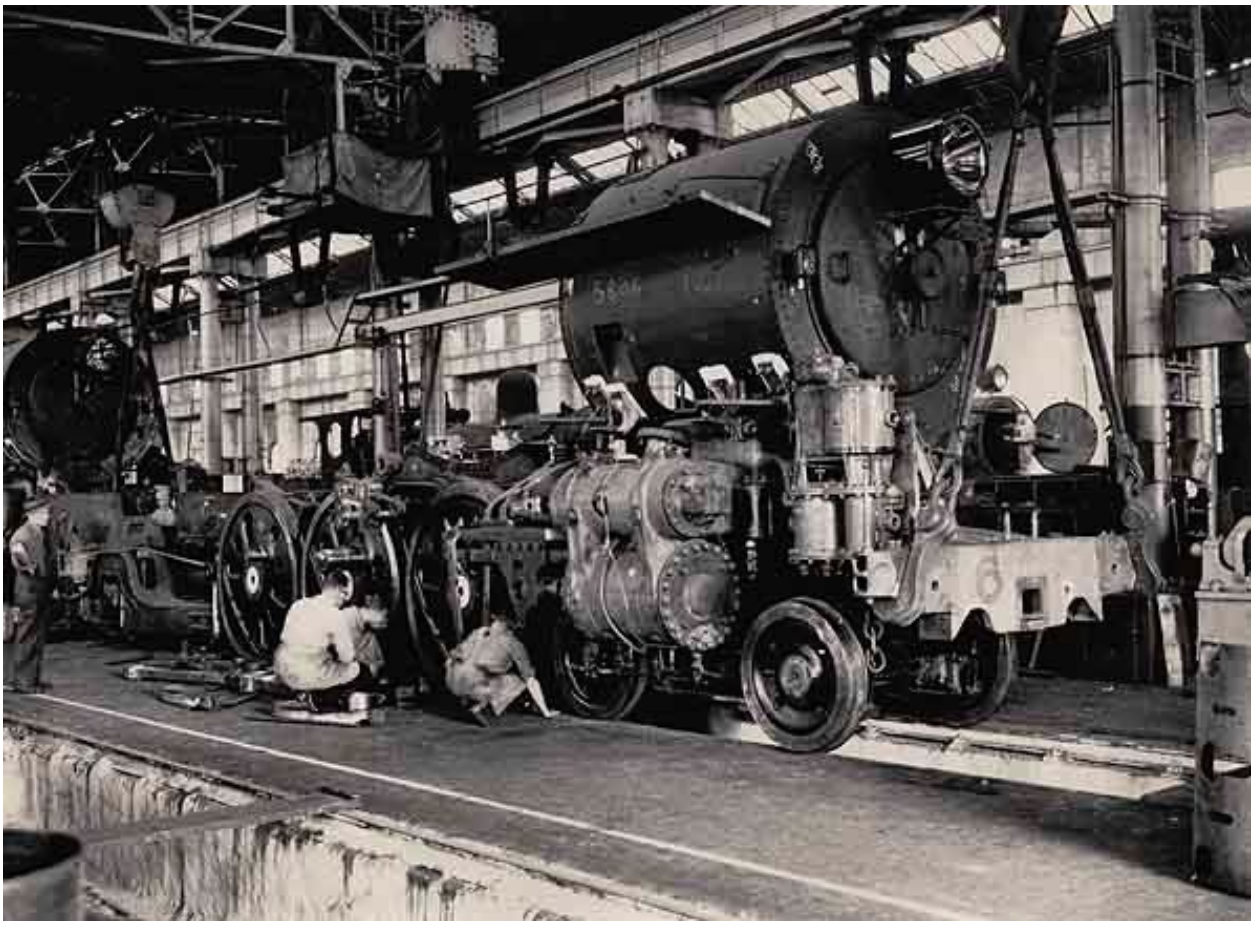
Rebuilding Rails

Get Your Hands Dirty and Build Your Own Ruby Web Framework



"Locomotive Construction", New South Wales State Records

Version: March 2020

0. Rebuilding Rails for Yourself

Why Rebuild Rails?

Knowing the deepest levels of any piece of software lets you master it. It's a special kind of competence you can't fake. You have to know it so well you could build it. What if you *did* build it? Wouldn't that be worth it?

This book will take you through building a Rails-like framework from an empty directory, using the same Ruby features and structures that make Rails so interesting.

Ruby on Rails is known for being "magical". A *lot* of that magic makes sense after you've built with those Ruby features.

Also, Ruby on Rails is an opinionated framework; the Rails team says so, loudly. What if you have different opinions? You'll build a Rails-like framework, but you'll have plenty of room to add your own features and make your own trade-offs.

Whether you want to master Rails exactly as it is or want to build your own personal version, this book can help.

Who Should Rebuild Rails?

You'll need to know some Ruby. If you've built several little Ruby apps or one medium-sized Rails app, you should be fine. If you consult the pickaxe book as you go along, that helps too ("https://ruby-doc.com/docs/ProgrammingRuby/"). You should be able to write basic Ruby without much trouble.

If you want to brush up on Rails, Michael Hartl's tutorials are excellent: "https://railstutorial.org/book". There's a free HTML version of them, or you can pay for PDF or screencasts. Some

concepts in this book are clearer if you already know them from Rails.

In most chapters, we'll use a little bit of Ruby magic. Each chapter will explain as we go along. None of these features are hard to understand. It's just surprising that Ruby lets you do it!

Working Through

Each chapter is about building a system in a Rails-like framework, which we'll call Rulers (like, Ruby on Rulers). Rulers is much simpler than Rails. But once you build the simple version, you'll know what the complicated version does and a lot of how it works.

Later chapters have a link to source code -- that's what book-standard Rulers looks like at the end of the previous chapter. You can download the source and work through any specific chapter you're curious about.

Late in each chapter are suggested features and exercises. They're easy to skip over, and they're optional. But you'll get much more out of this book if you stop after each chapter and think about what you want in your framework. What does Rails do that you want to know more about? What doesn't Rails do but you really want to? Does Sinatra have some awesome feature that Rails doesn't? The best features are the ones you care about!

Cheating

You can download next chapter's sample code from GitHub instead of typing chapter by chapter. You'll get a **lot** more out of the material if you type it yourself, make mistakes yourself and, yes, painstakingly debug it yourself. But if there's something you just can't get, use the sample code and move on. It'll be easier on your next time through the book.

It may take you more than one reading to get everything perfectly. Come back to code and exercises that are too much. Skip things but come back and work through them later. If the book's version is hard for you to get, go read the equivalent code in Rails, or in a simpler framework like Sinatra. Sometimes you'll need to see a concept explained in more than one way.

There are exercises at the end of each chapter. There are answers to the exercises near the end of the book.

At the end of the chapter are pointers into the Rails source for the Rails version of each system. Reading Rails source is optional. But even major components (ActiveRecord, ActionPack) are still around 25,000 lines - short and readable compared to most frameworks, with great test coverage. And generally you're looking for some specific smaller component, often between a hundred and a thousand lines.

You'll also be a better Rails programmer if you take the time to read good source code. Rails code is very rich in Ruby tricks and interesting examples of metaprogramming.

0.5 Getting Set Up

You'll need:

- Ruby
- a text editor
- a command-line or terminal
- Git (preferably)
- Bundler.
- SQLite, only for one later chapter... But it's a good one!

If you don't have them, you'll need to install them. This book contains an appendix with current instructions for doing so. Or you can install from source, from your favorite package manager, from RubyGems, or Google it and follow instructions.

Nothing here uses recently-added Ruby features, so any vaguely recent Ruby is great.

By "text editor" above, I specifically mean a programmer's editor. More specifically, I mean one that uses Unix-style newlines. On Windows this means a text editor with that feature such as Notepad++, Sublime Text or TextPad. On Unix or Mac it means any editor that can create a plain text file with standard newlines such as TextEdit, Sublime Text, AquaMacs, vim or TextMate.

I assume you type at a command line. That could be Terminal, xterm, Windows "cmd" or my personal favourite: iTerm2 for Mac. The command line is likely familiar to you as a Ruby developer. This book instructs in the command line because it is the most powerful way to develop. It's worth knowing.

It's possible to skip git in favour of different version control software (Mercurial, DARCS, Subversion, Perforce...). It's highly

recommended that you use some kind of version control. It should be in your fingers so deeply that you feel wrong when you program without version control. Git is *my* favourite, but use *your* favourite. The example text will all use git and you'll need it if you grab the (optional) sample code. If you "git pull" to update your sample repo, make sure to use "-f". I'm using a slightly weird system for the chapters and I may add commits out of order.

Bundler is just a Ruby gem -- you can install it with "gem install bundler". Gemfiles are another excellent habit to cultivate, and we'll use them throughout the book. Ruby 2.6 and higher include Bundler as part of Ruby.

SQLite is a simple SQL database stored in a local file on your computer. It's great for development, but please don't deploy on it. The lessons from it apply to nearly all SQL databases and adapters in one way or another, from MySQL and PostgreSQL to Oracle, VoltDB or JDBC. You'll want some recent version of SQLite 3. As I type this, the latest stable version is 3.7.11.

You may see minor differences in Ruby or Bundler output, depending on version. Small differences are to be expected: software changes frequently.

1. Zero to “It Works!”

Now that you’re set up, it’s time to start building. Like Rails, your framework will be a gem (a Ruby library) that an application can include and build on. Throughout the book, we’ll call our framework “Rulers”, as in “Ruby on Rulers”.

In the Rough

First create a new, empty gem:

```
$ bundle gem rulers
      create  rulers/Gemfile
      create  rulers/lib/rulers.rb
      create  rulers/lib/rulers/version.rb
      #...
Initializating git repo in src/rulers
Gem 'rulers' was successfully created. For more
information on making a RubyGem visit https://
bundler.io/guides/creating_gem.html
```

Rulers is a gem (a library), and so it declares its dependencies in rulers.gemspec. Open that file in your text editor. You can customize your name, the gem description and so on if you like. You can customize various sections like this:

```
# rulers.gemspec
spec.name        = "rulers"
spec.version     = Rulers::VERSION
spec.authors     = ["Singleton Ruby-Webster"]
```

```
spec.email       = ["webster@singleton-rw.org"]
spec.homepage    = ""
spec.summary     = %q{A Rack-based Web Framework}
spec.description = %q{A Rack-based Web Framework,
                      but with extra awesome.}
```

Traditionally the summary is like the description but shorter. The summary is normally about one line, while the description can go on for four or five lines.

Make sure to replace “FIXME” and “TODO” in the descriptions - “`gem build`” won't work as long as they're there.

There's also a section that starts with a comment about "Prevent pushing this gem to RubyGems.org". The whole purpose of the section is to prevent you from pushing your gem to RubyGems.org with a "rake push" command. You're just building a learning framework - and if it's named "rulers" then you can't push it to RubyGems anyway, because that name is taken. So you can just remove the whole section, both the "if" and "else" parts, rather than filling in all the "TODO" strings with real data.

In a bit, you’ll need to add a dependency at the bottom. They look roughly like this (***don't add these examples***):

```
spec.add_development_dependency "pry"
spec.add_runtime_dependency "rest-client"
spec.add_runtime_dependency "some_gem", "1.3.0"
spec.add_runtime_dependency "other_gem", ">0.8.2"
```

Each of these adds a runtime dependency (needed to run the gem at all) or a development dependency (needed to develop or test the gem). Now, add the following:

```
spec.add_runtime_dependency "rack"
```

(NOTE: this should be a ***runtime*** dependency, not a development dependency like the other ones in rulers.gemspec!)

Rack is a gem to interface your framework to a Ruby application server such as Thin, Puma, Passenger, WEBrick or Unicorn. An application server is a special type of web server that runs server applications, often in Ruby. In a real production environment you would run a web server like Apache or NGinX in front of the application servers. But in development we'll run a single application server and no more. Luckily an application server also works just fine as a web server.

We'll cover Rack in a lot more detail in the Controllers chapter, and again in the Middleware chapter. For now, you should know that Rack is how Ruby turns HTTP requests into code running on your server.

I'm also going to change the version number to 0.0.1 - I have a sentimental attachment to it. To do that, open the file rulers/lib/rulers/version.rb and change the version from 0.1.0 to 0.0.1:

```
# rulers/lib/rulers/version.rb
module Rulers
  VERSION = "0.0.1"
end
```

Let's build your gem and install it:

```
> gem build rulers.gemspec
```

```
> gem install rulers-0.0.1.gem
```

Did it fail, complaining about invalid links, invalid URLs or TODOs? Go back into the gemspec and fix all the boilerplate entries like "TODO: Put your gem's website or public repo URL here" - replace them with the real thing. For "homepage" you're allowed to just use the empty string.

Eventually we'll use your gem from the development directory with a Bundler trick. But for now we'll do it the simple way - build and install the gem locally after each change. Repeating that technique will get it in your fingers. It's always good to know the simplest way to do a task -- you can fall back to it when clever tricks aren't working.

Be careful - if you type "bundle install" without the Rulers gem already installed, you may get the version of Rulers (or whatever you called your library) from RubyGems.org! In a later chapter we'll add a trick to fix that. But for now, remember that you'll need to install Rulers manually, and that typing "bundle install" isn't your friend.

Hello World, More or Less

Rails is a library like the one you just built. But what application will you run *with* your framework? We'll start a very simple app where you submit favourite quotes and users can rate them. Rails would use a generator for this ("rails new best_quotes"), but we're going to do it manually.

Make a directory and some subdirectories:

```
> mkdir best_quotes
> cd best_quotes
```

```
> git init
Initialized empty Git repository in src/
best_quotes/.git/
> mkdir config
> mkdir app
```

You'll also want to make sure to use your library. Add a Gemfile:

```
# best_quotes/Gemfile
source 'https://rubygems.org'
gem "rulers"  # Your gem name
```

Then run "bundle install" to create a Gemfile.lock and make sure all dependencies are available.

We'll build from a trivial rack application. Create a config.ru file:

```
# best_quotes/config.ru
run proc {
  [200, {'Content-Type' => 'text/html'},
    ["Hello, world!"]]
}
```

Rack's "run" means "call that object for each request". In this case the proc returns success (200) and "Hello, world!" along with the HTTP header to make your browser display HTML.

Now you have a simple application which shows "Hello, world!" You can start it up by typing "rackup -p 3001" and then pointing a web browser to "http://localhost:3001". You should see the text "Hello, world!" which comes from your config.ru file.

(Problems? If you can't find the rackup command, make sure you updated your PATH environment variable to include the gems directory, back when you were installing Ruby and various gems! A ruby manager like rvm or rbenv can do this for you. Also, make sure that the "rack" dependency in rulers.gemspec is a runtime dependency, not a development dependency!)

Making Rulers Use Rack

In your Rulers directory, open up lib/rulers.rb. Change it to the following:

```
# rulers/lib/rulers.rb
require "rulers/version"

module Rulers
  class Application
    def call(env)
      [200, {'Content-Type' => 'text/html'},
        ["Hello from Ruby on Rulers!"]]
    end
  end
end
```

Build the gem again and install it (gem build rulers.gemspec; gem install rulers-0.0.1.gem). Now change into your application directory, best_quotes.

Now you can use the Rulers::Application class. Under best_quotes/config, create a new file application.rb and add the following:

```
# best_quotes/config/application.rb
require "rulers"

module BestQuotes
  class Application < Rulers::Application
  end
end
```

The "BestQuotes" application object should use your Rulers framework and show “Hello from Ruby on Rulers” when you use it. To use it, open up your config.ru, and change it to say:

```
# best_quotes/config.ru
require './config/application'
run BestQuotes::Application.new
```

Now when you type "rackup -p 3001" and point your browser to "http://localhost:3001", you should see "Hello from Ruby on Rulers!". You've made an application and it's using your framework!

Review

In this chapter, you created a reusable Ruby library as a gem. You included your gem into a sample application. You also set up a simple Rack application that you can build on using a Rackup file, config.ru. You learned the very basics of Rack, and hooked all these things together so that they're all working.

From here on out you'll be adding and tweaking. But this chapter was the only time you start from a blank slate and create something from nothing. Take a bow!

In Rails

By default Rails includes many reusable gems. The actual "Rails" gem contains very little code. Instead, it delegates to the supporting gems. Rails itself just *ties* them together. So the “railties” gem is glue between all those components - so Rails doesn't even really tie them together. That's what railties does!.

The Rails command allows you to change many of its components - you can specify a different ORM than ActiveRecord, a different testing library, a different Ruby template library or a different JavaScript library. So the components below aren’t always 100% required for applications that customize heavily. Curious what you can customise? Type "rails --help" to check.

Below are the basic Rails gems — the declared dependencies of Rails itself.

- ActiveSupport is a compatibility library including methods that aren't necessarily specific to serving web applications. You'll see ActiveSupport used by non-Rails, non-web libraries because it contains such a lot of useful baseline functionality. ActiveSupport includes methods for changing words from single to plural, or CamelCase to snake_case. It also includes significantly better time and date support than the Ruby standard library.
- ActiveModel is how Rails handles persistence and models, but doesn’t require that the persistence use a database. For instance, if you want a URL for a given model, ActiveModel helps you there, even if the model is in memory, on disk or in non-SQL storage like Redis.
- ActiveRecord is an Object-Relational Mapper (ORM). That means that it maps between Ruby objects and tables in a

SQL database. When you query from or write to the SQL database in Rails, you do it through ActiveRecord. ActiveRecord also implements ActiveModel. ActiveRecord supports PostgreSQL, MySQL and SQLite, plus JDBC, Oracle and many others.

- ActionPack does routing - the mapping of an incoming URL to a controller action in Rails. It also sets up your controllers and views, and shepherds a request through its controller action. ActionPack uses Rack quite a bit.
- ActionView renders template files, which eventually become the final HTML. The template rendering is done through an external gem like Erubis (for Erb) or Haml. ActionView also handles action- or view-centred functionality like view caching.
- ActionMailer is used to send out email, especially email based on templates. It works a lot like you'd hope Rails email would, with controllers, actions and views - it's just that the views are email, not web content.
- ActiveJob is for job queueing. Not everything in your web app can or should be done instantly in response to an HTML request. Slow batch jobs, sending email and running long command-line processes all want to be done separately from your web server. ActiveJob is a compatibility layer around many other gems for this purpose such as Resque, Sidekiq or DelayedJob.
- ActionCable sets up a persistent connection between a rendered web page and your server. “Classic” HTTP requests are transactional - your browser requests a page, your server sends it back and you’re done. HTTP extensions like AJAX, Server-Sent Events (SSEs) and WebSockets blur this line by letting your rendered page keep communicating with the server after they’re rendered. It’s used for realtime data

updates, chat servers, event polling and many other things. ActionCable manages this multi-request connection for your Rails app.

Some of what you built in this chapter was in your application directory, not in Rulers. Go ahead and make a new Rails app - type "rails new test_app". If you look in config/application.rb, you'll see Rails setting up a Rails Application object, a lot like your Rulers Application object. You'll also see Rails' config.ru file, which looks a lot like yours. Right now is a good time to poke through the config directory and see what a Rails application sets up for you by default. Do you see anything that now makes more sense?

Exercises

Exercise One: Reloading Rulers

Let's add a bit of debugging to the Rulers framework.

```
# rulers/lib/rulers.rb
module Rulers
  class Application
    def call(env)
      `echo debug > debug.txt`;
      [200, {'Content-Type' => 'text/html'},
        ["Hello from Ruby on Rulers!"]]
    end
  end
end
```

When this executes, it should create a new file called debug.txt in best_quotes, where you ran rackup .

Try restarting your server and reloading your browser at "http://localhost:3001". But you won't see the debug file!

Try rebuilding the gem and reinstalling it (gem build rulers; gem install rulers-0.0.1.gem). Reload the browser. You still won't see it. Finally, restart the server again and reload the browser. Now you should finally see that debug file.

Rails and Rulers can both be hard to debug. In chapter 3 we'll look at Bundler's :path option as a way to make it easier. For now you'll need to reinstall the gem and restart the rack server before your new Rulers code gets executed. When the various conveniences fail, you'll know how to do it the old-fashioned way.

