Ruby on Rails

Code Review

Improve the quality of your code

by Geoffrey Grosenbach

PeepCode press

$9

CONTENTS

4 Store Sessions in the Database

7 Use Custom Configuration Files

11 Use Constants for Repeated Strings

14 Keep Time in UTC

16 Don't Loop Around ActiveRecord

22 Beware of Binary Fields

26 Cache Your Reports

31 Store Country Lists

34 Avoid Bloated Controllers

38 Keep Your Controllers and Views Skinny

45 Don't Store Objects in the Session

49 Avoid Heavy Response Processing

54 Use ActiveRecord Mailing

60 Monitor Your Servers

67 Don't Cut Costs on Hardware

71 Test-Drive

78 The Rest of the List

80 Revisions

INTRODUCTION

Writing better code starts with the basics. The tips in this book aren't the kind that will help you win awards. They're the kind that will keep you from being fired, or from inadvertently writing code that cuts your site's performance in half.

I've read through the code of both proprietary Rails applications and open source projects. All the concepts you see here came from real-world examples (but variable names have been changed to protect the innocent). Some of the examples, I'm afraid to admit, came from my own code.

You won't be able to follow each chapter as a complete project from start to finish. I've assumed that you already have a project in process that needs to be tuned, so treat these as a collection of good ideas that will need to be customized for your application.

The good news is that most of these tips are easy to implement. You'll get into the habit of switching your timezone to UTC, setting up database tables for sessions, and installing the `exception_notifier` every time you write a new Rails application. Once you've been developing an application for a while, you'll look forward to running it against `ruby-prof` or setting up good monitoring tools.

Although it may be embarrassing to buy a book targeted at beginners, it will save you from repeating the beginner's blunders. If you run into a problem that isn't listed here, I'd be glad to include it in a future edition of this book. I'll understand if you say that it was really the developer in the cubicle next to yours who wrote that awful code!

Geoffrey Grosenbach, August 2007

Store Sessions in the Database

CHAPTER 1

THE WRONG WAY

By default, Rails stores user sessions on the filesystem.

This is a quick, easy setup that requires no configuration. In fact, you could run a website with sessions even if you weren't connected to a database.

Unfortunately, that party won't last long. Sites as prominent as A List Apart (http://alistapart.com) have nearly come to a crashing halt because their hard drives filled up with thousands of session files.

Disk-based sessions are also more difficult to maintain. You need to write cryptic strings of shell commands to clear them out. You can't easily use them with multiple application servers unless they can all access the same hard drive.

THE RIGHT WAY

There are several ways to store sessions, but database-backed ActiveRecord sessions are the most common. Memcached is also a good way to go, but is better left for later stages of a site's growth when many servers are being used. If your site runs on up to three servers, ActiveRecord is the easiest and most efficient way to go.

First, generate the migration to build the `sessions` table in the database.

It is often thought that Memcached is always faster than ActiveRecord sessions. Not true! If your database is on the same machine as your application server, ActiveRecord will be as fast or faster than Memcached would be.

```
ruby script/generate session_migration
```

Next, run the migration.

```
rake db:migrate
```

Finally, uncomment this line in `config/environment.rb` to instruct Rails to use the `active_record_store` for sessions (you can put it all on one line):

```
# Use the database for sessions instead of the file system
# (create the session table with 'rake db:sessions:create')
config.action_controller.session_store =
                                        :active_record_store
```

> The comments in `environment.rb` tell you to create the sessions table with the `db:sessions:create` rake task. This generates the same migration file that the generator does, but oddly does not run the `db:migrate` task to create the table.

For maintenance, the following tasks are useful. Create a file in `lib/tasks/sessions.rake` with these contents:

```
namespace :sessions do
  desc "Count database sessions"
  task :count => :environment do
    count = CGI::Session::ActiveRecordStore::Session.count
    puts "Sessions stored: #{count}"
  end

  desc "Clear database-stored sessions older than two weeks"
  task :prune => :environment do
    CGI::Session::ActiveRecordStore::Session.delete_all [
      "updated_at < ?", 2.weeks.ago
```

```
    ]
  end
end
```

Run `rake sessions:count` to find out how many sessions are being stored in the database. Run `rake sessions:prune` to clear out sessions older than two weeks.

Rails provides a task to clear sessions, but it deletes all sessions in the database, including those for users who are currently logged in! However, this does have the benefit of being very quick and can be useful if you have many sessions in the database. I've had nearly 500,000 sessions at times and the `prune` task took 30 minutes to execute!

It's a good idea to run the `prune` task regularly. Set a daily cron job to run this command and your `sessions` table will be in good shape:

```
rake sessions:prune RAILS_ENV=production
```

Use Custom Configuration Files

CHAPTER 2

THE WRONG WAY

Web applications often need to connect to other servers. This usually means storing an API key, a username and password, or other server configuration details.

In the worst case, this secret information is littered throughout your code. I've even seen plugins that instruct you to modify the source of the plugin with your secret API key!

Hopefully your code is more organized. Maybe you've even stored this information in constants within `environment.rb`:

```
# The wrong way
AMAZON_API_KEY    = '1234567890'
AMAZON_API_SECRET = 'abcdefg'
FLICKR_API_KEY    = '1234567890'
```

But even this is not as organized as it should be.

- **Your secret keys go everywhere your code does**. If you have subcontractors working on your code, they will have access to your secret information.
- **They are not relevant to the environment**. If you need to use different credentials from your staging server or development environment, you'll have to add logic into `environment.rb` or scatter it again in `config/environments/development.rb` and its counter-

parts. The environment files often include non-secret settings that really do need to go everywhere your code does, so this isn't a complete solution.

THE RIGHT WAY

There are several ways to do this correctly. One way is to create a configuration settings file in YAML format. This looks a lot like the `database.yml` file, but is more flexible since you can put any kind of information in it.

Here is a sample that could be stored in `config/config.yml` (the ampersand syntax is a YAML alias for reusing information):

```
development: &non_production_settings
  amazon_api:
    key: 1234567890
    secret: abcdefg
  flickr_api:
    key: 1234567890
  mailer:
    server: mail.example.com
    username: bubba
    password: s0meth1ngs3cr3t

test:
  <<: *non_production_settings

production:
  amazon_api:
    key: 9999999999
    secret: aaaaaaa
  flickr_api:
    key: 222222222
  mailer:
    server: mail.example.com
    username: bart
    password: 1c2r3c4g,8,938,7
```

None of these values are required, so you can create whichever ones you need. It's best to keep them consistent across the different environments. You can also use the ampersand trick, but add other values for the same environment, which will override duplicated settings.

In `environment.rb`, load the current environment's settings with these two simple lines:

```
# Load custom config file for current environment
raw_config = File.read(RAILS_ROOT + "/config/config.yml")
APP_CONFIG = YAML.load(raw_config)[RAILS_ENV]
```

Anywhere in your application, you can now access the configuration settings for the current environment with a simple lookup like this:

```
APP_CONFIG['flickr_api']['key']
```

You can use this to store API keys, mail server settings, or any other kind of configuration data that your application needs.

In order to keep your production settings secret, store the production copy of this file on your server under `shared/config/config.yml` and copy it on deployment with this Capistrano task:

```
task :copy_config_files do
  filename = "config/config.yml"
  run "cp #{shared_path}/#{filename} #{release_
path}/#{filename}"
end

after "deploy:update_code", :copy_config_files
```

With this kind of setup, you can safely store a dummy copy of `config.yml` in your repository. The copy in the repository should not

contain the production passwords, but can contain sandbox-related passwords needed for development.

RESOURCES

- Using OpenStruct for easier key access (http://kpumuk.info/ruby-on-rails/flexible-application-configuration-in-ruby-on-rails)
- The Ampersand trick in YAML files (http://blog.bleything.net/2006/06/27/dry-out-your-database-yml)

Use Constants for Repeated Strings

CHAPTER 3

THE WRONG WAY

Whether or not you use the previous tip for storing sensitive information, you'll will still need to store some non-sensitive information in your application. Usually this *is* information that needs to travel with your application wherever it goes.

It's tempting to list this information as a simple string wherever it needs to be used. Examples are contact emails, strings like "do-not-reply@mydomain.com", and offsite URLs.

Repeating these strings throughout an application increases the possibility that one will be misspelled. These kinds of bugs are hard to track down.

Another problem is that your different environments often need different settings for these strings.

A specific example is ActionMailer. In order to use the route generating methods available to controllers and views, you need to declare your site's URL as an option.

```
class UserMailer < ActionMailer::Base

  include ActionController::UrlWriter
  # The wrong way
  default_url_options[:host] = 'peepcode.com'
```

Using a hard-coded string here makes the emails unusable when sent from the development environment, since they point at the production server.

THE RIGHT WAY

Instead, use Ruby constants and Rails environments. This is exactly the kind of thing Rails environment files were designed to help with.

```
# In environments/development.rb
APP_DOMAIN = 'localhost:3000'

# In environments/staging.rb
APP_DOMAIN = 'staging.peepcode.com'

# In environments/production.rb
APP_DOMAIN = 'peepcode.com'
```

Then reuse the constant in the rest of your code. For the example shown earlier, it would look like this:

```
default_url_options[:host] = APP_DOMAIN
```

Non-sensitive candidates for this treatment are

- FROM_EMAIL, DO_NOT_REPLY_EMAIL, CUSTOMER_SUPPORT_EMAIL. These don't need to be pulled from a database since they will probably not change very frequently. They *will* be used repeatedly throughout the application, so store a constant and reuse it in mailers and views.
- SERVER_DOMAIN or APP_DOMAIN. Set them separately based on the environment.
- Operating system-specific paths to command-line executables. If

you are developing on Mac OS X and deploying to Linux, this could come in handy.

- Application-wide flags such as BETA_MODE. When writing code for temporary features, surround those features with a conditional based on BETA_MODE. Instead of re-writing your authentication system when you go public, you'll be able to flip a setting in the environment and be up and running right away.

