

Extracted from:

Crafting Rails Applications

Expert Practices for Everyday Rails Development

This PDF file contains pages extracted from Crafting Rails Applications, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2011 The Pragmatic Programmers, LLC.

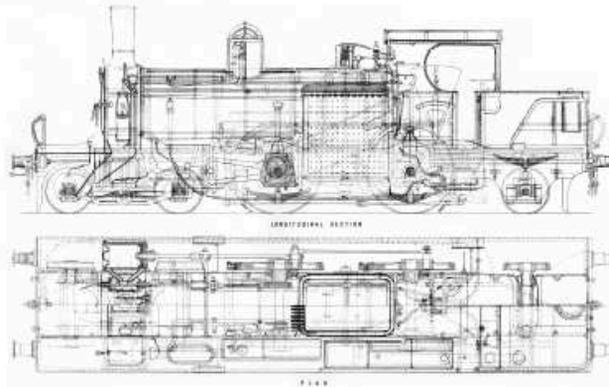
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The
Pragmatic
Programmers

Crafting Rails Applications

*Expert Practices for
Everyday Rails Development*



José Valim
edited by Brian P. Hogan

The Facets  of Ruby Series

In this chapter, we'll see

- Rails extensions and their basic structure
- how to customize the render method to accept custom options
- Rails rendering stack basics

Chapter 1

Creating our own renderer

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 XML requests. Those two scenarios can be illustrated in the index action below:

```
class PostsController < ApplicationController
  def index
    if client_authenticated?
      render :xml => 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.

Prior to Rails 3, there was no public API to add our own option to render and we needed to resort to methods like `alias_method_chain` to modify the rendering stack. Rails 3 changes this by introducing a new API which 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 in most chapters in this book, we'll develop the code as a Ruby Gem, making it easy to share the code across different Rails applications. To bootstrap those gems we will use a tool called *Enginex*² developed specifically for this book. In the same way the rails command generates a bare application, Enginex provides the `enginex` command that generates a bare gem for us.

Let's do it!

1.1 Generating projects with Enginex

Enginex is a Ruby gem that creates a bare project to be used within Rails 3 including a Rakefile, Gemfile and a ready-to-run test suite built on top of a Rails application. Enginex allows us to move from a simple gem to a Rails::Railtie and then to a Rails::Engine easily, as we will see in the next chapters. Let's install it:

```
gem install enginex
```

After we install Enginex, we are ready to craft our first gem for Rails 3. Let's call it `pdf_renderer`:

```
enginex pdf_renderer
```

The command's output is quite verbose; it tells us everything that is happening:

```
STEP 1 Creating gem skeleton
create
create pdf_renderer.gemspec
create Gemfile
create lib/pdf_renderer.rb
create MIT-LICENSE
create Rakefile
create README.rdoc
create test/pdf_renderer_test.rb
create test/integration/navigation_test.rb
create test/support/integration_case.rb
create test/test_helper.rb
create .gitignore
```

```
STEP 2 Vendoring Rails application at test/dummy
```

1. <http://github.com/sandal/prawn>
 2. <http://github.com/josevalim/enginex>

```

create
create README
create .gitignore
create Rakefile
create config.ru
create Gemfile
create app [...]
create config [...]
create db [...]
create doc [...]
create lib [...]
create log [...]
create public [...]
create script [...]
create test [...]
create tmp [...]
create vendor [...]

```

```

STEP 3 Configuring Rails application
force test/dummy/config/boot.rb
force test/dummy/config/application.rb
gsub test/dummy/config/environments/test.rb

```