Exercise Two: Your Library's Library

You can begin a simple library of reusable functions, just like ActiveSupport. When an application uses your Rulers gem and then requires it ("require rulers"), the application will automatically get any methods in that file. Try adding a new file called lib/rulers/array.rb, with the following:

```
# rulers/lib/rulers/array.rb
class Array
  def sum(start = 0)
    inject(start, &:+)
  end
end
```

Have you seen "&:+" before? It's a fun trick. ":+" means "the symbol +" just like ":foo" means "the symbol foo." The "&" means "pass as a block" -- the code block in curly-braces that usually comes after. So you're passing a symbol as if it were a block. Ruby knows to convert a symbol into a proc that calls the method of the same name. When you do that with "plus", you get "add these together" since that's what the method named "+" does.

Now add "`require "rulers/array"`" to the top of lib/rulers.rb. That will include it in all Rulers apps.

You'll need to go into the rulers directory and "git add ." before you rebuild the gem (git add .; gem build rulers.gemspec; gem install rulers-0.0.1.gem). That's because rulers.gemspec is actually calling git to find out what files to include in your gem. Have a look at this line from rulers.gemspec:

```
spec.files = `git ls-files -z`.split("\x0")
```

“git ls-files” will only show files git knows about -- the split is just to get the individual lines of output. If you create a new file, be sure to “git add .” before you rebuild the gem or you won’t see it!

Now with your new rulers/array.rb file, any application including Rulers will be able to write `[1, 2, 37, 9].sum` and get the sum of the array. Go ahead, add a few more methods that could be useful to the applications that will use your framework.

Exercise Three: Test Early, Test Often

Since we’re building a Rack app, the rack-test gem is a convenient way to test it. Let’s do that.

Add rack-test as a development (not runtime) dependency to your gemspec. If Minitest isn't there already, add that too:

```
# rulers/rulers.gemspec, near the bottom
  # ...
  gem.add_runtime_dependency "rack"
  gem.add_development_dependency "rack-test"
  gem.add_development_dependency "minitest"
end
```

Why use the Gemspec when you have a Gemfile? The gemspec is taken into account by apps and libraries that depend on your gem, so it’s necessary if other apps use your library. It's also where people look, for gems, since most of the dependencies *have* to be in the gemspec.

Now run “bundle install” to make sure you’ve installed rack-test. We’ll add one usable test for Rulers. Later you’ll write more.

Make a test directory:

```
# From rulers directory
> mkdir test
```

Now we'll create a test helper:

```
# rulers/test/test_helper.rb
$LOAD_PATH.unshift File.expand_path("../../lib",
                                    __FILE__)
require "rulers"
require "rack/test"

require "minitest/autorun"
```

The only surprising thing here should be the $LOAD_PATH magic. It makes sure that requiring "rulers" will require the local one in the current directory rather than, say, the one you installed as a gem. It does that by unshifting (prepending) the local path so it's checked before anything else in $LOAD_PATH.

We also do an expand_path so that it's an absolute, not a relative path. That's important if anything might change the current directory.

Testing a different local change to a gem you have installed can be annoying -- what do you have installed? What's being used? By explicitly prepending to the load path, you can be sure that the local not-necessarily-installed version of the code is used *first* and it doesn't matter what version you have installed.

Now you'll need a test, which we'll put in application_test.rb:

```
# rulers/test/application_test.rb
require_relative "test_helper"

class TestApp < Rulers::Application
end

class RulersAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    TestApp.new
  end

  def test_request
    get "/"

    assert last_response.ok?
    body = last_response.body
    assert body["Hello"]
  end

end
```

The require_relative just means “require, but check from this file’s directory, not the load path”. It’s a fun, simple trick.

This test creates a new TestApplication class, creates an instance, and then gets “/” on that instance. It checks for error with “last_response.ok?” and that the body text contains “Hello”.

To run the test, type “ruby test/application_test.rb”. It should run the test and display a message like this:

```
# Running tests:

.

Finished tests in 0.007869s, 127.0810 tests/s,
254.1619 assertions/s.

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

The line “get “/”” above can be “post “/my/url”” if you prefer, or any other HTTP method and URL.

Now, write at least one more test.

Exercise Four: Other Application Servers

When you run “rackup,” you’re seeing “WEBRick” in the output. That's the name of Ruby’s built-in web server. It’s not something you’d want to use in production, but it’s kind of cool that it’s there automatically.

For a real application in production, you’ll use a real application server, plus NGinX or Apache set up as a reverse proxy.

A “real” application server would be something like Passenger, Puma, Unicorn or Thin. All of them use Rackup files just like you have been here. For instance, to run Unicorn with your code, install Unicorn (“gem install unicorn”) and then run it:

```
# At the console:
unicorn -p 3001
```

Like “rackup”, Unicorn will automatically look for config.ru and you can tell it what port number to use. The other application servers are similar — if you set them up, they know how to use a config.ru file without a problem.

From here on out, if I tell you to run “rackup” you can install and use an app server of your choice. Everything in this book should work just fine with any application server you choose.

Exercise Five: Ignoring Files

When you ran "bundle gem rulers", it provided a reasonable .gitignore. It doesn't have absolutely everything you need, but it's not bad.

However, when you build gems, you get a .gem file that Git wants to check in. Ordinarily *you* do *not* want to check it in.

The best way to handle that is to add a line to the end of the .gitignore file, telling Git not to add it. For me, that line might look like:

```
rulers-*.gem
```

Add a line to the .gitignore file to keep yourself from accidentally checking in .gem files.

2. Your First Controller

In this chapter you'll write your very first controller and start to see how Rails routes a request.

You already have a very basic gem and application, and the gem is installed locally. If you don't, or if you don't like the code you wrote in the first chapter, you can download the sample source.

We'll bump up the gem version by 1 for every chapter of the book. If you're building the code on your own, you can do this or not.

To change the gem version, open up rulers/lib/rulers/version.rb and change the constant from "0.0.1" to "0.0.2". Next time you reinstall your gem, you'll need to type "gem build rulers.gemspec; gem install rulers-0.0.2.gem". You should delete rulers/rulers-0.0.1.gem, just so you don't install and run old code by mistake.

You may also need to "bundle update rulers" in best_quotes.

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_2_mine chapter_2" to create a new branch called "chapter_2_mine" for your commits.

On the Rack

Last chapter's big return values for Rack can take some explaining. So let's do that. Here's one:

```
[200, {'Content-Type' => 'text/html'},
  [ "Hello!" ]]
```

Let's break that down. The first number, 200, is the HTTP status code. If you returned 404 then the web browser would show a 404 message -- page not found. If you returned 500, the browser should say that there was a server error.

The next hash is the headers. You can return all sorts of headers to set cookies, change security settings and many other things. The important one for us right now is 'Content-Type', which must be 'text/html'. That just lets the browser know that we want the page rendered as HTML rather than text, JSON, XML, RSS or something else.

And finally, there's the content. In this case we have only a single part containing a string. So the browser would show "Hello!"

Soon we'll examine Rack's "env" object, which is a hash of interesting values. For now all you need to know is that one of those values is called PATH_INFO, and it's the part of the URL after the server name but minus the query parameters, if any. That's the part of the URL that tells a Rails application what controller and action to use.

Routing Around

A request arrives at your web server or application server. Rack passes it through to your code. Rulers will need to route the request -- that means it takes the URL from that request and answers the question, "what controller and what action handle this request?" We're going to start with very simple routing.

Specifically, we're going to start with what was once Rails' default routing. URLs of the form "http://host.com/category/action" will be routed to CategoryController#action.

Under "rulers", open lib/rulers.rb.

```
# rulers/lib/rulers.rb
require "rulers/version"
require "rulers/routing"

module Rulers
  class Application
    def call(env)
      klass, act = get_controller_and_action(env)
      controller = klass.new(env)
      text = controller.send(act)
      [200, {'Content-Type' => 'text/html'},
        [text]]
    end
  end

  class Controller
    def initialize(env)
      @env = env
    end
```

```
    def env
      @env
    end
  end
end
```

Our Application#call method is now getting a controller and action from the URL and then making a new controller and sending it the action. With a URL like "http://mysite.com/people/create", you'd hope to get PeopleController for klass and "create" for the action. We'll make that happen in rulers/routing.rb, below.

The controller just saves the environment we gave it. We'll use it later.

Now in lib/rulers/routing.rb:

```
# rulers/lib/rulers/routing.rb
module Rulers
  class Application
    def get_controller_and_action(env)
      _, cont, action, after =
        env["PATH_INFO"].split('/', 4)
      cont = cont.capitalize # "People"
      cont += "Controller" # "PeopleController"

      [Object.const_get(cont), action]
    end
  end
end
```

This is very simple routing, so we'll just get a controller and action as simply as possible. We split the URL on "/". The "4" just means

“split no more than 4 times”. So the split assigns an empty string to “_” from before the first slash, then the controller, then the action, and then everything else un-split in one lump. For now we throw away everything after the second “/” - but it’s still in the environment, so it’s not really gone.

The method “const_get” is a piece of Ruby magic - it just means look up any name starting with a capital letter - in this case, your controller class.

Also, you’ll sometimes see the underscore used to mean “a value I don’t care about”, as I do above. It’s actually a normal variable and you can use it however you like, but many Rubyists like to use it to mean “something I’m ignoring or don’t want.”

Now you’ll make a controller in best_quotes. Under app/controllers, make a file called quotes_controller.rb:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def a_quote
    "There is nothing either good or bad " +
      "but thinking makes it so."
  end
end
```

That looks like a decent controller, even if it’s not quite like Rails. You’ll need to add it to the application manually since you haven’t added magic Rails-style autoloading for your controllers yet. So open up best_quotes/config/application.rb. You’re going to add the following lines after “require 'rulers'” and before declaring your app:

```
# best_quotes/config/application.rb (excerpt)
$LOAD_PATH << File.join(File.dirname(__FILE__),
                        "..", "app",
                        "controllers")
require "quotes_controller"
```

The LOAD_PATH line lets you load files out of “app/controllers” just by requiring their name, as Rails does. And then you require your new controller.

Now, go to the rulers directory and type “git add .; gem build rulers.gemspec; gem install rulers-0.0.2.gem”. Then under best_quotes, type “rackup -p 3001”. Finally, open your browser to “http://localhost:3001/quotes/a_quote”.

If you did everything right, you should see a quote from Hamlet. And you’re also seeing the very first action of your very first controller.

If you didn’t quite get it, please make sure to include “quotes/a_quote” in the URL, like you see above -- just going to the root no longer works. If you see "Uninitialised constant Controller" then your URL is probably off.

It Almost Worked!

Now, have a look at the console where you ran rackup. Look up the screen. See that error? It’s possible you won’t on some browsers, but it’s likely you have an error like this:

```
NameError: wrong constant name
Favicon.icoController
  .../gems/rulers-0.0.3/lib/rulers/routing.rb:9:in
`const_get'
```

```
  .../gems/rulers-0.0.3/lib/rulers/routing.rb:9:in
`get_controller_and_action'
  .../gems/rulers-0.0.3/lib/rulers.rb:7:in `call'
  .../gems/rack-1.4.1/lib/rack/lint.rb:48:in
`_call'
  (...more lines...)
127.0.0.1 - - [21/Feb/2012 19:46:51] "GET /
favicon.ico HTTP/1.1" 500 42221 0.0092
```

You're looking at an error from the browser fetching a file... Hm... Check that last line... Yup, favicon.ico. Most browsers do this automatically. Eventually we'll have our framework or our web server take care of serving static files like this. But for now, we'll cheat horribly.

Open up rulers/lib/rulers.rb, and have a look at Rulers::Application#call. We can just check explicitly for PATH_INFO being /favicon.ico and return a 404:

```
# rulers/lib/rulers.rb
module Rulers
  class Application
    def call(env)
      if env['PATH_INFO'] == '/favicon.ico'
        return [404,
          {'Content-Type' => 'text/html'}, []]
      end

      klass, act = get_controller_and_action(env)
      controller = klass.new(env)
      text = controller.send(act)
```

```
      [200, {'Content-Type' => 'text/html'},
        [text]]
    end
  end
```

A horrible hack? Definitely. For now, that will let you see your *real* errors without gumming up your terminal with unneeded ones.

Review

You've just set up very basic routing, and a controller action that you can route to. If you add more controller actions, you get more routes. Rulers 0.0.2 would be just barely enough to set up an extremely simple web site. We'll add much more as we go along.

You've learned a little more about Rack -- see the "Rails" section of this chapter for even more. You've also seen a little bit of Rails magic with LOAD_PATH and const_get, both of which we'll see more of later.

Exercises

Exercise One: Debugging the Rack Environment

Open app/controllers/quotes_controller.rb, and change it to this:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def a_quote
    "There is nothing either good or bad " +
      "but thinking makes it so." +
      "\n<pre>\n#{env}\n</pre>"
  end
```

```
end
```

Now restart the server -- you don't need to rebuild the gem if you just change the application. Reload the browser, and you should see a big hash table full of interesting information. It should look very roughly like this:

```
{"GATEWAY_INTERFACE"=>"CGI/1.1", "PATH_INFO"=>"/
quotes/a_quote", "QUERY_STRING"=>"",
"REMOTE_ADDR"=>"127.0.0.1",
"REMOTE_HOST"=>"localhost",
"REQUEST_METHOD"=>"GET", "REQUEST_URI"=>"http://
localhost:3001/quotes/a_quote", "SCRIPT_NAME"=>"",
"SERVER_NAME"=>"localhost", "SERVER_PORT"=>"3001",
"SERVER_PROTOCOL"=>"HTTP/1.1",
"SERVER_SOFTWARE"=>"WEBrick/1.3.1 (Ruby/
1.9.3/2012-11-10)", "HTTP_HOST"=>"localhost:3001",
"HTTP_CONNECTION"=>"keep-alive",
"HTTP_CACHE_CONTROL"=>"max-age=0",
"HTTP_USER_AGENT"=>"Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_6_8) AppleWebKit/537.11 (KHTML, like
Gecko) Chrome/23.0.1271.64 Safari/537.11",
"HTTP_ACCEPT"=>"text/html,application/
xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
"HTTP_ACCEPT_ENCODING"=>"gzip,deflate,sdch",
"HTTP_ACCEPT_LANGUAGE"=>"en-US,en;q=0.8",
"HTTP_ACCEPT_CHARSET"=>"ISO-8859-1,utf-8;q=0.7,*;q=
0.3", "rack.version"=>[1, 1], "rack.input"=>#>,
"rack.errors"=>#>>, "rack.multithread"=>true,
"rack.multiprocess"=>false, "rack.run_once"=>false,
"rack.url_scheme"=>"http", "HTTP_VERSION"=>"HTTP/
1.1", "REQUEST_PATH"=>"/quotes/a_quote"}
```

That looks like a lot, doesn't it? It's everything your application gets from Rack. When your Rails controller uses accessors like "post?", it's checking the Rack environment to figure that out. You could easily add your own "post?" method to Rulers by checking whether env["REQUEST_METHOD"] == "POST".

Better yet, you can now see everything that Rails has to work with. Everything that Rails knows about the request is extracted from this same hash.

Exercise Two: Debugging Exceptions

Let's add a new action to our controller that raises an exception:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def exception
    raise "It's a bad one!"
  end
end
```

Re-run rackup. Go to "http://localhost:3001/quotes/exception" in your browser, which should raise an exception. You should see a prettily-formatted page saying there was a RuntimeError at /quotes/exception. The page should also have a big stack trace.

In chapter 8 we'll look deeply into Rack middleware and why you're seeing that. That page isn't built into your browser. You can turn it off by setting RACK_ENV to production in your environment. It's a development-only Rack debugging tool that you're benefiting from.

However, you don’t have to use it. You could add a begin/rescue/end block in your Rulers application request and then decide what to do with exceptions. Then Rack wouldn't do it for you.

Go into rulers/lib/rulers.rb and in your call method, add a begin/rescue/end around the controller.send() call. Now you have to decide what to do if an exception is raised -- you can start with a simple error page, or a custom 500 page, or whatever you like. Go to the page again in your browser and make sure you see the page you just added.

What else can your framework do with errors?

Exercise Three: Roots and Routes

It’s inconvenient that you can’t just go to “http://localhost:3001” any more and see if things are working. Getting an exception doesn’t tell you if *you* broke anything.

Open rulers/lib/rulers.rb in your text editor. Beneath the check for favicon.ico, you can add a check to see if PATH_INFO is just “/”.

First, return “/quotes/a_quote” if PATH_INFO is "/". Test in your browser. Then remove that, and instead try one of the following:

- Return the contents of a known file -- maybe public/index.html?
- Look for a HomeController and its index action.
- Extra credit: try a browser redirect. This requires returning a code other than 200 or 404 and setting some headers.

In chapter 9 we’ll build a much more configurable router, more like how Rails does it. Until then, you'll have a few hacks built into your framework.

In Rails

ActionPack in Rails includes the controllers. Rails also has an ApplicationController which inherits from its controller base class, and then each individual controller inherits from that. Your framework could do that too!

Different Rails versions had substantially different default routing. You can read about the current one in "*Rails Routing from the Outside In*": "http://guides.rubyonrails.org/routing.html". Your current routing is similar to old-style Rails 1 and 2 routing. Those Rails versions would automatically look up a controller and action without you specifying the individual routes. That's not great security, but it's very friendly to beginners just picking up your framework. Recent Rails routes aren't quite the same - they make you declare everything explicitly.

Rails encapsulates the Rack information into a "request" object rather than just including the hash right into the controller. That's a good idea when you want to abstract it a bit -- normalise values for certain variables, for instance, or read and set cookies to store session data. In chapter 6 you'll see how to make Rulers do it too. Rails also uses Rack under the hood, so it's doing it the same way you are.

Rack is a simple CGI-like interface. There's less to it than you'd think. If you're curious, have a look at the Rack spec at "http://rack.rubyforge.org/doc/SPEC.html" for all the details. It's a little hard to read, but it can tell you everything you need to know.

You'll learn more about Rack as we go along, especially in chapters 6 and 9. But for the impatient, Rails includes a specific guide to how it uses Rack: "http://guides.rubyonrails.org/rails_on_rack.html". It's full of things you can add to your own framework. Some of those tricks will be added in later chapters of this book.

3. Rails Automatic Loading

If you've used Rails much, it probably struck you as odd that you had to require your controller file. And that you had to restart the server repeatedly. In fact, all kinds of "too manual." What's up with that?

Rails loads files for you when it sees something it thinks it recognises. If it sees BoboController and doesn't have one yet it loads "app/controllers/bobo_controller.rb". We're going to implement that in Rulers in this chapter.

You may already know about Ruby's method_missing. When you call a method that doesn't exist on an object, Ruby tries calling "method_missing" instead. That lets you make methods with unusual names that don't explicitly exist in a .rb file.

It turns out that Ruby *also* has const_missing, which does the same thing for constants that don't exist. Class names in Ruby are just constants. Hmm...

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in BOTH rulers and best_quotes type "git checkout -b chapter_3_mine chapter_3" to create a new branch called "chapter_3_mine" for your commits.

Where's My Constant?

First, let's see how const_missing works.

Try putting this into a file called const_missing.rb and running it:

```
# some_directory/const_missing.rb
class Object
  def self.const_missing(c)
    STDERR.puts "Missing constant: #{c.inspect}!"
  end
end

Bobo
```

When you run it, you should see “Missing constant: :Bobo”. So that that means Bobo was used but not loaded. That seems promising. But we still get an error.

By the way -- I'll use "STDERR.puts" repeatedly instead of just "puts". When debugging or printing error messages I like to use STDERR because it’s a bit harder to redirect than a normal “puts”. You’re more likely to see your message even when using a log file, background process or similar.

You’ll also see a lot of “inspect” in my code. For simple structures, “inspect” shows them exactly as you’d type them into Ruby -- strings with quotes, numbers bare, symbols with a leading colon and so on. It’s great to train yourself to use "STDERR.puts" and “inspect” every time you’re debugging. When you have a problem where something is the wrong type, inspect will show you exactly what’s wrong. And STDERR.puts will make sure you always see the message. Make them a habit now!

Try creating another file, this one called bobo.rb, next to const_missing.rb. It should look like this:

```
# some_directory/bobo.rb
class Bobo
  def print_bobo
    puts "Bobo!"
  end
end
```

Pretty simple. Let's change const_missing.rb:

```
# some_directory/const_missing.rb
class Object
  def self.const_missing(c)
    require "./bobo"
    Bobo
  end
end

Bobo.new.print_bobo
```

Now try running it again. It prints "Bobo!" as you'd hope. So we just need to return something from const_missing, and that's what the unloaded constant acts like.

