directory will no longer point to the document root that you found. Your virtual host will handle the routing of those requests instead.

**Configuring the Load Balancer VM**

You’re going to create three files to cleanly separate concerns. All of them should be placed in the /etc/httpd/conf/apps directory in the httpd.conf file you edited earlier. If the directory doesn’t exist, create it.

- cluster.conf will define a load balancer that will distribute requests accross two or more machines.
- http_proxy.conf will define a virtual host that will proxy typical HTTP requests on port 80 to the load balancer.
- https_proxy.conf will define a virtual host that will proxy secure HTTPS requests on port 443 to the load balancer.

Let’s start with cluster.conf which contains the following:

```
# cluster.conf
<Proxy balancer://mycluster>
```
BalancerMember http://10.0.0.102:80
BalancerMember http://10.0.0.103:80
</Proxy>

This file defines the cluster definition itself. The Proxy stanza defines the load balancer named "mycluster". You are free to name the cluster whatever you like, and it should probably be something meaningful to your environment or application.

The balancer includes two BalancerMember entries - one for each of our virtual machines. Here I'm using the IP address and specifying the port explicitly. In certain cases you may have different servers running on different ports and apache lets you fully map requests received on port 80 of the proxy to any port on the balancer members. You'll see an example of that later when we map requests to our mongrel cluster.

By itself, cluster.conf would not impact your site very much. You need to somehow tell the server to use the balancer, which brings us to the next step: defining the virtual host to proxy the requests. The file you'll use to do that is http_proxy.conf:

```
# http_proxy.conf
<VirtualHost *:80>
  ServerName brainspl.at
  ServerAlias www.brainspl.at
  ServerAlias 10.0.0.101 # if you don't have a hostname yet

  ProxyPass / balancer://mycluster/
  ProxyPassReverse / balancer://mycluster/

  ErrorLog logs/http_proxy_error_log
  TransferLog logs/http_proxy_access_log
  LogLevel warn
</VirtualHost>
```

The first three lines, ServerName and ServerAlias, tell the virtual server which host names to intercept. These commands allow you to host multiple websites with different domain names on a single server easily. That's pretty nice stuff. I generally map the A record from my DNS configuration to the ServerName property if I can, and then map any relevant CNAME records in the ServerAlias. That implementation is clean and consistent with our DNS configuration.

The next two lines, ProxyPass and ProxyPassReverse, tell Apache to map the root path (/) of this virtual host to a load balancer called "mycluster".
Finally, you'll configure the location and name of our error and access logs, as well as set the log level, which is pretty self explanatory.

Next, I'll dive a little deeper and configure a secured port just like this. I'll put this configuration in a file called https_proxy.conf. It starts out exactly like the HTTP version, but then gets a little more complicated. Here is what the file should look like:

```apache
# https_proxy.conf
NameVirtualHost *:443
<VirtualHost *:443>
  ServerName brainspl.at
  ServerAlias www.brainspl.at
  ServerAlias 10.0.0.101 # if you don't have a hostname yet
  ProxyPass / balancer://mycluster/
  ProxyPassReverse / balancer://mycluster/
  ErrorLog logs/https_proxy_error_log
  TransferLog logs/https_proxy_access_log
  LogLevel warn

  RequestHeader set X_FORWARDED_PROTO 'https'

  SSLEngine on
  SSLCertificateFile /etc/httpd/conf/apps/ssl/crt/server.crt
  SSLCertificateKeyFile /etc/httpd/conf/apps/ssl/key/server.key
  <FilesMatch ".\.(cgi|shtml|phtml|php)$">
    SSLOptions +StdEnvVars
  </FilesMatch>
  <Directory "/var/www/cgi-bin">
    SSLOptions +StdEnvVars
  </Directory>
  BrowserMatch ".*MSIE.*"
      nokeepalive ssl-unclean-shutdown
      downgrade-1.0 force-response-1.0
</VirtualHost>
```

The first line tells Apache that you want to access your virtual host by names based on the URL of the incoming request. Without this, your server would likely never be invoked because the default handler will grab all incoming requests. With this enabled, when it sees the server name or one of the server aliases in the URL, it knows that this is the virtual host to use.

The next seven lines of https_proxy.conf look very much like its HTTP counterpart. In fact, you could have created an include file for the first four lines to avoid the duplication. However, I chose to duplicate it for readability and the fact that it’s four lines in two files in the same directory. I'll warn that there’s a nice balance between somewhere
between excessive duplication and excessive hierarchies of include files. Both can be overdone.

The rest of the file is quite different, starting with RequestHeader called X_FORWARDED_PROTO. Recall that in your cluster configuration both of your balancer members used basic, unencrypted HTTP on port 80. Without forwarding the protocol, your Rails application would be unaware that the request actually came through over a secure connection. This information is important to e-commerce applications that might redirect the user to a secure page when necessary. X_FORWARDED_PROTO tricks Rails into thinking that it was directly requested via an https protocol.

The rest of the lines are awkward to the extreme. This confusion loses friends for Apache. But there is good news. Your mod_ssl installation created a default configuration file, usually called/etc/httpd/conf.d/ssl.conf. In this file you’ll find all of these lines, and a few more, with full descriptions and documentation. I certainly won’t try to increase our page count by repeating the file here, so you should go give it a quick read.

There are two options that I should mention, though: SSLCertificateFile and SSLCertificateKeyFile. These two lines specify the location of your private key file and your certificate. SSL and public key cryptography is another one of those subjects that would double the size of this book. I’ll settle for giving you enough information to set up your test environment, but you should definitely research this topic further for your production environment. Luckily most hosting providers will help you out here, and may even properly set up the keys up for you. You do have to pay for these keys to be signed by a Certificate Authority for them to be valid. You don’t have to do that with your test certificates, but realize that they won’t be secure, and your user’s browsers will tell them so with an irritating warning.

Here’s how to generate the test certificates:

```
#root> # Generate a private key file with a passphrase
#root> openssl genrsa -des3 -out server.key 1024

#root> # TEST ONLY: Remove the passphrase to make our test environment easier to...
#root> openssl rsa -in server.key -out server.pem

#root> # Generate a certificate signing request.
#root> # DO NOT do this in production. The *.pem file is the key file without th...
#root> openssl req -new -key server.pem -out server.csr

#root> # TEST ONLY: Self sign the key.
```
In production the Certificate Authority's signature is what matters. An alternative to generating them is to just use the examples that Apache created when you installed mod_ssl. You can check the default /etc/httpd/conf.d/ssl.conf I mentioned before to find the locations of those example files.

In a true production situation you'll want keep the key file encrypted and lock down the file itself so that only root has access to it. If you lose this file or if it is compromised by a malicious party, you will have to buy a new certificate and immediately revoke the old one.

With that, your Apache based software load balancer is done. You can now test it by starting Apache on all three virtual machines. You can start, restart, stop or test the Apache configuration as follows:

- `sudo /etc/init.d/httpd start`
- `sudo /etc/init.d/httpd restart`
- `sudo /etc/init.d/httpd stop`
- `sudo /etc/init.d/httpd configtest`

The `configtest` option will often give some insight into any problems if the server is failing to start.

To test the cluster and see how it behaves, you should put something interesting and uniquely identifiable in the `/var/www/html/index.html` of your balancer members. That way you can watch how your balancer distributes requests between them by refreshing the balancer URL. If everything is working properly, you should see the page alternating between the two balancer members.

You're not here to serve static index.html pages though. It's now time to set up the balancer members with your Rails application! The cool thing is that it's really no different than what you've seen so far. It's just another proxy balancer on another server. You already know most of what you need, but let's step through it together.

**Apache as a Mongrel Proxy**

As you recall, Mongrel is a cool little web server that's custom tailored to serve up a single request through the Rails share-nothing architecture. Therefore, you need to run multiple mongrels to allow multiple users of your application to use your site concurrently. Otherwise your
users would all be waiting in line for a single Mongrel server. You’ve just learned in the previous section that Apache can balance requests across multiple servers, so let’s do that for Mongrel now.

First, remember that you’re no longer on the same virtual machine as before. You’re done with the software load balancer and are now working on one of the balancer members.

On each balancer member you’re going to create two files, only to keep the concerns separated as before. These files will also reside in `/etc/httpd/conf/apps/` and you’ll need to remember to add the `Include conf/apps/*.conf` to `/etc/httpd/conf/httpd.conf`. These are the files you’ll need:

- `mongrel_cluster.conf` will define a load balancer that will distribute requests across two or more Mongrel servers.
- `mongrel_proxy.conf` will define a virtual host that will proxy typical HTTP requests on port 80 to the mongrel cluster.

Note the lack of a secure SSL configuration. The reason for that omission is that our software load balancer configured in the last section will take care of that for us. A hardware load balancer would do the same. Secure requests are proxied through to a basic HTTP service on port 80 of our balancer members, which are inside our firewall. This keeps the configuration of the balancer members far cleaner. The exception is if you do want to expose your balancer members directly (e.g. www1.brainspl.at), then you may need additional configuration in the load balancer to support SSL for each one.

The Mongrel cluster configuration contained in `mongrel_cluster.conf` will look very familiar.

```bash
# mongrel_cluster.conf
<Proxy balancer://mongrelcluster>
  BalancerMember http://0.0.0.0.8000
  BalancerMember http://0.0.0.0.8001
  BalancerMember http://0.0.0.0.8002
  BalancerMember http://0.0.0.0.8003
</Proxy>
```

This time, you’ll notice that we’re not balancing to a different server. Recall our Capistrano roles. In this case it’s clear that this one server is filling both the web and app roles. If we wanted to separate the web and app roles, we’d run the mongrels on a different server and the IP addresses in the cluster would be to a remote server. For example:
# Alternative mongrel_cluster.conf
# with separated web and app roles

```xml
<Proxy balancer://mongrelcluster>
  BalancerMember http://10.0.0.104.8000
  BalancerMember http://10.0.0.104.8001
  BalancerMember http://10.0.0.104.8002
  BalancerMember http://10.0.0.104.8003
</Proxy>
```

Also remember that in our `/etc/mongrel_cluster/myapp.conf` there was a line restricting the IP address allowed to access the Mongrel servers.

```bash
# Partial /etc/mongrel_cluster/myapp.conf
# ...
address: 0.0.0.0
# ...
```

If you’re running both on the same server, you can use the IP address 0.0.0.0 or 127.0.0.1. If you’ve separated the web and app roles, this address line will have to match the web server / mongrel proxy – the server you’re working with now (e.g. 10.0.0.101). Note that these are all internal IP addresses we’re using here, not public IP addresses exposed to the Internet.

Also recall the following two lines from `/etc/mongrel_cluster/myapp.conf`

```bash
# Partial /etc/mongrel_cluster/myapp.conf
# ...
servers: 4
port: "8000"
# ...
```

These two values will determine how many balancer members you will have and what their port numbers will be. In this example, there would be four servers ranging from port 8000 to 8003. Thus our balancer configuration.

The second file that you need on each balancer member, and the final Apache configuration file we’ll deal with, is `mongrel_proxy.conf`. If you’re keen, you’re probably thinking you already know the answer here, as you can just add a virtual host that routes all requests to the balancer. That does indeed work, and the configuration would look nearly identical to the virtual hosts we defined on the load balancer.

```bash
# A very simple mongrel_proxy.conf
<VirtualHost *:80>
  ServerName www1.brainspl.at
  ServerAlias 10.0.0.102 # if you don't have a hostname yet
  RequestHeader set X_FORWARDED_HOST 'www.brainspl.at'
</VirtualHost>
```

Prepared exclusively for Arthur Pinkney
ProxyPass / balancer://mongrelcluster/
ProxyPassReverse / balancer://mongrelcluster/

ErrorLog logs/mongrel_proxy_error_log
TransferLog logs/mongrel_proxy_access_log
LogLevel warn
</VirtualHost>

I've added only one new element here. Did you spot it? The X_FORWARDED_HOST header is there to trick Rails into thinking the hostname is one thing, even if it is actually something else. The reason for this is that after all of this proxying and indirection, the original hostname will likely be lost in the shuffle. This trick just makes it easier for Rails to do things like redirect to its own server, without having to worry about unintentionally getting an internal hostname instead of the public web address.

