Understand Rails by Building Your Own Ruby MVC Web Framework

This Version: October 2022

Cover photo by zibik on Unsplash

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!

Books vs Videos

This ebook and the Rebuilding Rails video course can both be useful. If you signed up for my email list, you've already seen links to the first couple of video chapters. So how are they different?

The videos are intended to be shorter, simpler and more to-the-point. They'll walk you through very specific code without exercises or detours, and with fewer little interesting bits of Ruby

trivia. They're the fastest way to get from no framework, to having built a framework. There will be less debugging, less open-ended challenge and less creativity. They'll take less of your time, and you'll learn a smaller, more specific set of things, faster.

Which one is "better" depends a lot on your goals. If you're trying to say "I finished!" as fast as possible, the videos do that better. If you're trying to learn debugging and framework design, the book will do more of that. There's nothing wrong with a "hold your hand" learning experience. There's nothing wrong with an "open-ended exploration" learning experience. You probably have a preference, though.

A word of caution: ***do not mix code*** from the ebook and the videos. They're similar enough to *look like* they'll work together, but they don't. You can't work through chapter 3 of the ebook and then start chapter 4 of the videos from the same code. You ***can*** download the given chapter 4 code and then start chapter 4. See the next section for details.

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 already part of most recent Rubies. If you don't already have it, it's 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.

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 using "bundle gem rulers". Depending on your version of Bundler, you'll get slightly different output. Here's the old, simpler way:

```
$ 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
```

Newer Bundler will ask you a lot of questions about what to include. I recommend saying "no" to basically everything - any of it can be added later, and you won't be distributing this code.

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. The old way to do it looks roughly like this (***don't add these examples***):

```
# Don't Add These, Please
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). If you see them, add the following:

```
spec.add_runtime_dependency "rack", "~>2.2"
```

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

However, newer versions of Bundler use a different method call: "add_dependency" instead of "add_runtime_dependency". If you're using newer Bundler, use add_dependency instead of add_runtime_dependency throughout this book.

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
```

This directory should be ***next to*** rulers. Neither one is nested inside the other. You can make different directory structures work, but I'm going to teach you the one that's the most like Rails, and the most like making your own framework 'for real.'

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?

Not In Rails, But...

When you made your gem, a recent version of Bundler would ***not*** put Gemfile.lock into .gitignore. Any recent Bundler is of the opinion that gems should check their Gemfile.lock into Git. That's not how it's always been done. You can see Bundler's thinking on that at https://bundler.io/blog/2018/01/17/making-gem-development-a-little-better.html.

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 deeply_empty?
    empty? || all?(&:empty?)
  end
end
```

This is a silly method - it returns true if the Array is empty (just []) or if it contains only objects that return true for 'empty?' like {} or [] or "". Despite the name it will ***not*** return true for something like ["", ["", ""], {}] because that middle array returns true for deeply_empty? but not for plain empty?. You could fix this with more complexity.

Have you seen "&:" before? It's a fun trick. ":blank?" means "the symbol blank?" just like ":foo" means "the symbol foo." The "&" means "pass as a block" -- so pass that symbol as if it were a code block in curly-braces that you'd pass to 'all?'. 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 [[], []].deeply_empty? and have it check. Go ahead, add a few more methods that could be useful to the applications that will use your framework.

What's useful? Rails defines methods like "present?" and "blank?" this way. In general, ActiveSupport is a great place to look for what kind of monkeypatching Rails does — there's a lot of it.

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
  # ...
  spec.add_runtime_dependency "rack"
  spec.add_development_dependency "rack-test"
  spec.add_development_dependency "minitest"
```

```
end
```

(With newer Bundler, remember to use add_dependency rather than add_runtime_dependency.)

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 "https://github.com/rack/rack/blob/main/SPEC.rdoc" 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:

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.

Appendix: Installing Ruby, 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
```

Often you'll already have it, though.

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, Ruby 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.