Extracted from:

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

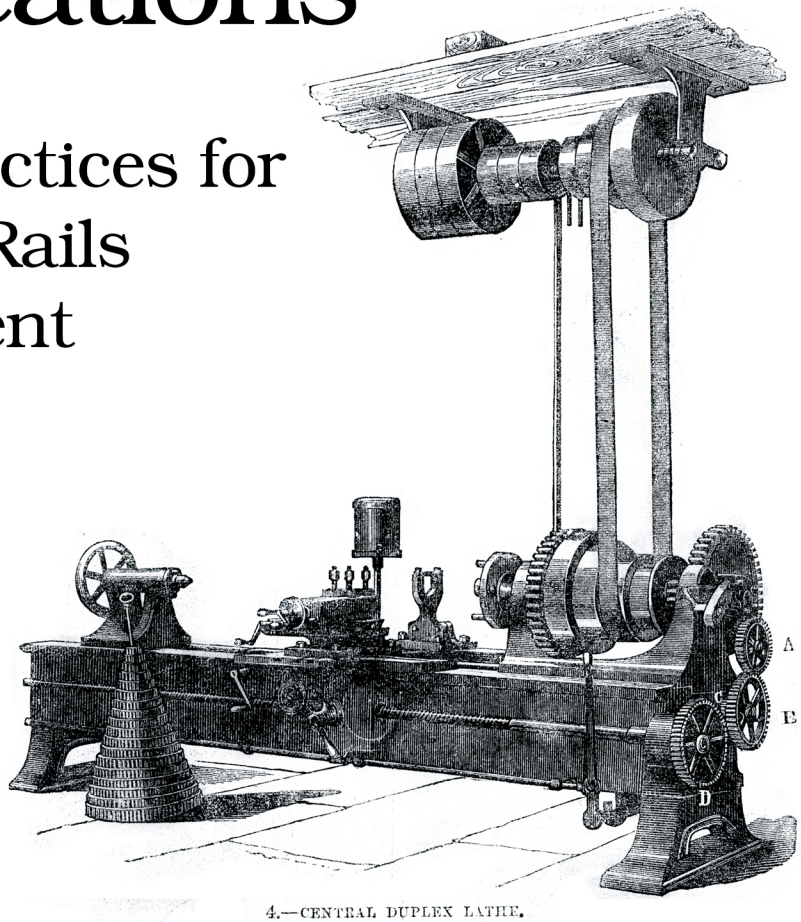
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development



4.—CENTRAL DUPLEX LATHE.

José Valim

edited by Brian P. Hogan

The Facets of Ruby Series

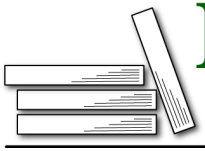
Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

Jose Valim

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-937785-55-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B2.0—June 21, 2013

Like many web frameworks, Rails uses the MVC architecture pattern to organize our code. The controller, most of the time, is responsible for gathering information from our models and sending the data to the view for rendering. On other occasions, the model is responsible for representing itself, and then the view does not take part in the request, as usually happens in JSON requests. Those two scenarios can be illustrated in the following index action:

```
class PostsController < ApplicationController
  def index
    if client_authenticated?
      render json: Post.all
    else
      render template: "shared/not_authenticated", status: 401
    end
  end
end
```

The common interface to render a given model or template is the render() method. Besides knowing how to render a :template or a :file, Rails also can render raw :text and a few formats like :xml, :json, and :js. Although the default set of options provided by Rails is enough to bootstrap our applications, we sometimes need to add new options like :pdf or :csv to the render() method.

To achieve this, Rails provides an API that we can use to create our own renderers. We'll explore this API as we modify the render() method to accept :pdf as an option and return a PDF created with Prawn,[1] a tiny, fast, and nimble PDF writer library for Ruby.

As most chapters in this book, we'll use the rails plugin generator to create a plugin that extends Rails capabilities. Let's get started!

