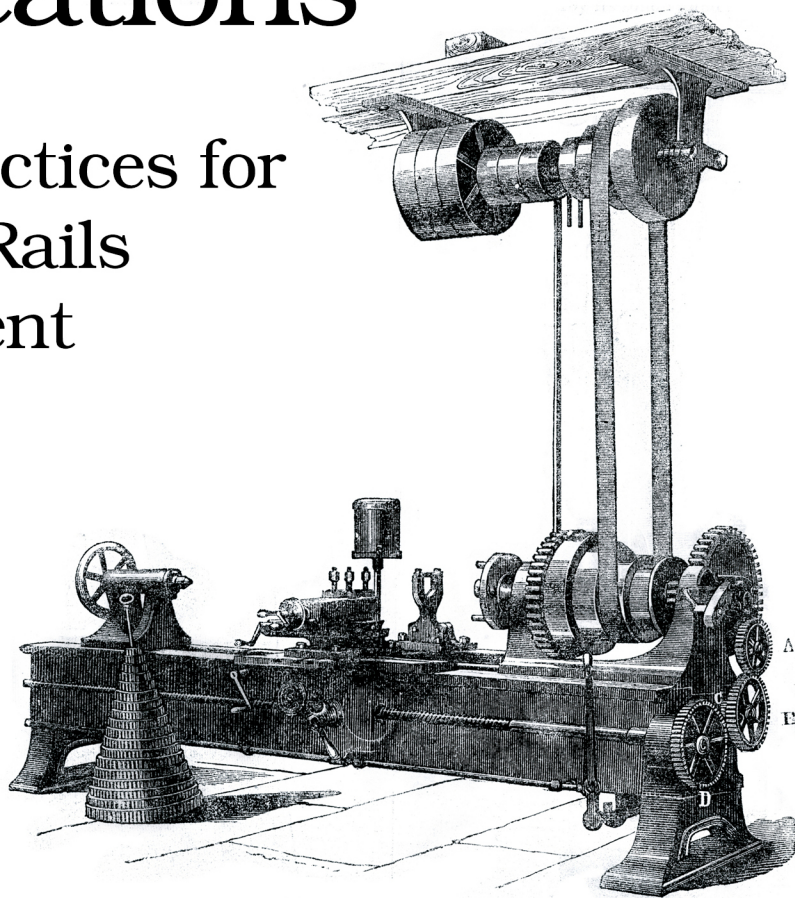


Crafting Rails 4 Applications

Expert Practices for
Everyday Rails
Development



4.—CENTRAL DUPLEX LATHE.

José Valim

edited by Brian P. Hogan





Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at pragprog.com/my_account. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at pragprog.com/titles/jvrails2/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Dave & Andy

Crafting Rails 4 Applications

Expert Practices for Everyday Rails Development

Jose Valim

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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>.

Copyright © 2013 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.

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

Contents

Changes	v
Preface	vii
1. Creating Our Own Renderer	1
1.1 Creating your first Rails Plugin	2
1.2 Writing the Renderer	5
1.3 Understanding The Rails Rendering Stack	9
1.4 Taking It to the Next Level	15
1.5 Wrapping Up	16
2. Building Models with Active Model	17
2.1 Creating Our Model	17
2.2 Integration tests with Capybara	28
2.3 Taking It to the Next Level	32
2.4 Wrapping Up	37
3. Retrieving View Templates from Custom Stores	39
3.1 Revisiting the Rendering Stack	39
3.2 Setting Up a SqlResolver	41
3.3 Configuring Our Resolver for Production	48
3.4 Serving Templates with Metal	56
3.5 Wrapping Up	60
4. Sending Multipart Emails Using Template Handlers	61
4.1 Playing with the Template Handler API	63
4.2 Building a Template Handler with Markdown + ERB	67
4.3 Customizing Rails Generators	71
4.4 Extending Rails with Railties	80
4.5 Wrapping Up	82

5.	<u>Streaming Server Events to Clients Asynchronously</u>	85
5.1	<u>Extending Rails with Engines</u>	86
5.2	<u>Live Streaming</u>	89
5.3	<u>File system notifications with Threads</u>	95
5.4	<u>Code loading techniques</u>	103
5.5	<u>Wrapping up</u>	107
6.	<u>Writing DRY Controllers with Responders</u>	109
6.1	<u>Understanding Responders</u>	110
6.2	<u>Exploring ActionController::Responder</u>	114
6.3	<u>The Flash Responder</u>	119
6.4	<u>HTTP Cache Responder</u>	124
6.5	<u>More Ways to Customize Generators</u>	128
6.6	<u>Wrapping Up</u>	134
7.	<u>Managing Application Events with Mountable Engines</u>	137
7.1	<u>Mountable and Isolated Engines</u>	137
7.2	<u>Storing Notifications in the Database</u>	139
7.3	<u>Rails and Rack</u>	146
7.4	<u>Middleware Stacks</u>	149
7.5	<u>Streaming with Rack</u>	157
7.6	<u>Wrapping Up</u>	161
8.	<u>Translating Applications Using Key-Value Backends</u>	163
8.1	<u>Revisiting Rails::Application</u>	164
8.2	<u>118n Backends and Extensions</u>	168
8.3	<u>Rails and Sinatra</u>	171
8.4	<u>Taking It to the Next Level with Devise and Capybara</u>	177
8.5	<u>Wrapping Up</u>	184

Changes

Beta 2 - June 21st, 2013

You'll find two new chapters: [Chapter 5, *Streaming Server Events to Clients Asynchronously*, on page 85](#) and [Chapter 8, *Translating Applications Using Key-Value Backends*, on page 163](#), making the book content-complete.

In addition, all the code has been updated to support Rails 4.0 RC2.

All outstanding errata has been addressed. Thanks to everyone who submitted items that needed fixing. Keep them coming!

Preface

When Rails was first released in 2004, it revolutionized how web development was done by embracing concepts like Don't Repeat Yourself (DRY) and convention over configuration. As Rails gained momentum, the conventions that were making things work so well on the golden path started to get in the way of developers who had the urge to extend how Rails behaved or even replace whole components.

Some developers felt that using DataMapper instead of Active Record was a better fit. Other developers turned to MongoDB and other nonrelational databases but still wanted to use their favorite web framework. Then there were those developers who preferred RSpec to Test::Unit. These developers hacked, cobbled, or monkey-patched solutions together to accomplish their goals because previous versions of Rails did not provide a solid API or the modularity required to make these changes in a clean, maintainable fashion.

With time, Rails started to listen to those developers and after years, the end result is a robust and wide set of plugin APIs, targetted to developers that want to customize their workflows, replace whole components, bending Rails to their needs without messy hacks.

This book guides you through these plugin APIs through practical examples. In each chapter, we will use test-driven development to build a Rails plugin or application that covers those APIs and how they fit in the Rails architecture. By the time you finish this book, you will understand Rails better and be more productive while writing more modular and faster Rails applications.

Who Should Read This Book?

If you're an intermediate or advanced Rails developer looking to dig deeper and make the Rails framework work for you, this is for you. We'll go beyond the basics of Rails; instead of showing how Rails lets you use its built-in features to render HTML or XML from a controller, we'll show you how the

`render()` method works so you can customize it to accept custom options, such as `:pdf`.

Rails Versions

All projects in *Crafting Rails Applications* were developed and tested against Rails 4.0.0. Future stable releases, like Rails 4.0.1, 4.0.2, and so forth, should be suitable as well. You can check your Rails version with the following command:

```
rails -v
```

And you can use `gem install` to get the most appropriate version:

```
gem install rails -v 4.0.0
```

This book also has excerpts from Rails' source code. All these excerpts were extracted from Rails 4.0.0.

Most of the APIs described in this book should remain compatible throughout Rails releases. Very few of them changed since the release of the first edition of this book.¹

Note for Windows Developers

Some chapters have dependencies that rely on C extensions. These dependencies install fine in UNIX systems, but Windows developers need the DevKit,² a toolkit that enables you to build many of the native C/C++ extensions available for Ruby.

Download and installation instructions are available online at <http://rubyinstaller.org/downloads/>.

Alternatively, you can get everything you need by installing RailsInstaller,³ which packages Ruby, Rails, and the DevKit, as well as several other common libraries.

What Is in the Book?

We'll explore the inner workings of Rails across eight chapters.

In [Chapter 1, *Creating Our Own Renderer*, on page 1](#), we will introduce rails plugin, a tool used throughout this book to create Rails plugins, and customize

-
1. <http://www.pragprog.com/titles/jvrails/>
 2. <http://rubyinstaller.org/downloads/>
 3. <http://railsinstaller.org>

render() to accept :pdf as an option with a behavior we will define. This chapter starts a series of discussions about Rails' rendering stack.

In [Chapter 2, *Building Models with Active Model*, on page 17](#), we will take a look at Active Model and its modules as we create an extension called Mail Form that receives data through a form and sends it to a preconfigured email.

Then in [Chapter 3, *Retrieving View Templates from Custom Stores*, on page 39](#), we will revisit the Rails rendering stack and customize it to read templates from a database instead of the filesystem. At the end of the chapter, we will learn how to build faster controllers using Rails' modularity.

In [Chapter 4, *Sending Multipart Emails Using Template Handlers*, on page 61](#), we will create a new template handler (like ERB and Haml) on top of Markdown.⁴ We'll then create new generators and seamlessly integrate them into Rails.

