Extracted from:

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

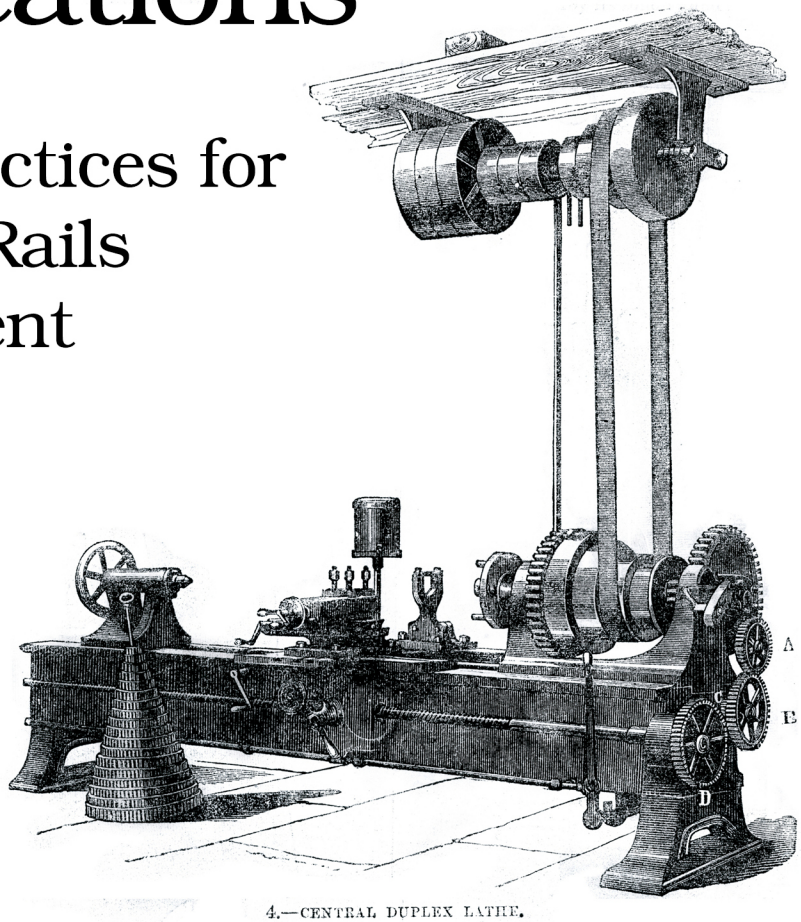
The Pragmatic Bookshelf

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

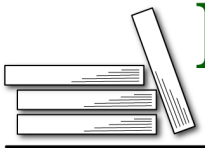
Crafting Rails 4 Applications

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

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 a controller's responsibility is to normalize the rendering options and send them to the view renderer. Based on these options, the view renderer asks the lookup context to search for a specific template in the available resolvers, also taking into account the locale and format values hold by the lookup context.

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

```
ActionView::Template.registered_template_handler("erb")
  #=> #<ActionView::Template::Handlers::ERB:0x007fc722516490>
```

The responsibility of the *template handler* in the rendering stack is to compile a template to Ruby source code. This source code is finally executed inside the view context, returning the rendered template. Figure 8, *Objects involved in the rendering stack*, on page 6 summarizes this process:

In order to understand how a template handler really works, we will build a template handler to solve a particular issue. Even though the foundation for today's emails was created in 1970 and version 4 of the HTML specification dates from 1997, we still cannot rely on sending HTML emails to everyone since many email clients cannot render these properly.

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

While Action Mailer makes creating multipart emails a breeze, the only issue with this approach is that we have to maintain two versions of the same email 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, that is intended to be as easy to read and easy to write as possible. Markdown's syntax consists entirely of punctuation characters and allows you to embed custom HTML whenever required. Here's an example of Markdown text:

```
Welcome
=======
```