It turns out, that's exactly what Rails does!

CamelCase and snake_case

There's one minor difficulty, though. When you use a constant like BadAppleController, it's not loading a file named BadAppleController.rb. It has to load bad_apple_controller.rb. So we'll need to convert from one to the other.

CamelCase is the name for alternating capital and lowercase letters to show where words begin and end. The other way is called snake_case when you use lowercase_and_underscores.

"CamelCase" is called that because it has "humps" (capital letters) in the word, while "snake_case" looks low and slithery with the smaller letters and the underscores. Neat, no?

Since we know we need to convert controller names from MyFooController to my_foo_controller.rb, let's add a method for case conversion to Rulers. We'll create a new rulers/lib/rulers/util.rb file:

```
# rulers/lib/rulers/util.rb
module Rulers
  def self.to_underscore(string)
    string.gsub(/::/, '/').
    gsub(/([A-Z]+)([A-Z][a-z])/,'\1_\2').
    gsub(/([a-z\d])([A-Z])/,'\1_\2').
    tr("-", "_").
    downcase
  end
end
```

That's kind of a weird chunk of code. I copied it from ActiveSupport in Rails. It's used to convert CamelCase to snake_case. Let's break down what it does. It'll help if you've used regular expressions a bit.

First, to_underscore calls gsub (replace-all) on double-colons with slashes. In Rails, a constant like "Namespace::MyController" means you want a subdirectory called "namespace" under app/controllers. We don't do that in Rulers, but you could.

Next, it gsubs any two or more consecutive capital letters followed by a lowercase letter... And replaces it with \1_\2. If you've used regular expressions, you know that \1 means "the first thing in parentheses" and \2 means "the second thing in parentheses". So with a little work, you can see that this would change, say, "BOBSays" into "BOB_Says". Huh. That's pretty cool! If you're having trouble figuring it out, open up irb and start trying things like "BOBSays".gsub(/([a-z\d])([A-Z])/,'\1_\2') to see what they do. It's worth trying out each step here if you're confused.

Next, it gsubs from lowercase-number-uppercase to lowercase-number-underscore-uppercase. That is, it changes "a7D" into "a7_D" or "b4M" into "b4_M".

Finally it turns all dashes into underscores, and converts everything to lowercase.

It's worth working through a few examples to convince yourself that this works. TryAnyWords that YouPutTogether and see what it does!

Then rebuild rulers again. Getting tired of doing that yet? We're about to fix it. Remember to "git add" the new util.rb file or "gem build" won't include it in the new installed gem! We're about to fix that, too.

Reloading Means Convenience

Before we move on to the main reloading code, let's have one quick piece of reloading convenience for *you*. We have best_quotes and rulers next to each other in a directory. Adjust the path below if you have them elsewhere... But open up best_quotes/Gemfile and add a :path to the Rulers gem:

```
# best_quotes/Gemfile
```

```
source 'https://rubygems.org'
gem "rulers", :path => "../rulers"
```

Now do a quick "bundle install" from best_quotes. After that, you shouldn't need to do the gem build and gem install every time!

Why haven't we told you this sooner? Because it can cause mysterious-looking problems. This trick actually relies on deep Bundler trickery and requires you to *always* "bundle exec" before running things like rackup. If you forget that, it can look like the gem isn't there or (worse) look like an old version you installed earlier.

More importantly, the way you improve is to learn everything manually and *then* learn the convenient way. Readers who already knew about :path and used it in chapter 1 and 2 would get some extra convenience… But unlike them, *you* learned a valuable skill by doing it manually first.

That's why you're reading this book, right?

By the way - you should also "gem uninstall rulers". That's because if you ever forget to "bundle exec" something, you want Ruby to complain that it can't find rulers instead of quietly using an old version until you notice. Remember: with :path, you *always* have to "bundle exec" or bad things happen (no gem or old gem.) Given the choice, you would ***always*** rather have no gem found than an old version used.

Putting It Together

Now that you can convert a CamelCaseConstantName to a snake_case_file_name, let's add magic constant loading to Rulers.

Open up a new file, rulers/lib/rulers/dependencies.rb:

```
# rulers/lib/rulers/dependencies.rb
class Object
  def self.const_missing(c)
    require Rulers.to_underscore(c.to_s)
    Object.const_get(c)
  end
end
```

You'll also need to add a bit to rulers/lib/rulers.rb. See those requires up at the top? Add a couple. They should look like this when you're done:

```
# rulers/lib/rulers.rb (snippet, top)
require "rulers/version"
require "rulers/routing"
require "rulers/util"
require "rulers/dependencies"
```

That includes your new stuff. Since you don't need to rebuild, have a quick sigh of relief.

Now we need to test it. Open up best_quotes/config/application.rb. Remember how we had to manually require "quotes_controller"? Get rid of that require so that we can load it up automatically.

```
# best_quotes/config/application.rb
require "rulers"

$LOAD_PATH << File.join(File.dirname(__FILE__),
```

```
                          "..", "app",
                          "controllers")
# --> No more require here! <--

module BestQuotes
  class Application < Rulers::Application
  end
end
```

Restart your rackup *with bundle exec*. That is, type “bundle exec rackup -p 3001”. Reopen your browser to “http://localhost:3001/quotes/a_quote”.

If you’ve done everything right, you’ve just seen your framework do the same kind of automagic constant loading that makes Rails impress the new programmers so much.

Take a bow. Or go back in and debug, debug, debug.

Review

In this chapter, you learned about const_missing and Gemfile :path. You also got a bit of a refresher on const_get, and learned how Rails uses const_get and const_missing to save you a lot of require statements.

This is also one of the first pieces of deep Rails magic that most people never understand. And now, you do. Have a look at exercise one below and think about the weird error message from Rails that goes “YourController is not missing constant WhateverHelper!” Why would Rails try to load that? Why would it check? *How* would it check?

Think a little on that error message. This is a deceptively deep problem that most Rails programmers dismiss with “Rails is just

weird" or "Rails is too magical". Here's a hint: Rails is trying to load things into the right place, but Ruby looks up constants in a funny scoped way so there are several right places... So it can be hard to tell where you're looking something up. Imagine what would happen if you had a constant with that name already there, or Rails tried to load the class to somewhere unexpected...

Rails is actually solving a much harder problem than we are. It can have nested namespaces for controllers so it has to figure out how and where to load something in. Remember that Rails can't just peek inside the source file without parsing a lot of Ruby code, and people are allowed to define classes in all kinds of weird ways (like Class.new!). You just want to write MyController, but you might secretly mean that it's GameSimulation::MyController or Analytics::MyController or somewhere else entirely.

That's one of the down-sides of Rails trying to guess what you mean. Sometimes it's very hard to guess right.

Exercises

Exercise One: Did It Work?

When you load a file called whatever_class.rb, you're not actually guaranteed that it contains WhateverClass, or that the constant WhateverClass is actually a class. How would you check?

You might try calling const_get(:WhateverClass)... Except that you just made const_get try to load automatically. If you call it on an unloaded class *inside* the method call where you try to load, you'll recurse forever and get a "stack level too deep" and a crash. So const_get isn't the full answer.

But const_missing is calling const_get again, which is calling const_missing... You already own the code to const_missing. Do you see a hideously hacky way to make this work?

You could say “do the const_get... *unless* I’m already calling const_missing.”

Open up rulers/lib/rulers/dependencies.rb and change to this:

```
# rulers/lib/rulers/dependencies.rb
class Object
  def self.const_missing(c)
    return nil if @calling_const_missing

    @calling_const_missing = true
    require Rulers.to_underscore(c.to_s)
    klass = Object.const_get(c)
    @calling_const_missing = false

    klass
  end
end
```

Now const_missing will return nil if you’re already in the middle of another const_missing. So you can add a check to make sure that the right constant was loaded! You can actually put the check in that same method.

If you were going to follow Rails’ lead, you could even check whether there was already a constant by that name before you loaded the file... Just in case something weird had happened and you didn’t want to risk double-loading.

But there’s a reason I say “hideously hacky.” Think about ways this could break. For instance -- think about what would happen if you hit this in multiple threads at once. Oops!

Now come up with at least one more way to write this. Your way can still be hideously hacky, if you like. But fix at least one problem in my hack!

Exercise Two: Re-re-reloading

Rails still has a head start on us with one kind of reloading... When we change our classes, we have to restart the server ourselves.

Rails actually uses a pretty impressive hack to drop and recompile all the classes in your main Rails app on each controller request -- that's a lot of the reason that Rails development mode can be so slow! It won't always succeed when you change gems, plugins or other kinds of dependencies, but mostly it works well.

We can use the rerun gem (sorry Windows folks -- not on Windows!) to get very much the same effect.

Open the Gemfile in best_quotes and add a development dependency on the rerun gem:

```
# best_quotes/Gemfile
source 'https://rubygems.org'
gem 'rulers', :path => "../rulers"

group :development do
  gem 'rerun'
  gem 'listen', '=1.3.1' # for older Ruby
end
```

"Rerun" is a gem and command to re-run your server or whatever process you tell it every time it sees your files change.

So after bundle installing, try running this: “bundle exec rerun --rackup -p 3001”. That just adds “rerun” after “bundle exec”, but before “rackup”.

The “--” is an old Unix trick. It means “that’s all the arguments you get, the rest belong to somebody else.” Specifically, it tells rerun to ignore the “-p” later. If you don’t add the “--”, rerun won’t work because it will believe it should only watch a file called 3001, which doesn’t exist.

When rerun runs, you’ll see something like this:

```
20:40:53 [rerun] Watching ./**/*.
{rb,js,css,scss,sass,erb,html,haml,ru} using Darwin
adapter
```

Now edit the quotes_controller. You can just add a comment or a blank line. Rerun should kill your server and restart it:

```
20:40:57 [rerun] Change detected: 1 modified
20:40:57 [rerun] Sending signal TERM to 38753
[2012-11-29 20:40:57] ERROR SignalException:
SIGTERM
    webrick/server.rb:98:in `select'
20:41:01 [rerun] Sending signal INT to 38753
[2012-11-29 20:41:01] INFO  going to shutdown ...
[2012-11-29 20:41:01] INFO
WEBrick::HTTPServer#start done.

20:41:02 [rerun] Best_quotes restarted
```

That looks a little alarming, but all it means is that rerun is restarting your server because it saw a file change. Nice!

Of course, if you're really observant, you might have seen that path above: "./**/*.{rb,js,css,scss,sass,erb,html,haml,ru}" -- that means it's watching every file in every 'best_quotes' subdirectory with those extensions. But **not** any of the files under rulers.

There are a few obvious fixes -- tell rerun to watch rulers also, link the rulers directory from best_quotes, run best_quotes from one directory up... But other than moving the rulers directory under best_quotes, I haven't found one that works. So rerun is a fine way to do app development, but it's annoying when you're trying to develop your framework -- just like real Rails reloading!

In Rails

Rails uses ActiveSupport for its const_missing support. Have a look at rails/activesupport/lib/active_support/dependencies.rb for the full version. Most of the code is installing a const_missing that can call through to non-Rails versions of const_missing in other classes, and can be removed or re-added and is appropriately modular. It also works hard to support nested modules like MyLibrary::SubModule::SomeClass. In other words, the Rails code is a lot more complicated than yours because it's doing a lot of things you don't need right now.

Now you know the code to make it work if you don't need to worry about being compatible with so many third-party libraries.

Rails can use Gemfile tricks like :path as well. In older versions of Rails you would "vendor" gems. That means to install them in a directory under your application so that the version is frozen and you don't have to worry about a different version being installed later, or no version being available. You can use :path in the same way, or any of several similar Bundler features.

4. Rendering Views

Rails wouldn't be nearly as useful without views and the many fun things you can do in them. You know how to use views, of course. Want to know what they look like under the hood?

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_4_mine chapter_4" to create a new branch called "chapter_4_mine" for your commits.

Erb and Erubis

Before building views into Rulers, first you'll need to know how to use Erb. Erb is the language that Rails uses by default for views. It provides those funny <%= %> things with Ruby code in them.

Erubis is the implementation of Erb that Rails uses. It's quite fast, it's easy to hook new features into, and it's actually useful in all sorts of places outside of Rails views. Did you know that you can use Erb in the middle of your config/database.yml file? Rails already does that, you can just use it.

To use Erb you'll first need to open up rulers/rulers.gemspec. You haven't touched it in awhile. Near the bottom, there should be some dependencies:

```
# rulers/rulers.gemspec (excerpt)
# specify any dependencies here; for example:
gem.add_runtime_dependency "rack"
gem.add_runtime_dependency “erubis" # ADD THIS
gem.add_development_dependency "rack-test"
```

You’ll need to add another runtime dependency called “erubis”, like you see above. Now run “bundle install” to install the erubis gem.

Create a new test file, just to play with. Let’s call it erb_test.rb:

```
# some_directory/erb_test.rb
require "erubis"

template = <<TEMPLATE
Hello! This is a template.
It has <%= whatever %>.
TEMPLATE

eruby = Erubis::Eruby.new(template)
puts eruby.src
puts "=========="
puts eruby.result(:whatever => "ponies!")
```

You can see that we set up a template from a here-document. You could also read the template from a file like most Erb users do. The call to “.src” is to say “give me the code for this template.” And you can see that when we call .result(), we get to specify what value the variable inside has.

Now run it with “ruby erb_test.rb”. We get this:

```
bash-3.2$ ruby erb_test.rb
_buf = ''; _buf << 'Hello!   This is a template.
It has '; _buf << ( whatever ).to_s; _buf << '.
';
_buf.to_s
==========
Hello! This is a template.
It has ponies!.
```

The few lines starting with _buf are interesting. Erubis takes apart our string, appends it to _buf piece by piece, and adds the variables in as well after calling .to_s on them. Then it just returns _buf.

It doesn't look very magical, does it? That's the best kind of magic.

And Now, Back to Our Program

Back in Rulers, you'll want to add a call to render a view. Now that you know how to use Erubis that should sound pretty easy. It's high time to split off the controller into its own file. So first, open rulers/lib/rulers.rb. Add require "rulers/controller" with the requires, and remove class Controller from the file:

```
# rulers/lib/rulers.rb
require "rulers/version"
require "rulers/routing"
require "rulers/util"
require "rulers/dependencies"
require "rulers/controller" # ADD THIS
```

```
# Further down, remove class Controller.
# It will go into its own file.
```

Then open up rulers/lib/rulers/controller.rb:

```
# rulers/lib/rulers/controller.rb
module Rulers
  class Controller
    def initialize(env)
      @env = env
    end

    def env
      @env
    end
  end
end
```

After that, add a “require "erubis"” at the top of the file and then add a simple Erubis-based ‘render’ method to Controller:

```
# rulers/lib/rulers/controller.rb (excerpt)
def render(view_name, locals = {})
  filename = File.join "app", "views",
    "#{view_name}.html.erb"
  template = File.read filename
  eruby = Erubis::Eruby.new(template)
  eruby.result locals.merge(:env => env)
end
```

It should look pretty similar to your Erb test earlier. How do we use it?

Also, see how we're passing "env" through? That allows us to use "env" in the view to get the Rack environment. We won't use that in this chapter -- it's just an example of how your framework can make variables available in the view.

More Quotes, More Better

In best_quotes, you can change QuotesController#a_quote to just say "render :a_quote, :noun => :winking":

```
# best_quotes/app/controllers/quotes_controller:
class QuotesController < Rulers::Controller
  def a_quote
    render :a_quote, :noun => :winking
  end
end
```

You'll also want to make a new shiny app/views directory. We're not yet separating them by controller, so create a new file, best_quotes/app/views/a_quote.html.erb:

```
<%# best_quotes/app/views/a_quote.html.erb %>
<p>
  There is nothing either good or bad but
    <%= noun %> makes it so.
</p>

<p>
  Ruby version <%= RUBY_VERSION %>
```

```
</p>
```

Looks simple enough. Go ahead and run “bundle exec rackup -p 3001” and point your browser at “http://localhost:3001/quotes/a_quote”.

And look at that! It’s even finding your Ruby version via the automatic built-in RUBY_VERSION constant.

Controller Names

One thing that may strike you about this, other than having to explicitly call render for the template, is that it’s not finding our controller name like Rails would. Let’s fix that.

In Rails, a view is normally at a path like app/views/controller_name/view.html.erb. To do the same you’ll need to know the class of the controller object, like NicePeopleController, and then convert to a directory name like nice_people. Ruby makes it pretty easy to get the class of an object. And you already know how to convert to snake_case. So let’s add a method to rulers/lib/rulers/controller.rb:

```
# rulers/lib/rulers/controller.rb:
def controller_name
  klass = self.class
  klass = klass.to_s.gsub /Controller$/, ""
  Rulers.to_underscore klass
end
```

There we are. Now that looks more like the way Rails does it. Now you can move your view and update the render method to

use the controller name. The updated render method will look like this:

```
# rulers/lib/rulers/controller.rb (excerpt)
def render(view_name, locals = {})
  filename = File.join "app", "views",
    controller_name, "#{view_name}.html.erb"
  template = File.read filename
  eruby = Erubis::Eruby.new(template)
  eruby.result locals.merge(:env => env)
end
```

You'll need to move the view file to best_quotes/app/views/quotes/a_quote.html.erb, too!

Re-run and reload, just to make sure everything still works.

Review

In this chapter, you built a basic set of views with Erb support, similar to the kind that Rails uses. You found out how to turn an Erb template into text. And you found another use for that "to_underscore" method we wrote awhile back.

At this point you can start to write code that looks a lot like real code in a real Ruby framework. You're also prepared to start looking hard through the code of Sinatra, Camping or Padrino and learn from it. Sinatra uses a view rendering system very similar to what you just built.

Exercises

Exercise One: Mind Those Tests!

As your framework resembles Rails more, you'll need to update your tests to match. The test from chapter 1 is clearly out of date by now, as you'll know if you've run it lately. It's good practice to run your tests regularly, automatically for preference, so you find out when that happens.

Did you notice when the test broke, or any other tests you wrote? If so, good for you -- those habits will stand you in good stead.

If not, savor that uncomfortable feeling. It's the one that says, "oh, that doesn't work... And I don't even know how long it's been broken. If I had customers, this would be really bad."

(If you commit to Git religiously, it's a different feeling and you're thinking about how to "git bisect" this one. Also not bad.)

So first, let's fix the test. Go back and look at rulers/test/application_test.rb. I'll wait. Do you see what's going wrong? Take a minute if it's not immediately obvious.

There are actually several broken pieces and some are hard to fix. For instance, you don't have any renderable routes without an application now. And how do you add controllers or views to your test application?

Let's handle it by creating a TestController that inherits from Rulers::Controller and testing that. There's more than one way to do it -- here's one:

```
require_relative "test_helper"

class TestController < Rulers::Controller
  def index
    "Hello!"  # Not rendering a view
  end
end
```

```
class TestApp < Rulers::Application
  def get_controller_and_action(env)
    [TestController, "index"]
  end
end

class RulersAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    TestApp.new
  end

  def test_request
    get "/example/route"

    assert last_response.ok?
    body = last_response.body
    assert body["Hello"]
  end
end
```

This way of testing just overrides the routing in the test application object. That's important since there's no set of app directories -- for the same reason it doesn't test a view file.

How could we test that? I'll let you think about about it. There are at least three or four good answers to the problem. But to give you a hint about some of those ways... The best way to be sure you are testing an app is for there to be an actual, literal app somewhere that is being tested.

Now go in and update your other test. That's the one you wrote at the end of the testing exercise in chapter 1. You *did* write another test, didn't you?

Exercise Two: View Variables

You may remember that we pass "env" into our Erubis call earlier. Then we don't do anything with it. That "env" is actually available as a view variable -- you could do something like this in your view:

```
# best_quotes/app/views/a_quote.html.erb
<p>
  There is nothing either good or bad but
    <%= noun %> makes it so.
</p>

<p>
  Ruby version <%= env["PATH_INFO"] %>
</p>
```

That's fine, but not terribly polished. Add at least two interesting variables you could use in your view, and then write a view that shows them.

