Extracted from:

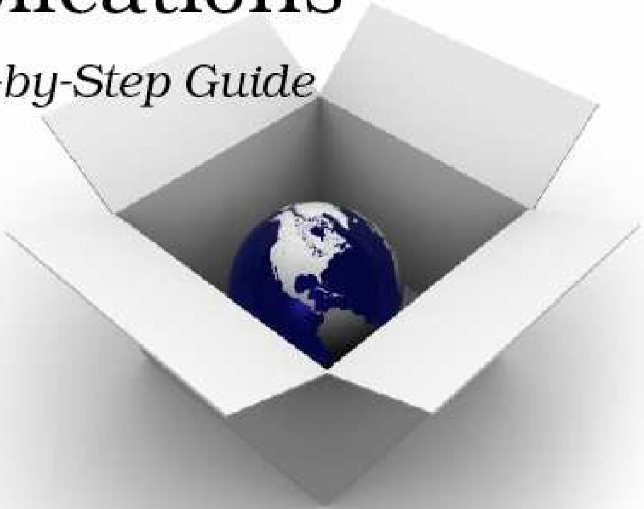
Deploying Rails Applications

A Step by Step Guide

The Pragmatic Programmers

Deploying Rails Applications

A Step-by-Step Guide



Ezra Zygmuntowicz,
Bruce Tate, and Clinton Begin
with Geoffrey Grosenbach and Brian Hogan

The Facets of Ruby Series

Printed in the United States of America.

ISBN-10: 0-9787392-0-5

ISBN-13: 978-09787392-0-1

Printed on acid-free paper with 50% recycled, 15% post-consumer content.

P1.0 printing, April 2008

Version: 2009-6-3

✓ Getting a Watchdog

Chapter 6

Managing Your Mongrels

By now, you've located a good home and moved in. If you've chosen to manage your own deployment and followed the steps in this book, you have a single Mongrel running your application. Things will start happening very quickly now. The next step is to make sure your house is running smoothly and that it is safe. Part of that job will be clustering and configuring Mongrel. Next, you'll want to get a watchdog to help keep an eye on things. In this chapter, you'll learn Mongrel configuration, clustering, and monitoring.

6.1 The Lay of the Land

Clustering Mongrel is the first step to achieving better scalability with Ruby on Rails. You'll find the process amazingly easy to do. First, you'll build a customized configuration file that will let you predictably and reliably restart Mongrel with an automated script. Then, you'll use a Mongrel cluster to launch more than one Mongrel so that your installation can share many simultaneous requests.

After you have a working cluster, you will place that cluster under a monitoring process called Monit. This watchdog process will take action when rogue Mongrel processes take up too much memory, stop responding, or misbehave in other ways. The Mongrel cluster under management from Monit is shown in Figure 6.1, on page 128.

6.2 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 can provide. You will want to cluster your Mongrels and probably run them as a service. Fortunately, configuring Mongrel and even enabling Mongrel clusters is surprisingly easy. As you recall, to start Mongrel, you want to run the following commands:

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

That command 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 and mongrel_rails stop to stop. But these commands simply take your dog for a walk. You are 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 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 a 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]

1. http://mongrel.rubyforge.org/docs/howto.html
2. http://mongrel.rubyforge.org/docs/