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

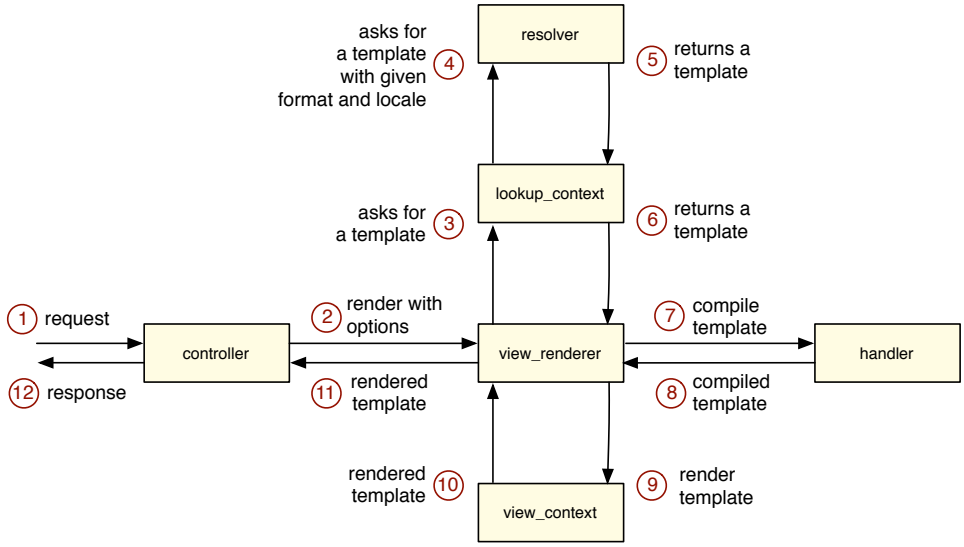


Figure 8—Objects involved in the rendering stack

```
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

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 next:

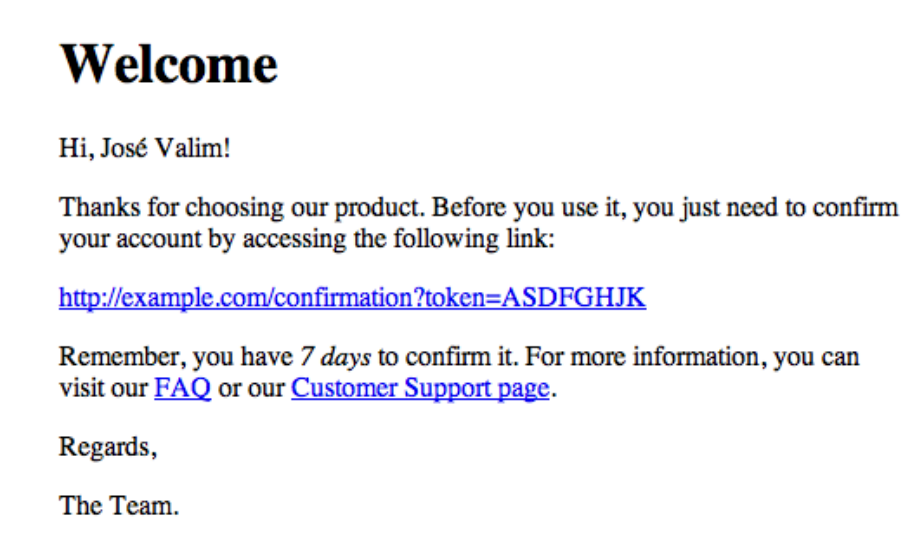


Figure 9—HTML generated from a Markdown template

Our template handler is going to use Markdown's features 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, we must first compile our templates with ERB and just then convert them using the Markdown compiler.

At the end of this chapter, we will hook into Rails' generators and configure the mailer generator to use our new template handler by default.

4.1 Playing with the Template Handler API

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 introduced in *Writing the Code*, on page ?, 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
```

To implement this, let's craft a new plugin called handlers using rails plugin:

```
$ rails plugin new handlers
```

Next, let's write a simple integration test for our template handler. Our goal is to render a dummy template at test/dummy/app/views/handlers/rb_handler.html.rb:

handlers/1_first_handlers/test/dummy/app/views/handlers/rb_handler.html.rb

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

Our integration test will need routes and a controller to serve that template, so let's add them:

handlers/1_first_handlers/test/dummy/config/routes.rb

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

handlers/1_first_handlers/test/dummy/app/controllers/handlers_controller.rb

```
class HandlersController < ApplicationController
end
```

Our integration test should make a request to the defined route at /handlers/rb_handler and assert our template was properly rendered:

handlers/1_first_handlers/test/integration/rendering_test.rb

```
require "test_helper"

class RenderingTest < ActionDispatch::IntegrationTest
  test ".rb template handler" do
    get "/handlers/rb_handler"
    expected = "This is my first <b>template handler</b>!"
    assert_match expected, response.body
  end
end
```