If you can't come up with any examples, here are a few possibilities:

- controller name
- user agent
- time of request start
- version of the rulers gem

Of course, if you wanted to do it full-on Rails style, you should be setting up a lot of instance variables and passing them through.

It turns out that Ruby lets you call ".instance_variables" on, say, a controller object to get the list of all instance variables currently set on that object.

Ruby *also* lets you call "obj.instance_variable_get :@myvar" to get the value of that variable. There's also an instance_variable_set method to set them.

Have another look at your "render" method in Rulers. Have you figured out how to do Rails-style instance-variable passing yet?

For extra credit, implement and test it.

Exercise Three: Rake Targets for Tests

Up to this point we've been running our tests directly and manually. There's nothing wrong with that approach. It a great first step. But we can do better.

Rake actually ships with a "Rake::TestTask" which generally does what we want.

Let's see what it looks like in the Rakefile:

```
# Rakefile
require "bundler/gem_tasks"
require "rake/testtask"

Rake::TestTask.new do |t|
  t.name = "test"  # this is the default
  t.libs << "test"  # load the test dir
  t.test_files = Dir['test/*test*.rb']
```

```
  t.verbose = true
end
```

Now if you run “rake -T” to show all Rake targets, it will show a “rake test” task. And when you run it, you should see the same listing of tests and assertions running:

```
# Running tests:

.

Finished tests in 0.010210s, 97.9432 tests/s,
195.8864 assertions/s.

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

A word of caution: Rake will always run your tests by loading them into the same Ruby process, then running each one in turn. This is a lot faster than running each one in an individual process. But it means that your tests can mess with each other in annoying ways. If you find yourself saying, “but I didn’t *set* that global variable in this test!” think about whether some other test might have done it. For extra fun, the tests don’t always run in any predictable order. Think about whether *any* other test might have done something that would cause that error!

(This is still true if you run the tests by hand on the command line -- at least, if you run them all with one Ruby command. Sorry!)

In Rails

Rails (until Rails 5) uses Erubis for its views by default, but there are several differences from what you did. For instance, Rails actually allows registering a number of different template engines at once with a number of different extensions -- .erb files are rendered with Erubis, but .haml files are rendered with the HAML templating engine.

You can find the Erb-specific view code in actionpack/lib/action_view.rb actionview/lib/action_view/template/handlers/erb.rb and actionview/lib/action_view/template/handlers/erb/erubis.rb.. Your current code may do less... but their code is around four pages, not six lines! You can see that the bulk of Rails' code is setup, interface and dealing with string encodings. You save yourself a lot of trouble by knowing that you're basically only dealing with ASCII and/or UTF-8 strings.

You can find all of the Rails template code in the ActionView and ActionPack gems if you look for it. Now that you have both controllers and views, you've built a surprisingly good mini-stand-in for ActionPack. In chapter 9 we'll improve our routing and make it work even better.

If Rails no longer uses Erubis, why not use Erubi, the fork that is used by Rails? Because it's less general-purpose, without the easy ability to pass variables through. Erubi is slightly faster and what Rails uses, but Erubis is more likely to help you on your own projects.

5. Basic Models

It's not common, but you *can* use Rails without using models. Our example app doesn't use any models at all yet. Apps that serve from memory, from constant data, from files or from REST APIs don't need a database. For that matter, Rails models don't *have* to be based on a database, though they usually are.

In this chapter you'll be building non-database models to see how they work. Diving full-on into building an Object-Relational Mapper can wait for chapter 7.

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_5_mine chapter_5" to create a new branch called "chapter_5_mine" for your commits.

File-Based Models

Our models will be based on files, not databases. We'll use JSON files. First, open rulers.gemspec to add a dependency for "multi_json". The bottom of the file will look like this:

```
# rulers/rulers.gemspec (excerpt)
  gem.add_runtime_dependency "rack"
  gem.add_runtime_dependency "erubis"
  gem.add_runtime_dependency "multi_json"
```

```
end
```

Notice that we keep adding gems. That pattern will continue. Rails depends on a *lot* of gems. Most large applications and libraries depend on many. It's not uncommon to need to install ten, twenty or even thirty gems when deploying a large project to a clean system. Many gems are quite small so this isn't as bad as it sounds.

Run "bundle install" to install multi_json.

Let's figure out what one of our files might look like. Since we have a "best quotes" application we'll want to be able to have a list of quotes. We'll want to find them individually, find them in groups and save new quotes.

JSON, like database rows, gives flexible, easy-to-read storage. We're using it here because it's quick to write and easy to read.

Here's how we'll create a JSON object for a single quote in the file best_quotes/db/quotes/1.json:

```
{
  "submitter": "Jeff",
  "quote": "A penny saved is a penny earned.",
  "attribution": "Ben Franklin"
}
```

Go ahead and create a best_quotes/db/quotes directory. "db" will be the directory for "databases", like in a Rails app. But in this case, a directory of files is acting like a table. The "quotes" subdirectory will be the "table" containing our quotes -- each one a JSON file.

Open best_quotes/db/quotes/1.json and put the text above into it. That can be the first “row” in our “table”.

To load it, we’ll need a Rulers object. First, open rulers/lib/rulers.rb and add a new dependency called file_model:

```
# rulers/lib/rulers.rb (excerpt)
require "rulers/dependencies"
require "rulers/controller"
require "rulers/file_model"
```

Then open rulers/lib/rulers/file_model.rb:

```
# rulers/lib/rulers/file_model.rb
require "multi_json"
module Rulers
  module Model
    class FileModel
      def initialize(filename)
        @filename = filename

        # If filename is "dir/37.json", @id is 37
        basename = File.split(filename)[-1]
        @id = File.basename(basename, ".json").to_i

        obj = File.read(filename)
        @hash = MultiJson.load(obj)
      end

      def [](name)
        @hash[name.to_s]
      end
```

```
        def []=(name, value)
          @hash[name.to_s] = value
        end

        def self.find(id)
          begin
            FileModel.new("db/quotes/#{id}.json")
          rescue
            return nil
          end
        end

      end
    end
  end
```

In initialize(), File.split will split on slashes on Unix or Mac, and split on backslashes on Windows. We then grab the final piece.

MultiJson has simple encode and decode to and from Ruby hashes. It's "multi" because it will use other JSON libraries as the back-end encoder if you ask it to, so you can make sure all your code is doing it the same way. It's what Rails uses.

We have a square-bracket method so we can use our object like a hash. Later we'll create methods with the names of the attributes, but not yet. For now, you can write obj["field"] instead of obj.field.

With ActiveRecord, you generally create an object from parameters or pass around its ID. If you pass around its ID, you have to be able to find an object by ID. Luckily, we have an easy way to do that - the filename. Have a quick look at "self.find(id)" above and satisfy yourself that it will look up an ID properly. It

works with numbers, too, since string interpolation (that's the #{} thing) automatically calls .to_s() on whatever is inside.

Our find() just returns nil if it can't find the ID - Ruby will raise a FileNotFound exception, the rescue will catch it and the method returns nil. Later, you could also raise a RecordNotFound exception if you like.

The code above looks okay. It gives us the ability to load up a file and write it back out. Now how do we use that code?

Open up best_quotes/app/controllers/quotes_controller.rb and add a new action:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  # (excerpt)
  def quote_1
    quote_1 = Rulers::Model::FileModel.find(1)
    render :quote, :obj => quote_1
  end
end
```

Looks good. And you'll also need a quote view:

```
<%# best_quotes/app/views/quotes/quote.html.erb %>
<p>
  &quot;<%= obj["quote"] %>&quot;
  <br/> &ndash; <%= obj["attribution"] %>
</p>

<p>
  Submitted by <%= obj["submitter"] %>
</p>
```

Now, “bundle install” in best_quotes to get the multi_json gem. Restart your server and point your browser at “http://localhost:3001/quotes/quote_1”. Seeing Ben Franklin?

You’ve just built your first usable model. As of this moment, what you’ve built is an opinionated, Rack-based MVC framework. Sound like any other framework you know?

Take another bow. You’ve earned it.

Inclusion and Convenience

Of course, seeing "Rulers::Model::FileModel" in your controller may have made you wince. Let’s fix that. Open up rulers/lib/rulers/controller.rb. We’ll add a require at the top, and then an include just after the Controller class declaration:

```
# rulers/lib/rulers/controller.rb (excerpt)
require "rulers/file_model"

module Rulers
  class Controller
    include Rulers::Model
    #...
  end
end
```

Here, we’re making our Controller include the Rulers::Model module. Your controllers inherit from Rulers’ Controller, so you can just say “FileModel” in your controller when you want one.

Queries

You can already look up an object by ID. ActiveRecord also provides a “find all” method so that you can have index actions. Let’s do that next. Make sure to add more JSON files to the directory if you want to see more quotes!

Open up rulers/lib/rulers/file_model.rb and we’ll add methods to FileModel:

```
# rulers/lib/rulers/file_model.rb (excerpt)
module Rulers
  module Model
    class FileModel
      def self.all
        files = Dir["db/quotes/*.json"]
        files.map { |f| FileModel.new f }
      end
    end
  end
end
```

Map takes a list, in this case of file names, and replaces them with the result of the block, in this case FileModel objects.

Open best_quotes/app/controllers/quotes_controller.rb. Add one more new action:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def index
    quotes = FileModel.all
```

```
      render :index, :quotes => quotes
    end
  end
```

And a new view:

```
<%# best_quotes/app/views/quotes/index.html.erb: %>
<% quotes.each do |q| %>
  <p> <%= q["quote"] %> </p>
<% end %>
```

See how we're adding new actions and views, like Rails does? This should be feeling familiar... Only it's with your own framework, where you typed out and debugged every line.

Where Do New Quotes Come From?

Another very basic object operation is creating and saving new instances of the model. Let's make that happen.

Here's one way to write FileModel.create:

```
# rulers/lib/rulers/file_model.rb (excerpt)
def self.create(attrs)
  hash = {}
  hash["submitter"] = attrs["submitter"] || ""
  hash["quote"] = attrs["quote"] || ""
  hash["attribution"] = attrs["attribution"] || ""

  files = Dir["db/quotes/*.json"]
  names = files.map { |f| File.split(f)[-1] }
  highest = names.map { |b| b.to_i }.max
```

```
    id = highest + 1

    File.open("db/quotes/#{id}.json", "w") do |f|
      f.write <<TEMPLATE
{
  "submitter": "#{hash["submitter"]}",
  "quote": "#{hash["quote"]}",
  "attribution": "#{hash["attribution"]}"
}
TEMPLATE
    end

    FileModel.new "db/quotes/#{id}.json"
  end
```

There are several fun things going on with this create. First, we read the three attributes we care about from the passed parameters. Mass assignment like this is a whole huge topic in Rails, not just to make it happen but to make it secure. Read up on “Strong Parameters” in Rails to see how it can be done.

We then read the list of JSON files in the directory and extract their IDs, and pick a new ID which is one higher than the highest we can find. We create the new file for the object. And finally we actually create a new FileModel in memory and return it.

(Why do we call .to_i on a filename? Because it takes just the integer part and ignores the extension ".json".)

What can we do with it? We can make our controller more complete:

```
  # best_quotes/app/controllers/quotes_controller.rb:
    # Excerpt
```

```
def new_quote
  attrs = {
    "submitter" => "web user",
    "quote" => "A picture is worth one k pixels",
    "attribution" => "Me"
  }
  m = FileModel.create attrs
  render :quote, :obj => m
end
```

Now when you point your browser at "http://localhost:3001/quotes/new_quote", you should see the "picture is worth one k pixels" quote. But then if you go to "http://localhost:3001/quotes/index", you'll see that a new one has been added. In fact, hit "reload" a few times on new_quote before going back to index and you'll see more of them. You can also look in best_quotes/db/quotes/ and you'll see that it's adding a new file every time. Neat!

You're one step closer to full CRUD - a silly acronym for Create/Read/Update/Delete. CRUD consists of the actions a Rails resource or scaffold can do with a new object by default.

Review

In this chapter, you built a simple system of models based on JSON files. You could actually use these models verbatim in Rails if you felt like it. Rails is quite flexible about models and about how you get data for your controllers.

You also added methods to create new model instances and handled some of the operations ActiveRecord does for you. In this chapter's exercises you'll do even more.

The big difference between what you did and "real" Rails models is that if you write to the ActiveModel interface, Rails will let you

use form helpers like form_for() directly with your models and URL helpers to route to them. It will also allow you to declare validations -- things like “validates_uniqueness_of :field”. You could build a version of FileModel that Rails would use with its helpers if you added a few extra methods - see “In Rails” below.

Exercises

Exercise One: Object Updates

In Rails scaffolding, you’ll generally see two methods called edit and update. “Edit” is a simple method that returns an HTML page for editing an existing object. You can reload the edit page as many times as you want and nothing will change.

“Update” is a POST-only method that changes the object and then usually redirects you to another view. Commonly the edit action will show a form with the object’s values. You can change the values in the browser and then submit the new values to the update action. Update will then look the object up by ID (normally in a hidden form field) and write the new values into place.

Go ahead and write just the update, not the edit, method. You may want to make it a POST-only method by checking env["REQUEST_METHOD"] -- some browsers have a habit of caching GETs so that only the first one actually gets to your server.

To debug POST, use or download a program called “curl” which allows you to make HTTP requests from the command line. Curl is incredibly useful for all web programming. To specifically POST an update, you’d type something like this:

```
> curl "http://localhost:3001/quotes/update_quote"
-v -d param1=value1 -d param2=value2
```

As the parameters to curl remind us, we don't have a way to get parameters yet. That will happen next chapter. For now, pick a quote (like 1.json) and update just one field (like its submitter).

You'll probably want to write a "save" method on FileModel, which could use MultiJson.dump() to turn @hash into JSON.

Exercise Two: Caching and Sharing Models

There are two common ideas of how to cache models in Ruby. The ActiveRecord method could be summed up as "I found it, my copy, no sharing." That means that if you call MyClass.find(37), your instance is probably separate from every other instance, everywhere. You can mess with it, update it and so on.

That also means you don't get much caching. It's pretty similar to what you have now, in fact.

The DataMapper model could be summed up as "why ever have more than one?" So if you call MyClass.find(37), you'll get the same object 37 as everywhere else.

Go ahead and add DataMapper-style caching to your find method. While Rails usually uses ActiveRecord, it will happily use DataMapper if you request it when you do "rails new".

By the way -- why would you ever use ActiveRecord caching over DataMapper caching? After all, the DataMapper version will always have better memory use and a smaller footprint!

Here's a hint: imagine writing a method that finds an object, changes some fields, then waits to see if it should commit them. Now imagine it calls a function that happens to call MyClass.find(37), then saves the result...

Now think about how that would work with multiple threads.

Both DataMapper and ActiveRecord encourage a tactical database workflow -- query, modify, save, then throw away the object that find() or create() returned. If you think about caching for long, you'll start to understand why. But the penalty for not doing it promptly is much more severe in DataMapper.

There are many other forms of caching that could be implemented, of course. Come up with one other kind of caching that is neither like ActiveRecord nor DataMapper. If it sounds useful or fun, try implementing it.

Exercise Three: More Interesting Finders

Rails lets you query by all kinds of database attributes, and that can get complicated fast. But old Rails used to let you find by attributes in a much simpler way.

Rails 3 or higher: "MyClass.where(:attr => 3).all"

Rails 2: "MyClass.find_all_by_attr(3)"

Go back to FileModel and add a "find_all_by_submitter" method. You can also add a simple route to your controller if you want to let you see all quotes by a given submitter. Then, make submitter names clickable in a view.

Your framework should be starting to feel like Rails by this point, even if you don't have all the bells and whistles. A lot of your framework is a lot like Rails version 1 or 2, in fact.

For bonus points, go add a method_missing() to FileModel. It should check the name of the method that was missing, and if it matches /^find_all_by_(.*)/, you can do a find on that attribute.

A side note for pedants like myself: always override respond_to_missing? if you override method_missing. Google it if you don't know why. For older Ruby versions, you may need to override respond_to? instead (or also).

In Rails

The obvious equivalent of FileModel in Rails is ActiveRecord. But a more interesting comparison for this chapter is ActiveModel. ActiveRecord is an Object-Relational Mapper so that each of your objects represents a database row. ActiveModel is the interface that Rails uses to all of storage including non-relational stores like Cassandra or MongoDB, to fit particular object types into Rails.

For a good overview of ActiveModel, have a look at a blog post from Yehuda Katz on that topic: http://yehudakatz.com/2010/01/10/activemodel-make-any-ruby-object-feel-like-activerecord/

The FileModel here is also directly usable in Rails. So it's also its own equivalent-in-Rails. Rails lets you use nearly any object as a model. Of course you can use any legal Ruby in your controllers.

By the way - don't use FileModel in Rails in a large application. If you have multiple servers, or use a multi-server hosted solution like Heroku, writing to local files just doesn't cut it. You need shared storage such as a real SQL database server, a shared network file system or a storage server like Redis, Memcached, Cassandra or MongoDB.

A less-obvious parallel in Rails is Fixtures -- database rows stores as YAML files, which have the same kind of bulk loading and limited querying, and a similar implementation.

6. Request, Response

Let's take a breath after an intense chapter on models to add some Rack-style conveniences. Rack provides a lot that we're not using yet. And some of Rack is quite simple to add to Rulers.

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_6_mine chapter_6" to create a new branch called "chapter_6_mine" for your commits.

Requests with Rack

Rack's Request object is just built from the environment. Remember that big imposing environment hash? Rack will parse it into a much friendlier form if you say "myrequest = Request.new(env)". Let's open up rulers/lib/rulers/controller.rb and make that happen:

```
# rulers/lib/rulers/controller.rb (excerpt)
require "rack/request"

module Rulers
  class Controller
    def request
      @request ||= Rack::Request.new(@env)
```

```
    end

    def params
      request.params
    end
  end
end
```

The “request” method creates a new request from the environment hash. The “@request ||=” caches the result. And params is just returning the params object that Rack::Request has already parsed for us. Convenient!

By the way, “@var ||= long_calculation()” is a common pattern in Ruby, and it’s worth knowing. The first time, it will calculate what you told it to. Every subsequent time it will just return the cached result. That’s because “value || calculation()” will return the value and not do the calculation (a.k.a. short-circuit evaluation).

Now we can use parameters much more easily in best_quotes. Let’s add a show method for any quote by ID:

```
# best_quotes/app/controllers/quotes_controller.rb
# (excerpt)
class QuotesController < ::Rulers::Controller
  def show
    quote = FileModel.find(params["id"])
    render :quote, :obj => quote
  end
end
```

This is starting to look like the same thing in Rails. Good. If you load "http://localhost:3001/quotes/show?id=1" in your browser, you should see quote #2, plain as day. Useful!

What can we do with request, other than getting the query parameters? Let's update the show action to show what browser is being used...

```
# best_quotes/app/controllers/quotes_controller.rb
# (excerpt)
class QuotesController < ::Rulers::Controller
  def show
    quote = FileModel.find(params["id"])
    ua = request.user_agent
    render :quote, :obj => quote, :ua => ua
  end
end
```

You'll also need to open the template file and display it there:

```
# best_quotes/app/views/quotes/quote.html.erb
<p>
  &amp;quot;<%= obj["quote"] %>&amp;quot;
  <br/> &amp;ndash; <%= obj["attribution"] %>
</p>

<p>
  Submitted by <%= obj["submitter"] %>
</p>

<p>
  Viewing with user agent: <%= ua %>
</p>
```

Now have a look at "http://localhost:3001/quotes/show?id=1". At the bottom you should see something like "`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.100 Safari/537.36`". That is, in fact, the string that Chrome sends to identify itself to your web server. Your string will look similar but not exactly the same, depending on what browser you use. Try a second browser (IE vs Chrome vs Firefox vs Safari, etc) and see what changes.

Responses with Rack

That was fun. We can do requests. What about responses? Plain-vanilla Rack responses aren't very exciting after you've done it all manually. You're too hard-core now to appreciate them fully - you can already do what they're protecting you from. But they're useful for building up a response string in pieces and have Rack keep track of the length for you. There are a few bits of cookie magic that they do which you may eventually care about.

Also, whoever uses your framework may not be as awesome as you, so let's use Rack::Response. We don't want to scare them with big raw nasty return values.

It would be nice for render() not to have to be the last line in an action. So let's do something about that too.

Open up rulers/lib/rulers/controller.rb:

```
# rulers/lib/rulers/controller.rb (excerpt)
module Rulers
  class Controller
    def response(text, status = 200, headers = {})
      raise "Already responded!" if @response
```

```
        a = [text].flatten
        @response = Rack::Response.new(a, status,
                                       headers)
      end

      def get_response  # Only for Rulers
        @response
      end

      def render_response(*args)
        response(render(*args))
      end
    end
  end
```

Also, open rulers.rb:

```
  # rulers/lib/rulers.rb:
  module Rulers
    class Application
      def call(env)   # Redefine
        if env['PATH_INFO'] == '/favicon.ico'
          return [404,
            {'Content-Type' => 'text/html'}, []]
        end

        klass, act = get_controller_and_action(env)
        controller = klass.new(env)
        text = controller.send(act)
        r = controller.get_response
        if r
         [r.status, r.headers, [r.body].flatten]
```

```
        else
          [200,
            {'Content-Type' => 'text/html'}, [text]]
        end
      end
    end
  end
```

Now, reopen quotes_controller and we'll render a response:

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController
  def show  # change the last line
    quote = FileModel.find(params["id"])
    ua = request.user_agent
    render_response :quote, :obj => quote,
      :ua => ua
  end
end
```

Reload everything, and voila... Nothing changes. You're doing the same thing you did before, just with Rack::Response. No problem. As an added bonus, you could add some debugging stuff after render_response and the response would still work... Simple return values can sometimes by inconvenient, which is why Rails doesn't use them for views.