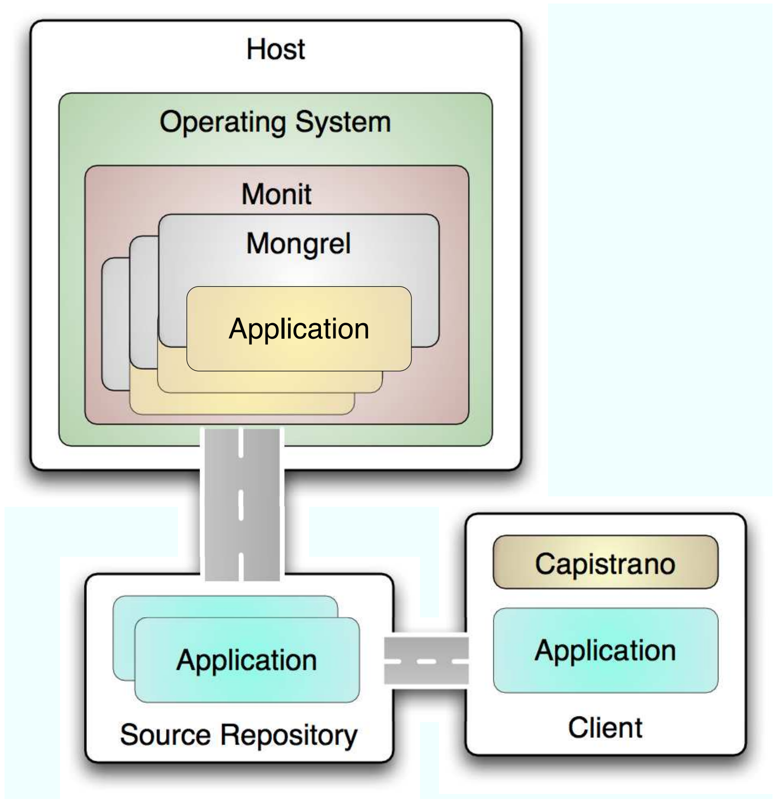


Figure 6.1: Deployment map for scaling out

You can specify all these 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 a 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 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 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 command is a convenience because Ctrl+C generates an interrupt signal and Ctrl+C is used in debug mode.
HUP	Internal reload. This command might not work well because sometimes doing an internal reload will not reload all the code in the system. 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. Your next step is to build a more flexible configuration for a cluster. First, you need to generate your mongrel_cluster.yml file. Let's configure a cluster of three Mongrels by running 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
---
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 hostname 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 three

Mongrels running on ports 8000, 8001, and 8002. Next, customize this config file a bit to take advantage of a few more attributes:

```
 ---
port: "8080"
cwd: /Users/ezra/railsapp
log_file: log/mongrel.log
environment: production
address: 127.0.0.1
pid_file: log/mongrel.pid
servers: 3
docroot: public
user: ezra
group: ezra
```

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. 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

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 MySQL. The service configuration keeps things consistent. The operating system will include Mongrel when automatically starting services each time your server starts or restarts. The service configuration works much like the Mongrel configuration you've already built.

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_cluster/myapp.conf. I recommend you replace myapp with the name of your application, but you can use anything you like. 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 following example configuration file:

```
# /etc/mongrel_cluster/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,
# put 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.

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.

Starting Mongrel Cluster on Boot

You can get your Mongrel cluster to start at boot time, and it should be fairly simple with most Linux distributions. The Mongrel cluster comes with a script ready to go. Installing it is simply a matter of finding it and copying it to the /etc/init.d/ directory. On my setup, the mongrel_cluster script file is located at the following location: /usr/lib/ruby/gems/1.8/gems/mongrel_cluster-<VERSION>/resources/mongrel_cluster.

Simply copy it to /etc/init.d/, and make it executable like this:

```
ezra$ sudo cp \
  /usr/lib/ruby/gems/1.8/gems/mongrel_cluster-1.0.5/resources/\
  mongrel_cluster \
  /etc/init.d
ezra$ sudo chmod +x /etc/init.d/mongrel_cluster
```

Now your Mongrel cluster is configured to load on boot, just like Apache and MySQL. As an added bonus, you can now also use /etc/init.d/mongrel_cluster [start|restart|stop] anywhere you read mongrel_cluster_ctl [start|restart|stop]. This is nice because it's very familiar to anyone who has used other service scripts like those for Apache and MySQL.

You might need to make a few changes to the PATH variable inside the script depending on your specific setup, Linux distribution, and hosting provider's custom configuration. Check with your host provider or the documentation for your Linux distribution in case yours is a little different.

6.3 Configuring the Watchdog

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 restart 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 will need to install the right version of mongrel_cluster. The minimum version of Mongrel you will want to run is 1.0.1.1.

Building Monit on RHEL or CentOS

You need to install a few dependencies 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.

Earlier versions do not support the --clean option. This is important because Mongrel 1.0+ will not start if there is a process identification (PID) 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 leftover 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, because you can build Monit from source, like this:

```
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
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:

```
check process mongrel_deployit_5000
  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 previous configuration is for one Mongrel only. The first directive, check_process, identifies a process to monitor. I have 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 Mongrel. The last three directives tell Monit what to do when certain pathological conditions exist. This configuration will restart Mongrel instances if the memory exceeds a threshold (110.0MB in the previous 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 this is fully automated and requires notification only in extreme circumstances.

Keep in mind that Monit will start your Mongrels with a completely clean shell environment. This means your normal $PATH will not be set up. You will need to use the fully qualified path to your mongrel_rails command. In the previous config I used /usr/bin/mongrel_rails, but you may need to adjust this path depending on where your system installed the command. You can figure out where the command was installed like this:

```
ezra$ which mongrel_rails
      /usr/bin/mongrel_rails
```

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

```
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 viewable only from the localhost, and sets /etc/monit.d to be the directory from which to include config files.