```

STEP 4 Removing unneeded files
remove test/dummy/.gitignore
remove test/dummy/db/seeds.rb
remove test/dummy/doc
remove test/dummy/Gemfile
remove test/dummy/lib/tasks
remove test/dummy/public/images/rails.png
remove test/dummy/public/index.html
remove test/dummy/public/robots.txt
remove test/dummy/Rakefile
remove test/dummy/README
remove test/dummy/test
remove test/dummy/vendor

```

First, it creates the basic gem structure, including `lib` and `test` folders. Next, it creates a Rails 3 application at `test/dummy`, allowing us to run our tests inside a Rails 3 application context. The third step modifies the dummy application load path and configuration while the last step removes unneeded files. Let's take a deeper look at those generated files.

Gemfile

The Gemfile lists all required dependencies to run the tests in our newly created gem. To install those dependencies, you will need *Bundler*³. Bundler locks our environment to only use the gems listed in the Gemfile ensuring the tests are executed using the specified gems.

The generated Gemfile by default requires the following gems: rails, copybara (for integration tests) and sqlite3-ruby. Let's install these gems by running `bundle install` inside the `pdf_renderer` directory.

Rakefile

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

```
rake clobber_package # Remove package products
rake clobber_rdoc    # Remove rdoc products
rake rdoc            # Build the rdoc HTML Files
rake rerdoc         # Force a rebuild of the RDOC files
rake test           # Run tests
```

pdf_renderer.gemspec

The `pdf_renderer.gemspec` provides a basic gem specification. If at the end of this chapter, you want to use the gem in Rails applications, you just need to push it to a git repository and reference it in your application Gemfile.

Notice the gem has the same name as the file inside the `lib`, 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 loaded.

Booting the dummy application

Enginex creates a dummy Rails 3 application inside our test directory and the booting process of this application is the same as a normal application created with the `rails` command.

Different from previous versions, in Rails 3 the `config/boot.rb` file has only one responsibility: to configure our application's load paths. The `config/application.rb` should then load all required dependencies and configure the application, which is initialized in `config/environment.rb`.

3. <http://github.com/carlhuda/bundler>

That said, Enginex simply changes `test/dummy/config/boot.rb` to add `pdf_renderer` to the load path and to use the Gemfile at our gem root:

```
require 'rubygems'
gemfile = File.expand_path('../ ../../../../Gemfile', __FILE__)

if File.exist?(gemfile)
  ENV['BUNDLE_GEMFILE'] = gemfile
  require 'bundler'
  Bundler.setup
end

$: .unshift File.expand_path('../ ../../../../lib', __FILE__)
```

And then `test/dummy/config/application.rb` is modified to load `pdf_renderer` just after all dependencies are loaded with `Bundler.require`:

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

require "active_model/railtie"
require "active_record/railtie"
require "action_controller/railtie"
require "action_view/railtie"
require "action_mailer/railtie"

Bundler.require
require "pdf_renderer"
```

Finally, notice that we don't require `active_resource/railtie`. This is because Active Resource won't be discussed in this book, since it wasn't substantially changed in Rails 3.0.

Running tests

Enginex creates two sanity tests for our gem. Let's run our tests and see them pass with:

```
rake test
```

You should see an output similar to this:

```
Started
..
Finished in 0.039055 seconds.
```

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

The first test, defined in `test/pdf_renderer_test.rb`, just asserts that a module called `PdfRenderer` was defined in `lib/pdf_renderer.rb`:

```
require 'test_helper'
```

```

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

```

The other test, inside `test/integration/navigation_test.rb`, ensures that a Rails application was properly initialized by checking that `Rails.application` points to an instance of `Dummy::Application`, which is the application class defined at `test/dummy/config/application.rb`:

```
require 'test_helper'
```

```

class NavigationTest < ActiveSupport::IntegrationCase
  test "truth" do
    assert_kind_of Dummy::Application, Rails.application
  end
end

```

Notice the test uses `ActiveSupport::IntegrationCase`, which is not defined by Rails but inside `test/support/integration_case.rb` as shown below:

```

# Define a bare test case to use with Capybara
class ActiveSupport::IntegrationCase < ActiveSupport::TestCase
  include Capybara
  include Rails.application.routes.url_helpers
end

```

The test case above simply includes *Capybara*⁴, which provides a bunch of helpers to aid integration testing, and our application url helpers. The reason we chose to create our own `ActiveSupport::IntegrationCase` instead of using `ActionController::IntegrationTest` provided by Rails is inline with Capybara philosophy, which we will discuss in the future.