And in [Chapter 5, *Streaming Server Events to Clients Asynchronously*, on page 85](#), we will build a Rails engine that streams data to clients. We will also see how we can use Ruby's Queue class in the Ruby Standard Library to synchronize the exchange of information in between threads, finishing with a discussion about thread safety and eager loading.

In [Chapter 6, *Writing DRY Controllers with Responders*, on page 109](#), we will study Rails' responders and how we can use them to encapsulate controllers' behavior, making our controllers simpler and our applications more modular. We will then extend Rails responders to add HTTP Cache and internationalized Flash messages by default. At the end of the chapter, we'll learn how to customize Rails' scaffold generators for enhanced productivity.

In [Chapter 7, *Managing Application Events with Mountable Engines*, on page 137](#), we will build a mountable engine that stores information about each action processed by our application in a MongoDB database and exposes them for further analysis through a web interface. We will finish the chapter talking about Rack and Rack' middleware stacks while writing our own middleware.

Finally, in [Chapter 8, *Translating Applications Using Key-Value Backends*, on page 163](#), we will learn about I18n and customize it to read and store translations in a Redis data store. We will create an application that uses Sinatra as a Rails extension so we can modify these translations from Redis through

4. <http://daringfireball.net/projects/markdown>

a web interface. We will protect this translation interface using Devise⁵ and show Capybara's⁶ flexibility to write integration tests for different browsers.

Changes in the Second Edition

All of the projects and code examples have been updated and tested to work with Rails 4. The projects also use more up-to-date workflows for creating Rails plugins and interfacing with the framework.

In addition, [Chapter 5, *Streaming Server Events to Clients Asynchronously*, on page 85](#) is brand new and covers Rails' support for Server Sent Events, and digs into eager loading and thread safety. We also explore isolated and mountable engines and single file Rails applications in this edition.

How to Read This Book

We'll build a project from scratch in each chapter. Although these projects do not depend on each other, most of the discussions in each chapter depend on what you learned previously. For example, in [Chapter 1, *Creating Our Own Renderer*, on page 1](#), we discuss Rails' rendering stack, and then we take this discussion further in [Chapter 3, *Retrieving View Templates from Custom Stores*, on page 39](#) and finish it in [Chapter 4, *Sending Multipart Emails Using Template Handlers*, on page 61](#). In other words, you can skip around, but to get the big picture, you should read the chapters in the order they are presented.

Online Resources

The book's website⁷ has links to an interactive discussion forum as well as errata for the book. You'll also find the source code for all the projects we build. Readers of the ebook can click the gray box above the code excerpts to download that snippet directly.

If you find a mistake, please create an entry on the errata page so we can address it. If you have an electronic copy of this book, there are links in the footer of each page that you can use to easily submit errata to us.

Let's get started by creating a Rails plugin that customizes the `render()` method so we can learn how Rails' rendering stack works.

José Valim

jose.valim@plataformatec.com.br

April 2013

5. <https://github.com/plataformatec/devise>

6. <https://github.com/jnicklas/capybara>

7. <http://www.pragprog.com/titles/jvrails2/>

In this chapter, we'll see

- Rails plugins 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 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. <https://github.com/prawnpdf/prawn>

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
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 Bundler² 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.

2. <http://gembundler.com/>

Finally, notice that our `gemspec` does not contain an 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 dependencies 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`.

3. <http://pryrepl.org/>

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__)
```

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

Prawn⁴ 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. 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 registered 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.

1.3 Understanding The Rails Rendering Stack

Both Action Mailer and Action Controller have several features in common, such as template rendering, helpers, layouts and others. In order to avoid code duplication, those shared responsibilities are centralized in Abstract Controller, which both Action Mailer and Action Controller use as their foundation. At the same time, there are some features that are required by only one of the two libraries. Given those requirements, Abstract Controller was designed in a way a developer can cherry-pick only the functionality he wants. 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 the ActionController::Rendering module in our object, leaving ActionController::Layouts out.

When we include ActionController::Rendering in an object, the rendering stack proceeds as shown in the following figure every time we call render().

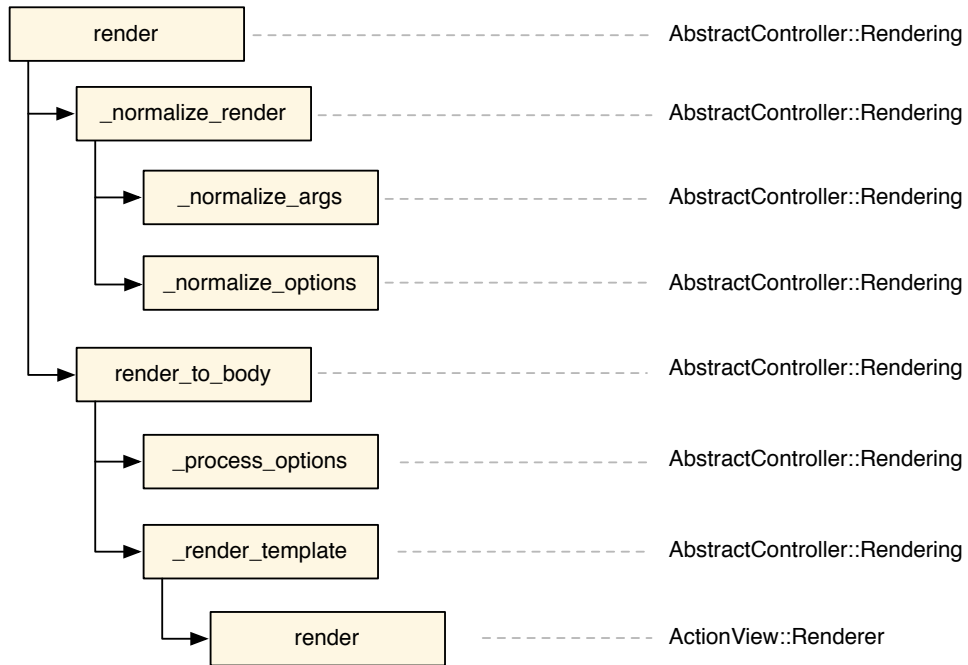


Figure 1—Visualization of the rendering stack when we call `render()` with `AbstractController::Rendering`

Each rectangle represents a method, followed by the classes or modules that implement it. The arrows represent method calls. For example, `render()` calls the `_normalize_render()` and then calls `render_to_body()`. The stack can be confirmed by looking at `AbstractController::Rendering` implementation in Rails source code:

```

rails/actionpack/lib/abstract_controller/rendering.rb
def render(*args, &block)
  options = _normalize_render(*args, &block)
  self.response_body = render_to_body(options)
end

def _normalize_render(*args, &block)
  options = _normalize_args(*args, &block)
  _normalize_options(options)
  options
end

def render_to_body(options = {})
  _process_options(options)
  _render_template(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 accepted by `ActionView::Renderer#render()` which will take care of finally rendering the template. Each method in the stack 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()`, invoked by `_normalized_render()`, 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, it becomes `partial: "show"` down the stack.

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. Although `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 the following:

```
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()`'s responsibility to intercept and handle this option and others.

After options processing, `_render_template()` is invoked and different objects start to collaborate. In particular, an instance of `ActionView::Renderer` called `view_renderer` is created and the `render()` method is called on it with two arguments: the `view_context` and our hash of normalized options:

```
rails/actionpack/lib/abstract_controller/rendering.rb
view_renderer.render(view_context, options)
```

The *view context* is an instance of `ActionView::Base` and it is called so because it is the context where our templates are evaluated. When you call `link_to()` in a template, it works because it is a method available inside `ActionView::Base`. When instantiated, the view context receives `view_assigns()` as argument. 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 highlight 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 and up, this responsibility is handled in the controller. We just need to override the `view_assigns()` method to return an empty hash:

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

By returning an empty hash, we ensure none of the actions in the controller pass assigns to the view.

With the view context and the hash of normalized options in hand, our `ActionView::Renderer` instance has everything it needs to find a template, based on the options, and finally render it inside the view context.

This modular and well-defined stack allows anyone to hook into the rendering process and add their own features. When we include `AbstractController::Layouts` on top of `AbstractController::Rendering`, the rendering stack is extended as shown in the following figure:

`AbstractController::Layouts` simply overrides `_normalize_options()` to support the `:layout` option. In case no `:layout` option is set when calling `render()`, one may be automatically set 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 that make sense only in the controller scope. Those extensions are broken into four main modules:

- `ActionController::Rendering`: Overall, it overrides `render()` to check whenever it's called twice, raising a `DoubleRenderError`, and also overrides `_process_options()` to handle options such as `:location`, `:status`, and `:content_type`

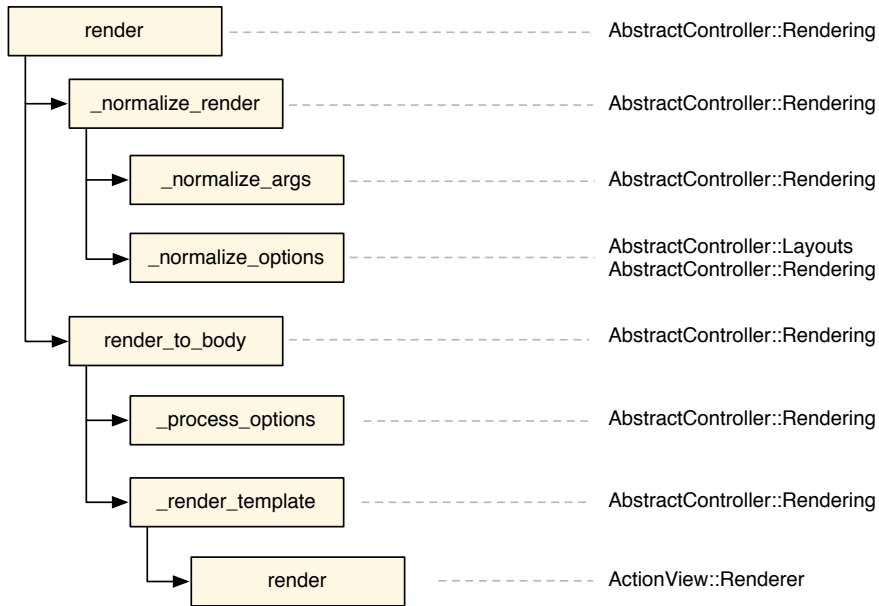


Figure 2—Visualization of the rendering stack when we call `render()` with `AbstractController::Rendering` and `AbstractController::Layouts`

- 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
- ActionController::Streaming: Overloads the `_process_options()` method to handle the `:stream` by setting the proper HTTP headers and also the `_render_template()` method to allow templates to be streamed

The final stack with Abstract Controller and Action Controller rendering modules is shown in [Figure 3, Visualization of the rendering stack when we call render with AbstractController and ActionController, on page 14.](#)

Now that we understand how the `render()` works, we are ready to understand how `render_to_string()` works. Let's start by seeing its definition in `AbstractController::Rendering`:

```

rails/actionpack/lib/abstract_controller/rendering.rb
def render_to_string(*args, &block)
  options = _normalize_render(*args, &block)
  render_to_body(options)
end

```

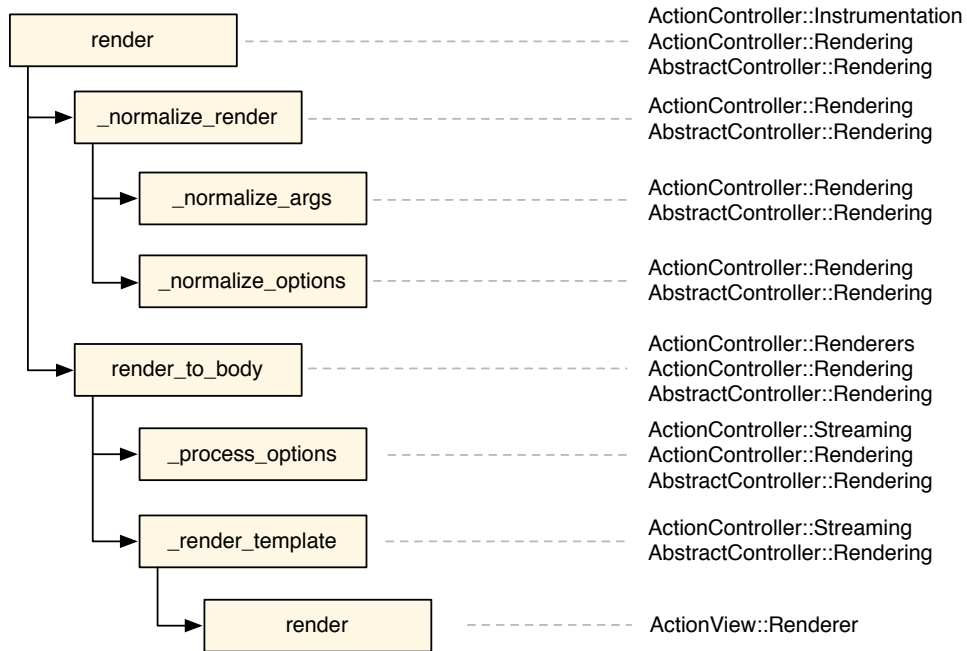



Figure 3—Visualization of the rendering stack when we call render() with AbstractController and ActionController

At first, the `render_to_string()` method looks in fact quite similar to `render()`. The only difference is that `render_to_string()` does not store the rendered template as the actual response body. However, 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 twice and that it won't raise a double render error, since those functionalities are added only to the `render()` method.

In some other cases, `render_to_string()` may be overloaded as well. When using Action Controller, the response body can be another object that is not a string, which is what happens on template streaming. For this reason, `ActionController::Rendering` overrides `render_to_string()` to always return a string, as the name says.

1.4 Taking It to the Next Level

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

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

In our renderer, it becomes:

```
pdf = Prawn::Document.new
pdf.text render_to_string({})
send_data(pdf.render, filename: "contents.pdf",
  disposition: "attachment")
```

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({})` simply passes `template: "#{controller_name}/#{action_name}"` to the view renderer object.

The fact that our renderer relies on `render_to_string()` allows us to also use the following options:

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

And internally, it's the same as the following:

```
pdf = Prawn::Document.new
pdf.text render_to_string(template: "path/to/template")
send_data(pdf.render, filename: "contents.pdf",
  disposition: "attachment")
```

This time `render_to_string()` receives an explicit template to render. To finish our PDF renderer, let's add a test to confirm the chosen template will indeed be rendered! Our test invokes a new action in `HomeController` that calls `render()` with both `:pdf` and `:template` options:

```
pdf_renderer/2_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:

```
pdf_renderer/2_final/test/dummy/config/routes.rb
get "/another", to: "home#another", as: :another
```

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

```
pdf_renderer/2_final/test/integration/pdf_delivery_test.rb
test "pdf renderer uses the specified template" do
  get another_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
```

Now run the tests and watch them pass once again!

1.5 Wrapping Up

In this chapter we created a renderer for the PDF format. Using these ideas, you can easily create renderers for formats such as PDF, CSV, and 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, such as the paid Prince XML⁵ library or the open source Flying Saucer,⁶ which is written in Java but easily accessible through JRuby⁷.

We also discussed the Rails rendering stack and its modularity. Since Rails itself relies on this well-defined stack to extend Action Controller and Action Mailer, this API is by consequence more robust because it was battle-tested by Rails' own features and different use cases. As we will see in the chapters that follow, this is a common practice throughout Rails' codebase.

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.

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

6. <http://xhtmlrenderer.java.net/>

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

In this chapter, we'll see

- Active Model and its modules
- How to make an object comply with the Active Model API required by Rails
- Rails' validators and Ruby constant lookup

CHAPTER 2

Building Models with Active Model

In the previous chapter, we talked briefly about Abstract Controller and how it reduced code duplication in the Rails source code since it's decoupled from both Action Mailer and Action Controller. Now let's look at Active Model.

Similar to Abstract Controller, Active Model was originally created to hold the behavior shared between Active Record and Active Resource¹. As in Abstract Controller, the desired functionalities can be cherry-picked by including only the modules you need. Besides, Active Model is also responsible for defining the API required by Rails controllers and views, so any other ORM can use Active Model to ensure Rails behaves exactly as it would with Active Record.

Let's explore both facets of Active Model in this chapter by writing a plugin called *Mail Form* that will be used in our controllers and views. Mail Form's goal is to receive a hash of parameters sent by a POST request, validate them, and email them to a specified email address. This abstraction will allow us to create fully functional contact forms in just a couple of minutes!

2.1 Creating Our Model

Mail Form objects belong to the models part in the MVC architecture as they receive the information sent through a form and deliver it to a recipient specified by the business model. Let's structure Mail Form in the same way Active Record works: we'll provide a class named `MailForm::Base` that contains the most common features we expect in a model, such as the ability to specify attributes and seamless integration with Rails forms. As we did in the previous chapter, let's use rails plugin to create our new plugin:

```
$ rails plugin new mail_form
```

1. Since then, Active Resource has been extracted from the Rails code base and is available at <https://github.com/rails/activerecord>.

Our first feature is to implement a class method called `attributes()` that allows a developer to specify which attributes the Mail Form object contains. Let's create a model inside `test/fixtures/sample_mail.rb` as a fixture to use in our tests:

```
mail_form/1_attributes/test/fixtures/sample_mail.rb
```

```
class SampleMail < MailForm::Base
  attributes :name, :email
end
```

And then add a test to ensure the defined attributes `name` and `email` are available as accessors in the Mail Form object:

```
mail_form/1_attributes/test/mail_form_test.rb
```

```
require "test_helper"
require "fixtures/sample_mail"

class MailFormTest < ActiveSupport::TestCase
  test "sample mail has name and email as attributes" do
    sample = SampleMail.new
    sample.name = "User"
    assert_equal "User", sample.name
    sample.email = "user@example.com"
    assert_equal "user@example.com", sample.email
  end
end
```

When we run the test suite with `rake test`, it fails because `MailForm::Base` is not defined yet. Let's define it in `lib/mail_form/base.rb` and implement the `attributes()` method:

```
mail_form/1_attributes/lib/mail_form/base.rb
```

```
module MailForm
  class Base
    def self.attributes(*names)
      attr_accessor(*names)
    end
  end
end
```

Our implementation delegates the creation of attributes to `attr_accessor()`. Before we run our tests again, we need to ensure that `MailForm::Base` is loaded. One option would be to explicitly require `"mail_form/base"` in `lib/mail_form.rb`. However, let's use Ruby's `autoload()` instead:

```
mail_form/1_attributes/lib/mail_form.rb
```

```
module MailForm
  autoload :Base, "mail_form/base"
end
```

`autoload()` allows us to lazily load a constant when it is first referenced. So we annotate that `MailForm` has a constant called `Base` defined in `mail_form/base.rb`. When `MailForm::Base` is referenced for the first time, Ruby loads the `mail_form/base.rb` file. This is frequently used in Ruby gems and in Rails itself for a fast booting process as it does not need to load everything up front.