The disadvantage here is performance. This configuration would have Mongrel serving up static content and images, as well as all cached rails pages. You didn't go through all of this work just to be as slow as Mongrel! You want to leverage the Apache web server and have it serve up the static stuff. The unfortunate solution here is yet another configuration that will have you turning up your nose a bit. It is not a trivial or easily digestible approach. Luckily, once again, you can pretty much take it line for line and use it in your own applications.

There is another approach that does away with the ProxyPass and ProxyPassReverse configurations and replaces them with a number of URL rewriting statements. The advantage is that Apache will serve static content directly, which is far faster and leaves your Mongrels available for more important work that actually requires application code to run.

The following listing contains the mongrel_proxy.conf file in its entirety. I've numbered the sections for later discussion, and identified the purpose in capital letters where relevant. Look through it and then I’ll walk you through it.

# A full featured mongrel_proxy.conf
<VirtualHost *:80>
  ServerName www1.brainspl.at
  ServerAlias 10.0.0.102 # if you don't have a hostname yet
  RequestHeader set X_FORWARDED_HOST 'www.brainspl.at'

  ### New configuration elements begin here. ###

  # 1. Document root specified to provide access to static files directly
  DocumentRoot /home/deploy/apps/myapp/current/public
  <Directory /home/deploy/apps/myapp/current/public>
    Options FollowSymLinks
  </Directory>
</VirtualHost>
AllowOverride None
Order allow,deny
Allow from all
</Directory>

RewriteEngine On

# 2. SECURITY: Don't allow SVN directories to be accessed
RewriteRule ^(.*/)?\.svn/ - [F,L]
ErrorDocument 403 "Access Forbidden"

# 3. MAINTENANCE: Temporary display of maintenance file if it exists
RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
RewriteCond %{SCRIPT_FILENAME} !maintenance.html
RewriteRule ^.*$ /system/maintenance.html [L]

# 4. PERFORMANCE: Check for static index and cached pages.
RewriteRule ^/$ /index.html [QSA]
RewriteRule ^([^.]+)$ $1.html [QSA]

# 5. OTHERWISE: If no static file exists, let Mongrel handle the request
RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME} !-f
RewriteRule ^/(.*)$ balancer://mongrel_cluster%{REQUEST_URI} [P,QSA,L]

# 6. PERFORMANCE: Compress text output for compatible browsers
AddOutputFilterByType DEFLATE text/html text/plain text/xml
BrowserMatch ^Mozilla/4 gzip-only-text/html
BrowserMatch ^Mozilla/4\.[0678] no-gzip
BrowserMatch \bMSIE !no-gzip !gzip-only-text/html

### End new configuration elements, logging below is the same. ###

ErrorLog logs/mongrel_proxy_error_log
TransferLog logs/mongrel_proxy_access_log
LogLevel warn
</VirtualHost>

The first thing you'll notice is a fairly typical looking web server document root stanza (#1). This is what allows static content to be served. However, before it attempts to serve any file, Apache must execute a number of rewrite rules. These rules are based on regular expressions that check for some condition and then rewrite the request to modify the resulting behavior. For example, the first rewrite rule (#2) blocks access to .svn directories that Capistrano tends to leave behind. The next group of rules form a maintenance (#3) rule that lets you gracefully respond to visitors while your site is not available. But the next rules (#4) demonstrate the overall goal: better performance. They look for static index pages when no file is specified in the URL, and also
serve up cached Rails pages if they exist. If Apache can’t find a static file matching the request, Rails handles the request through the last pair of rewrite rules(#5). Finally, to speed up network transfers at the expense of some processing power, you can optionally enable the mod_deflate output filter (#6) to compress outbound text data.

These are huge performance advantages for high-traffic sites. Not only is Apache faster at serving static content, but it keeps unnecessary load off of the Mongrels so they can efficiently handle only Rails requests.

You can now start up Apache and the Mongrel cluster on your balancer members. Better yet: have Capistrano do it! After all, that’s what Capistrano is for. Also, you only have a single database so far, and therefore you have to configure it to accept connections from remote servers. This database configuration is usually a simple matter of granting permissions to remote users (e.g. ’deploy’@’10.0.0.102’) on your database. I’ll cover more advanced MySql topics later in this chapter.

Congratulations! You’ve now configured a highly flexible and high performance clustered server architecture. If you look back and consider some of your options, you can scale this out to six servers quite easily: the software load balancer, two web proxy servers, two application servers and a database server. You can play around with your local virtual machines to try adding another web server, or to try separating the web and app roles onto different hosts. Otherwise, if you’ve found all of this to be a little overbearing, with too much unneeded flexibility and indirection, or if your server is strictly limited to low resources, you might need an alternative to Apache. In that case there’s NGinx.

7.7 NGinx, from Russia with Love.

NGinx(pronounced engine-x) is a fast, lightweight webserver written by Igor Sysoev. It is extremely well suited as a front end server for Mongrel clusters. Out of the box, some find that NGinx serves static files faster and under heavier load than Apache, often using a fraction of the resources under similar work loads. Where Apache focuses clearly on modularity and flexibility, NGinx focuses on simplicity and performance. That’s not to say NGinx isn’t feature rich, as it has nice built in rewrite and proxy modules and a clean configuration file syntax that makes it a pleasure to use.

The proxy module has similar capabilities to Apache’s mod_proxy_balancer so it works great for fronting clusters of Mongrels. Conditional if state-
ments and regular expressions matching allow precise control over which requests gets served as static content, and which dynamic requests nginx will proxy through to a Mongrel back end.

Nginx is surprisingly full featured considering how fast and efficient it is. It has support for SSL, HTTP AUTH, FastCGI, gzip compression, flv streaming, memcached and many other modules so you'll be able to handle the more advanced topics in this book like caching and clustering. The Nginx wiki http://wiki.codemongers.com/nginx is the definitive place go for Nginx documentation.

Nginx can do everything you configured Apache to do in the previous section: software load balancing, acting as a mongrel proxy and and serving static content. If you repeated everything all over again with Nginx you’d probably find it quite repetetive. Instead, I’ll show you the nginx configuration and separate the concerns as before. That should give you enough knowledge to configure Nginx as we did Apache, or however you like. I think you'll agree that the simplicity of the Nginx configuration will make it quite a bit more straightforward compared to Apache.

## Starting, Stopping and Reloading

If you haven’t already noticed, nginx has a more hardcore feel to it. Case in point: you have to compile it from source and even after its installed, it doesn’t come with all of the user-friendly service scripts like Apache does. You’ve already installed ngnix in Chapter 4, Virtual and Dedicated Hosts, on page 70. To start it you’ll be running the executable directly and to stop or restart it you’ll be using the `kill` program to send signals to the process. This sounds like it could get messy, but luckily the basics for using it are pretty simple. To start the server, run the command `/usr/local/nginx/sbin/nginx` as root. That command will load the configuration file at `/usr/local/nginx/conf/nginx.conf`. To load or test other configuration files, you’ll need a couple of options:

- `-c filename` Load the configuration file named filename instead of the default file.
- `†` Don’t run the server. Just test the configuration file.

As with other *nix applications, you can control nginx through the use of signals. To use signals, you’ll need the process id. Use this version of the `ps` command to get it:

```
ezra$ ps -aef | egrep '\(PID|nginx\)'
```

Report erratum
The master process PID is 6850, as you can see in the Parent Process ID (PPID) column of the output. Nginx is architected as a single master process with any number of worker processes. We are running 4 workers in our configuration. You can stop the master process with the kill 15 signal (kill -15 6850), this will also kill the worker processes. If you change a configuration file, you can reload it without restarting Nginx.

```
ezra$ # The -c option is only needed if your .conf file is in a custom location.
ezra$ sudo /usr/local/nginx/sbin/nginx -c /etc/nginx/nginx.conf
ezra$ sudo kill -HUP 614
```

The first command sets up the new configuration file. The second sends a kill signal with HUP. The HUP signal is a configuration reload signal. If the configuration is successful, ngnix will load the new configuration into new worker processes and kill the old ones. Now, you know enough to run the server, stop the server, and configure the server. The next step is building a configuration file.

**Configuring Nginx**

Nginx has a master configuration file that includes a number of other files, including our virtual host configurations. The master configuration file is stored at `/usr/local/nginx/conf/nginx.conf`. We’re going to add a line to the end of that file to include our virtual host configurations. You’re going to add only one line, but it’s important to put it in the right spot. You want it in the `http` block.

```
# Partial nginx.conf
# ...
http {
  # ...
  # Include the following line at the end of the http block
  include /etc/nginx/vhosts/*.conf;
}
```

Be sure to create the vhosts directory for yourself. This should seem familiar, as it is very much like the approach you saw when setting up Apache. However, the Nginx master configuration file is not well documented with inline comments so I’ve done that for you in Appendix A.
on page 219. Please take this time to refer to the nginx.conf file on your server, using Appendix A as a guide.

There is a default virtual host built into the master configuration file that you should remove. As you'll see in the next section, you'll be replacing the default virtual hosts with your own. You'll keep it in a separate file for cleanliness though.

**Virtual Host Configuration**

The Nginx virtual host configuration is similar to the Apache equivalent, but with less noise. The syntax is more like that of a domain specific language than a structured file format, which is more comfortable for Ruby programmers. You'll find most of the configuration elements familiar from our earlier discussions of the Apache configuration, but I'll go through each just to be sure.

The first thing you'll do in your virtual host configuration is set up a load balancing cluster for your mongrels. You'll then configure some typical HTTP details such as the server name, port, root directory, error files and logs. Finally, you'll set up the URL rewriting rules to make sure that you get the most out of Nginx by having it serve static files and cached rails files, to help keep unnecessary load off of your Mongrels.