Finally, note that both test files require `test/test_helper.rb`, which is the file responsible for loading our application and configuring our testing environment. With our gem skeleton created and a green test suite, we can move onto 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 it accepts, but we haven't formally described what is a *renderer*.

4. <http://github.com/jnicklas/capybara>

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

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

```
add :xml do |xml, options|
  self.content_type ||= Mime::XML
  self.response_body = xml.respond_to?(:to_xml) ? xml.to_xml(options) : xml
end
```

So whenever we invoke the following method in our application:

```
render :xml => @post
```

It will invoke the block given with the `:xml` renderer. The local variable `xml` inside the block points to the `@post` object, and the other options given 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 file 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. Below is an example of the API we want to provide:

```
render :pdf => "contents", :template => "path/to/template"
```

While Rails knows how to render templates and send files to the client, it does not know how to handle PDF files. For this, we will use Prawn.

Playing with Prawn

*Prawn*⁵ is a PDF writing library for Ruby. We can install it as gem with the following command:

```
gem install prawn -v=0.8.4
```

Let's test this out by opening `irb` and creating a simple PDF file:

```
require 'rubygems'
require 'prawn'
pdf = Prawn::Document.new
pdf.text("A PDF in four lines of code")
pdf.render_file("recipes.pdf")
```

5. <http://github.com/sandal/prawn>

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, while this gives us a flexible API, the drawback is that it cannot create PDF from HTML files.

Code in action

With *Prawn* installed, we are ready to develop our renderer. Let's add *prawn* as a dependency to our Gemfile:

```
Download pdf_renderer/1_first_test/Gemfile
```

```
gem "prawn", "0.8.4"
```

After installing the dependencies and before writing the code, let's write some tests first. 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 call the controller used in our tests `HomeController` and add the following contents:

```
Download pdf_renderer/1_first_test/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 both HTML and PDF views for the index action:

```
Download pdf_renderer/1_first_test/test/dummy/app/views/home/index.html.erb
```

```
<p>Hey, you can download the pdf for this page by clicking the link below:</p>
<p><%= link_to "PDF", home_path("pdf") %></p>
```

```
Download pdf_renderer/1_first_test/test/dummy/app/views/home/index.pdf.erb
```

This is your new PDF content.

The HTML view only contains a link pointing to the PDF download. Finally, let's add a route for the index action:

```
Download pdf_renderer/1_first_test/test/dummy/config/routes.rb
```

```
Dummy::Application.routes.draw do
  match "/home(.:format)", :to => "home#index", :as => :home
end
```

Now let's write an integration test that verifies a PDF is in fact being returned when we click the PDF link at `/home`:

Download pdf_renderer/1_first_test/test/integration/navigation_test.rb

```
require 'test_helper'

class NavigationTest < ActiveSupport::IntegrationCase
  test 'pdf request sends a pdf as file' do
    visit home_path
    click_link 'PDF'

    assert_equal 'binary', headers['Content-Transfer-Encoding']
    assert_equal 'attachment; filename="contents.pdf"',
      headers['Content-Disposition']
    assert_equal 'application/pdf', headers['Content-Type']
    assert_match /Prawn/, page.body
  end
end

protected

def headers
  page.response_headers
end
end
```

The test inherits from `ActiveSupport::IntegrationCase` and uses a few helpers defined in `Capybara`, like `visit` and `click_link`, providing a clean and easy-to-read DSL to our integration tests. The test uses the headers to assert that a binary encoded PDF file was sent as attachment, including the expected filename, and while we cannot assert anything about the PDF body since it's encoded, we can at least assert that it was generated by `Prawn`. Let's run our test with `rake test` and watch it fail:

```
1) Error:
test_pdf_request_sends_a_pdf_as_file(NavigationTest):
NameError: uninitialized constant Mime::PDF
  app/controllers/home_controller.rb:5:in `index'
  app/controllers/home_controller.rb:3:in `index'
```

The test fails because we are calling `format.pdf` in our controller, but Rails does not know anything about PDF mime types. To find out what formats Rails 3 supports by default, let's take a quick look at Rails source code:

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

```
# Build list of Mime types for HTTP responses
# http://www.iana.org/assignments/media-types/

Mime::Type.register "text/html", :html, %( application/xhtml+xml ), %( xhtml )
Mime::Type.register "text/plain", :text, [], %(txt)
Mime::Type.register "text/javascript", :js,
  %( 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 "application/xml", :xml, %( 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, %( text/yaml )

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

# http://www.ietf.org/rfc/rfc4627.txt
# http://www.json.org/JSONRequest.html
Mime::Type.register "application/json", :json,
  %( text/x-json application/jsonrequest )

# Create Mime::ALL but do not add it to the SET.
Mime::ALL = Mime::Type.new("*/*", :all, [])

```

As no PDF format is defined, we need to add one. Let's start by writing some unit tests in the `test/pdf_renderer_test.rb` file and removing the existing test in the file as it has nothing to add. The test file will look like the following:

[Download pdf_renderer/2_adding_mime/test/pdf_renderer_test.rb](#)

```

require 'test_helper'

class PdfRendererTest < ActiveSupport::TestCase
  test "pdf mime type" do
    assert_equal :pdf, Mime::PDF.to_sym
    assert_equal "application/pdf", Mime::PDF.to_s
  end
end

```

The test makes two assertions that ensures whenever `format.pdf` is called, it will retrieve the `Mime::PDF` type and then set `"application/pdf"` as the response content type. In order to make this test pass, let's register the pdf mime type at `lib/pdf_renderer.rb`:

[Download pdf_renderer/2_adding_mime/lib/pdf_renderer.rb](#)

```

require "action_controller"
Mime::Type.register "application/pdf", :pdf

```

The code above ensures that Action Controller was already loaded and then registers `Mime::PDF`, making our unit test pass. However, when we run the integration test again, it still fails, but for a different reason:

```

1) Failure:
test_pdf_request_sends_a_pdf_as_file(NavigationTest)
  <"binary"> expected but was

```

```
<nil>.
```

The test fails because no header was sent. This is expected since we still haven't implemented our renderer. So let's write it in a few lines of code inside `lib/pdf_renderer.rb`:

```
Download pdf_renderer/3_final/lib/pdf_renderer.rb

require "action_controller"
Mime::Type.register "application/pdf", :pdf

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",
    :type => "application/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 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 `bundle exec rails server` and test it by yourself by accessing <http://localhost:3000/home> and clicking the link.

While `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 Rails rendering process as a whole.

1.3 Understanding Rails rendering stack

In versions earlier than Rails 3, Rails used to have a lot of code duplication between Action Mailer and Action Controller due to the fact that both have several features in common, like template rendering, helpers, and layouts.

In Rails 3 those shared responsibilities are centralized in Abstract Controller, which both Action Mailer and Action Controller use as their foundation. Abstract Controller also allows us to cherry pick exactly the features we want. For instance, if we want an object to have basic rendering capabilities, where it simply renders a template but does not include a layout, we just need to include `AbstractController::Rendering` in our object.

