Extracted from:

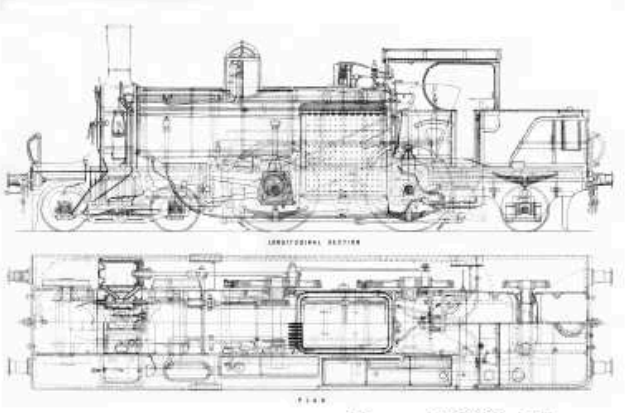
Crafting Rails Applications

Expert Practices for Everyday Rails Development

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 template handler API
- multipart templates with Action Mailer
- Rails generators and Railties

Chapter 4

Sending multipart e-mails with custom template handlers

To finish our tour of the Rails rendering stack, let's look at how templates are compiled and rendered by Rails. So far, we learned that *controllers'* responsibility is to normalize the rendering options and send them to the *view context*. Based on these options, the *view context* asks the *view paths* to find a template in the many *resolvers* it contains.

As we saw in Section 3.1, *Writing the code*, on page 60, the resolver returns instances of ActionView::Template and at the moment those templates are initialized, we need to pass along an object called handler as an argument. Each extension, like .erb or .haml, has its own *template handler*.

The responsibility of the *template handler* in the rendering stack is to compile a template to Ruby source code. And to understand how this happens, let's develop a few template handlers on our own.

Our template handler aims to solve a particular issue. Even though the foundation for today's e-mails was created in 1970 and the version 4 of the HTML specification dates from 1997, we still cannot rely on sending HTML e-mails to everyone since many e-mail clients cannot render these properly.

This means that whenever we configure an application to send an HTML e-mail, we should also send a TEXT version of the same, creating the so-called multipart e-mail. If the e-mail's recipient uses a client that cannot read HTML, it will fall back to the TEXT part.

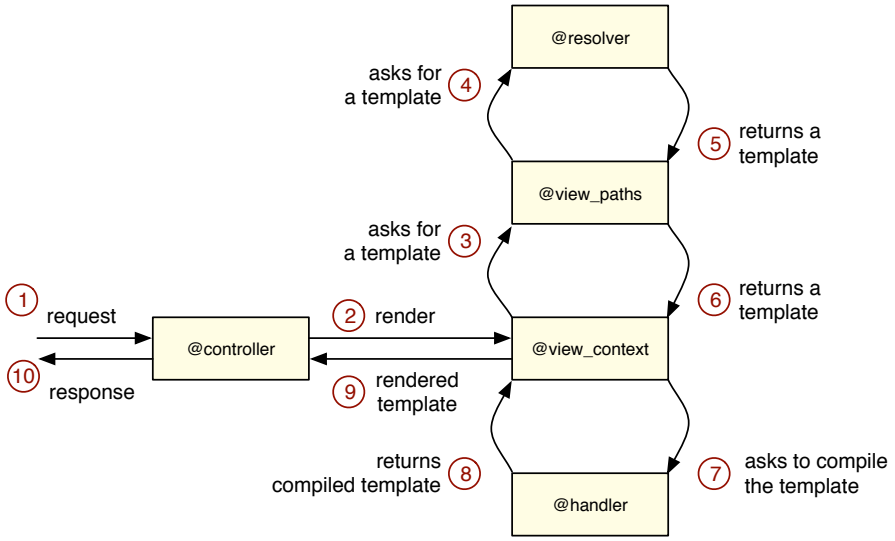


Figure 4.1: Objects involved in the rendering stack

While Action Mailer and the Mail gem make creation of multipart e-mails a breeze, the only issue with this approach is that we have to maintain two versions of the same e-mail message. Wouldn't it be nice if we actually have one template, that could be rendered both as TEXT and as HTML?