RESOURCES

- AppConfig plugin (http://agilewebdevelopment.com/plugins/app_config)

Keep Time in UTC

CHAPTER 4

THE WRONG WAY

Time is the bane of the programmer's existence. We are always running out of time for projects, and we also have to perform calculations against it.

At minimum, `created_at` and `updated_at` times are recorded for database records. Times might also be used to send reminders or generate calendars.

By default, Rails stores all these times in the local time zone of the database server. This makes no sense since web servers are often in a different time zone than the developer. Visitors are almost *always* in a different time zone than the web server.

THE RIGHT WAY

The most sensible thing to do is to store times internally in UTC (Coordinated Universal Time). This is the zero hour time zone that programming languages are built to perform calculations against. If all dates are stored as UTC, then any time can be easily converted to a user's local time.

Rails teases the developer with exactly the line of configuration needed to remedy this situation, but it's thinly veiled behind a comment in `environment.rb`. Uncomment this line and you'll be on your way to sensible time storage:

```
Rails::Initializer.run do |config|
  # Make Active Record use UTC-base instead of local time
  config.active_record.default_timezone = :utc
```

```
end
```

Now the possibilities are endless! Show local times with the help of client side Javascript, or use Jamis Buck's TzTime plugin to calculate dates on the server with Ruby (requires Rails 2.0 edge).

MySQL and PostgreSQL also have timezone-specific functions. These will perform calculations faster than Ruby can, but they will also make your queries more complicated. See CONVERT_TZ for MySQL or TIMESTAMP WITH TIME ZONE for PostgreSQL.

RESOURCES

- UTC (http://en.wikipedia.org/wiki/Coordinated_Universal_Time)
- Showing Perfect Time with Javascript (http://redhanded.hobix.com/inspect/showingPerfectTime.html)
- Jamis Buck's TzTime plugin (http://weblog.jamisbuck.org/2007/2/2/introducing-tztime)
- PostgreSQL timestamp with time zone (http://www.postgresql.org/docs/8.0/interactive/functions-datetime.html)
- PostgreSQL timezones (http://www.postgresql.org/docs/7.2/static/timezones.html)
- MySQL CONVERT_TZ (http://dev.mysql.com/doc/refman/5.0/en/time-zone-support.html)

Don't Loop Around ActiveRecord

CHAPTER 5

THE WRONG WAY

ActiveRecord makes database access quite simple. With a few lines of Ruby, you can jump across tables and drill down into associated objects. It's so easy that you may be tempted to do something like this:

```
# The wrong way
@league = League.find_by_name "Northwest"

@all_stars = []
@league.teams.each do |team|
  team.players.each do |player|
    @all_stars << player if player.all_star?
  end
end

@all_stars
```

Unfortunately, that will quickly become a very expensive operation. To see why, look at the Rails logfile after this query:

```
League Load (0.000331)   SELECT * FROM leagues WHERE
(leagues."name" = 'Northwest') LIMIT 1
Team Load (0.000573)   SELECT * FROM teams WHERE (teams.
league_id = 1)
Player Load (0.000693)   SELECT * FROM players WHERE
(players.team_id = 1)
Player Load (0.000658)   SELECT * FROM players WHERE
(players.team_id = 2)
Player Load (0.000623)   SELECT * FROM players WHERE
```

```
(players.team_id = 3)
```

What happened? We started with a single `league` object, then issued a separate SQL command to retrieve the teams for that league. Additional SQL commands were required to retrieve the players in each of those teams.

In this case there were only a few teams, but 5 queries were generated. A league with more teams, more players, or further nesting could easily result in dozens of SQL queries and a very slow response.

THE RIGHT WAY

There are two easy ways to improve this code. First, you could add a single @include@ directive to load all the teams and players with a single SQL command.

```
# A better way, with eager loading
@league = League.find_by_name("Northwest", {
  :include => {:teams => :players}
})

@all_stars = []
@league.teams.each do |team|
  team.players.each do |player|
    @all_stars << player if player.all_star?
  end
end

@all_stars
```

Looking at the log now shows a single query:

```
League Load Including Associations (0.002438)   SELECT
leagues."id" AS t0_r0, leagues."name" AS t0_r1, teams."id"
```

:include Options

The `include` argument is very powerful but can also be confusing. Here are a few issues to be aware of.

PLURALITY

The plurality of the `include` needs to match the plurality of the association you're joining on.

For example, here is a `Farm` that has many chickens (and also cows):

```
class Farm < ActiveRecord::Base
  has_many :chickens
  has_many :cows
end
```

If you wanted to get all the farms and all their chickens, you would use the plural `chickens` because that is how it is used in the `has_many` statement in the model:

```
Farm.find(:all, :include => :chickens)
```

In reverse, you would use the singular. `Chicken` belongs to the singular `farm`:

```
class Chicken < ActiveRecord::Base
  belongs_to :farm
  has_many :eggs
end
```

Use the singular `farm` to include a join from the `Chicken` model back to the `Farm` model.

```
Chicken.find(:all, :include => :farm)
```

HASHES AND ARRAYS

The include starts with a hash, but it can go either way depending on how many associations are being included. We already saw that a `Farm` has both `chickens` and `cows`. To include both, use an `Array` as the value for the `include` hash.

```
Farm.find(:all, {
  :include => [:chickens, :cows]
})
```

ALL THE WAY DOWN

You can go even further (within reason). We could retrieve all the farms, all their cows and chickens, and each of the chickens' eggs.

```
Farm.find(:all, {
  :include => [
    :cows,
    {:chickens => :eggs}
  ]
})
```

```
AS t1_r0, teams."name" AS t1_r1, teams."league_id" AS
t1_r2, players."id" AS t2_r0, players."name" AS t2_r1,
players."team_id" AS t2_r2, players."all_star" AS t2_r3 FROM
leagues LEFT OUTER JOIN teams ON teams.league_id = leagues.
id LEFT OUTER JOIN players ON players.team_id = teams.id
WHERE (leagues."name" = 'Northwest')
```

Even better, there will never be more than one query, no matter how many rows are in the database.

But this isn't the best way. We're still using Ruby to filter the records, which can be slow. We're still returning more records than we need, which results in extra traffic between your application server and your database server.

An even better way is to write a small amount of custom SQL that returns only the records we need. To do this, we'll need to rethink the query. It's not really about the league, it's about players. We still want to return the name of the team and the name of the league for each player, but we can start by filtering the players by their all-star status.

```
# A better way, with custom SQL
@all_stars = Player.find(:all, {
  :conditions => [
    "leagues.name = ? AND players.all_star = ?",
    "Northwest", true
  ],
  :include => {:team => :league}
})
```

Because we're writing a snippet of SQL, we have to break the veil of plurality. ActiveRecord table names are plural by default. When referring to the names of joined tables directly (as with the first value passed to `:conditions`), you'll need to use the plural. In the example above, `leagues` and `players` are named directly in the conditional, so

we have to use the plural table name, not the singular model name.

The log here is still only one query, but now the database is doing the work of filtering the players, which will usually be much faster than asking Ruby to do it.

```
Player Load Including Associations (0.002233)   SELECT
players."id" AS t0_r0, players."name" AS t0_r1,
players."team_id" AS t0_r2, players."all_star" AS t0_r3,
teams."id" AS t1_r0, teams."name" AS t1_r1, teams."league_
id" AS t1_r2, leagues."id" AS t2_r0, leagues."name" AS t2_r1
FROM players LEFT OUTER JOIN teams ON teams.id = players.
team_id LEFT OUTER JOIN leagues ON leagues.id = teams.
league_id WHERE (leagues.name = 'Northwest' AND players.
all_star = 't')
```

RESOURCES

- RailsCasts on include (http://railscasts.com/episodes/22)
- Include by Default plugin (http://blog.jcoglan.com/includebydefault)

:select and :include

There is a tricky interaction between the `:select` and `:include` options. If you are constructing special queries for reports, you'll need to understand how they work together.

:INCLUDE CLOBBERS :SELECT

If a query contains an `:include` directive, any `:select` options will be ignored. For example you might expect this to return an array of ids only. Not so!

```
# This won't work as you expect
Farm.find(:all, {
  :select  => "farms.id, chickens.id",
  :include => :chickens
})
```

The `:include` will force all fields to be returned, no matter what you pass to `:select`.

USE :JOIN WITH :SELECT

The way around this is to code the `:join` explicitly.

```
Farm.find(:all, {
  :select  => "farms.id, chickens.id",
  :join    => "LEFT JOIN ..."
})
```

Rails has softened my brain so that in my (not very) old age I often forget the exact SQL join syntax that I want to use. To be honest, I always had to look it up online even before I had access to a nice ORM library.

My modus operandi is to type my query into the Rails console and look at the output in the development log. I can then copy and paste the proper join string back into my code in the right place.

```
LEFT OUTER JOIN chickens ON chickens.farm_id =
farms.id
```

Beware of Binary Fields

CHAPTER 6

THE WRONG WAY

File storage can quickly become a problem for a growing site. If you're running more than one Mongrel application server, storing user-uploaded files on a single disk prevents other application servers from accessing them. The next step is to use networked file storage or a distributed solution like MogileFS.

Many sites try to solve this problem from the outset by storing files in the database. A table with a binary field stores the file and a controller action streams the file when it is requested.

Here is a simple migration that creates a `photos` table with a binary field to store the photo data.

```
class CreatePhotos < ActiveRecord::Migration
  def self.up
    create_table :photos do |t|
      t.column :caption, :string
      t.column :data, :binary
      t.column :updated_at, :datetime
    end
  end
```

I often use the Sqlite database for small prototypes (including the examples for this chapter). Unfortunately, Sqlite doesn't support the binary field properly, so your binary data will be mangled when you retrieve it. The lesson is to use MySQL or Postgres, or you'll spend an hour banging your head against a wall like I did while writing this chapter.