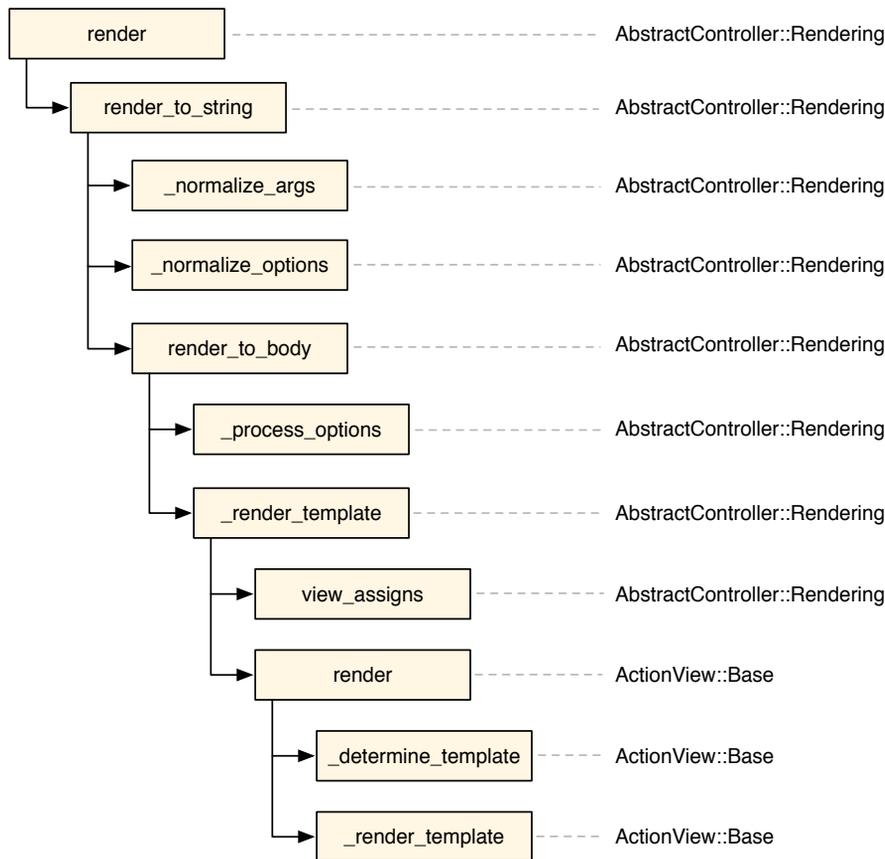


Figure 1.1: Visualization of the rendering stack when we call `render` with `AbstractController::Rendering`

When we include `AbstractController::Rendering` in an object, every time we call `render` the rendering stack proceeds as in Figure 1.1, on the next page.

Each rectangle represents a method, followed by the classes that implement it on the right. The arrows represent method calls. In this case, `render` calls the `render_to_string` method which calls three methods respectively: `_normalize_args`, `_normalize_options` and `render_to_body`. This can be confirmed by looking at both `render` and `render_to_string` implementations in Rails source code:

[Download rails/actionpack/lib/abstract_controller/rendering.rb](#)

```
def render(*args, &block)
  self.response_body = render_to_string(*args, &block)
end

def render_to_string(*args, &block)
  options = _normalize_args(*args, &block)
  _normalize_options(options)
  render_to_body(options)
end
```

Abstract Controller's rendering stack is responsible for normalizing the arguments and options given by you and converting them to a hash of options that complies to the public API defined by `ActionView::Base#render`. Each method showed in the stack in Figure 1.1, on the preceding page plays a specific role under this overall responsibility. These methods can be either private (starting with an underscore) or part of the public API.

The first relevant method in the stack is `_normalize_args` and it converts the arguments provided by the user into a hash. This allows the render method to be invoked as `render(:new)` which is converted by `_normalize_args` to `render(:action => "new")`. The hash returned by `_normalize_args` is then further normalized by `_normalize_options`. There is not much normalization happening inside `AbstractController::Rendering#_normalize_options` since it's the basic module, but it does convert `render(:partial => true)` calls to `render(:partial => action_name)`. So whenever you give `:partial => true` in a show action, the view will receive `:partial => "show"`.