1.1 Creating your first Rails Plugin

If you already have Rails installed, you are ready to craft your first plugin. Let's call this plugin pdf_renderer:

```
$ rails plugin new pdf_renderer
```

When we run this command we see the following output:

```
    create
    create  README.rdoc
    create  Rakefile
    create  pdf_renderer.gemspec
    create  MIT-LICENSE
    create  .gitignore
```

1. https://github.com/prawnpdf/prawn

```
    create  Gemfile
    create  lib/pdf_renderer.rb
    create  lib/tasks/pdf_renderer_tasks.rake
    create  lib/pdf_renderer/version.rb
    create  test/test_helper.rb
    create  test/pdf_renderer_test.rb
    append  Rakefile
vendor_app  test/dummy
       run  bundle install
```

This command creates the basic plugin structure, containing a pdf_renderer.gemspec file, a Rakefile, a Gemfile and the lib and test folders. The second to last step is a little more interesting; This command generates a full-fledged Rails application inside the test/dummy directory, which allow us to run our tests inside a Rails application context.

The generator finishes by running bundle install, which uses Bundler[2] to install all dependencies required by our project. With everything set up, let's explore the generated files.

pdf_renderer.gemspec

The pdf_renderer.gemspec provides a basic gem specification. The specification declares the gem authors, its version, its dependencies, the gem source files and more. This allows us to easily bundle our plugin into a Ruby Gem, making it easy for us to share our code across different Rails applications.

Notice the gem has the same name as the file inside the lib directory, which is pdf_renderer. By following this convention, whenever you declare this gem in a Rails application's Gemfile, the file at lib/pdf_renderer.rb will be automatically required. For now, this file contains only a definition for the PdfRenderer module.

Finally, notice that our gemspec does not contain a explicit version. Instead, the version is defined in lib/pdf_renderer/version.rb which is referenced in the gemspec as PdfRenderer::VERSION. This is a common practice in Ruby Gems.

Gemfile

In a Rails application, the Gemfile is used to list all sorts of dependencies, no matter if it is a development, test or production dependency. However, as our plugin already has a gemspec to list dependencies, the Gemfile simply reuses the gemspec dependencies. The Gemfile may eventually contain extra dependen-

2. http://gembundler.com/

cies which you find convenient to use during development, like the debugger or the excellent pry[3] gems.

To manage our plugin dependencies, we use Bundler. Bundler locks our environment to use only the gems listed in both the pdf_renderer.gemspec and the Gemfile, ensuring the tests are executed using the specified gems. Adding new dependencies and updating existing ones can be done by running the bundle install and bundle update commands in our plugin's root.

Rakefile

The Rakefile provides basic tasks to run the test suite, generate documentation and release our gem to the public. We can get the full list by executing rake -T at pdf_renderer's root:

```
$ rake -T
rake build         # Build pdf_renderer-0.0.1.gem into the pkg directory
rake clobber_rdoc  # Remove RDoc HTML files
rake install       # Build and install pdf_renderer-0.0.1.gem into system gems
rake rdoc          # Build RDoc HTML files
rake release       # Create tag v0.0.1 and build and push pdf_renderer...
rake rerdoc        # Rebuild RDoc HTML files
rake test          # Run tests
```

Booting the Dummy Application

rails plugin creates a dummy application inside our test directory, and the booting process of this application is similar to a normal application created with the rails command.

The config/boot.rb file has only one responsibility: to configure our application's load paths. The config/application.rb file should then load all required dependencies and configure the application, which is initialized in config/environment.rb.

The boot file generated by rails plugin can be found at test/dummy/config/boot.rb and it is similar to the application one, the only difference is that it needs to point to the Gemfile at the root of the pdf_renderer plugin. It also explicitly adds the plugin's lib directory to Ruby's load path, making our plugin available inside the dummy application:

pdf_renderer/1_prawn/test/dummy/config/boot.rb

```
# Set up gems listed in the Gemfile.
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../../../../Gemfile', __FILE__)

require 'bundler/setup' if File.exists?(ENV['BUNDLE_GEMFILE'])
$LOAD_PATH.unshift File.expand_path('../../../../lib', __FILE__)
```

3. http://pryrepl.org/

The boot file delegates the responsibility of setting up dependencies and their load path to Bundler. The test/dummy/config/application.rb is a stripped down version of the config/application.rb found in Rails applications:

pdf_renderer/1_prawn/test/dummy/config/application.rb

```
require File.expand_path('../boot', __FILE__)

require 'rails/all'

Bundler.require(*Rails.groups)
require "pdf_renderer"

module Dummy
  class Application < Rails::Application
    # ...
  end
end
```