You may be tempted to write a simple index action that lists the photos like this:

```
def index
  @photos = Photo.find :all
end
```

The problem here is that ActiveRecord retrieves *all fields* from the database unless told otherwise. The entire contents of each of your binary fields will be returned, even if you don't use them! If you have 20 rows that each contain a 100 KB file, you're transferring an extra 2 MB across the network from your database server (or at least loading them into memory if you're using a database on the same machine as the application server).

THE RIGHT WAY

Instead, select only the fields you need, at the time that you need them. In most cases this can be done by writing a class method in your `Photo` model:

```
def self.find_all_for_listing
  find(:all, :select => "id, caption, updated_at")
end
```

If you need more flexibility, use `with_scope` in a model class method to specify the fields.

```
def self.find_all_for_listing_scoped(*args)
  with_scope(:find => {
    :select => "id, caption, updated_at"
  }) do
    find(:all, *args)
  end
end
```

Now you can call `Photo.find_all_for_listing_scoped` with the hash arguments you would normally pass to a plain `find` method and it will return all the results without the binary field.

> Reviewers of this book warned that `with_scope` can

> become dangerous and lead to messy code. Use it only within model class methods. The words `with_scope` should never be found in a controller.

You can even use your custom class methods in chained calls:

```
user = User.find :first
user.photos.find_all_for_listing
```

You can call your custom class method when you just need to show a listing of the photos in the view. The `id` and `caption` are usually enough to show a gallery with a thumbnail and a link to the full resolution photo. You could add a `user_id` or other ownership-related fields if necessary.

Core member Rick Olson recommends the scope_out plugin (http://agilewebdevelopment.com/plugins/scope_out). It wraps similar functionality in a single method. I found it easier and more explicit to just use `with_scope` as shown above, but more complicated finds may benefit from the plugin.

You'll also need a controller to send the photo data when the image is requested by a web browser. This can be done all in one action with `respond_to` and an extra mime type. In `environment.rb`, declare the types of the images that you will be serving.

```
Mime::Type.register "image/jpeg", :jpg
Mime::Type.register "image/png",  :png
```

In your controller you can handle requests for the photo with a matching `respond_to` block.

```
def show
  @photo = Photo.find params[:id]
```

```
  respond_to do |format|
    format.html
    format.jpg do
      send_data @photo.data, {
        :disposition => "inline",
        :type => "image/jpeg"
      }
    end
  end
end
```

If you don't need to restrict access to the images, you can save processing power by letting Rails cache the images to disk the first time they are requested. Add this to the top of your `PhotosController`:

```
caches_page :show
```

> Using the `caches_page` directive will cache both the JPG version and the HTML version of the `show` action. The current edge (2.0) version of Rails has the ability to only cache certain data types.

RESOURCES

- scope_out plugin (http://agilewebdevelopment.com/plugins/scope_out)
- Dan Manges on scope_out (http://dcmanges.com/blog/21)

Cache Your Reports

CHAPTER 7

THE WRONG WAY

If your site sells a product, allows user input, or even just has visitors, you probably need to do some kind of reporting.

If you're like me, you start out with a simple report that runs fairly quickly. It gradually slows down as your tables fill with data. After a few months it takes 30 seconds to run.

Eventually, impatience wins out over the need to know what the report was going to tell you. You've unintentionally trained yourself to never look at it because it takes so long to respond and sends your server load to eleven! If it's really bad, Mongrel will timeout before it even completes the request.

You may find a temporary fix in adding a few database indexes to relevant tables. But of course that's only temporary and after a few weeks you're back to where you were.

THE RIGHT WAY

When you get to that point, you need to aggregate the results of your report and store them for future reference. I've used mem-cached for short-term storage, but it's worth the extra effort to add a few tables to your database and store it permanently.

The implementation details will depend on the contents of your report, but here's a flexible strategy I've used for several years.

For this example, I'll assume that I have a `views` table with one row for

every visit to a page on the site, which I'll call a resource. I'll create a report that shows the total number of visits to each resource for each day.

First, create a generic table to store all your aggregations:

```
create_table :reports do |t|
  t.column :label,       :string
  t.column :quantity,    :integer
  t.column :reported_on, :date
  t.column :type,        :string
end
```

Even if you use a third-party service like Google Analytics, you can use this technique to aggregate reports on other data within your site.

- `label` will be populated with the name of the resource being counted.
- `quantity` will hold the total number of visits for one day.
- `reported_on` records the day this statistic pertains to. If you need to store statistics with greater precision than a single day, make this a `datetime` field.
- You may also notice that I've added a `type` column to make this table compatible with the Single Table Inheritance pattern. The plan here is that you'll use this table to aggregate many different types of reports across different tables. Subclasses of `Report` will each know about the tables they need to report on and will use only the columns from `reports` that they need.

First, create a basic `Report` model. This will be abstract and won't be used for any actual report.

```
class Report < ActiveRecord::Base

end
```

Next, create a child class. This will still store its data in the `reports` table. However, the STI mechanism will be used to differentiate

reports from each other.

```
class ReportVisit < Report
```

I often want to be able to call it something like `Report::Visit`, but this becomes complicated when you want to create an association back to the regular `Visit` model. It's usually easier to just use a camel-cased name instead of a sub-module.

Now write your reporting code as class methods of the `ReportVisit` model. If you've followed the journey I described at the beginning of this chapter, you may have already written your reporting methods. In this case, you can call these methods instead of cluttering up this model.

Let's look at an example, then I'll comment on what it does:

```
def self.aggregate(day=(Date.today - 1))
  # Delete existing records for this date
  delete_all ['reported_on = ?', day]

  visits = Visit.report_for_day(day)
  visits.each do |visit|
    create({
      :label => visit.resource,
      :quantity => visit.hits,
      :reported_on => visit.created_on
    })
  end
end
```

Here's what's happening:

- Most reports will be run nightly, after midnight, and will summarize information from the previous day. I start the method with yesterday as the default `day` (which can be overridden).

- I like to write these methods so they can be run multiple times without inserting redundant records. The easiest way to do this is to start the method by deleting all the existing aggregated records for the date being reported on. Because I'm using STI, only the records for this report will be deleted, not records of other sub-classes of `Report`.
- Next, call a reporting method in another model with the `day` argument. You can also write the method that generates the report and put it in the `ReportVisit` model. If you do, be sure to make it a class method (i.e. prepended with `self`).
- Finally, loop through the records and save them to the current aggregation table with `create`. I'm not doing any error checking, but you could check the return value of `create` or call `create!` to throw errors on failure.

For reference, here is the `Visit` model that is being called from the `aggregate` method:

```
class Visit < ActiveRecord::Base

  def self.report_for_day(day)
    find(:all, {
      :select => "count(id) as hits,
                  LEFT(created_at, 10) as created_on,
                  resource",
      :conditions => ["left(created_at, 10) = ?", day],
      :group  => "created_on, resource",
      :order  => "created_on DESC, resource ASC"
    })
  end

end
```

Finally, call the `aggregate` method daily with a cron job. Here are two rake tasks for running `daily` or `yearly` aggregates.

```
namespace :report do
```

```
    desc "Aggregate daily views (yesterday only)"
    task :daily => :environment do
      ReportVisit.aggregate
    end

    desc "Aggregate past year's views"
    task :yearly => :environment do
      lastyear = 1.year.ago.to_date
      today    = Date.today
      (lastyear..today).each do |date|
        ReportVisit.aggregate(date)
      end
    end

  end
```

The beauty of this design is that it is very flexible. If you need to create a report that should join with a `products` table, just add a `product_id` foreign key column to the `reports` table and create a `ReportProduct` model that `belongs_to :product`.

RESOURCES

- PeepCode Screencast on Page, Action, and Fragment Caching (http://peepcode.com/products/page-action-and-fragment-caching)

Store Country Lists

CHAPTER 8

THE WRONG WAY

Most applications need to store an address at some point. If you are selling products, you may pay different taxes based on the country of residence of the person making the purchase.

Rails provides a nice helper for building a select list with most of the countries of the world.

```
<%= country_select 'user', 'country' %>
```

There are a few problems with this.

- **It's string-based**. If you build your database around this helper, you'll be storing each user's country as a string. This means that you can't easily run a report on all the purchases in a particular country (you'll need integer foreign keys for that).
- **It's not easily editable**. If you decide that you don't need to list all the countries provided, you're stuck. If you want to add more, you're in the same situation.
- **The only configuration is in the view**. If you want to modify the display of the list, you'll have to do it in the view.
- **It can't be validated**. If you allow users to signup via an API, you have no way of checking their input against an authoritative list of countries. Your database could fill up with fictional, or more likely, misspelled countries.

THE RIGHT WAY

It is much better to store the built-in country list in the database. This makes it possible to add fields for sorting the list. You can also take advantage of class methods to query the list of countries. The built-in list of countries can be used as a starting point.

First, make a model (and automatic migration) for the country table. (You could also use a full scaffold if you want to be able to edit the list in the browser.)

```
./script/generate model Country
```

Caveat: The TzTime plugin mentioned elsewhere in this book defines a subclass called `Country`. If you have conflicts while using this, you may have to refer to the `Country` model with `::Country` instead.

Edit the migration to include a `name` field in the `countries` table. I usually add a `special` field for marking countries that should be displayed at the top of the list. We'll use the built-in array of country names to populate the database.

```
create_table :countries do |t|
  t.column :name, :string
  t.column :special, :boolean, :default => false
end

names = ActionView::Helpers::FormOptionsHelper::COUNTRIES
names.each do |country_name|
  Country.create(:name => country_name)
end
```

Run the migration and you'll have a fully populated table with over 200 countries.

Now we have a `Country` model that can be enhanced with a few methods to make querying easier.