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:

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

When Monit starts, it will automatically boot your Mongrels. Then you can restart the Mongrels by their groups 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, 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 may be asking yourself "Who monitors Monit?" That is a great question. Monit is usually very stable, but certain conditions such as "out of memory" can cause Monit itself to crash. If you want to prevent this from happening, you can put Monit under the control of init. On a Linux system, init is responsible for running all the scripts in /etc/init.d. init can also respawn daemons if they die. The first step is to remove Monit from the /etc/init.d scripts. Consult the documentation for your system for information on how to remove a start-up script from the default run level. On Gentoo, you would do it by running rc-update del monit. The next step is to edit /etc/inittab and add the following lines near the bottom of the file:

```
mo:345:respawn:/usr/bin/monit -Ic /etc/monitrc
m0:06:wait:/usr/bin/monit -Ic /etc/monitrc stop all
```

Now you can have init to watch Monit. The first step is to stop Monit. Then you tell init to spawn Monit and keep it alive:

```
ezra$ sudo /etc/init.d/monit stop
ezra$ sudo telinit q
```

Now that Monit runs under init, the /etc/init.d/monit command will not work to start and stop the Monit daemon. Instead, you will have to kill Monit and let init pick it back up again, like this:

```
ezra$ sudo killall -9 monit
```

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 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 file:

```
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
  sudo "/usr/bin/monit restart all -g #{monit_group}"
end

desc <<-DESC
Stop the Mongrel processes on the app server.
DESC
task :stop_mongrel_cluster , :roles => :app do
  sudo "/usr/bin/monit stop all -g #{monit_group}"
end
```

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.

6.4 Keeping FastCGI Under Control

Our primary focus has been on Mongrel. I'm going to dedicate the rest of the chapter to FastCGI. If you should find yourself deploying with FastCGI, you'll want to read the next few sections. Otherwise, feel free to skip ahead to Section 6.5, *Building in Error Notification*, on page 140.

Zombie FastCGI Processes

During the dog days of summer in 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 *The Reaper* below.

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: Reloads only the application, not the framework (like the development environment). Reload sends the HUP signal.
- graceful: Marks all the processes for exit after the next request. Graceful sends the TERM signal.
- kill: Forcefully exits 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 is successful.

You can run the reaper without any arguments or request one of the previous actions such as the following:

```
ezra$ ./script/process/reaper --action=graceful
```

In my experience, the defaults don't 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

Couldn't find any process matching:
/data/deployit/releases/20060224192655/public/dispatch.fcgi
```

Reading through the reaper code revealed the exact command that the reaper used to find the list of running processes. I called that command manually:

```
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
```

6.5 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 are 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 is exactly what the exception_notification plug-in does.

You can read about the exception_notification plug-in at the Rails wiki (http://wiki.rubyonrails.org/rails/pages/ExceptionNotification). To install it, simply run the installation script like this:

```
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 in 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:

```
class ApplicationController < ActionController::Base
  include ExceptionNotifiable
  ...
end
```

Next, configure the email addresses that should get notified of Rails exceptions. Put the notification in config/environment.rb:

```
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:
     -------------------------------

       * 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          : /
       * 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

          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'
          #{RAILS_ROOT}/vendor/rails/actionpack/lib/action_view/base.rb:260:in ↩
           `render_file'
...
```

Voila! This email message is an actual email notification that helped solve a production problem in the code at ChangingThePresent.[3] The email contains a full complement 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 plug-in 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 a notification.

3. http://ChangingThePresent.org

6.6 Heartbeat

The exception_notification plug-in 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 system 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 custom 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 is down at ChangingThePresent:

Download managing_things/heartbeat.rb

```
#!/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

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 the developers on the project via an email address that 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.

6.7 Conclusion

The management strategies in this chapter don't cost anything, but they are surprisingly robust. Building repeatable Mongrel configurations rather than command-line options is easy and enables consistent clustering. Configuring your Mongrels in a cluster gives you good performance and some failover. Clustering Mongrel is important because of the Rails shared-nothing strategy.

Clustering is only the beginning of your managing strategy. To run production Mongrels, you need information and control. 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.