With `autoload()` in place, our first test passes. We have a simple model with attributes, but so far, we haven't used any of Active Model's goodness. Let's do that now.

Adding Attribute Methods

`ActiveModel::AttributeMethods` is a module that tracks all defined attributes, allowing us to add a common behavior to all of them dynamically. To show how it works, let's define two convenience methods called `clear_name()` and `clear_email()` which will clear out the value of the associated attribute when invoked. Let's write a test first:

```
mail_form/2_attributes_prefix/test/mail_form_test.rb
test "sample mail can clear attributes using clear_prefix" do
  sample = SampleMail.new

  sample.name = "User"
  sample.email = "user@example.com"
  assert_equal "User", sample.name
  assert_equal "user@example.com", sample.email

  sample.clear_name
  sample.clear_email
  assert_nil sample.name
  assert_nil sample.email
end
```

Invoking `clear_name()` and `clear_email()` set their respective attribute value back to `nil`. With `ActiveModel::AttributeMethods`, we can define both `clear_name()` and `clear_email()` dynamically in four simple steps, as outlined in our new `MailForm::Base` implementation shown next:

```
mail_form/2_attributes_prefix/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::AttributeMethods # 1) attribute methods behavior
    attribute_method_prefix 'clear_'      # 2) clear_ is attribute prefix

    def self.attributes(*names)
      attr_accessor(*names)

      # 3) Ask to define the prefix methods for the given attributes names
    end
  end
end
```

```

    define_attribute_methods(names)
  end

  protected

  # 4) Since we declared a "clear_" prefix, it expects to have a
  # "clear_attribute" method defined, which receives an attribute
  # name and implements the clearing logic.
  def clear_attribute(attribute)
    send("#{attribute}=", nil)
  end
end
end
end

```

Run rake test, and all tests should be green again. `ActiveModel::AttributeMethods` uses `method_missing()` to compile both the `clear_name()` and `clear_email()` methods when they are first accessed. Their implementation invokes `clear_attribute()`, passing the attribute name as a parameter.

If we want to define suffixes, instead of a prefix like `clear_`, we just need to use the `attribute_method_suffix()` method and implement the method with the chosen suffix logic. As an example, let's implement `name?()` and `email?()` methods, which should return true if the respective attribute value is present, as in the following test:

```

mail_form/3_attributes_suffix/test/mail_form_test.rb
test "sample mail can ask if an attribute is present or not" do
  sample = SampleMail.new
  assert !sample.name?

  sample.name = "User"
  assert sample.name?

  sample.email = ""
  assert !sample.email?
end

```

When we run the test suite, our new test fails. To make it pass, let's define `?` as a suffix, changing our `MailForm::Base` implementation to the following:

```

mail_form/3_attributes_suffix/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::AttributeMethods
    attribute_method_prefix 'clear_'

    # 1) Add the attribute suffix
    attribute_method_suffix '?'

    def self.attributes(*names)

```

```

    attr_accessor(*names)
    define_attribute_methods(names)
  end

  protected

  def clear_attribute(attribute)
    send("#{attribute}=", nil)
  end

  # 2) Implement the logic required by the '?' suffix
  def attribute?(attribute)
    send(attribute).present?
  end
end
end
end

```

Now we have both prefix and suffix methods defined and the tests passing. But what if we want to define both the prefix and the suffix at the same time? We could use the `attribute_method_affix()` method, which accepts a hash specifying both the prefix and the suffix.

Active Record uses attributes methods extensively. An example is the `attribute_before_type_cast()` method, which uses `_before_type_cast` as a suffix to return raw data, as received from forms. The dirty functionality, which is also part of Active Model, is built on top of `ActiveModel::AttributeMethods` and defines a handful of methods like `attribute_changed?()`, `reset_attribute!()`, and so on. You can check its source code in the Rails repository.²

Aiming for an Active Model–Compliant API

Even though we added attributes to our models to store form data, we need to ensure that our model complies with the Active Model API; otherwise, we won't be able to use it in our controllers and views.

As usual, we are going to achieve this compliance through test-driven development, except this time we won't need to write the tests because Rails already provides all of them in a module called `ActiveModel::Lint::Tests`. When included, this module defines several tests asserting that each method required in an Active Model–compliant API exists. Each of these tests, in order to run, expects an instance variable named `@model` to return the object we want to assert against. In our case, `@model` should contain an instance of `SampleMail`, which will be compliant if `MailForm::Base` is compliant. Let's create a new test file called `test/compliance_test.rb` with the following:

2. https://github.com/rails/rails/tree/4-0-stable/activemodel/lib/active_model/dirty.rb


```
mail_form/4_am_compliance/test/compliance_test.rb
require 'test_helper'
require 'fixtures/sample_mail'

class ComplianceTest < ActiveSupport::TestCase
  include ActiveSupport::Lint::Tests

  def setup
    @model = SampleMail.new
  end
end
```

When we run rake test, we get several failures, all with the reason:

The object should respond to to_model.

When Rails controllers and views helpers receive a model, they first call to_model() and manipulate the returned result instead of the model directly. This allows ORM implementations that do not want to add Active Model methods to their API to return a proxy object where these methods are in fact defined. In our case, we want to add Active Model methods directly to MailForm::Base. Consequently, our to_model() implementation should return self, as shown here:

```
def to_model
  self
end
```

Although we could add this method to MailForm::Base, we are not going to implement it ourselves. Instead, let's include ActiveSupport::Conversion which implements to_model() exactly as we discussed and three other methods also required by Active Model: to_key(), to_param() and to_partial_path().

The to_key() method should return an array of keys that uniquely identifies the model, if any exists, and it is used by dom_id() in views. The dom_id() method was added to Rails along with dom_class() and a bunch of other helpers to better organize views based on some conventions. For example, div_for(@post), where @post is an Active Record instance of the Post class with an id of 42, relies on both these methods to create a div where the id attribute is equal to post_42 and the class attribute is post. For Active Record, to_key() returns an array containing the record ID from the database.

On the other hand, to_param() is used in routing and can be overwritten in any model to generate a unique URL for that model. When we invoke post_path(@post), Rails calls to_param() in the @post object and uses its result to generate the final URL. For Active Record, the default is to return the ID as a string.

Finally, we have `to_partial_path()`. This method is invoked everytime we pass a record or a collection of records to `render()` in our views. Rails will go through each of these records and retrieve the path to their partial. For example, the path to an instance of the `Post` class is `posts/post`.

It is important not only to understand what those methods do but also what they allow us to achieve. For example, by customizing `to_param()`, we can easily change the URL of our objects. Imagine a `Post` class with `id` and `title` attributes, changing the URL of those posts to contain also the title is as easy as:

```
def to_param
  "#{id}-#{title.parameterize}"
end
```

Similarly, imagine that each `Post` have a different format. It can be a video, it can be a link, or it can be a bunch of a text, and each of those formats should be rendered differently. If we store the format of the blog post in the `format` attribute, one could write in the views:

```
@posts.each do |post|
  render partial: "posts/post_#{post.format}",
    locals: { post: @post }
end
```

However, by overriding `to_partial_path()`:

```
def to_partial_path
  "posts/post_#{format}"
end
```

Our view would simply call:

```
render @posts
```

This does not only make our code cleaner, but it also improves our application performance. In the first example, we end up going through Rails' rendering stack many times, looking up templates and duplicating efforts. However, by customizing `to_partial_path()`, we call `render()` just once, allowing Rails to efficiently look up all partials in one take.

The default `to_partial_path()` implementation available in `ActiveModel::Conversion` allows us to provide partials for `MailForm::Base` objects as in any Active Record object. However, since our objects are never persisted, they aren't uniquely identified, meaning that both `to_key()` and `to_param()` should return `The code tag should not be here..` This is exactly the behavior provided by `ActiveModel::Conversion`. Let's include it in our `MailForm::Base` class:

```
mail_form/4_am_compliance/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::Conversion
```

When we include this module and run rake test, we get different errors with the following messages (you may get them in different order):

The model should respond to model_name

The model should respond to errors

The model should respond to persisted?

To fix the first failing test, we need to extend the MailForm::Base class with ActiveModel::Naming:

```
mail_form/4_am_compliance/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::Conversion
    extend ActiveModel::Naming
```

After extending our class with ActiveModel::Naming, it now responds to a method called model_name() that returns an instance of ActiveModel::Name, which acts like a string and provides a few methods such as human(), singular(), and others that are inflected from the model name. Let's add a small test case to our suite showing these methods and what they return:

```
mail_form/4_am_compliance/test/compliance_test.rb
test "model_name exposes singular and human name" do
  assert_equal "sample_mail", @model.class.model_name.singular
  assert_equal "Sample mail", @model.class.model_name.human
end
```

This is similar to the behavior Active Record exhibits. The only difference is that Active Record supports internationalization (I18n) and Mail Form does not. Luckily, that can be easily fixed by extending MailForm::Base with ActiveModel::Translation. Let's write a test first:

```
mail_form/4_am_compliance/test/compliance_test.rb
test "model_name.human uses I18n" do
  begin
    I18n.backend.store_translations :en,
      activemodel: { models: { sample_mail: "My Sample Mail" } }

    assert_equal "My Sample Mail", @model.class.model_name.human
  ensure
    I18n.reload!
  end
end
```

The test adds a new translation to the I18n backend that contains the desired human name for the SampleMail class. We need to wrap the code in the begin...ensure clause since we need to guarantee the I18n backend is reloaded, removing the translation we stored. Let's update MailForm::Base to make the new test pass:

```
mail_form/4_am_compliance/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::Conversion
    extend ActiveModel::Naming
    extend ActiveModel::Translation
```

After adding naming and translation behaviors, rake test returns fewer failures, showing that we are moving forward. This time our tests fail due to the following reasons:

The model should respond to errors

The model should respond to persisted?