There are no changes to the config/environment.rb; it is exactly the same as you'd find in a regular Rails application:

pdf_renderer/1_prawn/test/dummy/config/environment.rb

```
# Load the rails application.
require File.expand_path('../application', __FILE__)

# Initialize the rails application.
Dummy::Application.initialize!
```

Running Tests

rails plugin generates by default one sanity test for our plugin. Let's run our tests and see them pass with the following:

```
$ rake test
```

The output looks something like this:

```
Run options: --seed 20094

# Running tests:

.

Finished tests in 0.096440s, 10.3691 tests/s, 10.3691 assertions/s.

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

The test, defined in test/pdf_renderer_test.rb, just asserts that a module called PdfRenderer was defined by our plugin.

pdf_renderer/1_prawn/test/pdf_renderer_test.rb

```
require 'test_helper'

class PdfRendererTest < ActiveSupport::TestCase
  test "truth" do
    assert_kind_of Module, PdfRenderer
  end
end
```

Finally, note that our test file requires test/test_helper.rb, which is the file responsible for loading our application and configuring our testing environment. With our plugin skeleton created and a green test suite, we can start writing our first custom renderer.

1.2 Writing the Renderer

At the beginning of this chapter, we briefly discussed the render() method and a few options that it accepts, but we haven't formally described what a *renderer* is.

A renderer is nothing more than a hook exposed by the render() method to customize its behavior. Adding our own renderer to Rails is quite simple. Let's take a look at the :json renderer in Rails source code as an example:

rails/actionpack/lib/action_controller/metal/renderers.rb

```
add :json do |json, options|
  json = json.to_json(options) unless json.kind_of?(String)

  if options[:callback].present?
    self.content_type ||= Mime::JS
    "#{options[:callback]}(#{json})"
  else
    self.content_type ||= Mime::JSON
    json
  end
end
```

So, whenever we invoke the following method in our application:

```
render json: @post
```

it will invoke the block defined as the :json renderer. The local variable json inside the block points to the @post object, and the other options passed to render() will be available in the options variable. In this case, since the method was called without any extra options, it's an empty hash.

In the following sections, we want to add a :pdf renderer that creates a PDF document from a given template and sends it to the client with the appropriate headers. The value given to the :pdf option should be the name of the file to be sent.

The following is an example of the API we want to provide:

```
render pdf: 'contents', template: 'path/to/template'
```

Although Rails knows how to render templates and send files to the client, it does not know how to handle PDF files. For this, let's use Prawn.

Playing with Prawn

Prawn[4] is a PDF-writing library for Ruby. Since it is going to be a dependency of our plugin, we need to add it to our pdf_renderer.gemspec:

pdf_renderer/1_prawn/pdf_renderer.gemspec
```
s.add_dependency "prawn", "0.12.0"
```

Next, let's ask bundler to install our new dependency and test it via interactive Ruby:

```
$ bundle install
$ irb
```

Inside irb, let's create a sample PDF:

```
require "prawn"

pdf = Prawn::Document.new
pdf.text("A PDF in four lines of code")
pdf.render_file("sample.pdf")
```

Exit irb, and you can see a PDF file in the directory in which you started the irb session. Prawn provides its own syntax to create PDFs, and although this gives us a flexible API, the drawback is that it cannot create PDF from HTML files.

4. https://github.com/prawnpdf/prawn

Code in Action

Let's write some tests before we dive into the code. Since we have a dummy application at test/dummy, we can create controllers as in an actual Rails application and use them to test the complete request stack. Let's name the controller used in our tests HomeController and add the following contents:

pdf_renderer/1_prawn/test/dummy/app/controllers/home_controller.rb

```
class HomeController < ApplicationController
  def index
    respond_to do |format|
      format.html
      format.pdf { render pdf: "contents" }
    end
  end
end
```

Now let's create the PDF view used by the controller:

pdf_renderer/1_prawn/test/dummy/app/views/home/index.pdf.erb

```
This template is rendered with Prawn.
```

And add a route for the index action:

pdf_renderer/1_prawn/test/dummy/config/routes.rb