(Why is this better? See exercise one, below.)

Review

Your framework is now using Rack's Request and Response objects, giving you a lot of things like parameter parsing and session variables for free. Otherwise you'd have to do those yourself.

This is like the Rails way of handling it, though Rails adds more functionality not found in Rack.

Exercises

Exercise One: Automatic Rendering

What we did with render_response was okay, but it still doesn't look like Rails. Specifically, we use render_response to override whatever value is returned from the action, but we never automatically render a view with the same name as the controller action like Rails does.

To fix that, you'll need to go into rulers/lib/rulers.rb. When no response has been set, instead you should automatically render a response with a name based on the current action.

In Rails the return value from the controller is ignored. If you don't call render (Rails' equivalent of render_response), it will automatically call it for you with the same controller name, and the view's name set to the same name as your action.

For this exercise, change the name of "render_response" back to "render", give "render" a new name or remove it, and your Rulers will work basically like Rails that way.

(But these exercises aren't mandatory, so I'm not going to change the Rulers method name in the remaining chapters of this book.)

Exercise Two: Instance Variables

In Rails the controller and view have very different methods available. For instance, the view has methods for escaping JavaScript and generating HTML tags like buttons and forms. The controller has methods for rendering different actions.

The Rails method of doing this is to have a view object, separate from the controller object. But then your variables passed to "render" don't work the same way.

The Rails answer is to set instance variables in the controller, then use them in the view. But an instance variable is on the *controller* instance, not the *view* instance.

Right now, your view code is executed directly in the controller object (via Erubis), but it has very few variables that it uses.

Try creating a new view object that uses Erubis to evaluate the view file. Then, make it easy to pass in a hash of instance variables which you'll set on the view object before doing the evaluation -- look up instance_variable_set for how to do it.

Once you've done that, make sure the controller creates one and uses it to render. You'll need to use "instance_variables" and "instance_variable_get" to query the controller's instance variables that were set in your action.

If you do this, you'll want to change the code in your existing controllers. Instead of passing "obj" as a local variable, for instance, you can do it Rails-style and set @obj in the controller, then use it in the view.

Are you feeling extra-magical now?

In Rails

Rails (more specifically, ActionPack) uses Rack in a very similar way, even exposing the Rack Request object with the “request” method. If you look in the actionpack gem under actionpack-3.X.X/lib/action_controller, you can find a lot of interesting Rack stuff. I especially recommend metal.rb and metal/*.rb. “Rails Metal” is a name for the lower-level Rails which goes mostly straight through to the “bare metal” -- that is, to Rack.

You can find a lot of the Rails implementation of Rack in these directories -- for instance, metal/redirecting.rb is the implementation of the redirect_to() helper which returns status 302 (redirect) and a location to Rack. You could steal the code and add a redirect_to to Rulers, if you wanted.

You can also find things like forgery (CSRF) protection, multiple renderers (i.e. Erb vs Haml), forcing SSL if requested and cookies in this directory. Some features are complex. Others call to Rack very simply and you could move the code right over to Rulers.

The biggest difference in the API is that Rails doesn’t return the string when you call “render” (well, usually - some calls to render do!) Instead, it keeps track of the fact that you called render and what you called it on. Then Rails raises an error if you call it again, or uses the defaults if you get to the end of a controller action without calling it, like in Exercise One above.

At this point your Rails clone is complete enough that you can understand most of what Rails is doing just by browsing its code. Most Rails files also have excellent comments at the beginning -- if you haven’t browsed through, say, the files in action_controller/caching you may find them easier to understand now.

7. The Littlest ORM

You have a fine Rack-based MVC framework, but we cheated a bit on the models. No real web framework uses only local files for data storage. If it did, you couldn't have multiple servers for your site! Rails may be scalable, but FileModel isn't... That's one reason real Rails apps usually use SQL databases.

We'll use SQLite and build a mini-ORM. You'll also see quickly how you could do the same thing in a different SQL database like MySQL or Postgres.

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_7_mine chapter_7" to create a new branch called "chapter_7_mine" for your commits.

What Couldn't FileModel Do? (Lots)

If you already know Ruby ORMs like Sequel, DataMapper or ActiveRecord, it was probably disappointing that FileModel didn't handle inheritance. SQLiteModel is going to handle inheritance just fine and automatically figure out your database schema.

We're not going to do full-on migrations, though. We have limited space in the chapter and something has to go!

No Migrations? But How Can I...?

Even if you have no migrations, you're still going to need tables. We'll use the Ruby SQLite gem directly. Make sure SQLite is installed on your machine (check this book's appendix and/or Google if you're not sure). Then we can install the Ruby SQLite gem, which uses the system SQLite library.

Go to rulers.gemspec and in the dependencies, add this:

```
# rulers/rulers.gemspec
#...
gem.add_runtime_dependency "erubis"
gem.add_development_dependency "rack-test"
gem.add_runtime_dependency "multi_json"
gem.add_runtime_dependency "sqlite3" # <- add this
```

You'll also need to run "bundle install" to install the sqlite3 gem. If you get compile errors, make sure to check your local (non-Ruby) SQLite installation -- you may need to install it or reinstall it, or tell Ruby where to find it. Google the error message you're getting if you can't easily fix it.

SQLite stores its databases in local files. But it still allows access from multiple processes and otherwise acts like a full database. And of course we still need to create tables. Let's create a new database file and a new table in it:

```
# best_quotes/mini_migration.rb
require "sqlite3"
conn = SQLite3::Database.new "test.db"
conn.execute <<SQL
create table my_table (
```

```
  id INTEGER PRIMARY KEY,
  posted INTEGER,
  title VARCHAR(30),
  body VARCHAR(32000));
SQL
```

You can run this with "`bundle exec ruby mini_migration.rb`".

The mini-migration makes a table with an id, two string fields called "title" and "body," and one integer field called "posted." That table is the only thing in a new database file called "test.db" which you should see in your directory. You can .gitignore it if you want.

Rulers needs to know about the table and its columns to use it. We can read the schema from SQLite.

Read That Schema

Let's see how we read the schema.

```
# rulers/lib/rulers/sqlite_model.rb
require "sqlite3"
require "rulers/util"

DB = SQLite3::Database.new "test.db"

module Rulers
  module Model
    class SQLite
      def self.table
        Rulers.to_underscore name
      end
```

```
      def self.schema
        return @schema if @schema
        @schema = {}
        DB.table_info(table) do |row|
          @schema[row["name"]] = row["type"]
        end
        @schema
      end
    end
  end
end
```

This is pretty simple. We make sure we have the table name from the class name (called “name”, above). We call table_info to get the schema. For each row, we just take the name and type.

Since those are self methods defined on the class object, @schema is actually an instance variable of the class itself, not on an instance -- so we’ll only have one for each class or subclass, not one for each instance of Rulers::Model::SQLite.

When Irb Needs a Boost

Usually we explore unknown code with irb, but we may load a specific library many times -- like Rulers. How do we handle that?

```
# best_quotes/sqlite_test.rb
require "sqlite3"
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite; end
STDERR.puts MyTable.schema.inspect
```

If you run the script, it should read the schema and print it out. Make sure Rulers is installed and run it: “bundle exec ruby sqlite_test.rb”.

If you get an empty hash as a result, make sure you ran the mini-migration, or do it now.

Or to play a bit in irb, you can just load Rulers! Start irb with -r rulers (“bundle exec irb -r rulers”). That way you can use anything in Rulers automatically. In irb, type “`require_relative "./sqlite_test.rb"` and you can keep playing with MyTable from there. You can make the command an alias, shellscript or batchfile -- “rails console” is a similar script for Rails.

If you create a MyTable instance, you’ll see that it’s not very useful yet. Let’s add some methods to the SQLite model.

Schema and Data

Let’s add some more interesting methods. Specifically, let’s create and count some new objects.

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
    class SQLite
      def initialize(data = nil)
        @hash = data
      end

      def self.to_sql(val)
        case val
        when NilClass
```

```
          'null'
        when Numeric
          val.to_s
        when String
          "'#{val}'"
        else
          raise "Can't change #{val.class} to SQL!"
        end
      end

      def self.create(values)
        values.delete "id"
        keys = schema.keys - ["id"]
        vals = keys.map do |key|
          values[key] ? to_sql(values[key]) :
            "null"
        end

        DB.execute <<SQL
INSERT INTO #{table} (#{keys.join ","})
  VALUES (#{vals.join ","});
SQL
        raw_vals = keys.map { |k| values[k] }
        data = Hash[keys.zip raw_vals]
        sql = "SELECT last_insert_rowid();"
        data["id"] = DB.execute(sql)[0][0]
        self.new data
      end

      def self.count
    DB.execute(<<SQL)[0][0]
SELECT COUNT(*) FROM #{table}
SQL
```

```
        end
      end
    end
  end
```

Now we can create new objects that also create rows in the database, and we can count how many are saved.

In SQLite the INTEGER PRIMARY KEY field we made, called “id”, will automatically increment to the next unused ID. Convenient! Notice that we don’t get an ID until we create or save. That’s exactly how ActiveRecord does it -- you can create a new object, play with it for awhile and then only save when you’re ready to add it to the database.

Here’s a fun side note on Ruby magic. Did you know you can declare a here-document and then keep going in the line? That’s what DB.execute(<<SQL)[0][0] is doing. That means “do all that, but substitute a string in for <<SQL, and that string goes from this line until SQL starts another line.” Pretty funky. You can even do it twice in one line!

Now let’s change the test program to create some new rows and then show us how many rows are in the database:

```
# best_quotes/sqlite_test.rb
require "sqlite3"
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite; end
STDERR.puts MyTable.schema.inspect

# Create row
mt = MyTable.create "title" => "It happened!",
```

```
  "posted" => 1, "body" => "It did!"
mt = MyTable.create "title" => "I saw it!"

puts "Count: #{MyTable.count}"
```

Every time you run it (`bundle exec ruby sqlite_test.rb`), the count should increase by two. Of course, we can't look them up and see the data... Yet.

Seek and Find

So let's add a few things. A .find() method should query for a particular id. Also, it would be nice to be able to query columns on an object:

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
      def self.find(id)
        row = DB.execute <<SQL
select #{schema.keys.join ","} from #{table}
where id = #{id};
SQL
        data = Hash[schema.keys.zip row[0]]
        self.new data
      end

      def [](name)
        @hash[name.to_s]
      end

      def []=(name, value)
```

```
        @hash[name.to_s] = value
      end
    end
  end
end
```

Let's change the test script to print out all the rows:

```
# best_quotes/sqlite_test.rb
require "sqlite3"
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite; end
STDERR.puts MyTable.schema.inspect

# Create row
mt = MyTable.create "title" => "I saw it again!"

puts "Count: #{MyTable.count}"

top_id = mt["id"].to_i
(1..top_id).each do |id|
  mt_id = MyTable.find(id)
  puts "Found title #{mt_id["title"]}."
end
```

Depending on how many times you ran each script, you should see something like this:

```
{"id"=>"INTEGER", "posted"=>"INTEGER",
"title"=>"VARCHAR(30)", "body"=>"VARCHAR(32000)"}
```

```
Count: 6
Found title It happened!.
Found title I saw it!.
Found title It happened!.
Found title I saw it!.
Found title I saw it again!.
Found title I saw it again!.
Found title I saw it!.
```

This is creating database objects for each ID in turn, and then printing out the title. It's like ActiveRecord Lite, but Lite-r!

Notice that we have to call .to_i explicitly on the ID. That's because we're getting the value back from SQL, but we're not converting it automatically.

I Might Need That Later

There's one remaining operation we'll cover before this chapter is done: saving objects. Let's have a look at minimal code for that, shall we?

```
# rulers/lib/rulers/sqlite_model.rb
module Rulers
  module Model
    class SQLite
      def save!
        unless @hash["id"]
          self.class.create
          return true
        end

        fields = @hash.map do |k, v|
```

```
          "#{k}=#{self.class.to_sql(v)}"
        end.join ","

        DB.execute <<SQL
UPDATE #{self.class.table}
SET #{fields}
WHERE id = #{@hash["id"]}
SQL
        true
      end

      def save
         self.save! rescue false
      end
    end
  end
end
```

Did you already know about Ruby having the same-line rescue? That thing where we say “rescue false” at the end and it just returns false if there’s an exception? Those make me happy. They also mask a lot of errors, so be really careful with them.

The code for save! is otherwise pretty unremarkable. ActiveRecord calls it “save!”, too. “Save!” is like save, but if it fails it raises an exception. For pure ActiveRecord compatibility we should actually be catching all exceptions and re-throwing our own... But I think the code is already long enough for a teaching example, don’t you?

Naturally we’ll need another test:

```
# best_quotes/sqlite_test.rb
require "sqlite3"
```

```
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite; end

# Create row
mt = MyTable.create "title" => "I saw it again!"
mt["title"] = "I really did!"
mt.save!

mt2 = MyTable.find mt["id"]

puts "Title: #{mt2["title"]}"
```

Run it (“bundle exec ruby sqlite_test.rb”) and you should see at the bottom “Title: I really did!”. That means you’ve successfully modified and saved the first object. Then when you reload it you see the new title that you set.

Review

You’ve put together a (very) small ORM that does the bare minimum to call itself that. And yet you’ve also found out a lot more about what “real” ORMs actually have to do, and a lot more about how their components fit together.

You’ve put together minimal versions of migrations, object creation and finding, saving, getting schemas and mapping your Ruby values into SQL. Each of these systems has a parallel in every ORM, and now you know what you’re looking for if you have to dig into ActiveRecord or Sequel to fix nasty bugs (take note, JRuby-on-Rails developers!).

Exercises

Exercise One: Column Accessors

Above, we defined [] and []= methods on SQLiteModel to let us change the column values. ActiveRecord and other real ORMs usually prefer to add a column accessor, so you'd say "obj.id" instead of "obj[:id]" or "obj['id']".

Ruby lets you called define_method to pass a block as a method definition. For instance:

```
class MyClass
  ["foo", "bar"].each do |method|
    define_method(method) {
      puts "Obj #{method}"
    }
  end
end
myobj = MyClass.new

myobj.foo
myobj.bar
```

This is an example where instead of the list above with "foo" and "bar", you could instead give a list of fields and write out accessors.

Remember that in Ruby you can reopen a class anywhere and add methods to it -- so even long after the class is defined, you can use define_method to add or redefine methods.

Now add code for method accessors to best_quotes and use them. In other words, add a call to define_method for each

column in the schema, and make sure it returns the column's value for the object.

Don't put too much work into it. We'll see an alternate, better way to do this in the next exercise.

Exercise Two: method_missing for Column Accessors

ActiveRecord doesn't do it quite the same way as Exercise One. Define_method is pretty slow, and it's easy to have a lot of model classes -- plus old-style ActiveRecord had all sorts of finders like "find_all_by_title_and_submitter". There's an exercise in chapter 5 where you created a finder like that. But you could easily wind up with hundreds or thousands of those being created every time you started a Rulers app. It's better if you add these accessors only if you need them.

The Ruby way to do that is with method_missing. When you call a method that doesn't exist on an object, "method_missing" will be called to tell you so. If you override method_missing to check for column names, you can return the column value from method_missing. Method_missing will return it from the original call.

But calling method_missing every time is *also* slow. So very clever libraries like ActiveRecord will define the column accessor or finder when method_missing is called. The first call is slow, and every later call is fast.

Go ahead and implement that. Remember that you can reopen the class, or you can call MyTable.class_eval and supply a block that gets evaluated inside. That block can also call define_method.

Exercise Three: from_sql and Types

Right now our column accessor returns the column's value from @hash, and @hash generally gets the value from SQLite. That

means it's whatever SQLite wants it to be. If you print out an integer field in sqlite_test.rb (e.g. with 'puts mt2["id"].class.to_s'), you'll see that the integer fields are coming through as Fixnum -- so that's just fine.

However, if you wanted more control over your fields, that wouldn't be enough. For example, if you had an integer field that you wanted to treat as a boolean (true/false) value, you'd have to convert it somehow. Or if you wanted to convert a Hash to JSON when you save and back from JSON to a Hash when you load, SQLite won't just do that for you.

You can add a method to the SQLite model that takes a column name and a type, and then when saving and loading that column, does something type-dependent to it, like the boolean or JSON fields above. Test it.

Now think about what would happen if you didn't convert on save or load, but converted whenever you got the value. How would it be different? Would you ever want to handle some fields one way and some another?

ActiveRecord allows both ways -- you can research the before_save and after_initialize callbacks for how to do it on save/load. And writing an accessor is easy in Ruby since you can redefine methods and reopen classes.

In Rails

The ActiveRecord gem is the obvious parallel to this system. ActiveRecord contains mappings of operations like our gem, but also migrations, cross-database compatibility and a huge amount of optimization and general complexity. And that's even though they use the ARel gem for most of the heavy lifting!

ActiveRecord also keeps a mapping of types between :int or :string and database equivalent. That's why we used types like varchar(32000) directly in the mini-migration instead of the type being “string”.

We also had a quick look at using irb and loading our library, which is a lot like “rails console”. Have you thought about how this would interact with our autoloading of classes? You can try it pretty easily if you’re curious.

8. Rack Middleware

With any Ruby web framework, you can modify how it works by adding Rack components around it. I like thinking of them as pancakes, because Rack lets you build your framework and your application like a stack of pancakes -- out of many thin layers. I also can't stand the word "middlewares" to refer to more than one.

Everything from this chapter can use your custom framework, Rulers. But all of this also works with Sinatra or any recent versions of Rails.

There are a lot of great Rack middleware modules out there to install as gems. Rack even comes with a lot of good built-ins so you can do simple things without any new code. It comes built right in.

Sample Source

Source for this chapter: not needed. You can use everything here with any Rack application, and you'll make an example one. Feel free to write a new Sinatra "hello world" app or use the code from last chapter if you want something more complicated.

Care and Feeding of Config.ru

You can tell you're looking at a Rack-based application (or framework) by the config.ru file. It's called a RackUp file -- that's what the "ru" stands for.

A RackUp file specifies the stack of middleware modules that makes this specific app.

The Rackup file can have intermediate layers or not. It has one single endpoint unless you use URLMap, as we'll see later on.

Endpoints are simple - Rack calls .call() on them, exactly like calling a Proc.

In fact, you can use a Proc. Remember the ultra-simple config.ru that we first started with in chapter 1? Here's what it looked like:

```
# sample_dir/config.ru
run proc {
  [200, {'Content-Type' => 'text/html'},
    ["Hello, world!"]]
}
```

This code just makes a new proc that returns success (HTTP 200), the content type HTML, and "Hello, world!". Then it calls "run" with that proc as an argument. Run the server ("rackup -p 3001" or "bundle exec rackup -p 3001"). Point your browser at localhost with the given port ("http://localhost:3001"). When you do, you should see "Hello, world!"