The first failure is related to validations. Active Model does not say anything about validation macros (such as validates_presence_of()), but it requires us to define a method named errors(), which returns a Hash, and each value in this hash is an Array. We can fix this failure by including ActiveModel::Validations in our model:

```
mail_form/4_am_compliance/lib/mail_form/base.rb
module MailForm
  class Base
    include ActiveModel::Conversion
    extend ActiveModel::Naming
    extend ActiveModel::Translation
    include ActiveModel::Validations
```

Now our model instance responds to errors() and also valid?() which behaves exactly as in Active Record. Furthermore, ActiveModel::Validations also adds several validation macros such as validates(), validates_format_of(), validates_inclusion_of(), and others.

For now, let's run rake test and see what is left to make our test suite green again:

The model should respond to persisted?

This time Rails won't help us. Luckily, it is easy enough to implement persisted?() ourselves. The persisted?() method is used by both our controllers and our views under different circumstances. For instance, whenever we invoke

`form_for(@model)`, it checks whether the model is persisted. If positive, it creates a form that does a PUT request; if not, it should do a POST request. The same happens in `url_for()` when it generates a URL based on your model.

In Active Record, the object is persisted if it's saved in the database, in other words, if it's neither a new record nor destroyed. However, in our case, our object won't be saved in any database, and consequently `persisted?()` should always return false.

Let's add the `persisted?()` method to our `MailForm::Base` implementation:

```
mail_form/4_am_compliance/lib/mail_form/base.rb
def persisted?
  false
end
```

This time, after running `rake test`, all tests pass! This means our model complies with the Active Model API. Well done!

Delivering the Form

The next step in our Mail Form implementation is to add the logic that delivers an email with the model attributes. The delivery is done by the `deliver()` method, which sends an e-mail to the address stored in the email attribute of our model. The e-mail body contains all model attributes and their respective values. Let's specify this behaviour by adding a new test to `test/mail_form_test.rb`:

```
mail_form/5_delivery/test/mail_form_test.rb
setup do
  ActionMailer::Base.deliveries.clear
end

test "delivers an email with attributes" do
  sample = SampleMail.new
  # Simulate data from the form
  sample.email = "user@example.com"
  sample.deliver

  assert_equal 1, ActionMailer::Base.deliveries.size
  mail = ActionMailer::Base.deliveries.last

  assert_equal ["user@example.com"], mail.from
  assert_match "Email: user@example.com", mail.body.encoded
end
```

When we run the new test, we get a failure because the `deliver()` method does not exist yet. Because our model has the concept of validity from `ActiveModel::Val-`

idations, the `deliver()` method should just deliver the email if the Mail Form object is `valid?()`:

```
mail_form/5_delivery/lib/mail_form/base.rb
def deliver
  if valid?
    MailForm::Notifier.contact(self).deliver
  else
    false
  end
end
```

The class responsible for creating and delivering the email is `MailForm::Notifier`. Let's implement it using Action Mailer:

```
mail_form/5_delivery/lib/mail_form/notifier.rb
module MailForm
  class Notifier < ActionMailer::Base
    append_view_path File.expand_path("../../views", __FILE__)

    def contact(mail_form)
      @mail_form = mail_form
      mail(mail_form.headers)
    end
  end
end
```

The `contact()` action in our mailer creates an assign called `@mail_form` and then invokes the `headers()` method in the given Mail Form object. This method should return a hash with email data as keys like `:to`, `:from`, and `:subject` and should not be defined in `MailForm::Base` but in each child class. This is a simple but powerful API contract that allows a developer to customize the email delivery without a need to redefine or monkey-patch the `Notifier` class.

Our `MailForm::Notifier` also calls `append_view_path()`, which adds `lib/views` inside our plugin folder as a new location to search for templates. The last step before we run the test suite again is to autoload our new class:

```
mail_form/5_delivery/lib/mail_form.rb
autoload :Notifier, "mail_form/notifier"
```

Then, let's define the `headers()` method in the `SampleMail` class:

```
mail_form/5_delivery/test/fixtures/sample_mail.rb
def headers
  { to: "recipient@example.com", from: self.email }
end
```

Now when we run rake test, it fails with the following message:

1) Failure:

```
test_delivers_an_email_with_attributes(MailFormTest):
ActionView::MissingTemplate: Missing template mail_form/notifier/contact
```

This is expected since we haven't added a template to our mailer. Our default mail template will show the message subject and print all attributes and their respective values:

`mail_form/5_delivery/lib/views/mail_form/notifier/contact.text.erb`

```
<%= message.subject %>
<% @mail_form.attribute_names.each do |key| -%>
  <%= @mail_form.class.human_attribute_name(key) %>: <%= @mail_form.send(key) %>
<% end -%>
```

In order to show all attributes, we need to keep a list of all attribute names and we don't keep such list. We can easily implement by defining a `class_attribute()` called `attributes_names()` that is updated every time we call `attributes()`:

`mail_form/5_delivery/lib/mail_form/base.rb`

```
# 1) Define a class attribute and initialize it
class_attribute :attribute_names
self.attribute_names = []
```

```
def self.attributes(*names)
  attr_accessor(*names)
  define_attribute_methods(names)
```

```
# 2) Add new names as they are defined
self.attribute_names += names
end
```

The advantage of using `class_attribute()` for defining the names is that it will automatically work with inheritance. So if a class eventually inherits from our `SampleMail` fixture, it will automatically inherit all of its attribute names too.

After running rake test, all tests should be green again, and our Mail Form implementation is finished. Whenever we need to create a contact form, we just create a class that inherits from `MailForm::Base`, define our attributes and the email headers, and we are ready to go! To ensure it works exactly as we expect, let's test the whole process with an integration test.

2.2 Integration tests with Capybara

In the previous chapter, we have used Rails testing facilities to ensure a PDF was delivered back. Now, to really guarantee our project works as a contact form, we should create an actual form, submit it to the appropriate endpoint, and verify the e-mail was sent. Those kind of tests are particularly hard to

write using only Rails' testing tools. Most of the times, we just end up writing direct requests to endpoints:

```
post "/contact_form", contact_form:
  { email: "jose@example.com", message: "hello"}
```

Writing a test using `post()` and explicit parameters is fine for some scenarios, particularly for APIs, but it falls flat when testing a contact form workflow. For example, how to guarantee there is an actual submit button on the page? What happens when we click it? Is the request send to the proper URL? What if we forget the e-mail field?

In order to guarantee all those questions are answered, it is common to use a more robust testing tool like Capybara³. Capybara makes this sort of testing trivial by providing an easy to use DSL which we will use throughout the book. The first step is to add it as a development dependency to our gemspec:

```
mail_form/5_delivery/mail_form.gemspec
s.add_development_dependency "capybara", "~> 2.0.0"
```

In order to use Capybara, let's define a brand new test case class called `ActiveSupport::IntegrationCase`. This class is built on top of `ActiveSupport::TestCase` and includes Rails' URL helpers and the Capybara DSL:

```
mail_form/5_delivery/test/test_helper.rb
require "capybara"
require "capybara/rails"

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

Now we are ready to write our first test using it:

```
mail_form/5_delivery/test/integration/navigation_test.rb
require "test_helper"

class NavigationTest < ActiveSupport::IntegrationCase
  setup do
    ActionMailer::Base.deliveries.clear
  end

  test "sends an e-mail after filling the contact form" do
    visit "/"
```

3. <https://github.com/jnicklas/capybara>


```

fill_in "Name",    with: "John Doe"
fill_in "Email",   with: "john.doe@example.com"
fill_in "Message", with: "MailForm rocks!"

click_button "Deliver"
assert_match "Your message was successfully sent.", page.body

assert_equal 1, ActionMailer::Base.deliveries.size
mail = ActionMailer::Base.deliveries.last

assert_equal ["john.doe@example.com"], mail.from
assert_equal ["recipient@example.com"], mail.to
assert_match "Message: MailForm rocks!", mail.body.encoded
end
end

```

The integration test navigates to the root path, which returns a form with name, email, and message fields. On submitting the form, the server delivers an email to the configured recipient with the given message and shows a success message to the user. To make the test pass, let's add the model, controller, views, and routes to our dummy app, starting with the latter:

```
mail_form/5_delivery/test/dummy/config/routes.rb
```

```

Dummy::Application.routes.draw do
  resources :contact_forms, only: :create
  root to: "contact_forms#new"
end

```

The controller and view follow next:

```
mail_form/5_delivery/test/dummy/app/controllers/contact_forms_controller.rb
```

```

class ContactFormsController < ApplicationController
  def new
    @contact_form = ContactForm.new
  end

  def create
    @contact_form = ContactForm.new(params[:contact_form])

    if @contact_form.deliver
      redirect_to root_url, notice: "Your message was successfully sent."
    else
      render action: "new"
    end
  end
end

```