After normalization, `render_to_body` is invoked. We can say that this is where the actual rendering starts to take place. The first step is to process all options that are meaningless to the view, using the `_process_options` method. While `AbstractController::Rendering#_process_options` is an empty method, we can look into `ActionController::Rendering#_process_options` for a handful of examples about what to do in this method. For instance, in controllers we are allowed to invoke:

```
render :template => "shared/not_authenticated", :status => 401
```

Here the `:status` option is meaningless to views, since `status` refers to the HTTP response status. So it's `ActionController::Rendering#_process_options` responsibility to intercept and handle this option and many others.

After options processing, `_render_template` is invoked and it creates an instance of `ActionView::Base`, called `view_context`, passing in the `view_assigns` and then calls `render` on it. If you are not familiar with the term *assigns*,

it references the group of variables available in the controller that will be accessible in the view. By default, whenever you set an instance variable in your controller, as `@posts = Post.all`, `@posts` is marked as an *assign* and will also be available in views.

At this point, it's important to note the inversion of concerns that happened between Rails 2.3 and Rails 3.0. In the former, the view was responsible for retrieving assigns from the controller, while in the latter, the controller *tells* the view which assigns to use.

Imagine that we want a controller that does not send any assigns to the view. In Rails 2.3, since the view automatically pulls in all instance variables from controllers, to achieve that we should either stop using instance variables in our controller or be sure to remove all instance variables before rendering a template. In Rails 3, since this responsibility is now in the controller, we just need to overwrite the `view_assigns` method to return an empty hash:

```
class UsersController < ApplicationController
  protected
  def view_assigns
    {}
  end
end
```

After assigns evaluation, the `render view` method is invoked and it breaks down in two main steps: `_determine_template` and `_render_template`. The former is responsible for finding the template, depending on the normalized options and hand it to the latter so the template is finally rendered.

This modular and well-defined stack allows anyone to hook into the rendering process and add his or her own features. This is what happens when we include `AbstractController::Layouts` in our object. The rendering stack is extended as exhibited in Figure 1.2, on the next page.

`AbstractController::Layouts` simply overrides `_normalize_options` to include the `:layout` option based on the value configured by the developer at the controller class level. Action Controller further extends the Abstract Controller rendering stack, adding and processing options which only makes sense in the controller scope. Those extensions are broken into three main modules:

- `ActionController::Rendering`: It overrides `render` to check whenever it's called twice raising a `DoubleRenderError` and also overrides `_process_options` to handle options like `:location`, `:status` and `:content_type`;