When we run the test suite, it fails because 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. Because the handler object is anything that responds to call() and returns a String, we can implement our handler simply using Ruby's lambda:

handlers/1_first_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 Ruby 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:

```
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 usually have big chunks of static contents, and handling those in the Ruby code would quickly become messy. That said, let's implement another template handler that is more suitable to handle static content but that still allows us to embed Ruby code. Since strings in Ruby support interpolation, our next template handler will be based on Ruby strings. Let's add a sample template to the dummy app:

handlers/1_first_handlers/test/dummy/app/views/handlers/string_handler.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. Let's define a new action with this instance variable in our HandlersController to be used as a fixture by our tests:

handlers/1_first_handlers/test/dummy/app/controllers/handlers_controller.rb
```
class HandlersController < ApplicationController
  def string_handler
    @what = "template handler"
  end
end
```

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

handlers/1_first_handlers/test/integration/rendering_test.rb
```
test ".string template handler" do
  get "/handlers/string_handler"
  expected = "Congratulations! You just created another template handler!"
```

```
  assert_match expected, response.body
end
```

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

handlers/1_first_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, the 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, because 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 it is time to look at more robust template handlers.

4.2 Building a Template Handler with Markdown + ERB

Several gems 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

We can create a template handler that compiles to Markdown in just a couple lines of code. Let's first add another test to our suite:

handlers/1_first_handlers/test/integration/rendering_test.rb

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

And then let's write our template in the filesystem:

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

handlers/1_first_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:

handlers/1_first_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 as a dependency to our plugin and run bundle install just after:

handlers/1_first_handlers/handlers.gemspec
```
s.add_dependency "rdiscount", "~> 2.0.0"
```

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 limitations on its own. Therefore, we are going to use ERB to embed Ruby code in our Markdown template and create a new template handler named .merb.

MERB 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:

handlers/1_first_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:

handlers/1_first_handlers/test/integration/rendering_test.rb
```
test ".merb template handler" do
  get "/handlers/merb"
  expected = "<p>MERB template handler is <strong>cool and fast</strong>!</p>"
  assert_match expected, response.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(), allowing us to break our implementation into smaller methods. In order to compile to ERB, we will simply use the ERB handler that ships with Rails, which can be retrieved using the ActionView::Template.registered_template_handler() method, as we did in *Writing the Code, on page ?*. Our .merb template handler is shown next:

handlers/1_first_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 like 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 that 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 is going to raise 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 example, since we introduced new lines in the compiled template, the exception backtrace would say the error happened in the second line of the template, which would be misleading. Notice we also need to use begin/end to wrap the code; otherwise, our handler would generate invalid Ruby code when the template contains more than one Ruby expression. Let's verify this by trying the following sample 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, making all tests 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. We just need to make a couple changes to our template handler in order to output different results depending on the template format.

Multipart Emails

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

handlers/2_final/test/dummy/app/mailers/notifier.rb

```
class Notifier < ActionMailer::Base
  def contact(recipient)
    @recipient = recipient

    mail(to: @recipient, from: "john.doe@example.com") do |format|
      format.text
      format.html
    end
  end
end
```

This code should look familiar; just like respond_to() in your controllers, you can give a block to mail() to specify which templates to render. However in controllers, Rails chooses only one template to render, while in mailers, the block specifies several templates that are used to create a single multipart email.

Our email has two parts, one in TEXT and another in HTML. Since both parts will use the same template, let's create a template inside our dummy app but without adding a format to its filename:

handlers/2_final/test/dummy/app/views/notifier/contact.merb

```
Dual templates **rock**!
```

And let's write a test for that using this mailer and view:

handlers/2_final/test/integration/rendering_test.rb

```
test "dual template with .merb" do
  email = Notifier.contact("you@example.com")
  assert_equal 2, email.parts.size
  assert_equal "multipart/alternative", email.mime_type

  assert_equal "text/plain", email.parts[0].mime_type
  assert_equal "Dual templates **rock**!",
    email.parts[0].body.encoded.strip

  assert_equal "text/html", email.parts[1].mime_type
  assert_equal "<p>Dual templates <strong>rock</strong>!</p>",
```

```
    email.parts[1].body.encoded.strip
end
```

The test asserts that our email has two parts. Since the TEXT part is an alternative representation of the HTML part, the email should have a MIME type equal to multipart/alternative, which is automatically set by Action Mailer. The test then proceeds by checking the MIME type and body of each part. The order of the parts is also important; if the parts were inverted, most clients would simply ignore the HTML part, showing only TEXT.