You can call run to specify any endpoint object that accepts .call(). That's what we're doing above with the proc, in fact. This will do the same thing as above:

```
# sample_dir/config.ru
obj = Object.new
def obj.call(*args)
  [200, {'Content-Type' => 'text/html'},
    ["Hello, world!"]]
end
run obj
```

We've also mentioned that Rack lets you build apps with intermediate layers. How do we include an intermediate layer? First, let's do the same kind of thing manually:

```
# sample_dir/config.ru
INNER_LAYER = proc {
  "world!"
}
OUTER_LAYER = proc {
  inner_content = INNER_LAYER.call
  [200, {'Content-Type' => 'text/html'},
    ["Hello," + inner_content]]
}
run OUTER_LAYER
```

You can see what we're doing here. The inner layer returns just "world!", while the outer layer wraps it up nicely and puts "Hello," in front. This is crude, but it's like how real Rack middleware works. The inner layer returns a result, and each layer modifies it in turn. Now let's see Rack's version.

The Real Thing

The actual Rack middleware doesn't know the next layer's name like OUTER_LAYER knows INNER_LAYER, above. It would be annoying to write reusable code if you always had to know the next layer. Instead, Rack will pass each layer its next layer as an argument to initialize(). The very bottom layer has no next layer. You tell Rack to set up the stack with "use." Like this:

```
# sample_dir/config.ru
use Rack::Auth::Basic, "app" do |_, pass|
```

```
  'secret' == pass
end

run proc {
  [200, {'Content-Type' => 'text/html'},
    ["Hello, world!"]]
}
```

Run this, point the browser at it and you'll get a username/password dialog. Call the user anything you want, but the password has to be the word "secret". You've just run a two-layer Rack app!

When we say "use", we tell it what class to create and what arguments to pass to the constructor. Let's make our own middleware class in config.ru:

```
# sample_dir/config.ru
class Canadianize
  def initialize(app, arg = "")
    @app = app
    @arg = arg
  end

  def call(env)
    status, headers, content = @app.call(env)
    content[0] += @arg + ", eh?"
    [ status, headers, content ]
  end
end

use Canadianize, ", simple"
```

```
run proc {
  [200, {'Content-Type' => 'text/html'},
    ["Hello, world"]]
}
```

Since you pass in “simple” in the initialize() argument to Canadianize, the final content is “Hello world, simple, eh?”. Point your browser at it. Simple, eh? The block you passed to Rack::Auth::Basic above was passed into its initialize method, just like “arg” is passed into your Canadianize middleware here. You can have as many or as few arguments as you like, but the arguments to initialize() must match what you pass to use().

Now if we look at the real config.ru from best_quotes in the earlier chapters, it should make a lot more sense:

```
# best_quotes/config.ru
require "./config/application"
run BestQuotes::Application.new
```

BestQuotes::Application has a .call() method so you can pass it to run(). And what it eventually returns is what the browser displays.

Powerful and Portable

Rack is used by all recent versions of Rails, by Rulers, by Sinatra and by basically every other Ruby web framework -- Cuba, Rum, Grape, Camping, Padrino, Ramaze, Maverick and many more you’ve never heard of. Generally you write a config.ru and start them up with “rackup”. If you don’t then the framework does.

That means that in any of these frameworks, you can use Rack middleware.

What does that buy you?

It gives you a fast way to tie in certain libraries to any of them. Remember Rack::Auth::Basic up above? You can use it to password-protect any application or API in any of these frameworks. Don't like HTTP Basic authentication? You can do the same thing with a Rack middleware for HTTP Digest authentication. Or you can move all the way up to something like Warden, a full authentication system with accounts... and it ties into whatever framework you like via Rack.

There's a lot of Rack middleware out there, and they basically all work with all frameworks.

Built-In Middleware

Rack has built-in middleware for the things below. They should each seem interesting but limited. That's because you're supposed to stack a bunch of them all up together with your own custom bits to make an application, not just use one by itself.

Rack::Auth::Basic - HTTP Basic authentication.

Rack::Auth::Digest - HTTP Digest authentication.

Rack::Cascade - Pass a request to a series of Rack apps, and use the first request that comes back as good. It's a way to mount one Rack app "on top of" another (or many).

Rack::Chunked - A Rack interface to HTTP Chunked transfer.

Rack::CommonLogger - Request logging.

Rack::ConditionalGet - Implement HTTP If-None-Match and If-Modified-Since with ETags and dates.

Rack::Config - Call a given block before each request.

Rack::ContentLength - Set Content-Length automatically.

Rack::ContentType - Try to guess Content-Type and set it.

Rack::Deflater - Compress the response with gzip/deflate.

Rack::Directory - Add Apache-style directory listings. This is an endpoint not an intermediate layer, so use it with “run.”

Rack::ETag - Generate ETags from MD5s of the content.

Rack::Head - Remove response body for HEAD requests.

Rack::Lint - Check your responses for correctness.

Rack::Lock - Only allow one thread in at once.

Rack::Reloader - Reload your app when files change.

Rack::Runtime - Times the request, sets X-Runtime in response.

Rack::Sendfile - Use the X-Sendfile header to ask your web server to send a file much faster than Ruby can.

Rack::ShowExceptions - Show a nice exception page if something breaks.

Rack::ShowStatus - Show a pretty page if the result is empty.

Rack::Static - Serve from certain directories as static files instead of calling your framework.

Rack::URLMap - Route different directories to different apps or different stacks. You can also use this with a “map” block in config.ru.

In all of these cases, the documentation can be pretty bad. Some of them are simple enough you can just “use” them and get the benefit. If not, Google can often help you - remember to look for “ruby rack” instead of just “rack” since “rack” matches a lot of

things that aren't software. But basically, you're very lucky that you now know how Rack middleware is built and you can read the source! You can build a lot of important stuff for your app (cache headers, static files, authentication, logging, compression) with only the middleware above if you know how to use it.

Third-Party Middleware

What's some other good middleware "in the wild"? Here are some popular choices and fun existing ideas:

Rack::Cache - Drop-in HTTP caching.

Rack::ChromeFrame - Require IE users to install ChromeFrame.

Rack::GoogleAnalytics - Add GA tracking code to each page.

Rack::HoneyPot - Foils spam-bots with fake form fields.

Rack-a-Mole - Monitor your app, with logging or persisting to MongoDB. Rack-a-mole can alert, catch exceptions and even tweet your app status.

Rack::Mount - Run different apps in different directories, a lot like Rack::URLMap.

Rack::Referrals - Figures out search words of search engine traffic to your site.

Rack::Throttle - Rate-limit incoming requests.

Warden - A powerful authentication framework.

These are usually documented better than the built-ins. Most are available via "gem install."

Again, remember to add the word "ruby" when Googling, or things like "rack mount" will give you completely wrong results.

More Complicated Middleware

I mentioned in Rack::URLMap that you could use “map” blocks in config.ru to use Rack::URLMap instead of doing it manually. Let’s see what that looks like:

```
# sample_dir/config.ru
require "rack/lobster"

use Rack::ContentType

map "/lobster" do
  use Rack::ShowExceptions
  run Rack::Lobster.new
end

map "/lobster/but_not" do
  run proc {
    [200, {}, ["Really not a lobster"]]
  }
end

run proc {
  [200, {}, ["Not a lobster"]]
}
```

If you run rackup and check in your browser, you should see “Not a lobster”. But if you go to “http://localhost:3001/lobster” (or other port if your console says so), you should see an ASCII-art lobster. Map is a way to tell Rack what paths go to what Rack apps - and if there’s could be two that match, the longer path always takes precedence, no matter what order the rules were defined in.

You can check the “longest path” thing with the /lobster/but_not path. You can move it before or after the “/lobster” block and it will still count as being more specific -- it displays “Really not a lobster” rather than displaying the ASCII lobster, even if a previous rule (“/lobster”) matches “first”. Instead, what matters is how long the matching prefix is for the rule, measured in characters.

We use Rack::ContentType to set the default HTML content type for everything. Since it’s at the top, outside the blocks, it applies to all the blocks.

We also use another layer, Rack::ShowExceptions, when using “/lobster”. That way if you hit the “crash” button under the ASCII-art lobster you’ll get a pretty page showing the details (try it!). It’s also a great way to see that inside each block you can have a full stack of middleware, not just a single “run” statement.

The lobster, by the way, is a simple test app built into Rack. You’ll see it as an example in many places.

Middleware Fast and Slow

At this point we could assemble a mish-mash of different stacks of Rack middleware at different paths with different authentication and whatnot. But that’s almost never actually useful. Besides, you’re definitely smart enough to figure it out on your own if you need to. So instead let’s see how we can use custom Rack middleware for benchmarking. We’re not even going to use the built-in Rack::Runtime middleware, though you could do that if you wanted to use curl and check headers to do your benchmarking.

```
# sample_dir/config.ru
```

```
require "rack/lobster"

use Rack::ContentType

class BenchMarker
  def initialize(app, runs = 100)
    @app, @runs = app, runs
  end

  def call(env)
    t = Time.now

    result = nil
    @runs.times { result = @app.call(env) }

    t2 = Time.now - t
    STDERR.puts <<OUTPUT
Benchmark:
  #{@runs} runs
  #{t2.to_f} seconds total
  #{t2.to_f * 1000.0 / @runs} millisec/run
OUTPUT

    result
  end
end

use BenchMarker, 10_000

run Rack::Lobster.new
```

This is a simple benchmark middleware that runs the next app (in this case, Rack::Lobster) many times and then figures out how

many milliseconds it took on average. The main discovery if you run it is that Rack::Lobster is *fast*. Here's what I get on console:

```
Benchmark:
  10000 runs
  0.476974 seconds total
  0.0476974 millisec/run
```

Which means that Rack is actually pretty fast. That means if your framework or application is slow, now you know how to find out what's slowing you down. In Rails (or Rulers!) you can even choose where to insert your Benchmark middleware to find out if it's an inner or an outer layer slowing you down. For Rails, see http://guides.rubyonrails.org/rails_on_rack.html.

Review

You've learned about using Rack middleware to build applications in small, modular pieces. You've learned what middleware is already built into Rack, and gotten a quick taste of what interesting middleware libraries are available.

You've also built useful middleware to benchmark your framework and applications, and seen a little of how to use it.

Exercises

Exercise One: Do It With Rulers

Take the benchmarking middleware example above and add it the best_quotes config.ru. You'll notice that it uses whatever route you specify, so try benchmarking a URL with just text, a URL with

a database call and a URL that loads from filemodel. Can you see a difference in speed?

Exercise Two: Middleware from Gems

You probably didn't actually install any of that third-party middleware earlier when we read about it. Rack::GoogleAnalytics is a nice simple one -- use that for an easy exercise, or Rack-A-Mole for a much less easy exercise. I'll leave it to your discretion whether you want to set up it up to use MongoDB... MongoDB setup isn't terribly easy, but it *is* highly educational.

Exercise Three: More Benchmarking

Go back to the benchmark exercise, above. The lobster is so fast it's nearly a "hello, world" app, so let's keep using it. Also add an even simpler "hello, world" route, and a route that does a "sleep 0.0001" -- which will be much slower, because that sleep is guaranteed to take at least 100 microseconds, or 0.1 milliseconds.

Now try adding other middleware to config.ru like ContentType or ContentLength -- or Rack-A-Mole, if you're adventurous. How much do they slow your app down? Does the same piece of middleware tend to subtract the same number of request/second to different routes? Or add the same number of milliseconds? Or neither?

Benchmarking is a very complex topic and I won't get into all the math here. But try a few things now, and you can Google later on if you get stuck and the results don't make sense.

In Rails

Rails has a config.ru file, of course. But it also has several separate Rack interfaces for things like routing.

You can find out about all the various Rails/Rack interfaces in the "Rails on Rack" Rails Guide at http://guides.rubyonrails.org/rails_on_rack.html. You can find out more about Rails Routing, the Rails equivalent of the URLMap stuff we're doing, at http://guides.rubyonrails.org/routing.html.

You can also type "rake middleware" in a Rails app for a list of the main stack of middleware, top to bottom. You'll probably see several you recognize from the big list of built-ins.

For more Rack middleware to use with Rails, see https://github.com/rack/rack/wiki/List-of-Middleware or do a little Googling. There's a lot of it out there. Remember to Google for "middleware" or (urgh) "middlewares" and probably "ruby". "Rack" is just too common a word by itself.

The primary Rack application object in Rails is called ActionController::Dispatcher. If you want to check the Rails source code for how Rack integrates, make sure to read it.

Also, ActionController::Base allows you to get mini-Rack-apps for each controller action because it inherits from Metal, the basic Rails Rack class. So you can call MyController.action(:myaction) and get a Rack app for that action in your controller. We'll talk more about that in the next chapter.

And finally, you can generally look through ActionDispatch inside the action_pack gem. Specifically, look in lib/action_dispatch and check the Ruby files there. Lots of good Rack stuff!

9. Real Routing

Right this second, Rulers will slice up any URL it receives and assume it's a controller and an action. That's not terrible - it got a lot of Rails 2.X apps through the Dark Ages. But we can do better by taking some concepts from modern Rails and adding a dash of Rack from last chapter.

This is the last big chapter of code. There's a lot of it. But you're nearly done. This is an area not many people really understand in Rails. In slightly less than one chapter, ***you will***.

Take a deep breath and jump in!

Sample Source

Sample source for all chapters is on GitHub:

http://github.com/noahgibbs/rulers

http://github.com/noahgibbs/best_quotes

Once you've cloned the repositories, in EACH directory do "git checkout -b chapter_9_mine chapter_9" to create a new branch called "chapter_9_mine" for your commits.

Routing Languages

Rails routers take a URL like "http://bobo.com/controller/action.css and turn it into a controller, an action and sometimes an ID or other parameters. Look back at chapter 2 if you want to see the code we used to do the same, but it was simple -- we split the string and assume we know what the parts are. That's quick and easy but it's not configurable.

Real routes might be specified more like this:

```
# example_routes.rb
match "/latte/:quality" =>
  "latte_controller#by_quality"

# Find a starbucks within :rad of the
# given location
match "/starbucks/:lat/:long/:rad" =>
  "sb_controller#storefind"
```

Actions don't always follow the "controller, action, id, params" formula of old-style Rails default routes. They also don't always follow the similar formula of your current router. Routes like the above need something a bit different.

In case "match" looks unfamiliar to you: it's the most versatile, but not the most common, way to specify a route in Rails. However, other routing methods like "get", "post" and "resource" are easy to write as thin wrappers around it.

An action should be able to put as many interesting parameters as it wants into the URL. The basic controller/action pattern in Rulers doesn't do any of that yet.

It would be convenient if you could write a more versatile router. Then your apps could have routes that match their needs.

Controller Actions are Rack Apps

Rack apps look quite a lot like a method or a proc. You already know some interesting tricks you can do with Rack apps. Rails takes this a half-step farther and makes every action on every controller a full-on Rack app that you can extract and use.

You already know from chapter 8 that you can do some routing and you can use middleware with Rack apps. Soon I'll show you a few more uses as well.

We'll need to be able to pass some extra parameters through to your controller action -- imagine a route called "/franklin-quote" that goes to an action like QuotesController#show(37) for your awesome Ben Franklin quote. How does that "37" get in as the ID for "show"? We can match against /franklin-quote and pass in ":id => 37" with the extra parameters.

Remember your Rulers application object? Here's what it looks like at the moment:

```
# rulers/lib/rulers.rb
module Rulers
  class Application
    def call(env)  # Before we change it
      if env['PATH_INFO'] == '/favicon.ico'
        return [404,
          {'Content-Type' => 'text/html'}, []]
      end

      klass, act = get_controller_and_action(env)
      controller = klass.new(env)
      text = controller.send(act)
      r = controller.get_response
      if r
       [r.status, r.headers, [r.body].flatten]
      else
        [200,
          {'Content-Type' => 'text/html'}, [text]]
      end
    end
```

```
  end
end
```

If you want to extract a controller's action into a Rack app, all of that will need to be wrapped in an object that looks like a proc. Let's create a method called "dispatch" to get the Rack app with "action," supply the extra parameters and wrap it all into a proc. Then we'll call that new "action" method to get a Rack app in Rulers::Application:

```
# rulers/lib/rulers.rb
module Rulers
  class Application
    def call(env)  # We're updating this method
      if env['PATH_INFO'] == '/favicon.ico'
        return [404,
          {'Content-Type' => 'text/html'}, []]
      end

      # Don't parse the route here,
      # use a new method we'll write.
      rack_app = get_rack_app(env)
      rack_app.call(env)
    end
  end
end


# rulers/lib/rulers/controller.rb
module Rulers
  class Controller
    include Rulers::Model
```

```
def initialize(env)
  @env = env
  @routing_params = {}  # Add this line!
end

def dispatch(action, routing_params = {})
  @routing_params = routing_params
  text = self.send(action)
  r = get_response
  if r
   [r.status, r.headers, [r.body].flatten]
  else
    [200, {'Content-Type' => 'text/html'},
      [text].flatten]
  end
end

def self.action(act, rp = {})
  proc { |e| self.new(e).dispatch(act, rp) }
end

# Change this too!
def params
  request.params.merge @routing_params
end
```

Now the Rulers::Application just gets the action and uses it like a Rack app when it gets called by Rack. And, we can take any controller action and get it as an actual Rack action.

So what did that buy us?

It lets us route to controller actions where we would previously use Rack apps. Let's play with that.

Rack::Map and Controllers

Remember the “map” statements in config.ru in chapter 8? Let’s use those to build a simple router purely in config.ru. But now we can map to our own controller actions.

Open best_quotes/config.ru again. Here’s what it looks like now:

```
# best_quotes/config.ru
require './config/application'
run BestQuotes::Application.new
```

We’ll use Rack map to serve the “index” action from the root:

```
# best_quotes/config.ru
require './config/application'

map "/" do
  run QuotesController.action(:index)
end

run BestQuotes::Application.new
```

You may have noticed that when you just pointed your browser at the top-level URL you would get a nasty error as it tried to look up a controller that didn’t exist. Now that doesn’t happen any more!

Putting map into config.ru is a great method for very simple routing, and it’s very fast. If your Rails, Rack or Rulers app has too many routes but they’re simple, this is a great way to pare that number down.

You ***could*** even write custom Ruby code to parse the route. Config.ru is just Ruby code, after all. Just keep in mind that custom Ruby code will be harder to speed-test and change later.

Many applications have more complex and interesting routes. Now that you can grab controller actions directly and put them in various places, let's do something more configurable.

Configuring a Router

The Rails router is all about "match". The complicated stuff like "resource" is all built on top of "match", usually pretty simply.

Match itself can be complicated, though:

```
# Example matches
match ":controller/:action/:id", :via => :post
match "posts/ajax/:action", :controller => /posts/
match "api/search/*rest", "api#search"
match "admin/login", "devise#new_session"
match "rack-intro", "pages#show",
  :defaults => { :id => 37 }
match "mini-rack/*args", my_rack_application
match “photos/:id", "photos#show",
  :defaults => { :user_id => 37 },
  :via => [ :get, :post],
  :constraints => { :user_id => /^\d+$/ }
```

The router just applies them in order -- if more than one rule matches, the first one wins.

Variables other than :controller or :action are added to the parameters that are passed through. If there's a second non-hash

argument to ‘match’, it’s usually of the form “controller#action”. That does the same as setting :controller and :action. Or the second argument can be a Rack application, which Rails then calls.

Wait, “second argument?” Aren’t there lots of arguments? Yes and no. All those extra bits of the form “:arg => data” are effectively hash keys, and they get lumped into a single hash table when Ruby parses them. So technically a call like this:

```
match "rack-intro", "pages#show",
  :defaults => { :id => 37 }
```

…actually has three arguments: the string “rack-intro”, the string “pages#show”, and a hash table with a single key, which is the symbol “:defaults”.

Ruby’s parser is complicated, yo.

There are also real keyword arguments in more recent versions of Ruby, which could be used for the same thing. Rails clearly feels a bit conflicted about them, and they’re still changing a bit in semi-compatible ways in Ruby 2.7. In other words, Matz ***also*** clearly feels a bit conflicted about their current implementation. So we’re going to do this the classical way.