```
Dummy::Application.routes.draw do
  get "/home", to: "home#index", as: :home
end
```

Finally, let's write an integration test that verifies a PDF is in fact being returned when we access /home.pdf:

pdf_renderer/1_prawn/test/integration/pdf_delivery_test.rb

```
require "test_helper"

class PdfDeliveryTest < ActionDispatch::IntegrationTest
  test "pdf request sends a pdf as file" do
    get home_path(format: :pdf)

    assert_match "PDF", response.body
    assert_equal "binary", headers["Content-Transfer-Encoding"]

    assert_equal "attachment; filename=\"contents.pdf\"",
      headers["Content-Disposition"]
    assert_equal "application/pdf", headers["Content-Type"]
  end
end
```

The test uses the response headers to assert that a binary-encoded PDF file was sent as an attachment, including the expected filename. Although we cannot assert anything about the PDF body since it's encoded, we can at least

assert that it has the string PDF in it, which is added by Prawn to the PDF body. Let's run our test with rake test and watch it fail:

```
  1) Failure:
test_pdf_request_sends_a_pdf_as_file(PdfDeliveryTest):
Expected /PDF/ to match "This template is rendered with Prawn.\n".
```

The test fails as expected. Since we haven't taught Rails how to handle the :pdf option in render, it is simply rendering the template without wrapping it in a PDF. We can make the test pass by implementing our renderer in just a few lines of code inside lib/pdf_renderer.rb:

pdf_renderer/1_prawn/lib/pdf_renderer.rb

```
require "prawn"
ActionController::Renderers.add :pdf do |filename, options|
  pdf = Prawn::Document.new
  pdf.text render_to_string(options)
  send_data(pdf.render, filename: "#{filename}.pdf",
    disposition: "attachment")
end
```

And that's it! In this code block, we create a new PDF document, add some text to it, and send the PDF as an attachment using the send_data() method available in Rails. We can now run the tests and watch them pass. You can also go to test/dummy, start the server with rails server, and test it by yourself by accessing *http://localhost:3000/home.pdf*.

Even though our test passes, there is still some explaining to do. First of all, observe that we did not, at any point, set the Content-Type to application/pdf. How did Rails know which content type to set in our response?

The content type was set correctly because Rails ships with a set of registed formats and mime types:

rails/actionpack/lib/action_dispatch/http/mime_types.rb

```
Mime::Type.register "text/html", :html, %w( application/xhtml+xml ), %w( xhtml )
Mime::Type.register "text/plain", :text, [], %w(txt)
Mime::Type.register "text/javascript", :js,
  %w(application/javascript application/x-javascript)
Mime::Type.register "text/css", :css
Mime::Type.register "text/calendar", :ics
Mime::Type.register "text/csv", :csv

Mime::Type.register "image/png", :png, [], %w(png)
Mime::Type.register "image/jpeg", :jpeg, [], %w(jpg jpeg jpe pjpeg)
Mime::Type.register "image/gif", :gif, [], %w(gif)
Mime::Type.register "image/bmp", :bmp, [], %w(bmp)
Mime::Type.register "image/tiff", :tiff, [], %w(tif tiff)
```

```
Mime::Type.register "video/mpeg", :mpeg, [], %w(mpg mpeg mpe)

Mime::Type.register "application/xml", :xml, %w(text/xml application/x-xml)
Mime::Type.register "application/rss+xml", :rss
Mime::Type.register "application/atom+xml", :atom
Mime::Type.register "application/x-yaml", :yaml, %w( text/yaml )

Mime::Type.register "multipart/form-data", :multipart_form
Mime::Type.register "application/x-www-form-urlencoded", :url_encoded_form

Mime::Type.register "application/json", :json,
  %w(text/x-json application/jsonrequest)

Mime::Type.register "application/pdf", :pdf, [], %w(pdf)
Mime::Type.register "application/zip", :zip, [], %w(zip)
```

Notice how the PDF format is defined with its respective content type. When we requested the /home.pdf URL, Rails retrieved the pdf format from the URL, verified it matched with the format.pdf block defined in HomeController#index and proceeded to set the proper content type before invoking the block which called render.

Going back to our render implementation, although send_data() is a public Rails method and has been available since the first Rails versions, you might not have heard about the render_to_string() method. To better understand it, let's take a look at the Rails rendering process as a whole.