First, add a class method that returns the records in alphabetical order. I've added an `order` clause to put any countries marked as

special at the top of the list.

```
##
# Find all countries in a special order.

def self.find_ordered
  find :all, :order => 'special DESC, name'
end
```

The last thing to do is to format the returned array for the select helper. This method uses the find_ordered method that we already wrote and returns an array of arrays that includes the name and id of each country.

```
##
# Find all countries and return in an Array usable
# by the select helper.

def self.find_for_select
  find_ordered.collect { |record| [record.name, record.id] }
end
```

Finally, we can use this with the normal select helper to show a list of countries to choose from.

```
<%= select 'user', 'country_id', Country.find_for_select %>
```

RESOURCES

- API Reference for country_select (http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html#M000508)

Avoid Bloated Controllers

CHAPTER 9

THE WRONG WAY

It's easy to keep adding actions to a controller. Some people even prefer to have many actions in a single controller because it seems easier to navigate a few files instead of many.

Unfortunately, your `ArticlesController` can soon get to look like this:

```
index
show
new
create
edit
update
destroy
add_comment
destroy_comment
approve_comment
mark_comment_as_spam
mark_comment_as_ham
mark_comment_as_should_have_been_spam
destroy_old_spam_comments
etc.
```

Of course the flip side is often a few actions that are too complicated.

```
# Too bloated! Don't do this.
def show
  if params[:id]
    if request.post?
      @article = Article.new(params[:article])
```

```
      if @article.save
        flash[:notice] = "Article was created!"
        redirect_to article_url(@article)
      else
        render "new"
      end
    else
      @article = Article.find(params[:id])
      render "show"
    end
  else
    @article = Article.new
    render "new"
  end
end
```

At any rate, the problem is usually finding a stopping point. How do you know when it's time to start a new controller?

THE RIGHT WAY

The architecture currently in vogue is called REST and dictates that you stick to the 7 CRUD actions for every controller.

- index
- show
- new
- create
- edit
- update
- destroy

There is some flexibility for adding a few custom methods that don't fit into this pattern. I often add one or two custom methods, but have set a hard maximum at 10. If I get to 10 actions, I probably need to

rethink the design of the application.

You don't need to use all 7 if you don't need them. For example, Rick Olson's `restful_authentication` plugin creates a `SessionsController` with only a `create` action for logging in and a `destroy` action for logging out. You can find these via the sensible aliases `http://peepcode.com/login` and `http://peepcode.com/logout`.

Here are a few fallacies that confuse people when trying to build around REST:

- **RESTful controllers dictate a certain kind of user interface**. FALSE! REST is about an API, not a visual interface. When most people think of CRUD, they think of a table with a "Delete" button on every row. Fortunately, it doesn't have to work that way. Almost any interface can be developed with a RESTful backend, so don't let the API change the way you design the user interface. For example, an email program does little more than show, update, delete, and create messages but there are many different kinds of email interfaces.
- **Controllers should correspond one to one with models**. FALSE! Many of my applications have more than one controller for a single model. In contrast, you could have one controller that touches several models.
- **RESTful actions should work destructively on their targets**. FALSE! A RESTful controller may represent the creation or deletion of only a concept. For example, a `SpamsController` may `create` spam by changing a comment's status to `spam` without adding any records to the database. In the real world, database tables don't always match up evenly with the concept of a resource, so additional controllers may be needed.

Here are some general guidelines if you have a controller with more than 7 actions.

- **Are the extra actions just changing an attribute of a model**? This is a good opportunity to make the attribute twiddling into it's own controller. You could use the `update` action if the attribute has many possible values, or `create` and `destroy` if it is a boolean field.
- **Am I working with a collection of things**? Sorting and searching are some of the hardest ideas to cram into a RESTful pattern. For searching, it's nice to use the HTTP GET method so a single URL recreates the search and can be emailed or bookmarked for future reference. Sorting (usually via a Javascript callback) is a type of update and should be done with a PUT. I often break the RESTful pattern for sorting and add an 8th action to update the database with the results of the sort.
- **Am I working with a third-party** API that requires a certain interface? PayPal sends people back to your site at a URL that you can specify. But depending on the type of payment used, the person could come back via POST or GET. In this case you could create a new controller that does nothing other than receive people who are returning from PayPal, or you could add an extra action to an `OrdersController` that can be accessed with either POST or GET. I've chosen to do the latter.

RESOURCES

- PeepCode Screencast on REST (http://peepcode.com/products/restful-rails)
- Trotter Cashion's Rails Refactoring to Resources (http://www.informit.com/store/product.aspx?isbn=0132417995)
- Abstraction via Hampton Catlin's make_resourceful Plugin (http://groups.google.com/group/make_resourceful)

Keep Your Controllers and Views Skinny

CHAPTER 10

THE WRONG WAY

When I first learned Rails, I would start an application by generating a few scaffolds. I would then open a controller file and start coding.

This made a lot of sense at the time because ActiveRecord does so much on the database side of things. Dynamic finders like `User.find_by_username_and_password` are so easy to use, and the controller's role is to shuttle things back and forth between the models and views anyway, right?

Write some controller code, print it out it in the view, refresh the web browser, repeat.

But I would end up with a lot of code in my controller actions and barely any code in the models.

```
# The wrong way
def fat_login
  @user = User.find_by_username_and_password({
    params[:user][:username],
    params[:user][:password]
  })
  if @user.verified?
    session[:user_id] = @user.id
    flash[:notice] = "You are now logged in"
    redirect_to "/"
    return
  else
    flash[:error] = "Wrong password!"
  end
```

```
end
```

You can find this inefficient strategy in many applications. Authentication, searching, reporting, and finds are often done in the controller.

The view is often the garbage pile for inadequate models. If you're doing complicated comparisons and calculations in your views, it's time to rethink and rework your models:

```
<%# The wrong way -%>
<% if article.updated_at < (Time.now - 60*60*24*31) %>
  New article!
<% end %>
```

THE RIGHT WAY

Jamis Buck (http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model) wrote a classic article about the flaws in this kind of strategy. It's hard to test, it's impossible to reuse, and it complicates your code. A much better way is to write custom methods in your models and call those from the controller.

For example, move all that model-specific controller code into the `User` model:

```
# Returns User object, or nil.
def self.authenticate(credentials)
  user = find_by_username_and_password({
    credentials[:username],
    credentials[:password]
  })
  return user.verified? rescue nil
end
```

Note that this is a class method that starts with `self`. This makes it possible to call the method on the `User` object as seen here:

```
def skinny_login
  if @user = User.authenticate(params[:user])
    session[:user_id] = @user.id
    flash[:notice] = "You are now logged in"
    redirect_to "/"
    return
  else
    flash[:error] = "Wrong password!"
  end
end
```

Now your controller is much simpler! Yes, you still have to manipulate the `flash` and the `session` in the controller, but you've made your code both more readable and more faithful to MVC. It's also simpler to pass the `:user` hash out of the `params` and let the model pull out the keys that it needs to know about.

It works with views, too. Those crazy conditionals from the view can become sensible methods on the `Article` model.

```
def recent?
  updated_at < 1.month.ago
end
```

Do you still need a bit of bit of control from the view? Take an argument, but start with a sensible default.

```
def more_recent_than?(compare_to=1.month.ago)
  updated_at < compare_to
end
```

Your view now becomes simpler and more descriptive.

```
<%# The right way -%>
<% if article.recent? %>
  New article!
<% end %>

<%# Or even... -%>
<% if article.more_recent_than?(2.months.ago) %>
  New article!
<% end %>
```

> There has been a lot of discussion recently about presenter objects that abstract views from models. This is still an experimental area and is not quite ready for majority of your code. Abstracting conditionals into models will still help you if you decide to write presenter objects later on.

GETTING THERE

As I've improved my Rails coding skills, and interesting thing has happened. I've found that I naturally put more code in the models. In fact, it would probably be better if `find` were a protected method that could only be called from other model methods. This would force developers to encapsulate complicated database-related code with custom model methods. Unfortunately, `find` is a public method and is often overused outside the context of the model. Somehow, I've changed my technique and write better code almost without trying. How did I learn this?

Test-first development.

If you start as I did a few years ago (by opening a controller file, writing some code, and refreshing the browser), it makes absolute sense to cram all kinds of code into the controller. Writing model code would take an extra step and would slow down the development

process. So you end up with heavy controllers and code that can't be reused in the console, in `rake` tasks, or in other controllers.

If, instead, your workflow starts with a list of expectations expressed in test code, you'll more naturally start by writing and testing code in your models. So the situation is reversed and it's actually easier to write well organized code.

The problem is partly about design, but it's also about workflow. Changing your workflow to use the principles of test-first development not only provides you with a nice test suite, it can also improve the implementation of your code.

In any case, here are a few practical tips for making the most of models.

STANDARD ENCAPSULATION

Most controllers contain a lot of database-specific code that can be pulled into class methods in a model. For example, a complicated `find` an a `User` model can become a class method in the `User` model:

```
def self.count_daily_signups
  find(:all, {
    :select => "count(id) as signup_count,
                left(created_at, 10) as signup_day",
    :group => "signup_day",
    :order => "signup_day ASC"
  })
end
```

Your controller can call the `count_daily_signups` method without any arguments:

```
User.count_daily_signups
```

> Developer Jay Fields (http://blog.jayfields.com) once suggested that `find` be made into a protected method, callable only within a model. This isn't a such a bad idea! Even now, you can treat it as such and write all your finds inside descriptively-named model methods.

WITH_SCOPE

If your query is used in several places and needs to take a few arguments, the `with_scope` method can be useful.