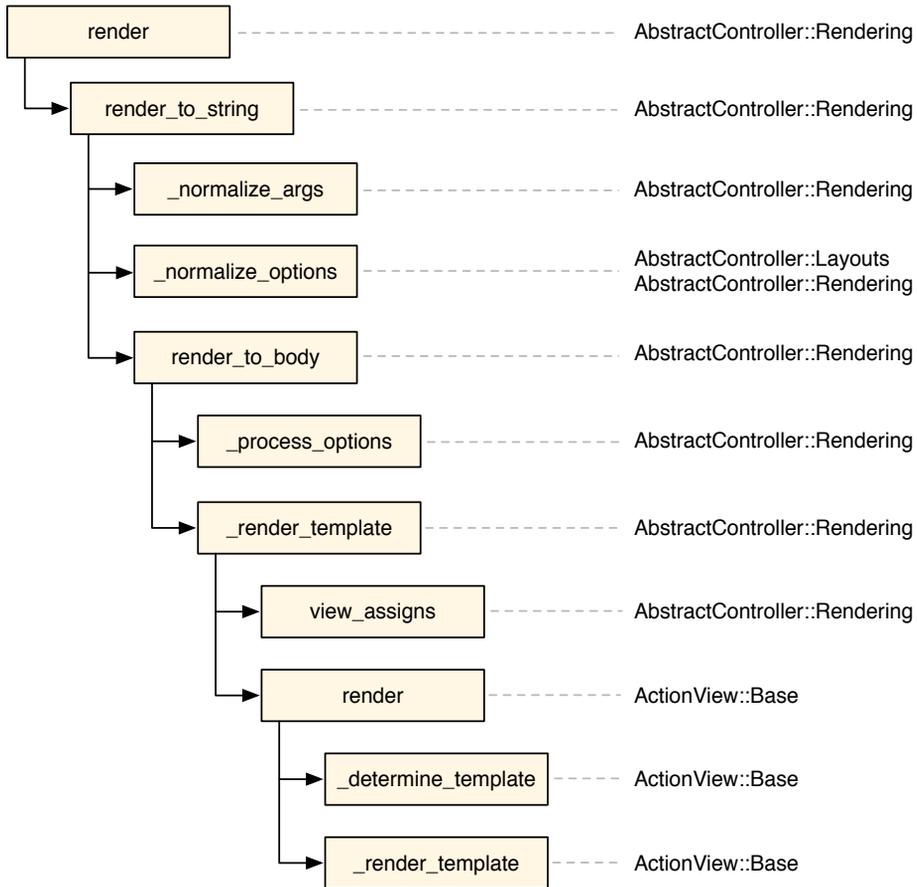


Figure 1.2: Visualization of the rendering stack when we call `render` with `AbstractController::Rendering` and `AbstractController::Layouts`

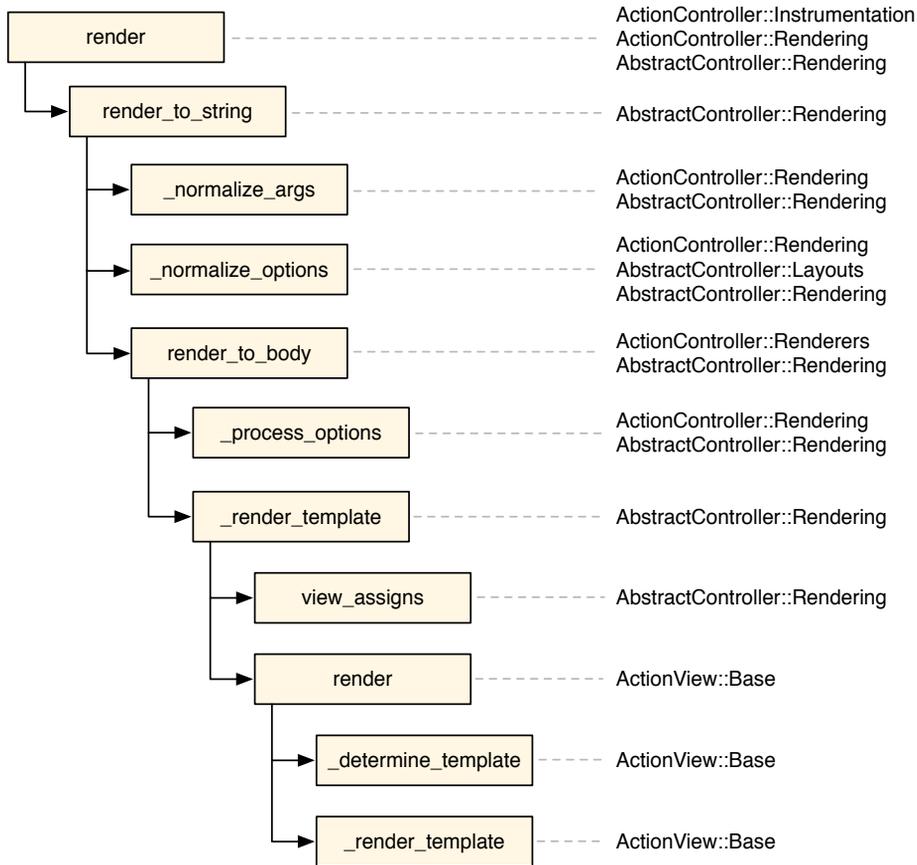


Figure 1.3: Visualization of the rendering stack when we call render with AbstractController and ActionController

- ActionController::Renderers: Adds the API we used in this chapter which allows us to trigger a specific behavior whenever a given key (like :pdf) is supplied;
- ActionController::Instrumentation: Overloads the render method so it can measure how much time was spent in the rendering stack;

The final stack with both Abstract Controller and Action Controller modules is shown in Figure 1.3, on the following page.