```
mail_form/5_delivery/test/dummy/app/views/contact_forms/new.html.erb
```

```

<h1>New Contact Form</h1>

<%= form_for(@contact_form) do |f| %>

```

```

<% if @contact_form.errors.any? %>
<div id="errorExplanation">
  <h2>Oops, something went wrong:</h2>
  <ul>
    <% @contact_form.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
  </ul>
</div>
<% end %>
<div class="field">
  <%= f.label :name %><br />
  <%= f.text_field :name %>
</div>
<div class="field">
  <%= f.label :email %><br />
  <%= f.text_field :email %>
</div>
<div class="field">
  <%= f.label :message %><br />
  <%= f.text_field :message %>
</div>
<div class="actions">
  <%= f.submit "Deliver" %>
</div>
<% end %>

```

And finally here's the model:

```

mail_form/5_delivery/test/dummy/app/models/contact_form.rb
class ContactForm < MailForm::Base
  attributes :name, :email, :message

  def headers
    { to: "recipient@example.com", from: self.email }
  end
end

```

Because our tests use flash messages, we need to add them to the layout, just before the yield call:

```

mail_form/5_delivery/test/dummy/app/views/layouts/application.html.erb
<p style="color: green"><%= notice %></p>

```

With everything in place, let's run the test suite and... get an unexpected failure:

```

1) Error:
test_sends_an_e-mail_after_filling_the_contact_form(NavigationTest):
ArgumentError: wrong number of arguments (1 for 0)
    app/controllers/contact_forms_controller.rb:7:in `initialize'

```

The failure occurs because the `initialize()` method in `MailForm::Base`, unlike `ActiveRecord`, does not expect a hash as argument. Notice that an Active Model compliant API does not say anything about how our models should be initialized. Let's implement an `initialize()` method, which receives a hash as an argument and sets attribute values:

```
mail_form/5_delivery/lib/mail_form/base.rb
def initialize(attributes = {})
  attributes.each do |attr, value|
    self.public_send("#{attr}=", value)
  end if attributes
end
```

After defining the previous method, our integration test works, showing that everything works as expected. Remember that if you go to the vendored application inside `test/dummy`, you can run `rails s` as in any other Rails application. Feel free to start your server, add some validations to your `ContactForm` class, and have some fun with it.

2.3 Taking It to the Next Level

In the previous section, we wrote our Mail Form plugin with some basic features and added some integration testing to ensure it works. However, we can do a lot more with Active Model. Let's take a look at some examples.

Validators

Every Rails developer is familiar with Rails validations as they are often showcased as an example of the productivity that can be achieved with Rails. In the Rails source code, each validation is backed up by a respective *validator* class. Let's see the `validates_presence_of()` macro as an example:

```
rails/activemodel/lib/active_model/validations/presence.rb
def validates_presence_of(*attr_names)
  validates_with PresenceValidator, _merge_attributes(attr_names)
end
```

The `validates_with()` method is responsible for initializing the given `ActiveModel::Validations::PresenceValidator` class, while `_merge_attributes()` converts the given attributes to a hash. When you invoke the following:

```
validates_presence_of :name
```

you're actually doing this:

```
validates_with PresenceValidator, attributes: [:name]
```

which is roughly the same as the following:

```
validate PresenceValidator.new(attributes: [:name])
```

This process is similar to what happens with the `validates()` method:

```
validates :name, presence: true
```

This, again, at the end is just the same as the following:

```
validate PresenceValidator.new(attributes: [:name])
```

The question is, how does Rails know that the `:presence` key should use the `PresenceValidator`? Simple: it first converts the `:presence` key to `"PresenceValidator"` and then tries to find a constant named `PresenceValidator` in the current class, just like the following:

```
const_get("#{key.to_s.camelize}Validator")
```

This is important to discuss because now we can add any validator to any class, relying solely on Ruby's constant lookup. To understand exactly how it works, let's start a new `irb` session and type the following:

```
module Foo
  module Bar
    end
end

class Baz
  include Foo
end

Baz::Bar # => Foo::Bar
```

Notice how the last line of the script in the previous code returns `Foo::Bar` even if `Bar` is not defined inside the `Baz` class. This happens because whenever a constant is looked up, Ruby searches inside all objects in the ancestor chain. Since `Foo` is included in `Baz`, `Foo` is an ancestor of `Baz`, allowing Ruby to find the `Foo::Bar` constant (you can check `Baz` ancestors by typing `Baz.ancestors` in the previous `irb` session).

To showcase how we can use this in practice, we are going to implement an absence validator in our `MailForm::Base`. Since a lot of spam usually comes through contact forms, we are going to use the absence validator as a honey pot.

The honey pot works by creating a field, such as `nickname`, and hiding it with CSS. This way, humans do not see the field and consequently do not fill it in, while robots will fill it in like any other field. So whenever the `nickname` value is present, the email should not be sent because it is definitely spam.

Given Ruby's constant lookup rules, we can add an `:absence` option to the `validates()` method of any class by implementing the `AbsenceValidator` inside a module and including this module in the desired class. Let's start by writing a simple test for it:

```
mail_form/6_final/test/mail_form_test.rb
test "validates absence of nickname" do
  sample = SampleMail.new(nickname: "Spam")
  assert !sample.valid?
  assert_equal ["is invalid"], sample.errors[:nickname]
end
```

The test shows the record must be invalid if the nickname field contains any value. Let's add the nickname field with `:absence` validation to our `SampleMail` object:

```
mail_form/6_final/test/fixtures/sample_mail.rb
attributes :nickname
validates :nickname, absence: true
```

When we run `rake test`, we get a test failure since `SampleMail` can no longer be loaded as `AbsenceValidator` is not defined anywhere. Let's create it:

```
mail_form/6_final/lib/mail_form/validators.rb
module MailForm
  module Validators
    class AbsenceValidator < ActiveRecord::EachValidator
      def validate_each(record, attribute, value)
        record.errors.add(attribute, :invalid, options) unless value.blank?
      end
    end
  end
end
```

Our validator inherits from `EachValidator`. For each attribute given on initialization, `EachValidator` calls the `validate_each()` method with the record, the attribute, and its respective value. For each attribute, we add an error message, unless the value is blank.

Next, let's include `MailForm::Validators` in `MailForm::Base`:

```
mail_form/6_final/lib/mail_form/base.rb
include MailForm::Validators
```

This will add `MailForm::Validators` to the `MailForm::Base` ancestors chain. So, whenever we give `:absence` as a key to `validates()`, it will search for an `AbsenceValidator` constant, find it inside `MailForm::Validators`, and initialize it, similar to what it did with the `PresenceValidator`. To ensure it really works, we just need to autoload our validators container:

```
mail_form/6_final/lib/mail_form.rb
autoload :Validators, 'mail_form/validators'
```

Run rake test, and all tests pass again. The beauty of this implementation is that adding the `:absence` key to `validates()` did not require us to register the option anywhere. Those options are discovered at runtime using Ruby's constant lookup.

Feel free to add the nickname field to our contact form views and hide it with CSS, fully enabling our honey pot. It's up to you to write an integration test for it, since we still have some more Active Model investigation to do.

Callbacks

Wouldn't it be cool if we could provide hooks around the `deliver()` method so we could add some behavior *before* and *after* the delivery? This is quite easy to achieve with `ActiveModel::Callbacks`. To illustrate the API we intend to achieve, let's change our `SampleMail` fixture:

```
mail_form/6_final/test/fixtures/sample_mail.rb
before_deliver do
  evaluated_callbacks << :before
end

after_deliver do
  evaluated_callbacks << :after
end

def evaluated_callbacks
  @evaluated_callbacks ||= []
end
```

We have added a `evaluated_callbacks()` method to keep all evaluated callbacks and we implemented both `before_deliver()` and `after_deliver()` callbacks. Our test should call the `deliver()` in our `SampleMail` instance and assert both callbacks were evaluated:

```
mail_form/6_final/test/mail_form_test.rb
test "provides before and after deliver hooks" do
  sample = SampleMail.new(email: "user@example.com")
  sample.deliver
  assert_equal [:before, :after], sample.evaluated_callbacks
end
```

Finally, let's add support to callbacks in `MailForm::Base`. This can be done in three steps: extend our class with `ActiveModel::Callbacks` functionality, then define our callbacks, and finally overwrite `deliver()` implementation to run the callbacks before and after delivering:

```

mail_form/6_final/lib/mail_form/base.rb
# 1) Add callbacks behavior
extend ActiveSupport::Callbacks

# 2) Define the callbacks. The line below will create both before_deliver
# and after_deliver callbacks with the same semantics as in Active Record
define_model_callbacks :deliver

# 3) Change deliver to run the callbacks
def deliver
  if valid?
    run_callbacks(:deliver) do
      MailForm::Notifier.contact(self).deliver
    end
  else
    false
  end
end
end

```

As Active Record callbacks, you can give procs, strings, symbols, and any object that responds to the callback name. Feel free to try these options!

A Base Model

There are many circumstances where a developer would like to achieve an Active Model compliant API: in contact forms, when implementing search functionality or even when splitting a sign-up flow into many steps. For this reason, Active Model provides an `ActiveModel::Model` module that can be included in any class:

```

class Person
  include ActiveSupport::Model
  attr_accessor :name, :age
end

person = Person.new(name: 'bob', age: '18')
person.name # => 'bob'
person.age # => 18

```

By including `ActiveModel::Model` we are guaranteed that it will pass all `ActiveModel::Lint::Tests` tests. Let's take a look at its source:

```

rails/activemodel/lib/active_model/model.rb
module ActiveSupport
  module Model
    def self.included(base)
      base.class_eval do
        extend ActiveSupport::Naming
        extend ActiveSupport::Translation
        include ActiveSupport::Validations

```