We’ll build a simpler match method and make our own router... Let’s start with how to get “match” called in the first place. Like a lot of Ruby domain-specific languages, we’ll make an object that is “self” for the block and just define methods on it:

```
# rulers/lib/rulers/routing.rb
module Rulers
  class RouteObject
    def initialize
```

```
      @rules = []
    end

    def match(url, *args)
    end

    def check_url(url)
    end
  end

  # And now, Rulers::Application:
  class Application
    def route(&block)
      @route_obj ||= RouteObject.new
      @route_obj.instance_eval(&block)
    end

    def get_rack_app(env)
      raise "No routes!" unless @route_obj
      @route_obj.check_url env["PATH_INFO"]
    end
  end
end
```

As you can see, this doesn't route yet. The match() and check_url() methods in RouteObject are still blank. But let's look at what it should do when we implement them.

We can take our Rulers application and call it like this:

```
# best_quotes/config.ru
require "./config/application"
```

```
app = BestQuotes::Application.new

# Standard Rack middleware works
# (see chapter 8)
use Rack::ContentType

app.route do
  match "", "quotes#index"
  match "sub-app",
    proc { [200, {}, ["Hello, sub-app!"]] }

  # default routes
  match ":controller/:id/:action"
  match ":controller/:id",
    :default => { "action" => "show" }
  match ":controller",
    :default => { "action" => "index" }
end

run app
```

As you've no doubt guessed, that's a routing table for best_quotes. By the end of the chapter we'll have a match that works correctly for each of those cases.

In the mean time, if you set up best_quotes with config.ru above and make match() print its arguments, you'll see that it calls into our RouteObject for the Rulers application. If you add methods with other names like "resource" or "root", you could make your router accept them as well.

Playing with Matches

First, we'll implement match. The match method doesn't immediately match. Instead, it remembers enough about its arguments to do a match later against real URLs:

```
# rulers/lib/rulers/routing.rb
module Rulers
  class RouteObject
    def match(url, *args)
      options = {}
      options = args.pop if args[-1].is_a?(Hash)
      options[:default] ||= {}

      dest = nil
      dest = args.pop if args.size > 0
      raise "Too many args!" if args.size > 0

      parts = url.split("/")
      parts.select! { |p| !p.empty? }

      vars = []
      regexp_parts = parts.map do |part|
        if part[0] == ":"
          vars << part[1..-1]
          "([a-zA-Z0-9]+)"
        elsif part[0] == "*"
          vars << part[1..-1]
          "(.*)"
        else
          part
        end
```

```
        end

        regexp = regexp_parts.join("/")
        @rules.push({
          :regexp => Regexp.new("^/#{regexp}$"),
          :vars => vars,
          :dest => dest,
          :options => options,
        })
      end
    end
  end
```

This uses a couple of neat tricks. The first few lines grab the hash of options off the end regardless of the number of parameters using “args.pop”.

Then, we take the URL, split it on slashes, map each piece to a little piece of regular expression, and put together a final regular expression from them. To me, this is awesome. You may have more going on in your life. While we do that, we keep track of a list of variables names (“vars”) so that after we run the regular expression we’ll be able to match up captured values from parenthesis expressions with names.

Finally, we put it all together into a hash describing that one routing rule. We save it for later so we can match against it.

Next let’s see what matching on those hashes looks like:

```
# rulers/lib/rulers/routing.rb
module Rulers
  class RouteObject
    def check_url(url)
```

```
  @rules.each do |r|
    m = r[:regexp].match(url)

    if m
      options = r[:options]
      params = options[:default].dup
      r[:vars].each_with_index do |v, i|
        params[v] = m.captures[i]
      end
      dest = nil
      if r[:dest]
        return get_dest(r[:dest], params)
      else
        controller = params["controller"]
        action = params["action"]
        return get_dest("#{controller}" +
          "##{action}", params)
      end
    end
  end

  nil
end

def get_dest(dest, routing_params = {})
  return dest if dest.respond_to?(:call)
  if dest =~ /^([^#]+)#([^#]+)$/
    name = $1.capitalize
    con = Object.const_get("#{name}Controller")
    return con.action($2, routing_params)
  end
  raise "No destination: #{dest.inspect}!"
end
```

```
    end
  end
```

Whew. That's pretty big. Let's go through it.

First, check_url goes through the rules in order until one matches. That makes sense. If the regular expression matches, it creates a "params" object and matches up variable names with the parts that the regular expression captured.

If the rule had a destination like "quotes#index", we call get_dest to turn that into a Rack application. If it didn't have a destination, we grab the :controller and :action variables and use them to get the Rack application through get_dest.

How does get_dest turn that into a Rack application? First, it checks whether the destination responds to call(), and returns it if it does. So if what you passed in was already a Rack app, we just use it.

Otherwise we split the name up on "#" and look it up as an action on a controller. When we get that controller action, we also pass in the routing parameters. And **that** is why we had to add the routing parameters to the action() method on the controller, way back toward the beginning of the chapter.

I told you I'd explain it later.

Putting It Together (Again)

Remember that config.ru from best_quotes, above? Now you can cd into best_quotes, type "bundle exec rackup -p 3001" and see it in action.

The root ("http://localhost:3001/") matches the quotes#index action. Going to "/sub-app" runs a little Rack app that's just a

proc, though you could put in a Sinatra app, or even a Rails app. The default routes at the end look a lot like Rails' default routes -- it would be easy for you to always include those in every Rulers app if you wanted to.

You can go to "/quotes" to see the index action and "/quotes/1" to see the "show" action. See config.ru for the full current set of routes.

Here's that config.ru again:

```
# best_quotes/config.ru
require_relative "config/application"

app = BestQuotes::Application.new

use Rack::ContentType

app.route do
  match "", "quotes#index"
  match "sub-app",
    proc { [200, {}, ["Hello, sub-app!"]] }

  # default routes
  match ":controller/:id/:action"
  match ":controller/:id",
    :default => { "action" => "show" }
  match ":controller",
    :default => { "action" => "index" }
end

run app
```

Take a bow. You've made it through nine chapters and the architecture of a modern web framework.

You're now ready to begin crawling through the internals of Rails to see all the little details that got left out. You've probably already started, in fact. And you're definitely better at seeing how it all fits together.

If you haven't already done the exercises, take a breath and a day or two before you go back and do that. This can be a lot all at once. If you **have** done the exercises, take that breath and two days, and then start building your own new features!

Whatever the next framework Big Thing is, somebody has to build it. It won't be exactly like Rails. But if you're the one to build it, it will use the best parts of Rails' design!

Review

In this chapter, you've built a full-on router. By now you could probably take apart Rails' router and figure out how it works. You also know how Rails connects lots of tiny Rack applications into a single overall application. It's a complicated, multi-layered construction. Now you know what all the layers look like.

Exercises

Exercise One: Adding Root

We never added a Rails-style "root" method. It's not complicated. It's just a match against "/". But go back and add it in. Little DSLs like this are *all over* in Ruby, so it's important that you get comfortable modifying them.

Exercise Two: HTTP Verbs

HTTP has various verbs - GET, POST, PUT, PATCH, HEAD and DELETE are the common ones. Rails resources, for instance, often have the same URL for getting an item (“show”) and modifying that item (“edit”), but with different HTTP verbs. Showing the item is GET, modifying it might be PUT or PATCH.

Partly this matters because of caching. A GET can be cached because if repeated it would return the same data. A POST, PUT or DELETE shouldn’t be cached because it’s being used for the side effect.

But it also matters because if you want Rails-like URLs, you’ll need to support HTTP verbs. You may notice the “:via” option in the example “match” calls at the start of the chapter. An option like “:via => ‘GET’” just means that the route only matches on a request with an HTTP GET verb, but not with a POST, a PUT, etc.

You already have the environment hash, which includes the HTTP verb - you can print it out and find the right hash key for it. You can add a “:via” option to the options hash, which should default to all HTTP verbs.

You see where this is going? Add a :via option to your routes, which will allow you to match a URL like “/posts/“ to different HTTP verbs, such as GET for the index action but PUT for the ‘create’ action.

Exercise Three: Default Routes

We talked repeatedly about default routes. In Rails, those are routes that get added to the route file automatically when it’s generated -- if you comment them out or remove them, they go away. You could also add mandatory default routes -- default routes that are already in the routing table at the beginning, before you add anything. That’s not like Rails, but it’s a neat idea!

Go in and add one or more of those. You can steal a default route from config.ru above if you like.

Now try running the app again and look at /sub-app. If you see the an error looking for sub-app's controller, you needed to make sure the default routes are always checked *last*.

But our default routes still don't look like Rails, partly because we never made it possible to make a matched section be optional. Go in and make *that* happen, which might take a lot of lines of code (or might not).

If you're enjoying this, adding the optional sections showed you at least one more thing you could add to routing. Go add it.

Exercise Four: Resources

(You'll want to do Exercise Two before this one.)

Have another look at the Rails routing guide ("http://guides.rubyonrails.org/routing.html"). Scroll down to the section with the specific HTTP verbs, paths and actions ("Crud, Verbs, and Actions").

You're looking at exactly how to implement the Rails "resource" routing command in terms of match. And you just built match. Want to add an adjective like "RESTful" to the framework you built? Go make it happen.

In Rails

Rails uses routers like the one you built. But Rails' own routers are still a little different from yours.

Routing can quickly get slow if you have many routes. It was common in Rails 2 for route-heavy applications to spend 20% CPU or more just figuring out how to route actions. Current Rails has some fixes for that.

The current Rails router does a lot of careful precomputation on your routes using a component called "Journey" to build the table. So part of Rails' fix for slow routes was just to ship with a more complicated, generally much faster router. ActionDispatch, inside ActionPack, has the code for routing including Journey.

The other major fix for this problem is Rails Metal -- the ability to route directly to Rack applications without going through the Rails router stack. So if you have a particular section of your app with horrifically complex routing, you can break it out into its own Rack app and handle that logic in a custom way before you get to the main router. This trick works fine in Rulers, too. See the previous chapter for details of creating and using middleware.

Now that you know what to look for, it's worth profiling your next Rails app to see how much time is spent in routing. It's not huge for most apps. But if you've got a case where routing is slow, now you know a very direct way to fix it!

Rails controllers also work a bit differently from how Rulers is doing it. Specifically, each Rails controller keeps track of a mini-Rack stack of middleware which can be specified per-action like before_filters. That makes their Controller.action() method significantly more complicated than yours was.

Rails also has a somewhat more complex match() method signature than you do. You can read about Rails' version here:

http://api.rubyonrails.org/classes/ActionDispatch/Routing.html

In general, you can read about Rails routing and get more ideas. See exercise 3 above for an example. So here's another link to the Rails routing guide:

http://guides.rubyonrails.org/routing.html

And the Rails Rack stuff is also thoroughly non-mysterious to you now. So here's one last link to the "Rack in Rails" guide:

http://guides.rubyonrails.org/rails_on_rack.html

Do you see that disclaimer at the beginning about needing to be basically familiar with Rack concepts and laugh yet? You should. You're now more familiar with Rack concepts than anybody who hasn't worked heavily on a Rack framework.

You're also ready to dig into Rack and see how it works if you want to. Or hey, rebuild it. But that would be a different book.

Answers to Exercises

The exercises start simple and well-defined, and get more open-ended as you go along. That's on purpose, and I think it's a very good thing. As a result you may disagree with some of my answers, especially later on. And you **certainly** may have a different answer.

Good! Programming, Rails and you are all changing over time. It's a very good thing that we don't all agree on everything.

These are *some* answers. I think they're *good* answers. But don't take them too seriously as the *only right* answers.

Chapter 1

Exercise One: just type the code as written and run it.

Exercise Two: just type the code as written and run it.

Exercise Three: after typing the code as written and running it, add another test. That can be as simple as copying the test_request method, changing the name, and changing the get to "/my/url". For now it doesn't matter what URL you use since the application doesn't check it.

Chapter 2

Exercise One: run the code given, then look over the hash. What's different from the one I gave? You'll be using a different version of Rack, a different browser and probably other things. What isn't the same as the hash I gave?

Exercise Two: Follow the steps as written. To return a custom 500 page, you can just return the text of it, or return

File.read("path/to/my/500page"). Another thing you could do on error would be to take the current context (request data, call stack, etc.) and save it to a file in /tmp. It would even be possible to save that data in the framework in an array or hash, though you'd have to figure out how to retrieve it. You can get the current call stack by calling the "caller" method, which returns an array of strings.

Exercise Three: The check for a URL of "/" in Rulers::Application#call is straightforward:

```
module Rulers
  class Application
    def call(env)
      if env['PATH_INFO'] == '/favicon.ico'
        return [404,
          {'Content-Type' => 'text/html'}, []]
      end

      # -> Here it is <-
      if env['PATH_INFO'] == '/'
        return [200,
          {'Content-Type' => 'text/html'},
          [File.read "public/index.html"]]
      end

      klass, act = get_controller_and_action(env)
      # ...
    end
  end
end
```

The version above returns the contents of public/index.html. You can return the controller and action by calling the same code as below for the given controller, or by resetting env. For instance,

inside the if statement you could write “env = { ‘PATH_INFO’ => ‘/home/index.html' } and not return, thus allowing the later code to treat that as the URL.

For a redirect, you’ll need to return a status of 301 or 302 instead of 200. And where you currently return the Content-Type header, you’ll want to return something like “{ ‘Location' => ‘/home/index.html’ }”. With a redirect, the client reads the Location header to find out where to redirect to.

Chapter 3

Exercise One: the code as given will fail if it’s in *any* call to const_missing. So one easy fix is to check the name of the constant, and only die if it’s in the middle of a call to *that* constant. That allows you to load a class, and then load another class that it requires. Here’s what that looks like:

```
class Object
  def self.const_missing(c)
    @calling_const_missing ||= {}
    return nil if @calling_const_missing[c]

    @calling_const_missing[c] = true
    require Rulers.to_underscore(c.to_s)
    klass = Object.const_get(c)
    @calling_const_missing[c] = false

    klass
  end
end
```

Fixing the multithreading problem is harder, and normally involves using a Mutex to prevent reloading a class while another thread

might be using or reloading it. A partial solution would be to keep a Mutex to prevent multiple threads from loading classes at once.

Some other possible improvements:

- Automatically add more directories to the LOAD_PATH
- Allow separate const-autoloading directories, not necessarily the normal Ruby LOAD_PATH.
- Look in both snake_case and CamelCase locations for the file, allowing either kind of naming

Exercise Two: just follow the code as given.

Chapter 4

Exercise One: first, follow the code as given. The next question is basically, “what if you wanted to test the code differently, with the application itself?” If you want to do that, it’s easiest to do it as a test in the application rather than the framework, naturally.

Some ways to do that:

- Create controller objects within the test like the example code
- Add your test to best_quotes, not rulers
- Create a test app within rulers, including controllers and views
- Create a test app object but use mocks to fake methods

I’m sure there are other methods.

To update your other test, you’ll need to add a route for it in the test app, and make it possible to return it, assuming it uses a different URL.

Exercise Two: the env method is just an instance method on the controller. So you can add more instance methods on the controller and get the other possibilities mentioned. For instance:

```
# rulers/lib/rulers.rb
module Rulers
  class Controller
    def controller_name
      self.class
    end

    def user_agent
      @env[]
    end

    def request_start_time
      # Note: for this, you'll
      # need to set @start_time
      # to Time.now at the beginning
      # of Application#call().
      @start_time
    end

    def rulers_version
      Rulers::VERSION
    end
  end
end
```

The render call takes a hash of variables. So if you want all the @instance variables from the controller set, one way is to pass them with the same name, but without the leading @, by defining

them all as variables. The easiest way to do this is to default the render call to getting all those variables by default. The end result looks something like this:

```
# rulers/lib/rulers/controller.rb
module Rulers
  class Controller
    def initialize(env)
      @env = env
    end

    def env
      @env
    end

    def instance_vars
      vars = {}
      instance_variables.each do |name|
        vars[name[1..-1]] =
          instance_variable_get name.to_sym
      end
      vars
    end

    def render(view_name, locals = instance_vars)
      filename = File.join "app", “views",
        "#{view_name}.html.erb"
      template = File.read filename
      eruby = Erubis::Eruby.new(template)
      eruby.result locals.merge(:env => env)
    end
  end
end
```

```
# best_quotes/app/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def test_rendering
    @noun = "a BlendTec blender"
    @another_noun = "a mutton, lettuce and " +
      "tomato sandwich"
    render :test_rendering
  end
end

# best_quotes/app/views/test_rendering.html.erb
<p>
  There is nothing either good or bad but
  <%= noun %> makes it so.
</p>

<p>
  Except <%= another_noun %>. It's so perky. I
  love that!
</p>
```

This lets you pass along all instance variables. If you want to do it full-on Rails style, you can invoke Erubis differently and define everything as an @instance variable instead of a local. I'll leave that as a further, extra-extra credit exercise for you.

Exercise Three: just run the code as given.

Chapter 5

Exercise One: the exercise says to just pick one quote (like #1) and just adjust one field. But let's show you how to do this the fun

way, which isn't too hard either. No shame if you just did what I asked, obviously!

In any case, you'll need to be able to save a FileModel. Let's look at how to do that:

```
# rulers/lib/rulers/file_model.rb (excerpt)
module Rulers
  class FileModel
    def save
      File.open(@filename, "w") do |f|
        f.write <<TEMPLATE
{
  "submitter": "#{@hash["submitter"]}",
  "quote": "#{@hash["quote"]}",
  "attribution": "#{@hash["attribution"]}"
}
TEMPLATE
      end
    end
  end
end
```

It should look a lot like the update. We already have @filename, though you could use @hash["id"] too.

The action in the controller is complicated if you want to get parameters from the request. This gets them from rack.input (the body) instead of the query params, because it's a POST, not a GET. HTTP is a bit odd that way. You could just assign the parameters in the action and call it a day.

```
# best_quotes/controllers/quotes_controller.rb
class QuotesController < Rulers::Controller
  def update
    raise "Only POST to this route!" unless
      env["REQUEST_METHOD"] == "POST"
    body = env["rack.input"].read
    astr = body.split("&")
    params = {}
    astr.each do |a|
      name, val = a.split "="
      params[name] = val
    end
    quote = FileModel.find(params["id"].to_i)
    quote["submitter"] = params["submitter"]
    quote.save

    render :quote, :obj => quote
  end
end
```

That's not too awful. Here's how you'd test it with curl:

```
curl http://localhost:3001/quotes/update \
  -d submitter=Frodo -d id=1
```

That backslash means "continue the line without hitting enter", a standard shell convention. In other words, don't literally type the backslash, but type everything else on a single line (or ***do*** type the backslash, but then follow it with a newline.)

When you type the curl command above, it will set quote 1's submitter to "Frodo". You can change the name to a different one, which will change the submitter to that instead.

You can check http://localhost:3001/quotes/quote_1 to see what the current submitter is from your browser.

Exercise Two: the kind of caching we're talking about here is normally done in FileModel.find() or similar methods that create a new object instance in memory. If we had multiple model classes, we'd keep a hash table for each class, with keys for class name and row ID. In this case, we have only a single model type, so we use only a single hash table with IDs as the keys.

```
# rulers/lib/rulers/file_model.rb (excerpt)
module Rulers
  class FileModel
    def self.find(id)
      id = id.to_i
      @dm_style_cache ||= {}
      begin
        if @dm_style_cache[id]
          return @dm_style_cache[id]
        end
        m = FileModel.new("db/quotes/#{id}.json")
        @dm_style_cache[id] = m
        m
      rescue
        return nil
      end
    end
  end
end
```

There can be ugly issues with keeping a cache around, figuring out how to garbage collect and so on. That is, if you keep a copy of every object, eventually you can run out of memory. We're going to ignore those issues, though there *are* solutions to them.

The reason you'd use ActiveRecord caching over DataMapper caching is that by keeping your instance specific to you, you're making sure it never gets saved partway through, or when you don't expect it. It also never gets modified by a method you call, or by another thread, if those happen to do a .find() on the same row.

Neither way is perfect. But the ActiveRecord method scales better to more threads and deeper call stacks at the cost of reduced performance.