Take a look at an Nginx virtual host configuration file now. I'll name mine after my site and call it brainspl.at.conf.

```conf
# brainspl.at.conf
# Nginx virtual host configuration file
# to be included by nginx.conf

# Load balance to mongrels
upstream mongrel_cluster {
    server 0.0.0.0:8000;
    server 0.0.0.0:8001;
    server 0.0.0.0:8002;
}

# Begin virtual host configuration
server {
    # Familiar HTTP settings
    listen 80;
    server_name brainspl.at *.brainspl.at;
    root /data/brainspl.at/current/public;
    access_log /var/log/nginx/brainspl.at.access.log main;
    error_page 500 502 503 504 /500.html;
    client_max_body_size 50M;
}```
# First rewrite rule for handling maintenance page
if (-f $document_root/system/maintenance.html) {
    rewrite ^(.*)$ /system/maintenance.html last;
    break;
}

location / {
    index index.html index.htm;

    # Forward information about the client and host
    # Otherwise our Rails app wouldn't have access to it
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_max_temp_file_size 0;

    # Directly serve static content
    location ~ ^/(images|javascripts|stylesheets)/ {
        expires 10y;
    }
    if (-f $request_filename) {
        break;
    }

    # Directly serve cached pages
    if (-f $request_filename.html) {
        rewrite (.*) $1.html break;
    }

    # Otherwise let Mongrel handle the request
    if (!-f $request_filename) {
        proxy_pass http://mongrel_cluster;
        break;
    }
}

The upstream block defines a load balancing cluster of three Mongrel servers, which in this case happen to be on the same host as the Nginx server. You can of course separate the mongrels onto their own application server, and then simply specify a remote IP address here. You can define as many clusters like this as you need, but be sure to name each one uniquely and descriptively. In this case I named it fairly generically as mongrel_cluster.

The server block is the virtual host directive. You will need one server block for each rails app you want to run, and pair it with a cluster as you did in your upstream block. If you only have a single virtual host accessing the cluster, you might as well keep it in the same file. But if
you have two or more virtual hosts accessing it, you might want to put the upstream block in its own include file. That way it’s easier to find and you can maintain your separation of concerns as before.

The contents of the server block begins with some basic HTTP configuration. This HTTP configuration is quite self explanatory, a result of the simplicity of the NGinx configuration syntax. Read through it now. First, I’m configuring nginx to listen on the standard HTTP port 80. The server is named brainspl.at and I’ve also aliased all subdomains (*.brainspl.at). I’ve set the document root to the /public directory of my Rails application, which was deployed with capistrano. I pointed the main access log at a file specific to this server and set the default error page for 50x class error codes. Finally, for security reasons, I’ve limited the maximum response size to avoid attempts to overload the server.

Just below the basic HTTP configuration, you’ll see the first rewrite rule. You’re already familiar with rewrite rules from the Apache configuration, however I think you’ll agree that these are cleaner and easier to read. That first rewrite rule checks for the existence of a maintenance.html and displays it (and only it) if it exists. This rule allows you to gracefully respond to your visitors while you are making changes requiring an application shutdown.

The location block allows Nginx to set custom configuration options for different urls. In this default configuration I’m setting configuration options for the root (/”). The index directive tells nginx which file to load for requests like / or/foo/. Here I’ll use the standbys index.htm andindex.html.

Next you’ll see a group of proxy_* directives that forward information about the client and spoof information about the host to help hid the Nginx proxy from the Rails application. This way, Rails has all of the information it would normally have if it were serving requests directly to the user from one of the Mongrel servers. The forwarded information includes the IP of the client and the hostname that we want Rails to see (e.g. www.brainspl.at instead of the internal address of our Mongrel server). This is important for the request objects work properly.

The sections immediately after the proxy_* directives are the meat of the URL rewriting. The expires directive sets the Expires header for any urls that match images, javascripts or stylesheets. This rule will make the client download only these respective assets once and then use their local cached versions for 10 years from the first download. When Capis-
trans deploys Rails apps, Capistrano appends a timestamp to the url so that clients will think they are new assets and download them again. This allows you to push changes to the same filename and properly invalidate the cache.

Next, if the requested filename exists on disk just serve it directly and then finalize the request by breaking out of the block. If the static file doesn’t exist, Rails then checks for the requested filename with .html appended. In other words, it checks for a cached Rails view. So when a user requests a url like /post/foo, Nginx will check for /post/foo.html and serve it directly instead of proxying to rails for a dynamic page.

Finally if none of the other rules matched, the config proxies the request to one of the mongrels defined in our upstream block.

Secure Connections

To use ssl with Nginx you use a configuration that matches the server block as above, but with the following differences:

- The secure HTTPS/SSL server will normally listen on port 443.
- SSL needs to be enabled.
- You need to create secure certificate and private key files, and put them in the appropriate directories, just as we did with Apache.
- You need to forward one more header, to ensure Rails knows when we have a secure connection or not.

You can put the configuration in another configuration file called brinssl.conf. Remember to consider putting the upstream block from the non-ssl configuration into its own file and include it in the nginx. You can add the relevant lines like this:

```
server {
    listen 443;
    ssl on;
    # path to your certificate
    ssl_certificate /etc/nginx/certs/server.crt;
    # path to your ssl key
    ssl_certificate_key /etc/nginx/certs/server.key;
    
    # put the rest of your server configuration here.

    location / {
        # set X-FORWARDED_PROTO so ssl_requirement plugin works
        proxy_set_header X-FORWARDED_PROTO https;
    }
}
```
# standard rails+mongrel configuration goes here.

Notice I didn’t define the mongrel cluster again. I can use the same one.

Taking it to the Next level

You can get more life out of your web tier by first introducing hardware load balancing to replace the Apache based load balancer you built earlier. Hardware load balancers offer many of the same features plus more, including enhanced security, proxying HTTPS/SSL requests and server affinity (sticky load balancing).

Another excellent enhancement is to offload serving of large static content from your own web servers, and instead employ a Content Delivery Service. These services will cache images, videos, music, PDF files, and any other large files that you need not bother your own servers with. It doesn’t make your application any faster of course, but it certainly will reduce the load on your web server’s network, memory and disks.

7.8 Clustering MySQL

I’ve shown how you might cluster multiple web and application servers behind a software load balancer, but a single database on the back end can only take you so far. The database would be thrashed severely if you had 10 servers all vying for its time and resources, so you need some solution to allow for multiple databases in your application infrastructure. I should warn you that you must customize clustering solutions to fit the application that you are building, considering carefully the balance of performance against data integrity concerns.

The Challenges with MySQL Clustering

In this book, I’ve chosen MySQL as it is by far the most common and best supported database for Rails applications. Truth be told, for the most extreme scalability problems, there are better databases on the market that offer more powerful clustering features. As it turns out, MySQL is actually not a very good candidate for clustering at all, due to some challenges.

Challenge 1: The Relational Database

MySQL is a relational database. Relational databases do not scale well to super high loads like those experienced by Google and Facebook.
Instead, such sites will often use simpler, faster, systems such as the Oracle Berkeley DB, which is an in-process non-relational data storage solution that offers far better performance and stability, even in a distributed environment. Others may use in memory object-oriented databases or even flat files! Each of these has their pros and cons, but the relational database can actually be among the slower options, often favoring features over performance.  

**Challenge 2: Asynchronous vs. Synchronous Replication**

MySQL only supports asynchronous replication. With this scaling model, you will have a lag between the time a transaction completes on one of the databases and when MySQL replicates that row to all of the others in the cluster. This lag time can cause you pain when you are dealing with unique indices such as the primary key. It’s especially challenging with other unique indices. Say that a given user must have a unique login. If two users took the same name at the same time, you’d have inconsistent data. The only solution is often your own custom algorithm for guaranteeing uniqueness. A few ambitious developers have formed projects to implement a synchronous clustering solution for MySQL, one that would eliminate the time-lag problem. One such project called google-mysql-tools includes a feature called SemiSyncReplication. Another is solidDB for MySQL, by Solid Information Technology. It also offers synchronous replication and other high availability features. If you’re serious about clustering MySQL, you should watch these projects.

**Challenge 3: Clustering vs. Sharding**

Clustering in general does not perform well enough for the most extreme database loads. Even if you do manage to get a synchronous cluster set up, certain kinds of transactions must update all of the databases before completing. This limitation means that if you have ten database servers in your synchronous cluster and update a number of records that are centrally dependent to your system, your transaction will need to write to all ten servers! Your database cluster can easily become more of a hindrance than a help. Furthermore, it’s hard to cluster in a geographically diverse way, so it becomes difficult to put one server in Japan and another in Canada and have them belong to a cluster. The performance implications of synchronizing a database across a wide network is not at all practical, regardless of the other benefits. So as

---

an alternative to clustering, many modern high-load websites use an approach called sharding.

Sharding means splitting your database up into groupings of cohesive data, such that all of the data that a certain user or function requires is co-located in one database, and data for other users or functions is stored in another database. For example, you might choose to store all of the user’s who’s names start with A - L in one database, and M - Z on another. Or, you may choose to keep records based on geography, so you could store Japanese records in a database in Japan, and Canadian records in a database located in Canada. Perhaps you can easily organize your database by data and time: a news site may store the most recent articles on one server and older archived ones on another. Sharding is highly application dependent so I can’t really do a good job of discussing approaches in a generic way within the scope of this book. You can imagine how complex this may make your application though.

As you may be thinking by now, a scale-up approach for the database may be an easier solution. After all, you can buy a monstrous database server with 16 cores, 16GB of RAM, 4 independent network interfaces, 16 SCSI drives and a RAID controller with a battery backup and 64MB of cache RAM. But that sounds expensive, and this isn’t a hardware book! So pretend that you’re a startup with a Rails application experiencing medium to high load, and that you’ve made an informed decision to cluster our MySQL database despite these challenges.

**Separating Reads and Writes**

The first approach to scaling out your database would be to separate the reads from the writes. MySQL has decent support for this model through its Master/Slave Replication. The idea here is that all data is written to one of the single databases, the Master, while all data reads occur on the Slave. This strategy lets you to optimize the databases more specifically for read or write performance and split the database up on to separate servers.

**Configuring MySQL Master/Slave Replication**

I have kind of a conservative nature, so I don’t like to mess around with existing data too much. So regardless of whether or not I have an existing database, I take the same approach to introducing the Master/Slave configuration to my environment. I’ll take the time to build a new Master and Slave virtual server from scratch. I find it less risky
because I can work offline, and less problematic because I'm not bal-
ancing between old and new configurations. Once I've prepared the new
Master/Slave servers, its simply a task of dumping your data from
the old database, uploading to the new one. Then, I can shutdown
the old server and introduce the new one into the environment. This
approach may sound like you would have a lot of downtime, but really,
you only need to be down for as long as it takes to dump the old data
and import it into the new database. You won't have to rebuild your
database infrastructure very often.

To set this up you'll need to MySQL 5.0+ database servers, on separate
virtual machines. The first thing you'll need to do is give each of your
servers an identity. On the master machine, open /etc/my.cnf and add
the following two lines to the [mysqld] section.

# /etc/my.cnf on MASTER
# These lines added below the [mysqld] section
log-bin=mysql-bin
server-id=1

The first line, log-bin, tells MySql to log activity in a binary format file
using the prefix specified on the right side of the assignment. This
binary logging only needs to be done on the master, and it's what the
slave will read from when replicating data. The second line, server-id, is
a unique identifier for the server, in this case, 1. Then restart your the
Master server using:

# Restart MySQL -- My conservative nature leads
# me to avoid restart scripts for major changes.
/etc/init.d/mysqld stop
/etc/init.d/mysqld start

Next, set up the Slave, and that's even easier. You don't need to use the
binary logging on the Slave. Add the server-id to the Slave, choosing a
different ID of course, and then restart the Slave as you did the master.

# /etc/my.cnf on SLAVE
# Following line added below the [mysqld] section
server-id=2

You should now log into the Master server as root an query replication
status using SHOW MASTER STATUS, which produces output like the follow-
ing. The important bits to note are the File and the Position. You'll need
those to configure the slave.

mysql> # ON MASTER;
mysql> SHOW MASTER STATUS;

1. row

Prepared exclusively for Arthur Pinkney
File: mysql-bin.000001
Position: 98
Binlog_Do_DB:
Binlog_Ignore(DB:
1 row in set (0.00 sec)

Now log into the slave and set the master using CHANGE MASTER TO. I usually use the same user as my Rails app, as it would normally have all of the necessary permissions (ALL PRIVILEGES). You'll need to grant privileges to the remote user though, for example: 'deploy'@'10.0.0.3'.

Notice that I used the name of the file and position that I retrieved from the Master server in the previous step. You

mysql> # ON SLAVE;
mysql> CHANGE MASTER TO
   -> MASTER_HOST='10.0.0.3',
   -> MASTER_USER='deploy',
   -> MASTER_PASSWORD='deploypassword',
   -> MASTER_LOG_FILE='mysql-bin.000001',
   -> MASTER_LOG_POS=98;
Query OK, 0 rows affected (0.04 sec)

You can now start the slave by using the START SLAVE command and stop it using STOP SLAVE. Stopping is more like pausing: when you start it up again, MySQL will catch up on anything it's missed while it was stopped.

It can be frustrating if you make a mistake, which is another reason why I start with a fresh server. If you make a mistake and somehow the servers get out of sync and become deadlocked, you can start over. Simply use the RESET MASTER and RESET SLAVE. The slave should be stopped, before resetting them. You should requery the master status with SHOW MASTER STATUS and rerun CHANGE MASTER TO to reconfigure the slave before restarting it. It's a worthwhile exercise to tinker around with this on a couple of test database instances and make mistakes on purpose to see how to get yourself out of them. If you have data to import, you can simply run your dump script against the Master once everything is set up correctly. The slave should replicate all of the imported data assuming it's all set up correctly.

You can also set up multiple read-only databases. If your application is far heavier on reads than writes, then you may want more than one read-only database. The cool thing is that adding more read-only slaves can be a simple matter of stopping the slave temporarily, copying the slave virtual machine and of course configuring the machine's identity. Recall that this includes the hostname, IP address and also the MySQL
server ID! Don’t forget that. You can then start up the slaves again and the new one will pick up as if it has always been there.

To test out your configuration, connect to the Master server, create a database, a table and write a row or two to it. Then log onto the Slave server to see if the changes were replicated.

**Configuring Your Rails Application**

Now that you have two databases, one for writing and one for reading, we need to tell Rails how to use them. Unfortunately Rails does not support this out of the box. So you can either write something yourself to override the finders and which database they can connect to, or you can search for something that someone else has already built. I did, and I found something I really like: acts_as_readonlyable (yes that’s the name).


Names aside, the acts_as_readonlyable uses a very clean syntax and simple configuration for dealing with separate read and write databases. And it supports multiple read-only databases too! You can install it as a plugin with:

```
script/plugin install svn://rubyforge.org/var/svn/acts-as-with-ro/trunk/vendor/p ...
```

Once it’s installed, you can configure additional read-only databases in your `database.yml` file.

```
production:
  database: my_app_master
  host: master_host

read_only_a:
  database: my_app_slave
  host: slave-a

read_only_b:
  database: my_app_slave
  host: slave-b
```

Applying the plugin to your model classes is straightforward, and when you do so, your finders will behave differently. They will use the read-only databases specified in the parameter of the declaration. For example:

```ruby
class Product < ActiveRecord::Base
  acts_as_readonlyable [:read_only_a,:read_only_b]
end
```
Here I've chosen to use two read-only databases. However, I think we can do one better. We can use this approach to achieve a sort of "poor man's sharding", but simply being selective about which read-only database is used by each model. For example:

```
# partial database.yml
#...
read_only_products:
  database: my_app_slave
  host: slave-products
read_only_articles:
  database: my_app_slave
  host: slave-articles

# product.rb
#...
class Product < ActiveRecord::Base
  acts_as_readonlyable [:read_only_products]
end

# article.rb
#...
class Article < ActiveRecord::Base
  acts_as_readonlyable [:read_only_article]
end
```

You can also get a little extreme and just apply the plugin to all of your model classes and cross your fingers, like this:

```
# environment.rb
# not recommended....
class << ActiveRecord::Base

  def read_only_inherited(child)
    child.acts_as_readonlyable :read_only
    ar_inherited(child)
  end

  alias_method :ar_inherited, :inherited
  alias_method :inherited, :read_only_inherited

end
```

I highly recommend against this. Due to the asynchronous replication in MySQL, there will be certain data you'll always want to read from the Master database. Rails sessions are a good example where lag between the write and read may create instability in your application. Also, beware of anything to do with money!

You've seen how to enjoy multiple read-only databases, but are still
constrained by only being able to write to a single database. It’s time to enable two read/write databases.

**Multi-Master, Read/Write Clustering**

The advantage to having two or more databases that accept writes is mostly stability and redundancy. Performance isn’t greatly improved, because eventually the data does have to be written to each database in the cluster. But if you lose a server to a catastrophic failure, you can rest assured that your data will be mostly intact. I say "mostly" again because of the asynchronous replication. There is still a slight chance that data will be lost in a crash, before it can propagate throughout the cluster.

Now that we’ll be reading and writing to the cluster, the challenges of asynchronous replication is doubled. Not only do we have to worry that the data may not be there when reading, but we also have to worry that it might already be there upon writing! If somehow a duplicate value was written to the same primary key column in two databases within the cluster, we’d have a real problem. MySQL allows us to configure offsets to keep autogenerated primary keys unique. Server 1 could have odd keys and server 2 could have even keys, for example. But if you have any other unique indices on your tables, you will have to find your own solution for preventing conflicts.

**Configuring the Asynchronous Multi-Master MySQL Cluster**

The configuration doesn’t look much harder, but it can definitely be frustrating initially. So if you’re going to try this, definitely bring a full bag of patience. Since you’re already familiar with the `my.cnf` file, I’ll spare you some time and tackle both master configurations in one shot. The following configuration files have comments identifying each server:

```plaintext
# /etc/my.cnf on FIRST MASTER
# These lines added below the [mysqld] section
server-id=1
log-bin=mysql-bin
log-slave-updates
replicate.same-server-id=0
auto_increment_increment=10
auto_increment_offset=1

# /etc/my.cnf on SECOND MASTER
# These lines added below the [mysqld] section
server-id=2
log-bin=mysql-bin
```

---

Report erratum

Prepared exclusively for Arthur Pinkney

this copy is (B2.0 printing, December 2007)
log-slave-updates
replicate-same-server-id=0
auto_increment_increment=10
auto_increment_offset=2

Notice the differences compared to the Master/Slave configuration we discussed before. Of course you still need the server-id to identify each server. The first difference though, is that both servers now use log-bin to enable binary logging. Each server will produce its own binary log and read the binary log from the other server. The log files allow each server to pick up the writes from the other server. It’s the same concept as Master/Slave, but done twice over. You’re using the same file prefix for the binary log files, but you can use whatever you like. The server name works perfectly well.

The log-slave-updates option ensures that if you chain more than one slave together in a circular arrangement, MySQL will forward along all updates received from other servers. Since you don’t want to send the same update around in an infinite replication loop, the next option replicate-same-server-id tells a server to ignore its own updates.

Finally, the last two lines of each file help MySQL deal with asynchronous auto increment key generation. The first, auto_increment_increment tells MySQL to increment auto increment fields by 10 each time, essentially dividing the total number of possible keys a server can generate by 10. The auto_increment_offset is basically added to the increment. By making this offset different for each server, you will avoid key collisions. The first master will generate keys like 1, 11, 21, and 31, and the second master will generate keys like 2, 12, 22, and 32. Having an increment level of 10 basically leaves room in the keyspace for 10 servers in your cluster. You can tune it higher or lower depending on the number of servers you expect to have, but I don’t think introducing more than 10 servers in an asynchronous cluster is practical. In fact, I probably wouldn’t do more than two or three, and would favor a hardware solution or replacing MySQL with a more capable clustered database solution.

The remainder of the configuration is the same as for the Master/Slave setup, but doubled up. As before, execute SHOW MASTER STATUS, but on each server. Take the the values from each master and using them in the appropriate file and position parameters of the CHANGE MASTER TO statement, on each slave. Then, START SLAVE on both servers to bind them to each other. Refer back to these steps in the Master/Slave dis-
Testing your work is pretty straightforward. Create a database, some tables and insert some rows on each server to ensure that MySQL updates both databases. It’s pretty cool when you get it up and running. However, as you’ll see in the next section, it’s not all sunshine and roses.

**Configuring Your Application for the Multi-Master Cluster**

The good news about a Multi-Master, read/write cluster is that you may not have to change your application at all. You won’t need any plugins or special software. Your application sees a fully functional database and is unaware of the cluster. This transparency is nice, especially compared to handling the clustering by hand. But remember, being explicit with each update has advantages too, along the lines of the sharding solutions I presented earlier.

The Multi-Master approach has its own challenges – serious challenges. Again, because of the asynchronous nature of the replication, you really have to be careful when you write sensitive data. In addition, you have to be aware of situations where a single user may make multiple requests in rapid succession, perhaps without even knowing it. One of my favorite web application patterns is the Redirect-After-Post pattern. Posting is annoying because when the user hits the back button in their browser, they get prompted to resend the data. However, if you redirect them after they post a form of data, their back button behaves nicer and their overall experienced is improved. However, this pattern results in a very quick update to the database in one request, and then a subsequent read within a second or two from the redirected request. Often, the next request will query for the data that you just wrote. In the case of a multi-master cluster, if the user posts to one server and is then redirected to a query for the same data on another server, that data may not have been written yet! This inconsistency could cause random instability in your site that’s hard to track down. So when you use a multi-master cluster, use sticky load balancing to ensure that a given user remains on the same server for the duration of their browser session. Sticky balancing is not the best performing solution, but I’ll always choose stability over performance.

Most web servers and hardware load balancers offer support for sticky load balancing. With Apache, the configuration is very simple. Recall the Apache cluster definition from our software load balancer solution...
earlier in this chapter. It looked like the following, but we've made some changes:

```bash
# cluster.conf
<Proxy balancer://mycluster>
   lbmethod=byrequests sticky_session=BALANCEID
   BalancerMember http://10.0.0.102:80 route=www1
   BalancerMember http://10.0.0.103:80 route=www2
</Proxy>
```

Did you spot the changes? In the first line I added `lbmethod` to tell the balancer to load balance every request, alternating the members between them. However, the `sticky_session` option overrides the default behavior if it finds a cookie called `BALANCEID` with a valid route value. If the value of `BALANCEID` matches any of the `route` values of any balancer members, then it will ensure to only balance among balancer members with the same value. It basically locks a user's browser session into a specific balancer member or group of balancer members (yes, multiple balancer members can have the same route value).

You're not quite done yet though. Your software load balancer itself does not set the cookie so you need to ask your web servers to do that for you. Luckily it's a one liner, an ugly line, but one line nevertheless.