This method takes a variable number of `args` and passes them on to the `ActiveRecord` finder. The `with_scope` block wraps it in a condition. This condition will be combined with any other arguments passed in.

```
def self.find_recent(*args)
  with_scope(:find => {
    :conditions => ["created_at > ?", 1.week.ago]
  }) do
    find(:all, *args)
  end
end
```

For example, this query will combine the `username` condition with the `created_at` condition.

```
User.find_recent(:conditions => ["username = ?", 'jim'])
```

The generated SQL will look like this:

```
SELECT * FROM users WHERE (created_at > '2007-08-22
17:38:37') AND (username = 'jim')
```

INSTANCE METHODS

Model instance methods are a great way to make your code more readable. Instead of scattering complicated conditions all around your controllers and views, encapsulate the condition in the model.

```
def recent?
  updated_at < 1.month.ago
end
```

In your view, you can call the `recent?` method on any instance of `Article`, like this:

```
@article.recent?
```

RESOURCES

- Jamis Buck on Skinny Controller, Fat Model (http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model)
- Jay Fields on the Presenter Pattern (http://blog.jayfields.com/2007/03/rails-presenter-pattern.html)
- Courtenay Gasking on the Presenter Pattern (http://blog.caboo.se/articles/2007/8/23/simple-presenters)
- Bruce Williams on Views (http://codefluency.com/search?q=views)
- API Docs on with_scope (http://api.rubyonrails.org/classes/ActiveRecord/Base.html#M001024)
- with_scope (http://blog.caboo.se/articles/2006/02/22/nested-with_scope)

Don't Store Objects in the Session

CHAPTER 11

THE WRONG WAY

The Rails session is useful for keeping information about users between requests. The most common use of this is to keep track of users as they login and visit protected parts of your website.

Many examples encourage you to store an entire `User` object in the session like this:

```
def login
  user = User.authenticate(params[:user])
  session[:user] = user
  redirect_to '/'
end
```

There are several problems with this:

- An object uses up **memory**. If thousands of people are viewing your site, you will run out of memory quickly.
- ActiveRecord objects can contain **cached references** to other large objects, such as the user's photos or the contents of their shopping cart. These will also be included in the session and will use resources.
- A database object can become **stale**. The record in the session will not reflect changes in the record in the database. Changes could cause the two to become out of sync, resulting in confusing problems.

- **Custom objects** may not be available across applications. If you're using Rails together with the Merb framework, you'll see errors if Rails-only objects are stored in the session. This could even happen between applications written with Rails. For example, you may have one Rails application for your main site, but host a message board application like Beast (http://beast.caboo.se) on a subdomain.

THE RIGHT WAY

Instead of storing an entire object, store only integers, strings, arrays, and hashes. Yes, these are also Ruby objects, but they are small objects and only use classes built in to Ruby.

So instead of storing the entire `User` object, store only the user's id:

```
def login
  user = User.authenticate(params[:user])
  session[:user_id] = user.id
  redirect_to '/'
end
```

When you need to retrieve an object, you can pull it from the database. I find it helpful to follow the technique used in the restful_authentication plugin (http://svn.techno-weenie.net/projects/plugins/restful_authentication). Write a method that pulls the `id` from the session and does a lookup on the corresponding object. Add this `protected` method to your `ApplicationController`:

```
protected

# Fetch an order from the session.
def current_order
  if session[:order_id].nil?
    return nil
  else
    Order.find(session[:order_id])
  end
end
```

You can also use an instance variable to cache the fetched object for the duration of the request. Instance variables are cleared at the end of every request, so there is a lesser risk of a memory leak.

The last step is to make this available to all your views. In the `ApplicationController`, declare the method as a `helper_method`:

```
helper_method :current_order
```

The restful_authentication plugin uses this technique internally for storing `User` information. However, it is still good to be aware of this concept since you may be tempted to put other large objects in the session. You can still follow the same strategy and store the smallest amount of information needed to rebuild or retrieve a record.

MEMCACHED

Do you really, really need to store an object? Then memcached is the way to go.

It uses existing RAM to store objects and can automatically expire them after a specified amount of time. You can configure several servers to run the memcached daemon and information will be automatically distributed across all of them.

In the reporting area of PeepCode I use memcached to store some database queries so they can be used in HTML views and image-based graphs without making duplicate calls to the database.

It's better to start without memcached and add it later when your site really needs it. If you follow the advice elsewhere in this book about organization of model code, you'll be in good shape to wrap a few database queries with calls to memcached when your site grows.

RESOURCES

- memcached (http://danga.com/memcached)
- Nuby on Rails article about Rails and memcached (http://nubyonrails.com/articles/2006/08/17/memcached-basics-for-rails)
- memcache-client gem (http://seattlerb.rubyforge.org/memcache-client)
- cache_fu plugin (http://plugins.require.errtheblog.com/browser/cache_fu)

Avoid Heavy Response Processing

CHAPTER 12

THE WRONG WAY

All requests are not created equal. One controller action may just need to display text out of the database, and another may need to make 5 API calls to remote web applications. Yet another may need to make several thumbnails of a photo.

Performing all your processing during a response makes the site seem slower to the user. Why should the user wait for a photo to be resized if she won't need to view that photo immediately anyway? Why should someone wait while their comment to a blog is verified against a remote API?

Some tasks are inherently asynchronous, yet web applications are written in ways that make them synchronous when they shouldn't be. For example, sending a confirmation email to a new user doesn't need to happen before the user is shown a confirmation screen on the website.

If every task is performed during the response, a web process is monopolized and won't be available to serve other requests.

THE RIGHT WAY

Web developers often get into the rut of thinking only of the request-response cycle, yet there are other ways to design an application.

PREEMPTIVE CONVERSION

A common expenditure of CPU cycles is text conversion with Textile or Markdown. Rails provides a view helper called `textilize` that will convert Textile to HTML. Don't use it!

If you use the `textile` helper in your view, you're converting the text over and over every time it is viewed. You only need to convert when the data changes, not every single time you display the content.

Add a `description_html` field to the relevant table and automatically convert it before every `save` (or `create`) with these simple lines:

```
before_save :convert_description

def convert_description
  return if self.description.nil?
  self.description_html = RedCloth.new(self.description).
to_html
end
```

QUEUEING SYSTEMS

There are many messaging and queuing systems available, including Ezra Zygmuntowicz's BackgroundRB and Amazon's SQS. But this involves additional daemons and even more remote API calls. A simple combination of `rake` tasks and `cron` jobs can be a good starting point. When your site grows you will be in a good place to use a more robust message queue system.

The `ar_mailer` gem mentioned elsewhere in this book is a good example of a queuing system. The general idea is that controller actions should only add or delete rows from database tables (which can be done quickly). Each row will have a boolean flag or multi-state column that identifies whether or not it has been processed. A cron job will periodically query the database for these records and perform the appropriate processing.

MULTI-STAGE TASK QUEUEING WITH ACTS_AS_STATE_MACHINE

Most of the queueing, querying, and processing code should reside in the application's models. Scott Barron's `acts_as_state_machine` plugin is a great tool for this purpose. In this example, I'll mark new blog comments as `:pending`. A class method will look for `pending` comments and check them against the Akismet spam filtering service, marking them as `spam` or `ham`.

First, install the `acts_as_state_machine` plugin.

```
ruby script/plugin install http://elitists.textdriven.com/
svn/plugins/acts_as_state_machine/trunk/
```

Next, declare an initial state in your `Comment` model.

```
acts_as_state_machine :initial => :pending
```

We'll need a few states to work with. Here is a simple set of states:

```
state :pending
state :spam
state :ham
state :denied
```

The `acts_as_state_machine` plugin also provides a callback system that can fire events when a record changes from one state to another. Useful events are sending emails after approval (via another queue) or crediting an account when an invoice has been paid.

Now, define events and transitions that will change the record from one state to another. These will be called as `@comment.spam!` or `@comment.ham!`.

```
event :spam do
```

```
    transitions :to => :spam, :from => :pending
  end

  event :ham do
    transitions :to => :ham, :from => :pending
  end
```

This is a strict set of events. I often find the need to add other transition directives so a comment can be changed from `spam` back to `ham` or vice versa.

Finally, write a class method to process pending comments. This will be called from a rake task and makes calls to the Akismet API.

```
  def self.process_spam(akismet_key, akismet_blog_name)
    akismet = Akismet.new(akismet_key, akismet_blog_name)

    find_all_by_state('pending').each do |pending_comment|
      comment_request = pending_comment.parsed_request
      akismet_data = {
        :user_ip => comment_request.env['REMOTE_ADDR'],
        :user_agent => comment_request.env['HTTP_USER_AGENT'],
        :referrer => comment_request.env['HTTP_REFERER'],
        :permalink => pending_comment.article.permalink,
        :comment_type => "comment",
        :comment_author => pending_comment.author,
        :comment_author_email => pending_comment.email,
        :comment_author_url => pending_comment.url,
        :comment_content => pending_comment.body
      }
      if akismet.is_spam?(akismet_data)
        pending_comment.spam!
      else
        pending_comment.ham!
      end
    end
  end
```

This method can be called regularly with the following rake task:

```
desc "Process comments and mark as spam or ham"
task :process_spam => :environment do
  akismet_key = APP_CONFIG['akismet']['key']
  blog_name   = APP_CONFIG['akismet']['blog_name']
  Comment.process_spam(akismet_key, blog_name)
end
```

Although I used a specific example, it's more important to treat this as a way of approaching application tasks. If you don't need to complete a task immediately, find a way to queue it and hand it off to another daemon or queue. In most cases, you can complete the task within a minute or less and the user will never notice. The result will be a much faster experience while browsing your web application.

Other tasks like log parsing or reporting don't need a queue but can still be completed with a combination of rake tasks and cron jobs.

RESOURCES

- Acts as State Machine plugin (http://elitists.textdriven.com/svn/plugins/acts_as_state_machine/trunk)
- BackgroundRB (http://backgroundrb.rubyforge.org)

Use ActiveRecord Mailing

CHAPTER 13

THE WRONG WAY

ActionMailer wraps several Ruby libraries with methods that make it simple to send email during an HTTP response. This works well for one-shot emails such as signup confirmations, password reminders, and order notifications.

It doesn't work as well when you need to send dozens, hundreds, or thousands of emails. The process of connecting to your email server and sending each email may cause Mongrel or FastCGI to timeout before all the emails are sent.

Things get worse if you are using a remote SMTP server such as Gmail. Sending 100 emails will be as slow as 100 sequential hits to the remote server.