```

        include ActiveSupport::Conversion
      end
    end

    def initialize(params={})
      params.each do |attr, value|
        self.public_send("#{attr}=", value)
      end if params
    end

    def persisted?
      false
    end
  end
end

```

As we can see, this basic module contains a subset of the behaviour we've implemented in this chapter. It's a good starting point whenever these functionalities are needed in our applications!

2.4 Wrapping Up

In this chapter, we learned how to use Active Model to quickly create our own models that play seamlessly with Rails controllers and views. We talked about `ActiveModel::AttributeMethods`, `ActiveModel::Conversion`, `ActiveModel::Naming`, `ActiveModel::Translation`, `ActiveModel::Validations`, and finally `ActiveModel::Callbacks`. We also dove into Rails validators and how we can easily extend the `validates()` method behavior.

Even after all that, Active Model has a couple other modules to explore, like `ActiveModel::Dirty` and `ActiveModel::Serialization`. They let us bring dirty attributes and serializers such as `to_xml()` and `to_json()`, right into our models, just like Active Record models.

Finally, if you enjoyed Mail Form here, check out the Mail Form by Plataformatec,⁴ which is a production-ready gem, created with the same concepts explored in this chapter. It also has additional features, such as attachment handling and the ability to append request information.

In the next chapter, we'll go back to studying the Rails rendering stack and extend it to look for a template in the database instead of the filesystem, keeping an eye on performance.

4. https://github.com/plataformatec/mail_form

In this chapter, we'll see

- How to customize Rails rendering stack to look up templates from the database
- How Ruby Hash lookup works
- How to speed up controllers with ActionController::Metal

CHAPTER 3

Retrieving View Templates from Custom Stores

When Rails renders a template, it has to get that template from somewhere. By default, Rails serves templates from the file system, but it doesn't need to be limited like that. Rails provides hooks that allow us to retrieve templates from anywhere we want, as long as we implement the required API. Let's explore this by building a mechanism that lets us serve templates from a database, templates which can be created, updated and deleted through the web interface. But first, let's take a deeper look at Rails' rendering stack.

3.1 Revisiting the Rendering Stack

In [Section 1.3, *Understanding The Rails Rendering Stack*, on page 9](#), we analyzed Rails' controllers rendering stack and learned that its main responsibility is to normalize options and send them to an instance of `ActionView::Renderer`. When invoked, the renderer receives an instance of `ActionView::Base` called view context and a hash of normalized options used to find, compile, and render a specific template.

Whenever we render a template in Rails, its source must be first compiled into executable Ruby code. Every time some Ruby code is executed, its execution happens inside a given context and, in a Rails application, views are executed inside the view context object. All helpers available in our templates, such as `form_for()` and `link_to()`, are actually defined in modules included in the view context object.

Besides the view context, the view renderer also has access to an instance of `ActionView::LookupContext` usually referred as `lookup_context`. The *lookup context* is shared between controllers and views and it holds all the information required

to find a template. For example, whenever a JSON request comes, the request format is stored in the `lookup_context` object, so Rails will only look for templates tied to the JSON format.

The lookup context is also responsible for holding all view paths. The *view paths* is a collection of objects able to find templates given a set of conditions. All controllers in a Rails application have one view path by default, which is a filesystem path pointing to `app/views`. Given a set of conditions like template name, locale, and format, this view path finds a specific template under `app/views`. For instance, whenever we have an HTML request at the index action of a `UserController`, this default view path will attempt to pick a template at `app/views/users/index.html.*`. If the desired template is found, it's then compiled and rendered, as shown in the following image:

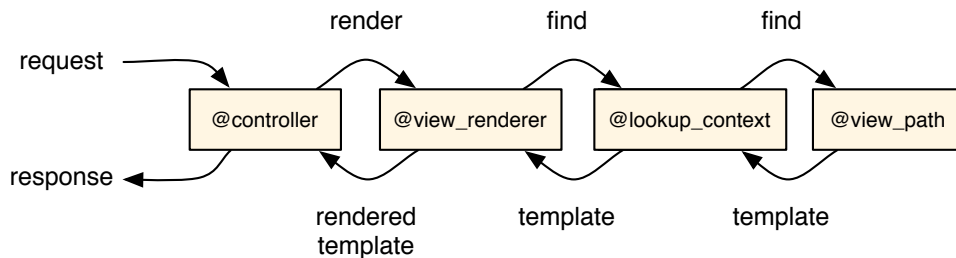


Figure 4—Rendering workflow between controller, view renderer, lookup context and view path

In [Delivering the Form, on page 26](#), we manipulated the view path in our `MailForm::Notifier` object to include another path in the template lookup:

```

module MailForm
  class Notifier < ActionMailer::Base
    append_view_path File.expand_path("../..views", __FILE__)
  end
end

```

The code above states that if a template cannot be found under `app/views`, the mailer should look within the `lib/views` directory next.

Although we mostly set new view paths as strings, representing file system paths, Rails actually provides a well defined API for adding any object as a view path. This means we are not forced to store view templates in the filesystem. We can store templates anywhere we want as long as we provide an object that knows how to find them. Although externally those objects are historically called view paths, internally Rails calls them *template resolvers*, and they must comply with the Resolver API.

Rails provides an abstract resolver implementation called `ActionView::Resolver`. In this chapter we're going to use it to create a resolver that uses the database as a template store, so we can store our pages in the database and edit them through a web interface using our favorite template handler (such as Liquid, ERB, or Haml). The nice thing is that we can implement this functionality with one scaffold and a few lines of code!

3.2 Setting Up a SqlResolver

This time, instead of using rails plugin to implement the desired functionality, we will develop the template management system by building a Rails application called `templer`. Let's create it using the command line:

```
$ rails new templer
```

Next, let's define the model that will hold templates in the database using the Rails scaffold generator:

```
$ rails generate scaffold SqlTemplate body:text path:string \
  format:string locale:string handler:string partial:boolean
```

The `body` attribute is a text column used to store the whole template, the `path` should store a string similar to a filesystem path (for instance the `index()` action under `UserController` will have `users/index` as the path), `format` and `locale` hold the template format and its locale, the `handler` stores the template handler (for example, Liquid, ERB, or Haml), and finally `partial` tells us whether the stored template is a partial.

Before executing the generated migration, let's just make one change in it, setting `false` as the default value for the `partial` attribute:

```
t.boolean :partial, default: false
```

And now we are ready to run our migrations:

```
$ bundle exec rake db:migrate
```

So far, no surprises. Next, let's create a *template resolver*, which will use the `SqlTemplate` model to read templates from the database and expose them according to the Resolver API described next.

The Resolver API

The Resolver API is composed of a single method, called `find_all()`, which should return an array of templates and has the following signature:

```
def find_all(name, prefix, partial, details, cache_key, locals)
```

For an HTML request at the `index()` action of a `UserController`, those arguments are exactly as shown here:

```
find_all("index", "users", false, { formats: [:html],
  locale: [:en, :en], handlers: [:erb, :builder, :rjs] }, nil, [])
```

For this simple request, we can see that `name` maps to the action name, while `prefix` refers to the controller name. Next, `partial` is a boolean that tells whether the template being rendered is a partial, and `details` is a hash with extra information for the lookup, such as the request formats, the current I18n locale followed by the default locale, and the available template handlers. The last two arguments are the `cache_key`, which we will consider to be The code tag should not be here. for now, and the `locals` variable, which is an empty array as `locals` are only used when rendering a partial.

Rails provides an abstract resolver implementation, called `ActionView::Resolver`, which we are going to use as the base for our resolver. Part of its source code is shown next, focusing on the `find_all()` and `find_templates()` methods:

`rails/actionpack/lib/action_view/template/resolver.rb`

```
module ActionView
  class Resolver
    attr_accessor :caching
    self.caching = true

    def initialize
      @cache = Cache.new
    end

    def clear_cache
      @cache.clear
    end

    def find_all(name, prefix=nil, partial=false, details={}, key=nil, locals=[])
      cached(key, [name, prefix, partial], details, locals) do
        find_templates(name, prefix, partial, details)
      end
    end

    private

    def find_templates(name, prefix, partial, details)
      raise NotImplementedError
    end
  end
end
```

The `find_all()` method implements a basic caching mechanism where the block given to `cached()` is yielded only if no previous entry exists in the cache. When

the block is invoked, it calls `find_templates()` which raises a `NotImplementedError`, indicating it should actually be implemented in child classes. Notice that the `cache_key` and `locals` are only used by the cache mechanism, they are not passed down to template lookup.

Let's inherit from `ActionView::Resolver` and implement the `find_templates()` method using the `SqlTemplate` model to retrieve templates from the database, resulting in the same template lookup, as shown next:

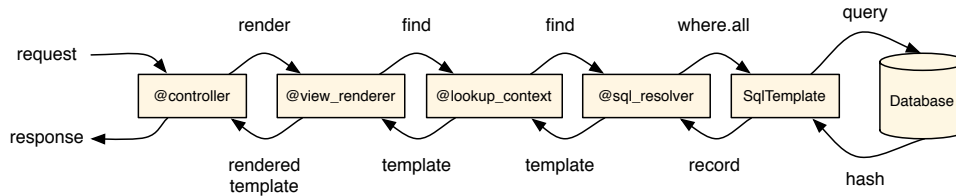


Figure 5—Template lookup with SqlTemplate

Writing the Code

Let's call our resolver implementation `SqlTemplate::Resolver` and implement it in three main steps. The first receives the name, prefix, partial, and details as arguments and normalizes them. From the normalized arguments, we create a SQL statement and query the database. This query returns an array of records from the database that is then used to create an array of `ActionView::Template` instances.

Let's write a test first to demonstrate the functionality we want.