```bash
# Partial mongrel_proxy.conf
# on 10.0.0.102
#...
RewriteEngine On
RewriteRule .* - [CO=BALANCEID:balancer.www1:.brainspl.at]
```

Writing the cookie makes use of the rewrite engine as you should recall from our earlier Apache discussions. The cookie value is `balancer.www1`, where the "www1" matches the route of the appropriate balancer member. Note that we set the domain explicitly to ".brainspl.at" so that the cookie can be read, written or overwritten from any server on our domain.

**Combining the Approaches**

There just isn’t any pleasing some people, so you may want both the redundancy and failover capabilities of a Multi-Master cluster, as well as the performance of read-only databases in Master/Slave configuration. This is entirely possible. Although, you will not only be inheriting...
the benefits of both, but also the challenges of both as well. Getting a configuration like this right will take time, patience and some documentation on your part. Don’t build something like this and expect everyone to understand it at a glance. So draw a picture of your environment and include some high level documentation.

I won’t go through the step-by-step configuration details all over again, because they’re the same as they were above. Essentially what you’ll do is build a Multi-Master cluster with sticky load balancing. But each balancer route will lead to an set of application and database servers that includes one master database server from the Master cluster, and a number of read-only slave servers dedicated to that Master server in particular. [[Author: A diagram of combined multi-master, master/slave would be helpful.]]

Don’t forget that in addition to the sticky load balancing, you will also have to use the acts_as_readonlyable plugin or a similar solution to handle the read-only databases. If you follow all of the rules, you shouldn’t get bitten by the asynchronous bug too hard or too often. If your environment is getting this complex, then you might want to seriously start consider an alternative to MySQL or perhaps do away with the relational database entirely.

7.9 Summary

If I haven’t exhausted the subject of scaling-out, then at the very least I must have exhausted you. In this chapter I’ve given you a number of tools to build convincingly high performance infrastructure for Ruby on Rails. You learned to configure A records and CNAMES for a cluster. Then, you learned to dabble in virtualized servers with VMWare and Parallels. You also learned to deploy to those virtual servers with Capistrano.

Next, you learned to build both Apache and ngnix servers to serve as load balancers, secured servers and static proxies. Apache demonstrated its flexibility and modularity as a Mongrel proxy, load balancer, and load balancer. For those interested in simplicity and performance, NGInx shone brightly. With some better documentation and a few more battle scars, NGInx looks like it could become a popular alternative to Apache, and possibly the new king of lightweight web servers.

Finally, I tackled one of the more daunting tasks of Rails deployments: clustering a MySQL database. Although there are more powerful databases
for clustering. MySQL ultimately handles the job effectively, if not grace-
fully. I’m happy to have MySQL despite its shortcomings, as it makes
up for them in spades in other areas, including community and Rails
support. And you can’t beat the price.

Before I move on, I should warn you that you can find books about most
of the topics in this chapter. If you’re a developer, I encourage you to
explore each one more deeply if you’re serious about deploying a mas-
vously scalable website. Stepping out of your developer shoes and deal-
ing with operational tasks will grow you as a person and a developer.
Or at the very least, you’ll be a little nicer to your System Administrator
due to your new appreciation for the role. And if you’re an admin, you
can better understand the foundations of your deployment. I’ve really
just scratched the surface.

In the next chapter, I’ll dive into some basic performance topics. You’ll
learn how to benchmark and profile systems for performance. You’ll
also see some solutions to common Rails bottlenecks such as caching
and eager loading.
Chapter 8

Deploying on Windows

If you asked for a show of hands of how many people deployed Rails on Windows at a Ruby conference you would get almost no one to admit to that crime. The truth is something very different. Microsoft has sold a bunch of copies of Windows to someone, and many companies and educational institutions simply don’t have access to other platforms for deployment. For many companies, bringing in a Linux-based server isn’t an option because politics or a lack of experience or the perceptions of management. If you are deploying small departmental applications in such a company, Windows may be your best bet.

This chapter explores a few strategies that you can use to get a Rails application deployed within a Windows server environment. I’ve used each one of these methods at various times to serve out applications to various audiences. I’ll cover using single instances of Mongrel, load balancing with Pen or Apache, and finally, a strategy to integrate Rails apps into an existing IIS web server using a special ISAPI filter and custom Rails plugin.

8.1 Setting up the server

In order to serve Ruby on Rails applications in our Windows environment, you need to do a few things to get your machine ready. You have to install Ruby, Gems, and Rails, as well as Subversion. I’ll also demonstrate how to get your Rails application talking to a Microsoft SQL Server.
**InstantRails, Apache and FastCGI**

InstantRails, a popular and easy way to get started with Rails development on Windows, won’t be discussed in this chapter because it’s really not meant to be a production deployment solution. Some people claim to have deployed applications with it to varying degrees of success, but the lack of load-balancing and inability to run as a service disqualify it as a good solution.

The setup of Apache and FastCGI will not be covered in this chapter. There are many issues with using this method on Windows including random server errors, poor performance, and really long startup times for Apache.

I’ll stick to the deployment options that will give you the best possible chance of success.

---

**Installing Ruby on Rails**

Getting Ruby, Gems, and Rails on a Windows server is extremely easy thanks to the work done by Curt Hibbs. His One Click Ruby Installer package makes the installation of Ruby and Gems painless. Just do the following:

1. Download the One Click Ruby Installer from RubyForge. Download version 1.8.6 or higher, as previous versions had issues with security and lacked proper debugging support.

2. Install the package by double-clicking the One Click Ruby Installer, and accept all of the default settings.

3. Open a Command Prompt and run the following command:

   ```
   C:/>gem install rails --include-dependencies
   Successfully installed rails-1.2.3
   Successfully installed activesupport-1.4.2
   Successfully installed activerecord-1.15.3
   Successfully installed actionpack-1.13.3
   Successfully installed actionmailer-1.3.3
   Successfully installed actionwebservice-1.2.3
   Installing ri documentation for activesupport-1.4.2...
   Installing ri documentation for activerecord-1.15.3...
   Installing ri documentation for actionpack-1.13.3...
   ```

---

Setting up the server

Installing ri documentation for actionmailer-1.3.3...
Installing ri documentation for actionwebservice-1.2.3...
Installing RDoc documentation for activesupport-1.4.2...
Installing RDoc documentation for activerecord-1.15.3...
Installing RDoc documentation for actionpack-1.13.3...
Installing RDoc documentation for actionmailer-1.3.3...
Installing RDoc documentation for actionwebservice-1.2.3...

The gem install rails command installs the latest version of Rails on your system. Next, I'll show you how to install Subversion.

Installing Subversion

You'll need to have the Subversion client tools installed on your machine in order to install the necessary Rails plugins. If you use Subversion to manage your projects, you can easily copy your applications to your production server. You can find a handy Windows installation of Subversion at Tigris.² Download the most recent Windows installer.

The Subversion installation alters your PATH environment variable to include the path of the Subversion executables. You don't have to restart your machine for these path changes to take affect, but you will need to close any open command windows.

Configuring Ruby on Rails to use Microsoft SQL Server

Microsoft SQL Server doesn't work with Rails without some tweaking. You will have to take a few minor steps to establish a successful connection. If you don't plan to use SQL Server with your Rails applications you can safely skip this section.

1. Download the latest stable version of Ruby-DBI.³ Look for a file with the name dbi-0.1.0.tar.gz or something similar. Extract this file to a temporary location like C:\TEMP.

2. Grab one file from that archive called ADO.rb. If you have extracted the files to C:\TEMP, you can find the file in the folder C:\TEMP\lib\dbd.

3. Copy this file to C:\ruby\lib\ruby\site_ruby\1.8\DBD\ADO. You will need to create the ADO folder, because it won't exist.

4. Please see the Rails Wiki\(^4\) for more information on using SQL Server with your Rails applications.\(^5\)

While Microsoft SQL Server is a common database for the Microsoft platform, it's not the only popular choice. You can also use several of the popular open source databases with Rails, including MySQL, SQLite, and Oracle.

**MySQL on Windows**

If you intend to use MySQL instead of SQL Server, you'll be happy to know you can do so easily. I'm going to assume you've already got MySQL installed and working. Rails has built-in support for MySQL, but to avoid potential problems such as speed and performance, you'll need to install the MySQL/Ruby for Windows adapter. You will see better performance when using this C-based library instead of the pure Ruby library. Until recently, installing MySQL/Ruby was a pain, but the gem now comes complete with a binary version for Windows. Installing it is as simple as opening a command prompt and installing the gem:

```
C:\>gem install mysql
```

```
Bulk updating Gem source index for: http://gems.rubyforge.org
Select which gem to install for your platform (i386-mswin32)
 1. mysql 2.7.3 (mswin32)
 2. mysql 2.7.1 (mswin32)
 3. mysql 2.7 (ruby)
 4. mysql 2.6 (ruby)
 5. Skip this gem
 6. Cancel installation
> 1
Successfully installed mysql-2.7.3-mswin32
Installing ri documentation for mysql-2.7.3-mswin32...
Installing RDoc documentation for mysql-2.7.3-mswin32...
While generating documentation for mysql-2.7.3-mswin32
... MESSAGE: Unhandled special: Special: type=17, text="<!-- $Id: README.html, v 1.20 2006-12-20 05:31:52 tommy Exp $ -->"
... RDOC args: --op c:/ruby/lib/ruby/gems/1.8/doc/mysql-2.7.3-mswin32/rdoc --exc lude ext --main README --quiet ext README docs/README.html
```

(continuing with the rest of the installation)

Be sure to select the highest-numbered version for Windows.\(^6\) That's really all there is to it. You have the prerequisites installed. It's time to

---

4. [http://wiki.rubyonrails.org/rails/pages/HowToConnectToMicrosoftSQLServer](http://wiki.rubyonrails.org/rails/pages/HowToConnectToMicrosoftSQLServer)

5. You can also use ODBC DSNs to connect to SQL Server from Rails, but there are some tricky permissions issues with ODBC and Mongrel as a service, so I don't typically recommend going that route.

6. Future versions of RubyGems will automatically install the correct version for you.
turn your attention to the server you’ll use to serve your Rails application. I’ll first show you Mongrel, and then a few tricks you can use to enhance the installation.

8.2 Mongrel

As with *nix platforms, if you’re running Ruby applications, Mongrel is usually the way to go. Not only is Mongrel relatively fast, but it’s also extremely easy to install and use. You can install the Windows version of Mongrel as a Windows service. There’s no mongrel_cluster for Windows yet, but don’t worry. I will show you how to manually build a cluster of Mongrels.

Installing Mongrel

Installing Mongrel on Windows is easy. Open a command prompt and install Mongrel with the `gem` command. Choose the highest numbered win32 option. The latest version is always at the top of the list so pay close attention to both the version number and the platform, like this:

```shell
C:\> gem install mongrel --include-dependencies
Select which gem to install for your platform (i386-mswin32)
1. mongrel 1.1.1 (ruby)
2. mongrel 1.1.1 (jruby)
3. mongrel 1.1.1 (mswin32)
4. mongrel 1.1 (mswin32)
5. mongrel 1.1 (ruby)
6. mongrel 1.1 (jruby)
7. Skip this gem
8. Cancel installation
> 3
Successfully installed mongrel-1.1.1-mswin32
Successfully installed gem_plugin-0.2.2
Successfully installed cgi_multipart_eof_fix-2.3
Installing ri documentation for mongrel-1.0.1-mswin32...
Installing ri documentation for gem_plugin-0.2.2...
Installing ri documentation for cgi_multipart_eof_fix-2.3...
Installing RDoc documentation for mongrel-1.0.1-mswin32...
Installing RDoc documentation for gem_plugin-0.2.2...
Installing RDoc documentation for cgi_multipart_eof_fix-2.3...
```

Next, install the Mongrel Service plugin. This plugin provides the necessary commands to get Mongrel installed and running as a Windows service. To do so, just run the following command:

---

7. RubyGems version 0.9.5 lets you skip the platform selection step
> gem install mongrel_service --include-dependencies

Select which gem to install for your platform (i386-mswin32)
1. mongrel_service 0.3.3 (mswin32)
2. mongrel_service 0.3.2 (mswin32)
3. mongrel_service 0.3.1 (mswin32)
4. mongrel_service 0.1 (ruby)
5. Skip this gem
6. Cancel installation

> 1

Select which gem to install for your platform (i386-mswin32)
1. win32-service 0.5.2 (ruby)
2. win32-service 0.5.2 (mswin32)
3. Skip this gem
4. Cancel installation

> 2

Successfully installed mongrel_service-0.3.3-mswin32
Successfully installed win32-service-0.5.2-mswin32
Installing ri documentation for mongrel_service-0.3.2-mswin32...
Installing ri documentation for win32-service-0.5.2-mswin32...
Installing RDoc documentation for mongrel_service-0.3.2-mswin32...
Installing RDoc documentation for win32-service-0.5.2-mswin32...

Watch closely for the gem for the win32-service. The win32 version is not always at the top of the list like it is for mongrel_service. Whenever installing gems, always take note of the version number and the platform to make sure you get the right version; the ruby ones won't install on Windows when you're using the One-Click Ruby Installer's Ruby interpreter.

**Test Mongrel**

Now that you've installed Mongrel, you should test it against your application to ensure that Mongrel can serve pages. I typically test Mongrel like this:

1. Create a folder on your hard drive called c:\web.

2. Open up the Command Prompt and navigate to c:\web.

3. Create a new Rails application in that folder:

   C:\web> rails mytestapp
   create
   create  app/controllers
   create  app/helpers
   create  app/models
   create  app/viewslayouts
   create  config/environments
   ...
   create  doc/README_FOR_APP
create log/server.log
create log/production.log
create log/development.log
create log/test.log

If you have a working Rails application that you would like to try, you should place that in a subfolder of c:\web and then reference that path in all future steps. Also, make sure that the database configuration for the production environment is correct before proceeding. Review config/database.yml to ensure that your production database is defined properly.

To test your application, execute the following command:

C:\web\mytestapp>mongrel_rails start -e production -p 4001
** Starting Mongrel listening at 0.0.0.0:4001
** Starting Rails with production environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready. INT => stop (no restart).
** Mongrel available at 0.0.0.0:4001

This command starts Mongrel on port 4001. Navigate to http://localhost:4001 and make sure that your application works before moving on.

If you don’t get a response, make sure port 4001 isn’t being blocked by the Windows Firewall or by your router.

Once you know your application works in production mode, stop the server with \(CTRL+C\).

Install Mongrel as a Windows Service

Now that you know your application works and that you have Mongrel installed correctly, you can install your application as a Windows service. You’ll install this application using production mode, so make sure your database.yml file points to a working production database if you’re using your own application with this tutorial.

1. Stop Mongrel by pressing \(CTRL+C\).

2. Execute the following command to install the application as a service:

C:\web\mytestapp>mongrel_rails service::install -N MyTestApp_4001 -p 4001 -e pro...*TRUNC*

** Copying native mongrel_service executable...
Mongrel service 'MyTestApp_4001' installed as 'MyTestApp_4001'.
This command creates a new Windows service with the name MyTestApp_4001, which you can view in the Control Panel Services applet as shown in Figure 8.1.

3. You can start the service from the Services applet or from the command line by executing the command: mongrel_rails service::start -N MyTestApp_4001

   • To stop the service from the command line, use mongrel_rails service::stop -N MyTestApp_4001.

   • Later, you can remove the service at any time using mongrel_rails service::delete -N MyTestApp_4001

Adding the port number to the service name is a really helpful way to keep track of the ports you’ve used. As you add additional services, the port number becomes even more useful. It’s not a requirement, but it’s a good convention to follow.

**Creating a second instance of Mongrel for your application**

Your application is now hosted by Mongrel, running as a service. Now, it’s time to kick up your feet and pop open a tall, frosty one, right? Be careful, though. One mongrel is not likely to be enough. Rails is not always the fastest available framework, and a single instance of Mongrel can only handle one request at a time. As load increases, you need to add more instances of Mongrel and then balance the requests.

Those lucky *nix guys get to use a tool called mongrel_cluster which can start and stop multiple instances of Mongrel with ease. Windows doesn’t have mongrel_cluster yet, so you’ll need to improvise. You can just just create another instance of Mongrel as a service that points
to the same Rails application. You can then use a load balancer to distribute traffic to each instance.

Creating your custom cluster manually is not as hard as it sounds. You'll create another service that points to the same application, but this time use port 4002 by running the following commands:

```
mongrel_rails service::install -N MyTestApp_4002 -p 4002 -e production
mongrel_rails service::start -N MyTestApp_4002
```

If you look in the Control Panel under Services, you will see both service running. Adding the port number to the service name makes it easier to remember what port each service uses. If you look at Figure 8.2, you'll see an example of both services installed.

Keep in mind that these services aren’t set to automatically start when you reboot the server, so your app isn’t going to be available after a restart. Let’s remedy that. Configure each service to start up automatically by right-clicking on each service name and setting the startup type to Automatic.

Test each address to make sure that the requests work and that the services are in fact serving your web application. http://localhost:4001 and http://localhost:4002 should both be serving the same application.

So, you have two instances of your application running on different ports. That’s better than one, but the setup is not that useful yet; You need to load-balance them, and you make that happen by using using PEN or Apache.
8.3 **Mongrel and Pen**

Pen is a nice, simple way to load balance an application without having to do too much setup. Pen is great for small sites without a huge number of expected connections, and it’s great because you don’t have to spend a lot of time learning how to configure it. Pen handles reverse-proxying and balancing very well, but it isn’t a web server, which means that you can’t serve static content with it. Your static content will be served by your back-end Mongrel services. For small apps with more than a couple of users, this setup should work just fine.

You can download a Windows binary of Pen from [ftp://siag.nu/pub/pen/pen-0.17.1.exe](ftp://siag.nu/pub/pen/pen-0.17.1.exe)

Pen supports SSL, but its support is very experimental so if you need to host a secure site you’ll have to use some sort of proxy server in front of this setup.

### Setting up Pen

Create a folder to store Pen. C:\pen will do just fine. Save the file you just downloaded in this folder and rename it to pen.exe so it’s easier to call.

In order to run Pen on Windows, you’ll need to download the file cygwin1.dll\(^8\) and place it in C:\pen. The file is compressed, so you will need to unzip it.

Once both files have been downloaded you can give Pen a quick test, but first make sure your Rails app is running with Mongrel as described

previously. Open a command window and navigate to C:\pen. Execute
the following command to start Pen:

\texttt{pen -f 80 localhost:4001}

This tells Pen to listen on port 80 and forward all requests to localhost:4001. If your Mongrel service is still running there, this command will make your application available on port 80.

Now open up \texttt{http://localhost} in your browser and you should see your Rails app. It’s just that simple. The -f switch keeps Pen from going into the background. If you forget this switch then you’ll have to kill Pen using the Windows Task Manager.

Use \texttt{CTRL+C} to stop Pen and return to the command prompt.

\section*{Load balancing with Pen}

Load balancing with Pen is as easy as adding each remote host and port to the command line. If you had two Mongrel instances running, one on port 4001 and the other on port 4002, you would use the following command:

\texttt{pen -f 80 localhost:4001 localhost:4002}

\section*{Installing Pen as a service}

If you decide that Pen is right for you, you should install it as a service so you can have it automatically start just like your Mongrel services do. There’s a relatively easy (and free) way to do that.

1. Download the Windows 2003 Server Resource Kit from Microsoft\textsuperscript{9} and install it.

2. Open a command prompt and run the following command:

\begin{verbatim}
"C:\Program Files\Windows Resource Kits\Tools\instsrv.exe" Pen
"C:\Program Files\Windows Resource Kits\Tools\srvany.exe"
\end{verbatim}

The service was successfully added!

Make sure that you go into the Control Panel and use the Services applet to change the Account Name and Password that this newly installed service will use for its Security Context.

\footnote{\url{http://www.microsoft.com/downloads/details.aspx?familyid=9d467a69-57ff-4ae7-96ee-b18c4790cffe}}
The commands in the following list will create a new registry entry containing the configuration for your new service.

3. Open regedit and locate the key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Pen.

4. Create a new key beneath that key called Parameters

5. Select the Parameters key and create a new String Value with the name Application. Enter c:\pen\pen.exe for the value


7. Create a third String Value called AppDirectory. Enter c:\pen for the value.

8. Close regedit and open a command prompt.

9. Start the service by typing the following command:

   C:\>net start pen
   The Pen service is starting.
   The Pen service was started successfully.

   You can stop the Pen service just as easily:

   C:\>net stop pen

   The Pen service was stopped successfully.

That’s it. This setup should work well for single applications that need to handle a lot of users. Though you see a lot of steps, they only take about 15 minutes from beginning to end. If you decide that this method is not for you, you can remove the service with the command sc delete pen.

Before moving on, you should know that a single instance of Pen is only going to work for one Rails application. If you are trying to serve multiple Rails applications with Pen, you’ll need to copy pen.exe to another file like my_app_name_pen.exe and then set up the service with a new name. You could end up with quite a few services if you’re serving lots of apps.

So Pen works great for small apps that need a bit of load-balancing help, but what do you do when you have to handle a lot more requests? You’re going to have to use what some Windows system administrators refer to as "the A word".
8.4 Using Apache 2.2 and Mongrel

If you need to handle load, there’s no better solution on Windows than Apache 2.2. Not only can you load-balance with relative ease, but you can also make Apache serve all of the static content like cached pages, CSS, JavaScript, and images. If you’re taking advantage of page caching (and you should if you’re doing a public site), then Apache is going to be your best bet for high performance.

Another great advantage of using this approach is that you can “borrow” a lot of configuration files from the *nix guys.

Install Apache

Download the Apache 2.2 Windows Binary from Apache. Be sure to grab the latest release.

Install Apache 2.2 using the installer you downloaded.

The wizard should be pretty easy to handle. I’ll just walk you through a few of the highlights. Most importantly, be careful when you pick a port. I usually choose to install Apache for a single user on port 8080 to prevent conflicts with IIS on port 80. I’ll install the Windows service manually later.

Install Apache to \apache, or some other directory you can easily find later. The Apache configuration should complete without any problems.

Configuring Apache to serve your Rails applications

Apache uses the file httpd.conf to hold all of the configuration settings. You know the drill. I’m not going to talk about them all, only the ones you will need to know for this Windows installation. You can always consult their excellent documentation to learn more. In fact, I’d recommend you familiarize yourself with the contents of that file before you roll out into production. Apache’s big, it has lots of options, and you really want to make sure you don’t have any gaping security holes.

Locate the httpd.conf file. It’s in the folder C:\apache\conf

First, locate the section of the file that starts with

# Dynamic Shared Object (DSO) Support
This section contains all of the modules that can be loaded by Apache. Each hash mark (#) means that the line is commented out. You need to uncomment the following lines to activate the proxy balancer:

- `LoadModule proxy_module modules/mod_proxy.so`
- `LoadModule proxy_balancer_module modules/mod_proxy_balancer.so`
- `LoadModule proxy_http_module modules/mod_proxy_http.so`

You also need to enable URL Rewriting support by uncommenting this line:

- `LoadModule rewrite_module modules/mod_rewrite.so`

Next, you’ll want to enable the deflate module to allow your content to be compressed as it is served:

- `LoadModule deflate_module modules/mod_deflate.so`

Finally, add this line to the bottom of the file:
Include conf/httpd-proxy.conf

This allows you to split up your configuration for your application into another file. At this point, you should have the httpd.conf configuration file.

Create a new file in C:\Apache\Apache2.2\conf called httpd-proxy.conf containing the following contents:

```xml
<VirtualHost *:8080>
  ServerName yourdomain.com
  DocumentRoot c:/web/mytestapp/public

  <Directory "c:/web/mytestapp/public">
    Options FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
  </Directory>

  # Configure mongrel instances

  <Proxy balancer://mongrel_cluster>
    BalancerMember http://127.0.0.1:4001
    BalancerMember http://127.0.0.1:4002
  </Proxy>

  RewriteEngine On

  # Uncomment for rewrite debugging
  #RewriteLog logs/your_app_deflate_log deflate
  #RewriteLogLevel 9

  # Check for maintenance file and redirect all requests
  RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
  RewriteCond %{SCRIPT_FILENAME} !maintenance.html
  RewriteRule ^.*$ /system/maintenance.html [L]

  # Rewrite index to check for static
  RewriteRule ^/$ /index.html [QSA]

  # Rewrite to check for Rails cached page
  RewriteRule ^\([^./]+\)$ $1.html [QSA]

  # Redirect all non-static requests to cluster
  RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ balancer://mongrel_cluster%{REQUEST_URI} [P,QSA,L]