Here's where Markdown comes in. *Markdown*[1] is a lightweight markup language, created by John Gruber and Aaron Swartz, which is intended to be as easy-to-read and easy-to-write as possible. Markdown's syntax is based entirely of punctuation characters and allows you to embed custom HTML whenever required. Here's an example of Markdown text:

```
Welcome
=======

Hi, José Valim!

Thanks for choosing our product. Before you use it, you just need
to confirm your account by accessing the following link:

http://example.com/confirmation?token=ASDFGHJK
```

1. http://daringfireball.net/projects/markdown

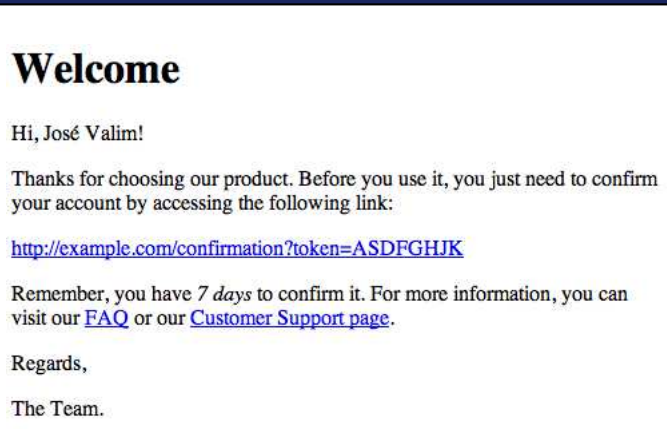


Figure 4.2: HTML generated from a Markdown template

```
Remember, you have *7 days* to confirm it. For more information,
you can visit our [FAQ][1] or our [Customer Support page][2].

Regards,

The Team.

[1]: http://example.com/faq
[2]: http://example.com/customer
```

Indeed, it's quite readable! The best part is that it can be transformed into HTML which is rendered as shown in Figure 4.2.

Our template handler is going to use the features of Markdown to generate both TEXT and HTML views using just one template. The only issue with Markdown is that it does not interpret Ruby code. To circumvent this, let's first compile our templates with ERb and then convert them using the Markdown compiler.

Finally, let's also hook into the Rails 3 generators and configure the mailer generator to use our new template handler instead of ERb.

4.1 Playing with the template handler API

In order to have an object compliant with the handler API, it just needs to respond to the call method. This method receives as an argument an instance of ActionView::Template, which we already discussed in Sec-

tion 3.1, *Writing the code*, on page 60, and should return a string containing valid Ruby code. The Ruby code returned by the handler is then compiled into a method, so rendering a template is as simple as invoking this compiled method.

Before diving into our Markdown + ERb handler, let's create a few template handlers to get acquainted with the API.

Ruby template handler

Our first template handler simply allows arbitrary Ruby code as a template. This means the following template is valid:

```
body = ""
body << "This is my first "
body << content_tag(:b, "template handler")
body << "!"
body
```

In order to implement this, let's craft a new gem called handlers using enginex:

```
enginex handlers
```

Next, let's write a simple integration test for our template handler:

Download handlers/1_handlers/test/integration/navigation_test.rb

```
require 'test_helper'

class NavigationTest < ActiveSupport::IntegrationCase
  test '.rb template handler' do
    visit '/handlers/index'
    expected = 'This is my first <b>template handler</b>!'
    assert_match expected, page.body
  end
end
```

The test makes a request to the /handlers/index path, let's define it in our router:

Download handlers/1_handlers/test/dummy/config/routes.rb

```
Dummy::Application.routes.draw do
  get "/handlers/:action", :to => "handlers"
end
```

Since our new route points to HandlersController, let's implement it as well:

Download handlers/1_handlers/test/dummy/app/controllers/handlers_controller.rb

```
class HandlersController < ApplicationController
```

```
end
```

And create our Ruby template at test/dummy/app/views/handlers/index.html.rb:

Download handlers/1_handlers/test/dummy/app/views/handlers/index.html.rb

```
body = ""
body << "This is my first "
body << content_tag(:b, "template handler")
body << "!"
body
```

When we run the test suite, it fails as Rails still does not recognize the .rb extension in templates. To register a new template handler, we invoke ActionView::Template.register_template_handler with two arguments: the template extension and the handler object. As the handler object is anything that responds to call and returns a String, we can implement our handler simply using Ruby's **lambda**. Ruby's **lambda** accepts a block and returns a Proc object that executes the given block once we invoke call and is a perfect fit as our template handler implementation is very short:

Download handlers/1_handlers/lib/handlers.rb

```
require "action_view/template"

ActionView::Template.register_template_handler :rb,
  lambda { |template| template.source }

module Handlers
end
```

Run the test suite and the test we just wrote now passes. Our lambda receives as an argument an ActionView::Template instance. Since our template handler needs to return a String with Ruby code and our template in the filesystem is written in Ruby, we just need to return the template.source.

As, since Ruby 1.8.7, symbols implement a to_proc method and :source.to_proc is exactly the same as lambda { |arg| arg.source }, we can make our template handler even shorter:

Download handlers/1_handlers/lib/handlers.rb

```
ActionView::Template.register_template_handler :rb, :source.to_proc
```

String template handler

Our .rb template handler is quite simple but has limited usage. Rails views are constituted mainly of static contents and handling big chunks

of strings in the Ruby code would quickly become messy. That said, let's implement another template handler more suitable to handle static content but still allows us to embed Ruby code. Since strings in Ruby supports interpolation, our next template handler will be based on strings and allow the following syntax:

Download handlers/2_more_handlers/test/dummy/app/views/handlers/show.html.string

```
Congratulations! You just created another #{@what}!
```

Our new template uses string interpolation and the interpolated Ruby code references an instance variable named @what. This variable is defined in controllers and given by the view_assigns method to the view, as we discussed in Section 1.3, *Understanding Rails rendering stack*, on page 25. So let's define a new action with this instance variable in our HandlersController to be used as fixture by our tests:

Download handlers/2_more_handlers/test/dummy/app/controllers/handlers_controller.rb

```
class HandlersController < ApplicationController
  def show
    @what = "template handler"
  end
end
```

And write a small test for it in our integration suite:

Download handlers/2_more_handlers/test/integration/navigation_test.rb

```
test '.string template handler' do
  visit '/handlers/show'
  expected = 'Congratulations! You just created another template handler!'
  assert_match expected, page.body
end
```

To make our new test pass, let's implement this new template handler, once again in lib/handlers.rb, as follow:

Download handlers/2_more_handlers/lib/handlers.rb

```
ActionView::Template.register_template_handler :string,
  lambda { |template| "%Q{#{template.source}}" }
```

Run the test suite and our new test passes. Our template handler returns a string created with the Ruby shortcut %Q{} which is then compiled to a method by Rails. When this method is invoked, Ruby interpreter evaluates the string and returns the interpolated result.

This template handler works fine for simple cases, but has two major flaws: adding the "}" character to the template causes syntax errors unless the character is escaped, and the block support is limited, as it

needs to be wrapped in the whole interpolation syntax. In other words, both of the following examples are invalid:

```
This } causes a syntax error

#{2.times do}
  This does not work as in ERb and is invalid
#{end}
```

So let's look at more robust template handlers next.

4.2 Building a template handler with Markdown + ERb

There are several gems out there that can compile Markdown syntax to HTML. For our template handler, let's use *RDiscount*[2], which is a Ruby wrapper to the fast Markdown compiler library called *Discount*, written in C.