When we run this test, it fails because our text/plain part contains HTML code and not only TEXT. This is expected, since our template handler always returns HTML code. To make it pass, we will need to slightly change the implementation of Handlers::MERB.call() to consider the template.formats:

handlers/2_final/lib/handlers.rb

```
def self.call(template)
  compiled_source = erb_handler.call(template)
  if template.formats.include?(:html)
    "RDiscount.new(begin;#{compiled_source};end).to_html"
  else
    compiled_source
  end
end
```

We inspect template.formats and check whether it includes the :html format. If so, we render the template as HTML; otherwise, we just return the code compiled by ERB, resulting in a TEXT template written in Markdown syntax. This allows us to send an email with both TEXT and HTML parts but using just one template!

With this last change, our template handler does exactly what we planned at the beginning of this chapter. Before we create generators for our new template handler, let's briefly discuss how template.formats is set.

Formats Lookup

In *Writing the Code*, on page ?, we discussed that the resolver is responsible for giving the :format option to templates. The resolver looks at three places to decide which format to use:

- If the template found has a valid format, it's used. In templates placed in the filesystem, the format is specified in the template filename, as in index.html.erb.
- However, if the template found does not specify a format, the resolver asks the template handler whether it has a default format.

- Finally, if the template handler has no preferred format, the resolver should return the same formats used in the lookup.

Because our contact.merb template does not specify a format, the resolver tries to retrieve the default format from our Handlers::MERB template handler. This default format is retrieved through Handlers::MERB.default_format(), but since our template handler does not respond to default_format(), the second step is also skipped. So, the resolver last option is to return the format used in the lookup. As we are using format.text and format.html methods, they automatically set the formats in the lookup to, respectively, TEXT and HTML.

For instance, if we defined Handlers::MERB.default_format() in our implementation to return :text or :html, our last test would fail. Our resolver would never reach the third step and would always return a specific format in the second step.

4.3 Customizing Rails Generators

With our template handler in hand and rendering multipart emails, the final step is to create a generator for our plugin. Our generator will hook into Rails' mailer generator and configure it to create .merb templates instead .erb.

In general, Rails generators have a single responsibility and provide hooks so other generators can do the remaining work. A quick look at the mailer generator in the Rails source code reveals the hooks it provides:

rails/railties/lib/rails/generators/rails/mailer/mailer_generator.rb

```
module Rails
  module Generators
    class MailerGenerator < NamedBase
      source_root File.expand_path("../templates", __FILE__)

      argument :actions, type: :array,
        default: [], banner: "method method"
      check_class_collision

      def create_mailer_file
        template "mailer.rb",
          File.join("app/mailers", class_path, "#{file_name}.rb")
      end

      hook_for :template_engine, :test_framework
    end
  end
end
```

Although we are not familiar with the whole Generators API yet, we can see that its main behavior is to copy a mailer template to app/mailers, which is implemented in the create_mailer_file() method. Notice the mailer generator does not say anything about the template engine or the test framework; it provides only hooks. This allows other libraries like Haml and RSpec developers to provide their own hooks, customizing the generation process.

The Active Model API and the decoupling in Rails generators are the major keys to agnosticism in Rails. We have already discussed the former in Chapter 2, *Building Models with Active Model*, on page ?, and now we are going to play with the latter.

The Structure of a Generator

To briefly describe how a generator works, let's take a deeper look at the Rails::Generators::MailerGenerator shown in code on page 16. The mailer generator inherits from Rails::Generators::NamedBase. All generators that inherit from it expect an argument called NAME to be given when the generator is invoked from the command line. Let's verify it by executing the following command inside a Rails application:

```
$ rails g mailer --help
Usage:
  rails generate mailer NAME [method method] [options]

Options:
  -e, [--template-engine=NAME]  # Template engine to be invoked
```

```
                                # Default: erb
  -t, [--test-framework=NAME]   # Test framework to be invoked
                                # Default: test_unit
```

Back to our generator code, the Rails::Generators::MailerGenerator class also defines :actions as an argument, on line 6. Since a default value was provided (an empty array), these actions are optional and appear between brackets in the previous help message.

Next, we invoke the class_collisions_check() method, which verifies that the NAME given to the generator is not already defined in our application. This is useful since it raises an error if we try to define a mailer named, for instance, Object.

On the next lines, we define the create_mailer_file() method, reproduced here for convenience:

```
def create_mailer_file
  template "mailer.rb",
    File.join("app/mailers", class_path, "#{file_name}.rb")
end
```

Rails generators work by invoking all public methods in the sequence they are defined. This construction is interesting because it plays well with inheritance: if you have to extend the mailer generator to do some extra tasks, you just need to inherit from it and define more public methods. Skipping a task is a matter of undefining some method. Whenever your new generator is invoked, it will first execute the inherited methods and then the new public methods you defined. As with Rails controllers, you can expose or run actions by accident by leaving a method declared as public.