  # Deflate
  AddOutputFilterByType DEFLATE text/html text/plain text/xml
  BrowserMatch ^Mozilla/4 gzip-only-text/html
```

Report erratum
this copy is (82.0 printing, December 2007)
BrowserMatch ^Mozilla/4\.[0-9][0-9]* no-gzip
BrowserMatch \bMSIE !no-gzip !gzip-only-text/html

# Uncomment for deflate debugging
#DeflateFilterNote Input input_info
#DeflateFilterNote Output output_info
#DeflateFilterNote Ratio ratio_info
#LogFormat "%r" %{output_info}n/%{input_info}n (%{ratio_info}n%%) deflate
#CustomLog logs/your_app_deflate_log deflate
ErrorLog logs/your_app_error_log
CustomLog logs/your_access_log combined
</VirtualHost>

Save the file and double-check to make sure it’s in the same folder as httpd.conf.

**Explaining the proxy**

The important part of the file is this section:

```
# Configure mongrel_cluster
<Proxy balancer://mongrel_cluster>
    BalancerMember http://127.0.0.1:4001
    BalancerMember http://127.0.0.1:4002
</Proxy>
```

This section is the load balancer configuration. Each `BalancerMember` points to one of your backend instances of Mongrel. This configuration supports only two backends but you can easily add more. Keep in mind that changing this configuration file requires a restart of Apache.

When a request comes in, Apache checks for a static page. If no static page is found, the system will forward the request to the Rails application, just like a stand.

**Test Apache’s configuration**

Open a command prompt and navigate to `c:\apache\bin` and execute the command

```
httpd
```

If you receive no errors, Apache is running and listening on Port 8080. If you have Mongrel instances listening on ports 4001 and 4002 then you can test the configuration by pointing your browser to `http://localhost:8080/`. You should see your Rails application.

This section showed you how to host a Rails application using Apache on port 8080. If you wanted to host the application using the standard port 80, you would simply need to change the virtual host definition.
Installing Apache as a Windows Service

Now that Apache has been correctly configured and tested, you can safely install the service.

Open a new command prompt and enter the command

```
cd\apache\bin
httpd -k install
```

You should see the following output

```
Installing the Apache2 service
The Apache2 service is successfully installed.
Testing httpd.conf....
Errors reported here must be corrected before the service can be started.
```

You should now see the service in your Services panel. Ensure that the startup type is set to "Automatic" so it will restart when you restart your server.

You might see a Windows Firewall prompt. You’ll need to unblock Apache or disable your Windows Firewall service for things to work properly. Apache is now configured to load balance between two back-end Mongrel processes. You can test this configuration by opening a browser and navigating to `http://localhost:8080/`. Your Rails application will appear.

Now you can go a step further and hide your Apache server behind IIS. Taking this step may seem strange to you, but read on.

8.5 IIS Integration

IIS is a popular web server in Windows-based organizations. Despite its reputation it can be a very good static web server. In this section,
What about just serving Rails through IIS?

Different groups of people have tried to serve Rails directly through IIS. Some tried to use the FastCGI ISAPI filter, but that configuration requires registry hacks and leads to instability. When I tried the FastCGI approach on three machines, I was only able to make it work successfully on one, and it wasn’t very reliable. This option may become more viable soon, but for now, most Windows developers are going with the approaches outlined in this chapter.

Microsoft has actually taken some great steps to make FastCGI better in IIS 7.0, but they have made no specific commitment to serving Rails applications with IIS at the time this book is being written. This stuff moves pretty fast though, so it’s definitely something you should keep an eye on.

Using a proxied approach does provide for better long-term scalability, as it is much easier to move your Rails application to one or more separate physical machines. If you plan to host several Rails applications from one server, FastCGI is probably not a good option for you at all.

I’ll show you how to use IIS to forward requests to your Rails applications. This configuration will allow you to seamlessly integrate a Rails application into an existing IIS web site.

Using IIS has a couple of benefits. First of all, you can use the same SSL certificate for all of your Rails applications. Secondly, you have the flexibility to move the backend applications to another server at any time. If you find that Windows isn’t going to work for you as a deployment method, you can easily move your Rails applications to a Linux-based server and still have requests come through your main web server.

Before you can begin, you will need to get an additional piece of software. I am also going to assume that you will be performing all of this on a server where IIS is running on the default port (80) and that your Rails applications reside on the same machine.
Install ISAPI Rewrite

ISAPI Rewrite is a URL rewriting filter that provides some simple forward proxy support. Though it is not free, it is well worth the nominal fee its developers charge and you have access to an unrestricted trial version which will get you through this chapter.

Visit the ISAPI Rewrite site\(^{11}\) and download the trial version of ISAPI Rewrite 3.0. Launch the installation program and accept all of the default settings. The installation will restart your IIS service, as it needs to install an ISAPI filter on your server.

If you experience trouble with the installation, you'll need to refer to the developers of this product. The support forum \(^{12}\) is an excellent resource.

Forwarding requests to your application.

Say you want to forward all requests from \texttt{http://localhost/mytest/} to your Rails application. You need to ensure that one of the following is true:

- You allow script execution from our site root
- You allow script execution from the folder or virtual directory \texttt{mytest}.

Failing to allow script execution from one of those places will result in a \texttt{403.1} error message from IIS.

\texttt{C:\Program Files\Helicon\ISAPI_Rewrite3/httpd.conf} contains the rewrite rules that IIS uses to forward requests.

Forwarding a requested URL to a back-end server is really easy. To forward requests to \texttt{/mytest} to a Mongrel instance on the same machine running on port 4001, you use this rule:

\texttt{RewriteProxy /mytest(.*) http://localhost:4001$1 [I,U]}

If you're using Apache on port 8080, you just forward requests to that port instead.

\texttt{RewriteProxy /mytest(.*) http://localhost:8080$1 [I,U]}

You can even go to a different server.

\texttt{RewriteProxy /mytest(.*) http://backend.mydomain.com:4001$1 [I,U]}

\(^{11}\) \texttt{http://www.isapirewrite.com/}
\(^{12}\) \texttt{http://www.helicontech.com/forum/}
On some systems, especially those that have tightened security, this file is marked as read-only. You'll need to remove the read-only attribute before you can change the file. Also, ensure that the SYSTEM user can read that file.

**Testing it out**

Configure the filter to forward requests to your Mongrel instance on port 4001.

```
RewriteProxy /mytest(.*).http://localhost:4001$1 [I,U]
```

You can now pull up your Rails application via IIS by navigating to http://localhost/mytest/. Unfortunately, it’s not going to look very good. Read on to find out why.

### 8.6 Reverse Proxy and URLs

The big problem we’re faced with now is that the URLs that Rails creates internally, such as stylesheet links, url_for links and other links don’t work as you might expect.

For example, if you pull up the URL http://localhost/mytest/ in your browser, you should see that the application comes up just fine, but without the stylesheets. You will also notice that when you click a link, you’re transferred to http://localhost/ and in some cases your proxy will be exposed. This situation could be especially bad for your users if your application server happens to be behind a firewall that can’t be accessed from the Internet.

Neither IIS nor ISAPI_Rewrite has a method to handle reverse proxying. A reverse proxy rewrites the content served from the backend to mask the fact that the request was filtered through a proxy.

I developed a simple Rails plugin that modifies the way Rails creates URLs in order to address this issue. The plugin tells Rails to prepend our external URL to any URLs it creates through the system. This plugin will force all user requests to come back through the IIS proxy. The URLs are only altered when you run the application in production mode, so you don’t have to worry about changing routes or configuration files when you deploy your application. It’s also safe to keep the plugin with your application during development.
A note about relative_url_root

At first glance, it looks like most problems with the URLs could be solved simply by applying the following code to the environment.rb file:

```
ActionController::AbstractRequest.relative_url_root = '/mytest'
```

That change fixes most of the issues but it doesn’t fix any links written using `url_for :only_path => false`. The reverse_proxy_fix plugin that I wrote addresses these issues as well.

Installing the proxy plugin

Execute the command (but all on one line):

```
ruby script/plugin install http://svn.napcsweb.com/public/reverse_proxy_fix
```

from within your application’s root folder. Once the plugin has installed, it asks you for the base url. Enter `http://localhost/mytest` and press ‘enter’. If all goes well, the plugin will write the configuration file. If the configuration file can’t be modified, you can configure it yourself by editing the file `vendor/plugins/reverse_proxy_fix/lib/config.rb`.

Using the proxy plugin

Once you’ve installed the plugin, you’ll need to restart your Rails application. If you’re using multiple instances of Mongrel, you’ll need to restart all instances before the plugin will work. Once the applications restart, any internal links in your application will now be automatically corrected and your users will be routed back through the proxy.  

8.7 Strategies for hosting multiple applications

There are several strategies that you can use to host several applications and the one you choose depends mostly on the number of users who will use your system. When you have many users and long HTTP requests such as file uploads, you will have to address scaling through adding more backend processes. That was traditionally done

---

13. This assumes you used link_to and friends to generate your links and images. Hard-coded paths are not changed by this plugin.
by increasing the number of FastCGI processes, but now you can just add another instance of Mongrel to our cluster.

The next few sections will cover various strategies you can use to deploy several applications into production.

**Serve several small applications using IIS and Mongrel**

Serving many small applications is a simple approach. Each application is installed as a Windows service using Mongrel running on a different port. You can then use IIS with ISAPI_Rewrite and Reverse_Proxy_Fix plugin to mount each application to its own URL within IIS.


The ISAPI_Rewrite rules for this are simply