THE RIGHT WAY

There are many ways to solve this with messaging and queuing systems. Fortunately, there is also a solution that is fine-tuned to the problem of sending email from ActionMailer. It's called `ar_mailer`.

Author Eric Hodel says:

> *The majority of time spent sending an email from Rails is in opening and closing the SMTP socket. I've seen 1000+ emails sent in a reasonable amount of time with ar_mailer.*

In addition, you get the ability to queue many emails and then send a limited number at a time. If your ISP limits your ability to send

email, this is the only way to stay under the limit.

Hendy Irawan of Ruby Inside argues that `ar_mailer` also makes the whole process more reliable since `ar_sendmail` will keep trying to send an email until a successful connection with the SMTP server has been made. By default, Rails assumes that the SMTP server is available 100% of the time, so only one attempt is made to send any piece of email. Read Hendy's article and comments at RubyInside (http://www.rubyinside.com/ar_mailer-batch-send-actionmailer-messages-517.html).

There are some drawbacks. While reviewing this book, Courtenay Gasking (http://caboo.se) observed that it's not possible to retrieve errors from emails sent with ar_mailer. With direct email delivery, you can report back to the user that they entered an incorrect email address, but ar_mailer gives no such feedback.

And, it's not synchronous. You'll have to live with the fact that some emails may be delayed by 60 seconds. In most cases, this isn't a big issue since email delivery is not always instant anyway.

USING IT

Assuming your application is already setup to send email, installing `ar_mailer` involves only a few steps. First, install the `ar_mailer` and `ar_mailer_generator` gems with this single command:

```
sudo gem install ar_mailer_generator
```

I wrote the `ar_mailer_generator` while writing this chapter in order to assist `ar_mailer` and save a few installation steps. This is the easiest way to generate the database migration and model files needed by the gem. The installation of the `ar_mailer` gem also installs the `ar_sendmail` script that will be used to send the emails in production, so you'll need to install it on your production server.

Run the generator inside your Rails application. This will copy the migration and model to your application:

```
ruby script/generate ar_mailer
```

Require the gem in `environment.rb`.

```
require 'action_mailer/ar_mailer'
```

Next, all your mailer classes need to inherit from `ActionMailer::ARMailer` instead of `ActionMailer::Base`.

```
class UserMailer < ActionMailer::ARMailer
```

You also need to make sure that all your mail methods have a `from` address specified. I set a constant in `environment.rb` and use that throughout so I don't have to repeat the same address over and over again.

```
def signup_confirmation(user)
  @subject    = "Thanks for signing up!"
  @body       = { "user" => user}
  @recipients = user.email
  @from       = PEEPCODE_EMAIL
end
```

At this point, you're ready to use `ar_mailer`, but your emails will still be sent immediately unless you tweak a setting in the proper environment. This is a feature, not a bug! Usually, you'll want to leave the `development` and `test` environments in their normal state so emails will not be queued.

In `production`, however, you'll want to set the `delivery_method` to `:activerecord`. You can do this in `environments/production.rb` with

EXPERIMENTAL: MULTIPLE QUEUES

Many third-party hosts restrict the number of emails you can send each day (two examples are shared hosts and Gmail).

The `ar_sendmail` command has a `batch-size` option, but this doesn't work well if you want to queue only some types of messages. For example, you probably want to send new users a confirmation immediately, but general announcements are less time-sensitive.

It would be nice to have a separate queue for non-urgent messages that could be sent in batches of a few hundred per day (Gmail restricts you to 500 per day, other shared hosts have a per-hour limit).

Fortunately, `ar_sendmail` is configurable and can look in any table for pending emails. For normal, urgent emails, use the defaults by following the instructions in this chapter. For non-urgent emails, write messages to a separate table and tell `ar_mailer` to look there instead.

First, create a new model and migration for the `batch_emails` table.

```
ruby script/generate model BatchEmail
```

Next, create the `batch_emails` table with the same fields as the `emails` table. The easiest way to do this is to copy the field definitions from the default `emails` table migration created while installing `ar_mailer`.

Finally, add a class method to your `Mailer` class.

```
##
# Queue an email in the BatchEmail table for sending
# by ar_sendmail. This makes it possible to send
# existing emails immediately, but queue others
# separately for sending in limited batches (in
# order to not go over an SMTP server's limit).

def self.deliver_delayed(mail_name, args=nil)
  mail = class_eval("create_#{mail_name}(*args)")
  mail.destinations.each do |destination|
    BatchEmail.create({
      :mail => mail.encoded,
      :to => destination,
      :from => mail.from.first
    })
  end
end
```

You don't have to change anything for normally queued emails. To use the delayed queued to send a `new_product` email, use this syntax:

```
ProductMailer.deliver_delayed(:new_product, user)
```

In a daily cron task, run this command:

```
ar_sendmail --batch-size 300 --table-name BatchEmail
```

this single line:

```
config.action_mailer.delivery_method = :activerecord
```

You can also put that same line in `environments/development.rb` if you want to try it out in the `development` environment.

After all that work, you're finally ready to send some email. Trigger the event in your application that sends email (or use the console to call your mailer's delivery method).

The `ar_sendmail` command manages the final delivery of email and uses the email server settings you've specified in `environment.rb`. You can see all the options by typing

```
ar_sendmail --help
```

The `mailq` argument will show the contents of the unsent mail queue. These are all the emails which have been sent by Rails but not delivered to the mail server yet.

```
ar_sendmail --mailq
```

You should see something like this:

```
-Queue ID- --Size-- ----Arrival Time---- -Sender/Recipient-
         1      658 Tue Aug 14 00:31:42  from@example.com
                                         to@example.com
-- 0 Kbytes in 1 Requests.
```

Finally, execute this command to actually deliver the messages to your mail server:

```
ar_sendmail --once
```

On your production server, you'll want to run that command frequently with `cron`. Alternately, you can run `ar_sendmail` in continuous daemon mode with the `daemonize` argument.

The ar_sendmail daemon has been known to go crazy from time to time. It's a good idea to monitor and restart it if you notice it taking up too much memory.

If you're using the Exception Notifier plugin, you'll need to edit `exception_notifier.rb` in the plugin. Change the superclass from `ActionMailer::Base` to `ActionMailer::ARMailer` just as you did for the other mailer classes.

RESOURCES

- ActionMailer (http://api.rubyonrails.com/classes/ActionMailer/Base.html)
- ar_mailer (http://seattlerb.rubyforge.org/ar_mailer)

Monitor Your Servers

CHAPTER 14

THE WRONG WAY

A web application is a constantly changing creature, but many developers run their applications in the dark.

How much memory is your application using right now? Is this more or less than average?

How much disk space remains on your server? How fast is it filling up?

What is your average load? What is the ratio of hits to misses on your memcached servers?

If an important process dies, will anyone notice?

THE RIGHT WAY

There are several levels of monitoring. A complete solution will use different tools to keep an eye on the webserver, the Mongrel application servers, the Rails application itself, the database server, and the operating system in general. The good news is that you can start with one or two of these tools, then add more as your site grows.

IMMEDIATE ERROR NOTIFICATION

The best place to start is with Jamis Buck's Exception Notifier. This is both immediate and specific. You'll receive an email when a 500 error occurs on your site with the name of the action and `params` that caused it.

Large sites have been known to see thousands of these emails after launching, so you may want to use Rick Olson's exception logger (http://svn.techno-weenie.net/projects/plugins/exception_logger) which logs to a database instead of sending email. However you choose to use it, it's an invaluable tool for discovering errors before your users report them.

Installation is quite simple. Install the plugin:

```
ruby script/plugin install http://svn.rubyonrails.org/rails/
plugins/exception_notification/
```

Include the module in your `ApplicationController`:

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

Finally, set a few variables to identify who should receive the messages:

```
# You could also store these in config.yml
# and access with the APP_CONFIG constant
# as mentioned in chapter 2.
ExceptionNotifier.exception_recipients = [
  'sysadmin@example.com',
  'boss@example.com'
]
ExceptionNotifier.sender_address = 'error@example.com'
ExceptionNotifier.email_prefix = "[SITE ERROR] "
```

One of the most useful features of the exception notifier is that it sends the actual parameters used to access the action that erred. In some cases this will be enough to recreate the error. I've even used this to recreate the error in a functional test (and then fix it).

```
* URL: http://videosite.com/downloads/123456789?s3=true
* Parameters: {"s3"=>"true", "action"=>"show",
"id"=>"123456789", "controller"=>"downloads"}
* Rails root: /var/www/apps/videosite.com/current
```

If the `params` aren't enough, at least you'll have some user-specific information that can be used to track down the cause of the error.

> Sometimes you'll go for a few days without receiving an error email. To ease the nerves, write a controller with one action that intentionally raises an error (for example, `raise "This error for experimentation only."`). Then you can trigger an error email just to make sure that the whole thing is working properly. As an exercise for the reader, protect this action with a `before_filter` so spiders and bots don't accidentally trigger it.

ACTION PERFORMANCE WITH PL_ANALYZE

The Robot Co-op's `pl_analyze` script is useful for tracking both long-term and action-specific trends. It parses the Rails production log and generates a report that shows the most popular controller actions and how long each request took to serve.

Eric Hodel says:

> *There's no sense trying to figure out what to optimize by mimicking usage when all that information is easily extractable from your logs. Use the production_log_analyzer gem to figure out what pages get hit the most and how much they cost and look at where they spend their time.*

Install it with

```
gem install production_log_analyzer
```

The `pl_analyze` script was originally written to work with the operating system's syslog logging facility. This is great if you have many machines and want to consolidate logs to a single machine for analysis, but requires a few setup steps and familiarity with your specific flavor of Unix. See the full instructions (http://seattlerb.rubyforge.org/SyslogLogger/classes/SyslogLogger.html) for setting it up.