At first, it seems there are no differences between render and render_to_string, but when we analyze the whole rendering stack, we can see that some

Action Controller modules overload `render` to add additional behavior while leaving `render_to_string` alone.

For instance, by using `render_to_string` in our renderer, we ensure instrumentation events defined by `ActionController::Instrumentation` won't be triggered again and that it won't raise double render error, since both were added only to the `render` method.

1.4 Taking it to the next level

Going back to our renderer implementation, we now fully understand what happens when we add the following line to our controllers:

```
format.pdf { render :pdf => "contents" }
```

Internally, it becomes:

```
pdf.text render_to_string({})
```

And when we invoke `render_to_string` with an empty hash, the `_normalize_options` method in the rendering stack detects the empty hash and changes it to render the template with the same name as the current action. At the end, `render_to_string({})` is simply calling `render :template => "#{controller_name}/#{action_name}"` in the view object.

The fact that our renderer forwards the given options to the `render_to_string` method, allows us to also use the following syntax:

```
render :pdf => "contents", :template => "path/to/template"
```

And internally, it's the same as:

```
pdf.text render_to_string(:template => "path/to/template")
```

Which is sent straight to the view.

To finish this renderer, let's add a test ensuring this is exactly what happens! Our test invokes a new action in `HomeController` that calls `render` with both `:pdf` and `:template` options:

```
Download pdf_renderer/3_final/test/dummy/app/controllers/home_controller.rb
```

```
def another
  render :pdf => "contents", :template => "home/index"
end
```

Let's add a route for this new action:

```
Download pdf_renderer/3_final/test/dummy/config/routes.rb
```

```
match "/another(.:format)", :to => "home#another", :as => :another
```

Our test simply accesses `"/another.pdf"` and ensures a PDF is being returned:

```
Download pdf_renderer/3_final/test/integration/navigation_test.rb
test 'pdf renderer uses the specified template' do
  visit '/another.pdf'
  assert_equal 'binary', headers['Content-Transfer-Encoding']
  assert_equal 'attachment; filename="contents.pdf"',
    headers['Content-Disposition']
  assert_equal 'application/pdf', headers['Content-Type']
  assert_match /Prawn/, page.body
end
```

Now run the tests and watch them pass once again!

1.5 Wrapping up

In this chapter we created a renderer option for the PDF format. Feel free to play around with different options together with the `:pdf` option and understand how it works. From now on, you can easily create renderers for formats like PDF, CSV, ATOM and encapsulate any logic specific to your application in a renderer as well. You could even create a wrapper for other PDF libraries that are actually able to convert HTML files to PDF, like the paid *Prince XML*⁶ library or the open source *Flying Saucer*⁷ written in Java, but easily accessible through *JRuby*⁸.

We have also discussed Rails rendering stack and its modularity. Since Rails itself relies in this well-defined stack to extend Action Controller and Action Mailer, this API is by consequence more robust as it was battle-tested by Rails own features and different use-cases. As we will see in the chapters that follow, this was a common practice while designing most of the new Rails 3 APIs.

Rails' renderers open several possibilities to extend your rendering stack. But as any other powerful tool, remember to use it wisely.

Next, let's take a look at Active Model and its modules and create a Rails extension that can be used in Rails controllers and views.

6. <http://www.princexml.com/>

7. <http://xhtmlrenderer.dev.java.net/>

8. <http://jruby.org/>

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home Page for "Crafting Rails Applications"

<http://pragprog.com/titles/jvrails>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/jvrails.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)