The create_mailer_file() method invokes three methods: template(), class_path(), and file_name(). The first one is a helper defined in Thor,[3] which is the basis for Rails generators, while the others are defined by Rails::Generators::NamedBase.

Thor has a module called Thor::Actions, which contains several methods to assist in generating tasks. One of them is the previous template() method, which accepts two arguments: a source file and a destination.

The template() method reads the source file in the filesystem, executes the embedded Ruby code in it using ERB, and then copies the result to the given destination. All ERB templates in Thor are evaluated in the generator context, which means that instance variables defined in your generator are also available in your templates, as well as protected/private methods.

3. https://github.com/wycats/thor

The values returned by the two other methods, class_path() and file_name(), are inflected from the NAME given as an argument. To see all the defined methods and what they return, let's sneak a peek at the named_base_test.rb file in Rails' source code:

rails/railties/test/generators/named_base_test.rb

```
def test_named_generator_attributes
  g = generator ['admin/foo']
  assert_name g, 'admin/foo',  :name
  assert_name g, %w(admin),    :class_path
  assert_name g, 'Admin::Foo', :class_name
  assert_name g, 'admin/foo',  :file_path
  assert_name g, 'foo',        :file_name
  assert_name g, 'Foo',        :human_name
  assert_name g, 'foo',        :singular_name
  assert_name g, 'foos',       :plural_name
  assert_name g, 'admin.foo',  :i18n_scope
  assert_name g, 'admin_foos', :table_name
end
```

This test asserts that when admin/foo is given as NAME, as in rails g mailer admin/foo, we can access all those methods, and each of them will return the respective value given in the assertion.

Finally, the mailer generator provides two hooks: one for the template engine and another for the test framework. Those hooks become options that can be given through the command line as well. Summing it all up, the previous generator accepts a range of arguments and options and could be invoked as follows:

```
$ rails g mailer Notifier welcome contact --test-framework=rspec
```

Generators' Hooks

We already know Rails generators provides hooks. However, when we ask to use ERB as the template engine, how does the mailer generator know how to find and use it? Generators' hooks work thanks to a set of conventions. When you pick a template engine named :erb, the Rails::Generators::MailerGenerator will try to load one of the following three generators:

- Rails::Generators::ErbGenerator
- Erb::Generators::MailerGenerator
- ErbGenerator

And since all generators should be in the $LOAD_PATH, under the rails/generators or the generators folder, finding these generators is as simple as trying to require the following files:

- (rails/)generators/rails/erb/erb_generator
- (rails/)generators/rails/erb_generator
- (rails/)generators/erb/mailer/mailer_generator
- (rails/)generators/erb/mailer_generator
- (rails/)generators/erb/erb_generator
- (rails/)generators/erb_generator

If one of those generators is found, it is invoked with the same command-line arguments given to the mailer generator. In this case, it defines an Erb::Generators::MailerGenerator, which we are going to discuss next.

Template Engine Hooks

Rails exposes three hooks for template engines: one for the controller, one for the mailer, and one for the scaffold generators. The first two generate files only if some actions are supplied on the command line, such as in rails g mailer Notifier welcome contact or rails g controller Info about contact. For each action given, the template engine should create a template for it.

On the other hand, the scaffold hook creates all views used in the scaffold: index, edit, show, new and the _form partial.

The implementation of Erb::Generators::ControllerGenerator in Rails is:

rails/railties/lib/rails/generators/erb/controller/controller_generator.rb

```
require "rails/generators/erb"

module Erb
  module Generators
    class ControllerGenerator < Base
      argument :actions, type: :array,
        default: [], banner: "action action"

      def copy_view_files
        base_path = File.join("app/views", class_path, file_name)
        empty_directory base_path

        actions.each do |action|
          @action = action
          @path = File.join(base_path, filename_with_extensions(action))
          template filename_with_extensions(:view), @path
        end
      end
    end
  end
end
```

The only method we haven't discussed yet is filename_with_extensions(), defined in Erb::Generators::Base:

rails/railties/lib/rails/generators/erb.rb
```
require "rails/generators/named_base"

module Erb
  module Generators
    class Base < Rails::Generators::NamedBase
      protected
      def format
        :html
      end
      def handler
        :erb
      end
      def filename_with_extensions(name)
        [name, format, handler].compact.join(".")
      end
    end
  end
end
```