Markdown Template Handler

Creating a template handler that can compile Markdown code is quite straightforward. Let's add another test to our suite:

Download handlers/2_more_handlers/test/integration/navigation_test.rb

```
test '.md template handler' do
  visit '/handlers/rdiscount'
  expected = '<p>RDiscount is <em>cool</em> and <strong>fast</strong>!</p>'
  assert_match expected, page.body
end
```

And then write our template in the filesystem:

Download handlers/2_more_handlers/test/dummy/app/views/handlers/rdiscount.html.md

```
RDiscount is *cool* and **fast**!
```

Note that our template uses .md as the extension for Markdown. Let's register it in Rails:

Download handlers/2_more_handlers/lib/handlers.rb

```
require "rdiscount"
ActionView::Template.register_template_handler :md,
  lambda { |template| "RDiscount.new(#{template.source.inspect}).to_html" }
```

Since our template handler relies on RDiscount, let's add it to the Gemfile and run bundle install just after:

2. http://github.com/rtomayko/rdiscount

Download handlers/2_more_handlers/Gemfile

```
gem "rdiscount", "1.6.5"
```

When we run the test suite, our new test passes. While our Markdown template handler works like a charm, it does not allow us to embed Ruby code, so its usage becomes quite limited. To circumvent this limitation, we could use the same technique we used in our .string template handler, but it also has its limitations when using Ruby blocks. Therefore, we are going to use ERb to embed Ruby code in our Markdown template and create a new template handler named .merb.

Markdown + ERb Template Handler

First, let's add an example of our new template handler to the filesystem. This example should be inside our dummy app and will be used in our tests:

Download handlers/2_more_handlers/test/dummy/app/views/handlers/merb.html.merb

```
MERB template handler is **<%= %w(cool fast).to_sentence %>**!
```

And then let's write a test that renders this template and check the desired output:

Download handlers/2_more_handlers/test/integration/navigation_test.rb

```
test '.merb template handler' do
  visit '/handlers/merb'
  expected = '<p>MERB template handler is <strong>cool and fast</strong>!</p>'
  assert_match expected, page.body.strip
end
```

This time, to implement our template handler, we are not going to use a **lambda**. Instead, let's create a module that responds to call, so, as our implementation grows, we will be able to split and refactor it in several methods, something that would not be possible if we used a **lambda**. Also, let's use the ActionView::Template.registered_template_handler method to retrieve the ERb handler, as we did in Section 3.1, *Writing the code*, on page 60. The code is shown below and should be added to our lib/handlers.rb file:

Download handlers/2_more_handlers/lib/handlers.rb

```
module Handlers
  module MERB
    def self.erb_handler
      @@erb_handler ||= ActionView::Template.registered_template_handler(:erb)
    end

    def self.call(template)
```

```
      compiled_source = erb_handler.call(template)
      "RDiscount.new(begin;#{compiled_source};end).to_html"
    end
  end
end

ActionView::Template.register_template_handler :merb, Handlers::MERB
```

The ERb handler compiles the template and, as any other template handler, it returns a string with valid Ruby code. The result returned by this Ruby code is a String containing Markdown syntax which is then converted to HTML using RDiscount.

Finally, look how we wrapped the code returned by ERb in an inline **begin/end** clause. This must be done inline or it will mess up backtrace lines. For instance, imagine the following template:

```
<% nil.this_method_does_not_exist! %>
```

This template obviously raises an error when rendered. However, consider those two ways to compile the template:

```
RDiscount.new(begin
  nil.this_method_does_not_exist!
end).to_html

RDiscount.new(begin;nil.this_method_does_not_exist!;end).to_html
```

In the first case, it says the error was raised in the second line, while in the latter, it correctly accuses the first line. And we need to use begin/end to wrap the code, otherwise it's not valid Ruby code. Let's verify this by trying the following code in irb:

```
puts(a=1;b=a+1)           # => raises syntax error
puts(begin;a=1;b=a+1;end) # => prints 2 properly
```

The last line in our implementation registers our new handler, allowing all tests to pass. Our .merb template handler is already implemented, but it still does not render both TEXT and HTML templates as described at the beginning of this chapter, only the latter. So let's change our template handler to output different results depending on the template format.

Multipart e-mails

The best way to showcase the behavior we want to add to our template handler is using multipart e-mails in Action Mailer. So let's create a mailer inside our dummy application to be used by our tests:

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)