I've also written a small library (http://nubyonrails.com/articles/a-hodel-3000-compliant-logger-for-the-rest-of-us) that can be used with single machines or systems that don't have a proper implementation of syslog. You can still analyze the output with `pl_analyze` and run the `pl_analyze` command against it.

To use my logger, add this line to `environments/production.rb`:

```
require 'hodel_3000_compliant_logger'
config.logger =
            Hodel3000CompliantLogger.new(config.log_path)
```

In either case, run the report with `pl_analyze`:

```
pl_analyze log/production.log
```

It also takes arguments for automatically sending the report to recipients by email. Call it without any arguments to see the options.

```
pl_analyze log/production.log -e recipient@example.com
```

The output will look something like this:

```
Request Times Summary:    Count   Avg     Min     Max
```

```
ALL REQUESTS:             5539    0.173   0.000   11.87

ProductsController#show:  1209    0.164   0.005   2.189
ProductsController#home:  863     0.096   0.058   0.487
ProductsController#index: 833     0.374   0.048   1.127
PagesController#show:     260     0.093   0.041   0.558
OrdersController#index:   172     0.328   0.009   1.003
SessionsController#new:   103     0.082   0.054   0.358


Slowest Request Times:
        IpnsController#create took 11.878s
        NotificationsController#create took 11.847s
        IpnsController#create took 5.832s
        NotificationsController#create took 4.388s
        NotificationsController#create took 4.328s
        ItemsController#redeem took 3.935s
        NotificationsController#create took 3.561s
        ItemsController#redeem took 3.259s
        ItemsController#redeem took 3.213s
        NotificationsController#create took 3.069s

-------------------------------------------------------
```

This report is so useful that you may get all the knowledge you need just from looking at one run of it. However, it's also useful to store the results in a database and chart performance over time. You can use the `Analyzer` class in the gem to parse the log and store the results. Here's a sample:

```
require 'production_log/analyzer'
a = Analyzer.new('log/production.log')
a.process

# See also a.db_times and a.render_times
a.request_times.map do |resource, times|
  next if resource.nil?
  # times is an array of floats for each
  # request to the resource.

  # Here is where you would save some values
```

```
    # to the database.
    puts "#{resource} => #{times.join(", ")}"
  end
```

The built-in Ruby `Enumerable` class gives you `min` and `max`. The gem adds `sum`, `average`, and `standard_deviation`. You can call these on the `times` returned by the `Analyzer` class and store them in a database each day with a rake task.

I've done something similar with my Mint plugin (http://topfunky.net/svn/plugins/mint), but used the PHP-based Mint application to display the results.

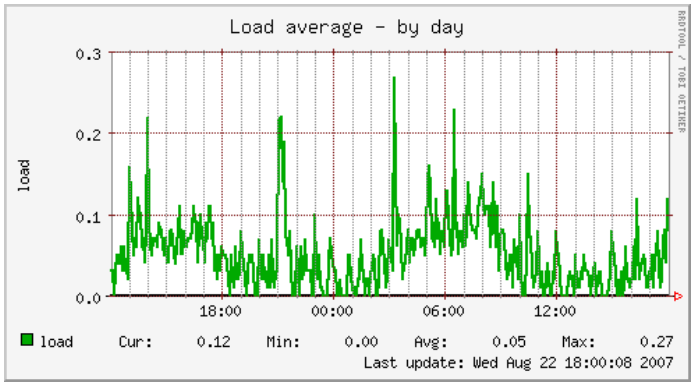
LONG TERM TRENDS

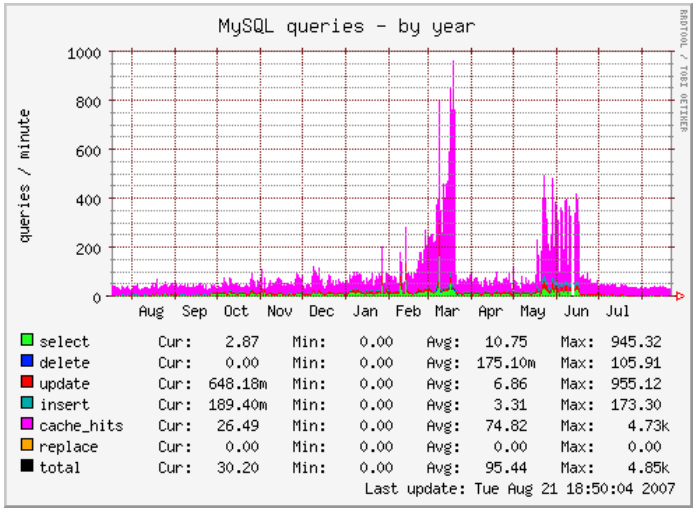
In addition to immediate errors and action-specific statistics, it's useful to have a tool that reports on the general health of your server. There are several open source and commercial products that can do this.

I run Munin (http://munin.projects.linpro.no) on all my servers. It's a tool that logs system statistics every few minutes and graphs them over time. You can choose from a basic suite of plugins or download third-party plugins. Munin provides informative graphs on disk usage, load, memory usage, firewall throughput, ping times, and other stats. It is useful for visualizing long-term trends and reacting to daily variations in a server's operation.

Installation is somewhat complicated, although there is a package for Debian (and Ubuntu). This article (http://www.howtoforge.com/server_monitoring_monit_munin) shows the steps for installing it on Debian.

The information you discover from viewing Munin won't tell you what the solution is to your performance problems. It may not even tell you what the problem is! However, it will give you a starting point and





a historical reference to judge by.

RESOURCES

- Exception Notifier (http://svn.rubyonrails.org/rails/plugins/exception_notification)
- Exception Logger (http://svn.techno-weenie.net/projects/plugins/exception_logger)
- dwatch (http://nubyonrails.com/articles/surviving-rails-1-1-with-server-monitoring)
- Munin (http://munin.projects.linpro.no)
- Installing Munin and Monit (http://www.howtoforge.com/server_monitoring_monit_munin)
- production_log_analyzer (http://seattlerb.rubyforge.org/production_log_analyzer)
- Mailing list message about pl_analyze (http://www.zenspider.com/pipermail/ruby/2007-March/003314.html)
- Hodel 3000 Compliant Logger (http://nubyonrails.com/articles/2007/01/03/a-hodel-3000-compliant-logger-for-the-rest-of-us)
- god (http://god.rubyforge.org)
- monit (http://www.tildeslash.com/monit)

Don't Cut Costs on Hardware

CHAPTER 15

THE WRONG WAY

Many startups try to cut costs by running on the cheapest hardware they can find. By some logic, it seems wasteful to pay for a quality server that is not being used by anyone outside of the development team.

Even after the site has been opened to the public, many startups take a "wait and see" approach, waiting to increase their server capacity until they have a large number of visitors.

What's wrong with this approach?

Waiting means you delay server issues until the worst possible time... the time that you have a sudden influx of traffic. Having early access to the operating system and hardware that your final site will run on means that you can find and fix issues during the development process.

Some startups have even observed that the number of new accounts correlates directly with the speed of the site. If you wait to add that stick of RAM until an extra 1,000 users signup, you may never get those 1,000 new users.

The monthly cost of a server seems expensive when considered alone, but how does it compare to your other costs?

Assume that your main developer is being paid $75/hour and also

performs some sysadmin work. Assume also that you are choosing between a bare-bones hosting plan that will cost you $10/month and a full-featured one that will cost $300 every month.

In this case, one month worth of hosting costs *less than one day's worth* of developer time. So if the full featured server would save your developer one day of work each month, you've covered your costs. Many quality hosts offer plans for even less than $100 per month, which makes this a no-brainer.

THE RIGHT WAY

After paying tens of thousands of dollars developing your site, you deserve quality hardware and support services to go along with it. A quality VPS will be able to increase your disk space, memory, bandwidth, or CPU cores with very little downtime.

If your site collects any kind of revenue or if you ever hope to, here are some minimal requirements:

- **Redundant disks**: RAID isn't a substitute for backups, but it does provide an extra level of redundancy if a single disk should fail. There are as many variations of RAID as there are levels of hell in *The Inferno*, but all are basically combinations of either mirroring, striping, or both. Mirroring gives redundancy and striping gives you speed.
- **Memory**: You should get at least 256 MB RAM and preferably 1 GB or more. Rails can be run with less memory, but it's better to have a little extra instead of trying to scrape by with barely enough.
- **Root Access**: A shared host is acceptable for a personal blog, but not for a production site. Get a virtual or dedicated server where you have full root access. Without root access, you'll be limited to what software you can install and run, which will negatively affect the architecture of your application and the growth of your site.

VIRTUALIZATION

Xen virtualization is pretty much the standard in VPS hosting. To many people, it's confusing why they would run several free-standing operating systems on one computer at the same time. There are a few benefits.

- **Upgrading is dead simple**. Need to add more RAM? Call your hosting provider and order it. Need to move to a different machine altogether? Just move the server image to the new hardware and you'll be up and running. There's no need to re-install all your software or reconfigure the operating system. Some sysadmins even choose to run single servers under Xen because of the ease of migration to new hardware.
- **Better use of resources**. A single server will spend most of its time in idle mode, with most of its memory unoccupied. Putting several server instances on a single piece of hardware will use the hardware more efficiently and reduce your costs, too. You can run your web server, database, and staging server on a single box. If you need to move each to its own machine, you'll be ready.

STAGING SERVER

Once your site goes live and is used by more than just your development team, you'll need a staging server. This will be used to try out new features in an environment that can interact with other web services. For example, PeepCode currently runs on a single server at RailsMachine (http://railsmachine.com) with 512 MB RAM and a RAID-10 disk array, but the staging server is a smaller VPS with only 256 MB RAM.

The staging server is configured with all the same software as the production server and even has its own third-party SSL certificate so it can send sample payments to the staging counterparts of PayPal and Google Checkout. Just today, I learned that a new version of the Nginx webserver will fix some problems I've been having. Instead

of rolling it out on the live site, I can try it on the staging server and verify that it works as expected.