Exercise Three: the interesting part of find_all is that you'll need to look through all the files unless you build some kind of index. For files, that generally means a directory scan. Here's what that looks like:

```
# rulers/lib/rulers/file_model.rb (excerpt)
module Rulers
  module Model
    class FileModel
      def self.find_all_by_attrib(attrib, value)
        id = 1
        results = []
        loop do
          m = FileModel.find(id)
          return results unless m

          results.push(m) if m[attrib] == value
          id += 1
```

```
        end
      end
    end
  end
end
```

You can implement find_all_by_submitter from this as find_all_by_attrib("submitter", value).

And to implement every finder, you can do this:

```
# rulers/lib/rulers/file_model.rb (excerpt)
module Rulers
  module Model
    class FileModel
      def self.method_missing(method, *args)
        if method.to_s[0..11] == "find_all_by_"
          attrib = method.to_s[12..-1]
          return find_all_by_attrib attrib, args[0]
        end
      end
    end
  end
end
```

Chapter 6

Exercise One: just rename the method. This is to show that it's more like Rails this way. You don't need to write any other code.

Exercise Two: for this exercise, we'll make a view object. I'll put mine in rulers/lib/rulers/view.rb, and include it from rulers/lib/rulers.rb.

Notice that we use a trick from the view chapter again: generating the source for the view with Erubis so that we can control how it gets evaluated. We also add an example view helper, called h(), just to show how that works.

```
# rulers/lib/rulers/view.rb
module Rulers
  class View
    def set_vars(instance_vars)
      instance_vars.each do |name, value|
        instance_variable_set(name, value)
      end
    end

    def evaluate(template)
      eruby = Erubis::Eruby.new(template)
      # Locals are in addition to
      # instance variables, if any
      eval eruby.src
    end

    # View helper methods
    def h(str)
      URI.escape str
    end
  end
end
```

We'll need to make the controller use it. Here's one way:

```
# rulers/lib/rulers/controller.rb (excerpt)
module Rulers
  class Controller
    def instance_hash
      h = {}
      instance_variables.each do |i|
        h[i] = instance_variable_get i
      end
      h
    end

    def render(view_name, locals = {})
      filename = File.join "app", "views",
        controller_name, "#{view_name}.html.erb"
      template = File.read filename
      v = View.new
      v.set_vars instance_hash
      v.evaluate template
    end
  end
end
```

And we'll need a route and view in best_quotes:

```
# best_quotes/app/controllers/quotes_controller.rb
(excerpt)
class QuotesController < Rulers::Controller
  def view_test
    @noun = "roller skating"
```

```
    render :view_test
  end
end


# best_quotes/app/views/view_test.html.erb
<p>
  I love <%= @noun %>
  and <%= h @noun %>!
<p>
```

This view demonstrates the view helper method and the instance variables.

Notice that the above method does *not* use the local variable hash that gets passed to render. If you want to do that, you'll want to write a method_missing on the view object that checks for a local variable and returns its value. I'll leave that one to you if you want to do it.

Chapter 7

Exercise one: for this one, you'll go through the list of columns and define getters and setters. One simple choice is to do it in SQLiteModel.schema, when we first get the list of columns:

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
    class SQLiteModel
      def self.schema
        return @schema if @schema
        @schema = {}
```

```
          DB.table_info(table) do |row|
            @schema[row["name"]] = row["type"]
          end

          @schema.each do |name, type|
            define_method(name) do
              self[name]
            end
            define_method("#{name}=") do |value|
              self[name] = value
            end
          end

          @schema
        end
      end
    end
  end
```

You'll also want a lightly modified sqlite_test.rb to test it. Here's one that can work fine:

```
# best_quotes/sqlite_test.rb
require "sqlite3"
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite; end

# Create row
mt = MyTable.create "title" => "I saw it again!"
mt.title = "I really did!"
mt.save!
```

```
mt2 = MyTable.find mt.id
puts "Title: #{mt2.title}"
```

The only interesting thing it's doing is getting and setting columns on the model.

Exercise two: this exercise can use the same test code as the previous one. But be sure to remove the define_method calls from exercise one or you'll never call method_missing!

Here's a method_missing that can do the right thing:

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
    class SQLiteModel
      def method_missing(name, *args)
        if @hash[name.to_s]
          self.class.class_eval do
            define_method(name) do
              self[name]
            end
          end
          return self.send(name)
        end

        if name.to_s[-1..-1] == "="
          col_name = name.to_s[0..-2]
          self.class.class_eval do
            define_method(name) do |value|
              self[col_name] = value
            end
```

```
            end
            return self.send(name, args[0])
          end

          super
        end
      end
    end
end
```

There are several fine ways you can refactor it, but this does the right thing.

Exercise three: this exercise is intentionally open-ended: there's more than one way to do it, and more than one suggestion for what to do. For this answer I'll show a JSON-serialized field, converted on load and save.

To specify the field, you'll want a method on the model class and to do something with fields that were marked. We'll do that by modifying from_sql to take the field name, not just the value — which means we need to change both calls to it to take both, not just the value. I'll leave that change to you.

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
    class SQLiteModel
      def self.json_field *fields
        @json_fields ||= []
        @json_fields.concat fields.map(&:to_s)
      end
```

```
      def self.to_sql(key, val)
        if @json_fields.include?(key.to_s)
          return "'#{MultiJson.dump(val)}'"
        end

        #...Same as before...
      end
    end
  end
end
```

That's fine, and kind of fun. Now how do you turn it back into a structure on load?

```
# rulers/lib/rulers/sqlite_model.rb (excerpt)
module Rulers
  module Model
    class SQLiteModel
      def self.from_sql(key, val)
        if @json_fields.include?(key.to_s)
          return MultiJson.load val
        end
        val
      end

      # Just add one loop with from_sql()
      def self.find(id)
        row = DB.execute <<SQL
select #{schema.keys.join ","} from #{table}
where id = #{id};
SQL
        data = Hash[schema.keys.zip row[0]]
```

```
        data.each do |k, v|  ### <- added
          data[k] = from_sql(k, v)
        end
        self.new data
      end
    end
  end
end
```

That's fun. So what does a test of it look like?

```
# best_quotes/sqlite_test.rb
require "sqlite3"
require "rulers/sqlite_model"

class MyTable < Rulers::Model::SQLite
  json_field :body
end

# Create row
mt = MyTable.create "title" => "I saw it again!",
  "body" => {
    "text" => "Yes",
    "tags" => [ "totally", "yo" ],
  }
mt.save!

mt2 = MyTable.find mt["id"]

puts "Body: #{mt2["body"].inspect}"
```

Finally, I'll answer the question of which fields use which approach: load/save conversion is great when the conversion is time-consuming or otherwise inconvenient, so you only do it as often as you need to. Load/save conversion plus a dirty bit on the fields can prevent conversion except when it's absolutely required. That's also good when the result of conversion takes a lot of memory or isn't well-suited to being manipulated in Ruby.

Converting on every assignment can keep you from converting when there's *no* assignment, even without a dirty bit. It can detect conversion errors earlier, during the assignment, instead of on save. And it can reduce the complexity of load and save.

So it's a tradeoff and both sides are often reasonable. It's quite possible that you'd handle some columns one way and some another.

Chapter 8

Exercise one: this one is mostly "do it yourself." Here's what my config.ru looked like when I added the benchmark code:

```
# best_quotes/config.ru
require './config/application'

class BenchMarker
  def initialize(app, runs = 100)
    @app, @runs = app, runs
  end

  def call(env)
    t = Time.now

    result = nil
```

```
      @runs.times { result = @app.call(env) }

      t2 = Time.now - t
      STDERR.puts <<OUTPUT
Benchmark:
  #{@runs} runs
  #{t2.to_f} seconds total
  #{t2.to_f * 1000.0 / @runs} millisec/run
OUTPUT

      result
    end
end

use BenchMarker, 10_000

run BestQuotes::Application.new
```

On my laptop, I get about .160 milliseconds per request for actions like a_quote:

```
Benchmark:
  10000 runs
  1.597576 seconds total
  0.1597576 millisec/run
127.0.0.1 - - [10/Dec/2015 16:42:37] "GET /quotes/
a_quote HTTP/1.1" 200 - 1.5979
```

You may get a different total time, of course. And you should run multiple times — my first few runs were between about .159 and .162, for instance, and some actions will vary far more.

I see slower actions as well. Here's the "index" action that reads one JSON file, for instance:

```
Benchmark:
  10000 runs
  2.400778 seconds total
  0.24007779999999998 millisec/run
127.0.0.1 - - [10/Dec/2015 16:43:03] "GET /quotes/
index HTTP/1.1" 200 - 2.4011
```

And with more JSON files, it gets slower, as you'd expect. With 5 JSON files, one of them a bit larger, it takes close to twice as long:

```
Benchmark:
  10000 runs
  4.304467 seconds total
  0.43044669999999996 millisec/run
127.0.0.1 - - [10/Dec/2015 16:48:50] "GET /quotes/
index HTTP/1.1" 200 - 4.3050
```

There are many more benchmarking experiments you could do. And this is all with the unmodified middleware, not trying to take anything else into account.

Exercise two: this one is just "set up third-party middleware." For Google Analytics, that's as simple as adding it to the Gemfile and config.ru — though for that one in particular, I also needed to require JSON.

```
# best_quotes/Gemfile
source :rubygems
gem 'rulers', :path => "../rulers"
gem 'rack-google-analytics'


# best_quotes/config.ru
require './config/application'
require 'rack/google-analytics'
require  'json'

use Rack::GoogleAnalytics,
  :tracker => 'UA-30520713-5'

run BestQuotes::Application.new
```

You'll need to run "bundle install" after modifying the Gemfile.

This Google Analytics middleware actually only shows up if you have a <head></head> section in your HTML, so if you want to see it work, add that to the front of your HTML template, then view source of a page using it. Here's an example template that does that:

```
<html>
<head>
</head>
<body>
<p>
  There is nothing either good or bad but
    <%= noun %> makes it so.
</p>

<p>
  Ruby version <%= RUBY_VERSION %>
```

```
</p>
</body>
</html>
```

Exercise three: for a basic rack/lobster benchmark without a framework, I get around 21 or 22 microseconds (0.021 milliseconds) per request:

```
Benchmark:
  10000 runs
  0.21571 seconds total
  0.021571 millisec/run
```

That's pretty quick compared to the version with Rulers. But the hello-world-iest route I can image clocks in much faster. If you return a straight up proc, here's the run statement and a benchmark:

```
# config.ru (excerpt)
run proc { [200, {}, []] }
```

```
Benchmark:
  10000 runs
  0.003107 seconds total
  0.00031069999999999996 millisec/run
127.0.0.1 - - [10/Dec/2015 16:59:46] "GET / HTTP/
1.1" 200 - 0.0035
```

My benchmarks vary a bit on this, but it comes in at around a third of a ***microsecond*** on most runs — as opposed to 160

microseconds for a Rulers-enabled “hello, world” route using a template, for instance.

To test it with Rack::ContentType, be sure to move that middleware below the use statement for Benchmarker. For me, that brings it up to 4 or 5 microseconds per request:

```
Benchmark:
  10000 runs
  0.037814 seconds total
  0.0037814 millisec/run
127.0.0.1 - - [10/Dec/2015 17:02:10] "GET / HTTP/
1.1" 200 - 0.0386
```

So: much slower than a Rack-only hardcoded version, but still pretty darn quick.

Feel free to benchmark whatever you like - it’s a big topic, and this is just a start.

For the ending questions: most middleware, in most cases, will add about the same number of milliseconds of delay to most routes. And the requests/second can be calculated by dividing that — so adding a constant delay in milliseconds will *not* change that in a constant, easily-predicted way. It won’t just subtract 30 requests/second, say. Instead, it depends how much delay the route starts with.

Chapter 9

Exercise one: this is just a matter of adding a “root” method that calls match. That adds one method to the RouteObject:

```
# rulers/lib/rulers/routing.rb (excerpt)
class RouteObject
  def root(*args)
    match("", *args)
  end
```

Exercise two: default routes also aren't hard. We'll take one of the routes labeled "default routes" in config.ru and make it actually be a default.

That's also a change to the RouteObject. We'll take the just-controller rule and make it a default. That's the one that matches against "/quotes", for instance. This is what the parsed form of the rule is, the one that gets pushed onto @rules:

```
# rulers/lib/rulers/routing.rb (excerpt)
class RouteObject
  def initialize
    @rules = [
      {
        :regexp=>/^\/([a-zA-Z0-9]+)$/,
        :vars=>["controller"],
        :dest=>nil,
        :options=>{:default=>{"action"=>"index"}}
      }
    ]
  end
end
```

Make sure to remove the same rule (the last one) in config.ru to test it — you can't check that it's a default if you still have it there explicitly.

To apply the default rules last, create a @default_rules object for the defaults, then in check_url() you can test against "(@rules + @default_rules).each" to get the right ordering instead of just @rules.each.

If you do it this way, you can make optional sections in default routes really easily — you can just put parens around a section of a regexp and add a question mark afterward. That's because you're giving the regexp directly, not creating it from a match() call.

That doesn't let you write routes like that in the routes DSL, but lets you hardcode them into the framework. Kinda feels like cheating that way, though, doesn't it? So let's look at what it would take to put optional routing chunks into the framework properly.

We're also going to add the :via option, also used next exercise, because why not? It's only a few lines. This way, you know where it goes in the more complicated code too.

Here's the route I'm using to test in best_quotes:

```
# best_quotes/config.ru (excerpt)
app.route do
  match ":controller/:id/:action.(:type)?"
end
```

In rulers, there will be a number of changes.

```
# rulers/lib/rulers/routing.rb (excerpt)
class RouteObject
  def initialize
```

```
  @rules = []
  @default_rules = [
    {
      :regexp=>/^\/([a-zA-Z0-9]+)$/,
      :vars=>["controller"],
      :dest=>nil,
      :via =>false,
      :options=>{:default=>{"action"=>"index"}}
    }
  ]
end

def root(*args)
  match("", *args)
end

def match(url, *args)
  options = {}
  options = args.pop if args[-1].is_a?(Hash)
  options[:default] ||= {}

  dest = nil
  dest = args.pop if args.size > 0
  raise "Too many args!" if args.size > 0

  # Split on appropriate punctuation
  parts = url.split /(\/|\(|\)|\?|\.)/
  parts.select! { |p| !p.empty? }

  vars = []
  regexp_parts = parts.map do |part|
    if part[0] == ":"
      vars << part[1..-1]
```

```
        "([a-zA-Z0-9]+)"
      elsif part[0] == "*"
        vars << part[1..-1]
        "(.*)"
      elsif part[0] == "."
        "\\."  # Dot means literal dot
      else
        # Parens, slash and question fall
        # through to here
        part
      end
    end

    regexp = regexp_parts.join("")
    @rules.push({
      :regexp => Regexp.new("^/#{regexp}$"),
      :vars => vars,
      :dest => dest,
      :via  => options[:via] ?
        options[:via].downcase : false,
      :options => options,
      :orig => url,
    })
  end
```

Here, instead of splitting on slash and rejoining on it, we split on a variety of punctuation (slash, parens, question mark, dot) and do slightly different things with each of them. You can make this routing language as complicated, or as much like Rails, as you feel like. This is already a miniature compiler and compilers can get complex and deep very quickly.

Notice that we're also passing ":via" as an attribute of the rule. We'll use that again as we check the rule.

```
# rulers/lib/rulers/routing.rb (excerpt)
  def check_url(url, verb)
    (@rules + @default_rules).each do |r|
      next if r[:via] && r[:via] != verb.downcase

      m = r[:regexp].match(url)

      if m
        options = r[:options]
        params = options[:default].dup
        r[:vars].each_with_index do |v, i|
          params[v] = m.captures[i]
        end
        dest = nil
        if r[:dest]
          return get_dest(r[:dest], params)
        else
          controller = params["controller"]
          action = params["action"]
          return get_dest("#{controller}" +
            "##{action}", params)
        end
      end
    end

    nil
  end
end
```

The check_url method doesn't change much, but notice that we now check the ":via" option, if there is one. No sense going through an expensive, elaborate regexp check if a simple equality check fails right at the beginning.

Give that a try with the route in best_quotes at the beginning of this answer - one route that matches it is "/quotes/1/show.json", which should then pass in the right ID and type.

Exercise three: Read the previous exercise for the answer to ":via" — it's mostly about passing the ":via" parameter into the rule unchanged and then checking if :via matches at the beginning of check_url.

But we'll also need to add a "resource" method to the routing object, and implement it in terms of "match". So let's do that.

First, here's the important chart from the Rails Routing Guide:

HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Now, here's a simple version in code, implemented via "match":

rulers/lib/rulers/routing.rb

```
def RouteObject
  def resource(name)
    match "/#{name}",
      :default => { "action" => "index" },
      :via => "GET"
    match "/#{name}/new",
      :default => { "action" => "new" },
      :via => "GET"
    match "/#{name}",
      :default => { "action" => "create" },
      :via => "POST"
    match "/#{name}/:id",
      :default => { "action" => "show" },
      :via => "GET"
    match "/#{name}/:id/edit",
      :default => { "action" => "edit" },
      :via => "GET"
    match "/#{name}/:id",
      :default => { "action" => "update" },
      :via => "PUT"
    match "/#{name}/:id",
      :default => { "action" => "update" },
      :via => "PATCH"
    match "/#{name}/:id",
      :default => { "action" => "destroy" },
      :via => "DELETE"
  end
end
```

I’m cheating a little here in that I don’t do singular versus plural resources or sub-resources. You can do that (feel free!), but the complexity is all in tracking the structure of the routing language, not in implementing the match routes.

The difficulty in checking this, of course, is in writing a sample app to prove it out. Which reminds us that we'd really like a better "redirect" method…

Which further reminds us…

Okay, I'm just going to leave this as an exercise for the reader. Do you ever notice how everything you build reminds you of three more things to build?

You've made it through the final exercise of Rebuilding Rails. Have fun, and go with my blessing.

Appendix: Installing Ruby 2.0, Git, Bundler and SQLite3

Ruby

Any recent version of Ruby should be fine. You're welcome to install it through RVM or ruby-build. But they can be complicated, so I'm not recommending that if you don't already.

Windows

To install Ruby on Windows, go to http://rubyinstaller.org/, hit "download", and choose a recent version. This should download an EXE to install Ruby.

Mac OS X

To install Ruby on Mac OS X, you can first install Homebrew ("http://mxcl.github.com/homebrew/") and then "brew install ruby". For other ways, you can Google "mac os x install ruby".

Ubuntu Linux

To install Ruby on Ubuntu Linux , use apt-get:

```
> sudo apt-get install ruby-dev
```

This should install Ruby.

Others

Google for "install Ruby on <my operating system>". If you're using an Amiga, email me!

Git (Source Control)

Windows

To install git on Windows, we recommend GitHub's excellent documentation with lots of screenshots: "http://help.github.com/win-set-up-git/". It will walk you through installing msysgit ("http://code.google.com/p/msysgit/"), a git implementation for Windows.

Mac OS X

To install git on Mac OS X if you don't have it, download and install the latest version from "http://git-scm.com/". You won't see an application icon for git, which is fine - it's a command-line application that you run from the terminal. You can also install through Homebrew.

Ubuntu Linux

To install git on Ubuntu Linux, use apt.

```
> sudo apt-get install git-core
```

Others

Google for "install git on <my operating system>".

Bundler

Bundler is a gem that Rails uses to manage all the various Ruby gems that a library or application in Ruby uses these days. The number can be huge, and there wasn't a great way to declare them before Gemfiles, which come from Bundler.

To install bundler:

```
> gem install bundler
Fetching: bundler-1.1.17.gem (100%)
Successfully installed bundler-1.1.17
1 gem installed
```

Other gems will be installed via Bundler later. It uses a file called a Gemfile that just declares what gems your library or app uses, and where to find them.

SQLite

Windows

Go to sqlite.org. Pick the most recent stable version and scroll down until you see “Precompiled Binaries for Windows”. There is a This is what you’ll be using.

Mac OS X

Mac OS X ships with SQLite3. If the SQLite3 gem is installed correctly it should use it without complaint.

Ubuntu Linux

You’ll want to use apt-get (or similar) to install SQLite. Usually that’s:

```
> sudo apt-get install sqlite3 libsqlite3-dev
```

Others

Google for “install sqlite3 on <my operating system>”.

Other Rubies

If you’re adventurous, you know about other Ruby implementations (e.g. JRuby, Rubinius). You may need to adjust some specific code snippets if you use one.