```
RewriteProxy /app1(.*) http://localhost:4001$1 [I,U]
RewriteProxy /app2(.*) http://localhost:4002$1 [I,U]
RewriteProxy /app3(.*) http://localhost:4003$1 [I,U]
```

You would then need to apply the Reverse_Proxy_Fix plugin to each of your applications, setting the BASE_URL parameter for each originating URL.

I don’t recommend doing this for production. There’s no support for page caching here, and there’s just no way to scale up. However, this is a really great approach to demo a site to your stakeholders quickly without going through a lot of complex setup.

**Serving several large applications**

There are several possibilities for serving large applications.

One method would be to use Pen to cluster several Mongrel instances and then use IIS to forward requests to Pen.

This configuration is exactly the same as if you were going directly to Mongrel. You would install multiple copies of Pen on different ports, each forwarding to their own group of Mongrel instances. You would then set up IIS to forward requests to each instance of Pen.

This is one of those solutions that works well for those cases where your organization has a “no Apache” policy.
Applications, Users, and Requests

How do you measure the size of an application, and how do you choose the method of deployment? People tend to think about application size by thinking about how many users the app will have. There’s a slight problem with that though.

I could have an application with hundreds of models. The application could be very complex, but if there are only 100 people using the application, it’s not going to be that problematic to just throw up one instance of Mongrel and let it do the work, provided that there aren’t any simultaneous requests.

I could also have an application with 5 models and 3 controllers, and this application gets hit 100,000 times a day by students who are registering for a summer orientation session at a university. A single instance of Mongrel would probably work, but there would be a lot of waiting going on.

The number of users an application can support is really not a good measure though. With AJAX becoming more and more popular, and Rails’ support of REST, you may see more hits to your application than you expect, whether it be from a user’s browser or another web service.

“Requests per Second” is a much better measure for your site. How many requests does your app need to support per second? 3? 6? 20? 100? It’s something you need to figure out by benchmarking existing applications and doing some forecasting. An application that supports 5 requests per second can serve 144,000 requests in an 8 hour period. That’s not too bad. The problem is that a Rails application is single-threaded. A single instance of Mongrel can only serve one request at a time. So, if you have an AJAX-based search on your site, the live updating that the search does can cause other requests to get stuck in a queue.

So, test your apps and determine the requests per second. If you determine that your small internal application can run on a single instance of Mongrel, that makes life easier. You can easily scale up using the techniques in this book.

Keep in mind that on Windows, your app will typically perform much slower than on another platform, so you may require more balanced backends to process the same number of requests.
This approach won’t be the best approach if your application makes extensive use of page caching, but it is easy to implement and can work fine for systems where every user needs to be authenticated on every request, making page caching a non-issue.

The most performant method is to simply use Apache on port 80. Using the proxy_balancer method, Apache can be configured for multiple virtual hosts with each virtual host serving a separate cluster of Mongrel instances.

Implementing this approach is a matter of creating separate groups of Mongrel instances and then creating a virtual host entry for each of these groups in the httpd-proxy.conf file created earlier. Your DNS and
Figure 8.7: IIS forwarding requests to multiple instances of Pen
Figure 8.8: Apache 2.2 with mod_proxy_balancer on multiple VHosts
local HOSTS file would then need to be configured for each virtual host. This approach yields good performance, scales well, and allows page caching to work effectively. It’s the fastest and most stable solution right now for Windows.

Finally, if you want transparent integration, you could make IIS send these requests to your various Apache virtual hosts. This is a more complicated approach with more points of failure, but it will allow you to use your IIS SSL certificates, it will allow you to place your database and Rails applications behind a firewall, and will make your applications appear to be integrated. Remember to make use of the reverse_proxy_fix plugin if you choose to use IIS to forward your requests.

**Performance on Windows**

Ruby does not perform as well on Windows as it does on Linux. Applications that routinely handle 60 req/second on a Linux box with one instance of Mongrel can handle only 6 to 9 requests per second on Windows. Any more than that and your users start seeing delays as each request is processed.

If you have a really powerful server, like the fastest thing available with lots of RAM and very little running then you might see around 35 requests per second with a single instance. Linux servers tend to provide much greater throughput with much less expensive hardware.

You can improve performance slightly by looking over the applications you plan to deploy and checking the following areas:

- Change how you use sessions. How are sessions managed in your application? The P-Store, or file-based store, can often be slow. Consider moving your session store into your database or investigate other session storing mechanisms.

- Go through your development logs and make sure you’re not making unnecessary calls to your database. Simply adding an :include to a finder can really help out an application’s performance and it is often missed.

- Make use of fragment, action, and page caching as much as you can. Since Ruby is slow on Windows, you want to make as much use of page caching as you possibly can so that Rails is never invoked.
• Ensure that nothing is interfering with the process. Certain security auditing software, quota managers, and virus scanners can drastically reduce the amount of requests you can handle. Watch your performance monitor for any spikes when testing your application.

8.8 Load-testing your applications.

There are few good choices for load testing your applications on Windows. If at all possible, get a Linux machine or a Mac and use httperf to test your application. If that’s just not going to work for you, try one of these alternatives.

- Microsoft Web Application Stress Tool

This tool, available online, is a free tool that lets you record your steps through a web application or web site and then play them back and generate loads. It can be used to control remote clients as well so you can generate more realistic loads against your application. It has quite a few bugs but it is still a very useful program. I have found its accuracy to be relatively good.

- WAPT

WAPT is a commercial tool that performs load and stress testing of web applications. Like the Microsoft offering, WAPT allows you to record your browsing session so it can be played back. Interpreting the reports can be trickier, but it’s worth it because WAPT has the ability to connect to secure sites.

Apache Benchmark (ab)

Apache Benchmark is a command-line tool that can be used to hit a URL repeatedly, but it is known to produce extremely misleading results. I only use this tool to generate loads against an application.

8.9 Final Thoughts

I believe that Windows is a good platform for developing Rails applications and does an adequate job of serving Rails applications that have

Testing tips

- Run your tests from a different machine than the one hosting the application. Testing the throughput on the same machine can lead to unrealistic and inaccurate results.

- Run the stress test against a baseline such as your public home page. This will give you a good idea of how your application compares to your existing services.

- Run the tests from inside and outside of your network. If you can, try running the test from a cable or DSL connection to see what kind of an impact that has.

- Stress test your app on the production server. Some sysadmins may cringe at this, but you should test the application where you plan to deploy it whenever possible. Do some testing during some scheduled downtime or during off-peak times when your server use is low. If you can’t do this, you should at least consider having a staging server that mirrors your production machine so you will be able to see accurate results and plan for the future.

- Use more than one tool and compare the results.

a moderate user base. However, as you begin to develop more applications and gain more users, your needs may change.

I also believe that Linux is the better choice right now. I have a 1Ghz desktop machine running Ubuntu and serving a single Rails application from one instance of mongrel that serves three times as many requests per second than a server with two Xeons at 2ghz.

In recent months, Windows machines are getting much faster, but they’re not quite as performant. If your applications aren’t performing at an acceptable level and you’ve done everything you can to optimize them then you should consider deploying some applications to a Linux test server. The information and strategies I have shared with you can be used to help you migrate some or all of your Rails applications to Linux servers transparently. You could still use IIS or Apache on Windows and move only the Rails applications to Linux, which is a great transitional
solution that I’ve employed a number of times with great success.

### 8.10 Developing on Windows and deploying somewhere else

If you develop on Windows but plan to deploy on a Linux server such as a virtual private host, a shared host, or a brand new shiny Linux server that your wonderful bosses purchased especially for you, you should be aware of some issues.

**Dispatch.fcgi Ruby interpreter**

When you deploy an application that you created on Windows to a Linux server, the very first line in your dispatch.fcgi file will be wrong.

On Windows, it usually reads:

```
#!/c:/ruby/bin/ruby
```

On Linux, it often reads:

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

You can figure out what path you should use by connecting to your remote host and typing `which ruby`.

This is a problem if your hosting platform is using FastCGI because the FastCGI server will be unable to locate the Ruby interpreter. If you’re using Mongrel then it’s not really a concern.

**Line breaks**

Linux uses different line breaks than Windows. This can sometimes be a problem because the extra character that Windows uses can interfere with how scripts are processed. Many Linux distributions have a program called dos2unix that you can use to convert the line breaks in your files.

The best solution is to find yourself a good editor for Windows that allows you to specify what type of line breaks are used. Eclipse, NetBeans IDE, Notepad++, and Crimson Editor are just a few examples of editors that are known to work correctly. Windows Notepad should be avoided at all costs, as well as Wordpad, as they are meant for Windows-formatted text files.
Permissions
When you deploy your application to a Linux server, you need to check permissions. If your server uses FastCGI, then you need to make sure that you allow the web server’s user and group the right to execute public/dispatch.fcgi or your application isn’t going to work. If your application uses page caching, ensure that the public/ folder is writable by the web server’s user, or Rails will be unable to write the static versions of the pages.

Preventing problems when deploying
Follow these tips to make deploying an application to production from Windows to Linux:

• Create the application on the Linux machine using the rails command. This will put all the files in the right locations and make sure that the paths are correct.

• If your production server or web host uses Apache and FastCGI for Rails application hosting, be sure to modify your copy of public/.htaccess on your server so that dispatch.fcgi is called instead of dispatch.cgi.

• Deploy your application files over the top of the ones you created, making sure to ignore overriding the dispatch.fcgi or .htaccess files. This can easily be scripted if you use Subversion.

• Edit your database configuration file on the production server and then make sure you never overwrite it when you redeploy. I don’t think it’s a good idea to store your database passwords in a code repository, so I never check the database.yml file in to the repository.

• Run your migrations in Production mode to ensure that your database is configured.

• Open the console in production mode (.script/console production) and attempt to retrieve some data. This will help test to see if you have any odd characters in your code that need to be converted.

• Test your application on the production server using WEBBrick in Production mode. (.script/server -e production). If your host allows you to connect on port 3000, try pulling up your site using that port.
• Configure your production server to use your new application. Some providers like Dreamhost have a control panel where you specify the public folder of your Rails app. Once your app is configured, try hitting it with the browser one more time to make sure it comes up.

• Check to see if the production log is being used. If it’s not, you’ll need to modify your environment.rb file to force production mode. Some shared hosts have been known not to set the environment in their Apache configuration.

A better approach than this is to automate your deployment using Capistrano. You can safely make all these alterations to your files at anytime and just check them in to your repository. You can then just deploy using a Capistrano recipe. Capistrano tasks can change permissions, alter files, and more. If you’re going through all the trouble to write a program, take a little more time to learn how to automate the deployment. You’re less likely to forget something later.

8.11 Wrapping up

This chapter talked about various strategies you can use to deploy your application. You have a lot of choices to make now, because each method will yield different results. Some might be better than others, but you need to figure out which will work for you and your situation. I can’t stress enough the importance of testing your stack. Run performance testing tools against your application before you deploy it, and keep a close eye on it when it’s running. You want to make sure you are ready to move to a better deployment solution before you need it.

Don’t be afraid to deploy on Windows though—many people, including myself, have been very successful deploying applications with these methods. It’s a great way to get Rails into a Windows-based environment. Once you prove you can be more efficient with Rails, you can push for a Linux deployment stack!
Appendix A

An Example ngnx Configuration

In Chapter 7, *Scaling Out*, on page 141, I based Apache and ngnx configurations on existing configurations. I used the base configurations that came with Apache as a foundation for that web server, but ngnx has no consensus base configuration for Rails. The following configuration, complete with comments, serves as the foundation for the ngnx configurations in this book.

```bash
# user and group to run as
user  ezra ezra;

# Nginx uses a master -> worker configuration.
# number of nginx workers, 4 is a good minimum default
# when you have multiple CPU cores I have found 2-4 workers
# per core to be a sane default.
worker_processes  4;

# pid of nginx master process
pid /var/run/nginx.pid;

# Number of worker connections. 8192 is a good default
# Nginx can use epoll on linux or kqueue on bsd systems
events {
    worker_connections  8192;
    use epoll;  # linux only!
}

# start the http module where we config http access.
http {
    # pull in mime-types. You can break out your config
    # into as many include's as you want to make it cleaner
    include /etc/nginx/mime.types;
}
```

Download nginx/nginx.conf
APPENDIX A. AN EXAMPLE NGNIX CONFIGURATION

# set a default type for the rare situation that
# nothing matches from the mimie-type include
default_type application/octet-stream;

# This log format is compatible with any tool like awstats
# that can parse standard apache logs.
log_format main '
    $remote_addr - $remote_user ["$time_local"]
    "$request" $status $body_bytes_sent "$http_referer"
    "$http_user_agent" "$http_x_forwarded_for"';

# main access log
access_log /var/log/nginx/access.log main;

# main error log - Do not comment out. If you do not want the log file set thi...
error_log /var/log/nginx/error.log notice;

# no sendfile on OSX
sendfile on;

# These are good default values.
tcp_nopush on;
tcp_nodelay on;

gzip on;
gzip_http_version 1.0;
gzip_comp_level 2;
gzip_proxied any;
gzip_types text/plain text/html text/css application/x-javascript
    text/xml application/xml application/xml+rss text/javascript;

# this will include any vhost files we place in /etc/nginx/vhosts as
# long as the filename ends in .conf
include /etc/nginx/vhosts/*.conf;

}
Index
First page of blurb
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

Deploying Rails Applications’ Home Page
http://pragprog.com/titles/fr_deploy
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 in the news.

Buy the Book

If you liked this PDF, perhaps you’d like to have a paper copy of the book. It’s available for purchase at our store: pragprog.com/titles/fr_deploy.

Contact Us

Phone Orders: 1-800-699-PROG (+1 919 847 3884)
Online Orders: www.pragprog.com/catalog
Customer Service: orders@pragprog.com
Non-English Versions: translations@pragprog.com
Pragmatic Teaching: academic@pragprog.com
Author Proposals: proposals@pragprog.com