The good news is that you don't have to buy multiple machines to accomplish this. You could buy a single server and split it into 3, 6, or even 12 virtual hosts. Each can have their own operating system and IP address. Then you'll be ready for the rush of traffic when it comes!

RESOURCES

- Nuby on Rails hosting article (http://nubyonrails.com/articles/the-host-with-the-most)
- Xen virtualization (http://www.xensource.com)

Test-Drive

CHAPTER 16

THE WRONG WAY

Most people know that they need to write tests, especially with Rails. The `test` directory sits there in the application's root directory. If you used the Rails generators to build your application, it may even have code in it!

I haven't met anyone who has blatantly deleted the `test` directory or who denies that tests provide a benefit, but I've met many people who leave it empty or never use it. There is usually a lingering sense of guilt. "Yeah, I know I should write tests...I'll get around to it eventually."

There's a world of difference between learning a new skill and unlearning bad habits. Test-first development is a skill (you have to know how to write expectations), but it's mostly the process of unlearning old habits that keeps people from doing it.

Some people try to learn new habits by force. They grit their teeth and start doing something they don't want to do in the hope that they will eventually like it. This rarely works! It's also the reason that people who write tests after they've written the implementation quickly tire of it. It's an auxiliary to their workflow, not a part of it.

THE RIGHT WAY

You *can* write tests, and it's OK to do it in the way that works best for you.

Test-first development didn't make sense to me until I started doing

it in a way that worked with how my mind thinks. I made it a part of by brainstorming process, not a parasite of it.

My advice is "Don't force it!" Learn to do test-first development in a way that works for you and provides a benefit to your workflow instead of being yet another thing to do.

Here are some situations you may want to start with.

START WITH SIMPLE SPIDERING

Courtenay Gasking wrote a plugin called spider_test (http://blog.caboo.se/articles/2007/2/21/the-fabulous-spider-fuzz-plugin). It's not a replacement for application-specific tests, but it will visit all the links in your site and attempt to submit values to forms, too. If you don't have any tests at all (or even if you have a few), it's a great place to start.

Install the plugin with from this url:

```
script/plugin install svn://caboo.se/plugins/court3nay/
spider_test
```

The instructions tell you to run a generator, but you can also just make a file called `test/integration/spider_test.rb` and copy these contents into it (this is the integration test file from PeepCode):

```
require "#{File.dirname(__FILE__)}/../test_helper"

class SpiderTest < ActionController::IntegrationTest

  fixtures :users, :products, :pages

  include Caboose::SpiderIntegrator

  def test_spider
    get '/'
    assert_response :success
```

```
    spider(@response.body, '/', {
      :verbose => true
    })
  end
end
```

> You need to load all the fixtures used by any of the `show` actions for pages on your site. Originally, I had a problem where the integration test would pass when run with the entire test suite, but fail when run alone. It turned out that my `:countries` fixture was being loaded by other tests and the data lingered in the database for the integration test. The solution was to find the missing fixture and load it explicitly in the integration tests that needed it.

You can run the test with the entire suite:

```
rake
```

Or alone

```
ruby test/integration/spider_test.rb
```

The `verbose` argument to the `spider` method will show you that it's actually doing something, and will list the pages that have been requested.

```
$ ruby test/integration/spider_test.rb

Loaded suite test/integration/spider_test

Spidering will ignore the following URLs ["/logout"]
```

```
. /pages/subscribe
. /pages/faq
. /pages/community
.....
Finished in 2.423417 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

START WITH BUGS

Consider these two facts:

- Testing happens. Either you test when you're developing a new feature, or your users test the site when they enter unexpected data and cause an error on the live site.
- Tests are meaningless unless they start with a failure.

Therefore, the best way to learn the value of testing is to write a failing test based on a bug that your users found on the site.

> If you're using the Exception Notification plugin, you'll get an error report with the parameters that were used to produce the error. Copy and paste these into a controller test (functional test) in order to recreate the bug. Often this isn't enough to fully recreate the bug since it may rely on user-specific data that is already in the database, but it's a good starting point.

First, write a test for the relevant controller or model. Run the tests until you see a failure (this shows that your test accurately describes the error condition).

Now, fix the bug. Run the test suite again and it should pass. If it doesn't, you haven't fixed the bug yet!

BRAINSTORM

I usually start the application design process on paper. I write down feature ideas or lists of what the application needs to do. This is a perfect start for a test suite!

rSpec (http://rspec.rubyforge.org) gives you the option to write an unimplemented example by listing an `it` string without a block. Sometimes I'll write 20 or 30 unimplemented examples in several files while I'm sketching out the beginnings of an application or a new feature.

```
it "should credit account after purchase"
```

When you run the examples, you'll see a report with unimplemented examples identified.

```
$ spec order_spec.rb

Order
- should credit account after purchase (NOT IMPLEMENTED)

Finished in 0.007886 seconds

1 example, 0 failures, 1 not implemented
```

For Test::Unit, Jay Fields posted a brief helper method (http://blog.jayfields.com/2006/09/testunit-test-creation.html) that makes brainstorming much easier. It's now available as the `dust` gem.

```
sudo gem install dust
```

Require it in your `test_helper.rb`:

```
require 'dust'
```

Now, you can write a test as a simple string, making it easier to write descriptive test names. While there is no tidy report, the unimplemented tests will still be present in the source.

```
test "should calculate total value of cart" do
  assert_equal @cart.total, 9.00
end
```

Another benefit of `dust` is name-checking. Ruby will let you write several methods with the same name, but `dust` will raise an error. I've occasionally copy-and-pasted method definitions without realizing that one was overwriting the other. That won't happen if you use `dust` for your tests.

USE AUTOTEST

Part of the pain of testing is running tests. Fortunately, other developers have had that same pain and have tried to alleviate it!

The `ZenTest` gem includes a tool called `autotest` that will run your tests whenever you make a change to a test or implementation file. Not only does this make it easier to run your tests more frequently, but it also makes them run faster. Autotest will run only the tests that relate to the current file being modified. When your test passes, it will run all tests for verification.

I've made a short movie (http://nubyonrails.com/articles/autotest-rails) of autotest in action. People have also enhanced it to send a Growl notification or play a sound upon completion.

There are times when I've needed something more fine-grained than `autotest`. For example, this PDF minibook is autogenerated with a Ruby script and I want to generate a copy anytime I save a source file. I run a script called `rstakeout` that can watch an arbitrary set of files and run an arbitrary command when any of them are modified.

The source and instructions for using rstakeout (http://nubyonrails.com/articles/automation-with-rstakeout) are detailed on my blog.

RESOURCES

- autotest (http://nubyonrails.com/articles/autotest-rails)
- ZenTest (http://zentest.rubyforge.org/ZenTest)
- spider_test plugin (http://blog.caboo.se/articles/2007/2/21/the-fabulous-spider-fuzz-plugin)
- rstakeout (http://nubyonrails.com/articles/automation-with-rstakeout)
- PeepCode Screencast on Test::Unit (http://peepcode.com/products/test-first-development)
- PeepCode Screencast on rSpec (http://peepcode.com/products/rspec-basics)

The Rest of the List

CHAPTER 17

Congratulations! You've made it this far.

Unfortunately, the previous chapters don't cover *everything* you need to know about building solid applications with Rails. Here is a short list of other topics you should look into.

- **Use database indexes** to speed up queries. Rails only indexes primary keys, so you'll have to find the spots that need more attention.
- **Profile your code**. The ruby-prof gem and plugin (http://cfis.savagexi.com/articles/category/ruby-prof) helped me make an application three times faster with only minimal changes to the code.
- **Minimize graphic-related dependencies**. If your application only needs to make a few thumbnails, don't waste memory by importing large graphics libraries. Look at mini-magick (http://rubyforge.org/projects/mini-magick) or image_science (http://seattlerb.rubyforge.org/ImageScience.html) for lightweight thumbnailing.
- **Avoid** excessive repeated rendering of **small partials**.
- **Use** CSS instead of inline tags to apply selective styling.
- Don't use ActiveRecord's `serialize` option to store **large objects** in database fields.
- Use `attr_protected :fieldname` in models to keep database fields from being manipulated from forms (or any calls to `Model.update_attributes(params[:model])`).
- Use Ruby **classes and inheritance** to refactor repeated controller code.
- Use **unobtrusive Javascripting** techniques to separate behavior

from markup.

- Package self-sufficient classes and modules as **plugins or RubyGems**.
- **Cache** frequently accessed data and rendered content where possible.
- Write **custom** Test::Unit **assertions** or rSpec **matchers** to help with debugging test suite errors.
- **Rotate** the Rails and Mongrel logfiles using the `logrotate` daemon on Linux.
- Build a reliable **backup** system.
- **Automate deployment** and maintenance with Capistrano or Vlad.
- Keep **method bodies short**. If a method is more than 10 lines long, it's time to break it down and refactor.
- **Run flog** to determine overly complex methods and clases.
- Don't use **too many conditionals**. Take advantage of `case` statements and Ruby objects to filter instead of multiply-nested `if` statements.
- **Don't be too clever**. Ruby has great metaprogramming features, but they are easy to overuse (such as `eval` and `method_missing`).
- Become familiar with the most **popular plugins**. Instead of re-implementing the wheel, save yourself some time by using well tested, popular plugins. Yes, you may be able to implement 80% of their features in 20% of time, but the last 20% of the features will take more time than you would have saved!

(Thanks to Courtenay Gasking (http://caboo.se) for many of these guidelines).

Revisions

CHAPTER 18

AUGUST 31, 2007

- First release

OCTOBER 24, 2007

- Finished unimplemented sections in *Skinny Controllers*, *Monitoring*, and *Testing*.

Every effort was made to provide accurate information in this document. However, Topfunky Corporation assumes no responsibility for any errors in the code or descriptions presented.

This document is available for US$9 at http://peepcode.com. Group discounts and site licenses can also be purchased by sending email to peepcode@topfunky.com.