The Erb::Generators::ControllerGenerator creates a view file in app/views using the configured format and handler for each action given. The template used to create such views in the Rails source code looks like this:

rails/railties/lib/rails/generators/erb/controller/templates/view.html.erb
```
<h1><%= class_name %>#<%= @action %></h1>
<p>Find me in <%= @path %></p>
```

This, for rails g controller admin/foo bar, outputs the following in the file app/views/admin/foo/bar.html.erb:

```
<h1>Admin::Foo#bar</h1>
<p>Find me in app/views/admin/foo/bar</p>
```

The Erb::Generators::MailerGenerator class simply inherits from the previous controller generator and changes the default format to be :text, reusing the same logic:

rails/railties/lib/rails/generators/erb/mailer/mailer_generator.rb
```
require "rails/generators/erb/controller/controller_generator"

module Erb
  module Generators
    class MailerGenerator < ControllerGenerator
      protected

      def format
        :text
```

```
      end
    end
  end
end
```

And the template created for mailers looks like this:

rails/railties/lib/rails/generators/erb/mailer/templates/view.text.erb

```
<%= class_name %>#<%= @action %>

<%%= @greeting %>, find me in app/views/<%= @path %>
```

If we take a glance at the ERB generator's directory structure in the Rails source code at the railties/lib directory; we can easily see which templates are available, as in the following figure:

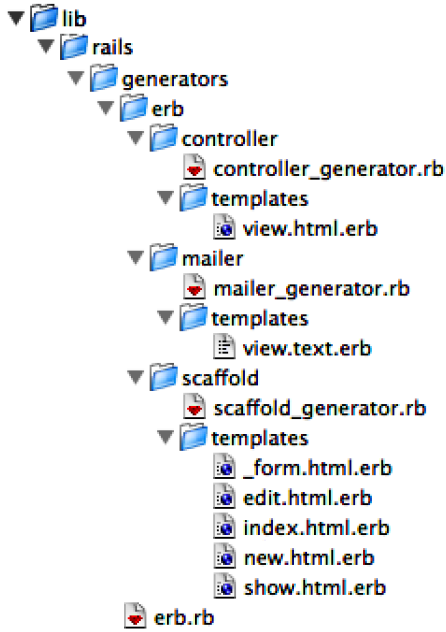


Figure 10—Structure for ERB generators

Therefore, if we want to completely replace ERB generators, we just need to create those generators and templates. And since Rails generators play well with inheritance, we can do that by inheriting from the respective ERB generator and overwriting a few configuration methods.

Creating Our First Generator

All we need to do to implement our .merb mailer generator is inherit from Erb::Generators::MailerGenerator and overwrite both format() and handler() methods defined Erb::Generators::Base. Our generator implementation looks like this:

handlers/2_final/lib/generators/merb/mailer/mailer_generator.rb

```
require "rails/generators/erb/mailer/mailer_generator"

module Merb
  module Generators
    class MailerGenerator < Erb::Generators::MailerGenerator
      source_root File.expand_path("../templates", __FILE__)

      protected
      def format
        nil # Our templates have no format
      end

      def handler
        :merb
      end
    end
  end
end
```

Note that we need to invoke a method called source_root() at the class level to tell Rails where to find the template used by our generator at lib/generators/merb/mailer/templates.

Since we chose nil as the format and :merb as the handler, let's create our template view.merb with the following content:

handlers/2_final/lib/generators/merb/mailer/templates/view.merb

```
<%= class_name %>#<%= @action %>

<%%= @greeting %>, find me in app/views/<%= @path %>
```

And that's it. Our template has the same contents as in the ERB generator, but you could modify it to include some Markdown by default. To try the generator, let's move to the dummy application inside our plugin at test/dummy and invoke the following command:

```
$ rails g mailer Mailer contact welcome --template-engine=merb
```

The previous command creates a mailer named Mailer with two templates named contact.merb and welcome.merb. The generator runs, showing us the following output:

```
create  app/mailers/mailer.rb
invoke  merb
create    app/views/mailer
create    app/views/mailer/contact.merb
create    app/views/mailer/welcome.merb
```

You can also configure your application at test/dummy/config/application.rb to use the merb generator by default, by adding the following line:

```
config.generators.mailer template_engine: :merb
```

However, you may not want to add this line to each new application you start. It would be nice if we could set this value as the default inside our plugin and not always in the application. Rails allows us to do it with a Rails::Railtie. This will be our last topic before we finish this chapter.