```
templatel/1_resolver/test/models/sql_template_test.rb
```

```
require 'test_helper'
```

```
class SqlTemplateTest < ActiveSupport::TestCase
```

```
  test "resolver returns a template with the saved body" do
```

```
    resolver = SqlTemplate::Resolver.new
```

```
    details = { formats: [:html], locale: [:en], handlers: [:erb] }
```

```
    # 1) Assert our resolver cannot find any template as the database is empty
```

```
    # find_all(name, prefix, partial, details)
```

```
    assert resolver.find_all("index", "posts", false, details).empty?
```

```
    # 2) Create a template in the database
```

```
    SqlTemplate.create!(
```

```
      body: "<%= 'Hi from SqlTemplate!' %>",
```

```
      path: "posts/index",
```

```
      format: "html",
```

```
      locale: "en",
```

```

    handler: "erb",
    partial: false)

# 3) Assert that a template can now be found
template = resolver.find_all("index", "posts", false, details).first
assert_kind_of ActionView::Template, template

# 4) Assert specific information about the found template
assert_equal "<%= 'Hi from SqlTemplate!' %>", template.source
assert_kind_of ActionView::Template::Handlers::ERB, template.handler
assert_equal [:html], template.formats
assert_equal "posts/index", template.virtual_path
assert_match %r[SqlTemplate - \d+ - "posts/index"], template.identifier
end
end

```

The `find_all()` method in our resolver should return an `ActionView::Template` instance. This template instance is initialized as follows:

```
ActionView::Template.new(source, identifier, handler, details)
```

The source is the body of the template stored in the database. The identifier is a unique string used to represent the template. We will ensure its uniqueness by adding the template ID in the database.

The handler is the object responsible for compiling the template. The handler is not a string—like we stored in the database—but an object that is retrieved using the method `registered_template_handler()` from `ActionView::Template`:

```

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

```

Finally, the last parameter given on template initialization is a hash with three keys: the `:format` of the template found, the last time the template was updated as `:updated_at`, and a `:virtual_path` that represents the template.

Since templates are no longer required to be in the filesystem, they do not necessarily have a path, and this breaks a couple of Rails features that depend explicitly on it. One example is the `I18n` shortcut `t("message")` inside your views. It uses the template filesystem path to retrieve the translation, so whenever you are inside a template at `app/views/users/index`, the shortcut attempts to find the `I18n` translation at `"users.index.message"`.

To circumvent this need for a path, Rails requires templates to provide a `:virtual_path`. You can store your templates anywhere and give them any source or any identifier, but you need to provide a `:virtual_path` that represents what the path would be if this template was stored in the filesystem. This allows `t("message")` to work as expected by setting the virtual path to be `users/index`.

With tests in place and an understanding of how templates are initialized, let's implement our resolver by inheriting from `ActionView::Resolver` and implementing `find_templates()`.

It's important to consider in our resolver that the order of the given details matters. In other words, if the locale array contains `[:es, :en]`, a template in Spanish (`:es`) has higher preference than the one in English, in case both exist. To solve this in our resolver, one option is to generate an order clause for each detail and get the result properly sorted from the database. Another option is to sort the returned templates in Ruby. However, for simplicity, instead of passing all locales and formats to the SQL query, let's simply pick the first ones from the array.

Without further ado, let's implement our resolver:

```
templatel/1_resolver/app/models/sql_template.rb
```

```
class SqlTemplate < ActiveRecord::Base
  validates :body, :path, presence: true
  validates :format, inclusion: Mime::SET.symbols.map(&:to_s)
  validates :locale, inclusion: I18n.available_locales.map(&:to_s)
  validates :handler, inclusion:
    ActionController::Template::Handlers.extensions.map(&:to_s)

  class Resolver < ActionController::Resolver
    protected

    def find_templates(name, prefix, partial, details)
      conditions = {
        path: normalize_path(name, prefix),
        locale: normalize_array(details[:locale]).first,
        format: normalize_array(details[:formats]).first,
        handler: normalize_array(details[:handlers]),
        partial: partial || false
      }
      ::SqlTemplate.where(conditions).map do |record|
        initialize_template(record)
      end
    end

    # Normalize name and prefix, so the tuple ["index", "users"] becomes
    # "users/index" and the tuple ["template", nil] becomes "template".
    def normalize_path(name, prefix)
      prefix.present? ? "#{prefix}/#{name}" : name
    end

    # Normalize arrays by converting all symbols to strings.
    def normalize_array(array)
      array.map(&:to_s)
    end
  end
end
```

```

# Initialize an ActionView::Template object based on the record found.
def initialize_template(record)
  source = record.body
  identifier = "SqlTemplate - #{record.id} - #{record.path.inspect}"
  handler = ActionView::Template.registered_template_handler(record.handler)

  details = {
    format: Mime[record.format],
    updated_at: record.updated_at,
    virtual_path: virtual_path(record.path, record.partial)
  }

  ActionView::Template.new(source, identifier, handler, details)
end

# Make paths as "users/user" become "users/_user" for partials.
def virtual_path(path, partial)
  return path unless partial
  if index = path.rindex("/")
    path.insert(index + 1, "_")
  else
    "_#{path}"
  end
end
end
end
end

```

Our implementation normalizes the given arguments, queries the database, and creates template objects from the result set. We also added several validations to our model, ensuring the body and path values cannot be blank, and guaranteeing a valid format, locale, and handler are supplied.

As a result of adding some validations to our models, some functional tests are failing since our fixtures now contain invalid data. To make them pass, let's change the fixture at `test/fixtures/sql_templates.yml` to include a valid format, locale, and handler:

```

templatery/1_resolver/test/fixtures/sql_templates.yml

```

```

one:
  id: 1
  path: "some/path"
  format: "html"
  locale: "en"
  handler: "erb"
  partial: false
  body: "Body"

```


Now with our resolver implemented and a green test suite, we get to create a new scaffold and make it use templates from the database, instead of the filesystem. Let's create a user scaffold by running the following command:

```
$ rails generate scaffold User name:string
```

And run our migrations next:

```
$ bundle exec rake db:migrate
```

We can now start the server, access `/users`, and perform all CRUD operations as usual.

Next, let's access the `/sql_templates` path and create a new template by filling the template body with the same contents as the file in `app/views/users/index.html.erb`; setting the path with `users/index`; setting the format, locale, and handler to `html`, `en`, and `erb`, respectively; and keeping the partial box unchecked.

Save this new template, and head back to the `/users` path. Now, delete the view file `app/views/users/index.html.erb`, and rerender the page. You should get a "Template is missing" error, but don't worry, because we expect that. The template is stored in the database, but we still haven't told the `UserController` to use our new resolver to retrieve it.

Let's do it by adding the following line to `UserController`:

```
templatizer/1_resolver/app/controllers/users_controller.rb
class UsersController < ApplicationController
  append_view_path SqlTemplate::Resolver.new
```

When we refresh the page at `/users`, we see the whole index page once again, retrieved from the database! And, while the template is in the database, the layout still comes from the filesystem. In other words, in a single request, we can get templates from different resolvers in our view paths.

Feel free to head back to `/sql_templates`, manipulate the body of the stored template, and notice that the `index()` action in the `UserController` will change accordingly. The fact we can achieve this behavior in so few lines of code shows the power of the `ActionView::Resolver` abstraction.

Before we move to the next section, let's run the test suite once again. It turns out that a test is failing with an error message:

```
1) Error:
test_should_get_index(UsersControllerTest)
ActionView::MissingTemplate: Missing template users/index,
  application/index with {:locale=>[:en], :formats=>[:html],
    :handlers=>[:erb, :builder, :raw, :ruby, :jbuilder, :coffee]}. Searched in:
```

```
* "templatier/app/views"
* "#<SqlTemplate::Resolver:0x007f9774fbc0d0>"
```

This happens because we deleted the template from the filesystem. Although we added the same template to our development database, our test database remains clean, raising this `MissingTemplate` error in the test environment. To fix this, let's change our `sql_templates` fixture.

```
templatier/1_resolver/test/fixtures/sql_templates.yml
users_index:
  id: 2
  path: "users/index"
  format: "html"
  locale: "en"
  handler: "erb"
  partial: false
  body: "<h1>Listing users</h1>
<table>
  <tr>
    <th>Name</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <%% @users.each do |user| %>
    <tr>
      <td><%= user.name %></td>
      <td><%= link_to 'Show', user %></td>
      <td><%= link_to 'Edit', edit_user_path(user) %></td>
      <td><%= link_to 'Destroy', user,
        data: { confirm: 'Are you sure?' }, method: :delete %></td>
    </tr>
  <%% end %>
</table>
<br />
<%= link_to 'New user', new_user_path %>"
```

Our fixture is just a copy of the template. The only caveat is that Rails parses fixtures with ERB, so we need to escape our ERB tags using `<%% ... %>`. And that's all—tests are all green again.

3.3 Configuring Our Resolver for Production

To ensure the template lookup is fast in production, Rails provides some caching conveniences. Let's explore those conveniences so we understand how to enable caching for our templates and the strategies available to us to expire this cache whenever a template is saved to the database.

As mentioned earlier, Rails gives our resolver a `cache_key` through the `find_all()` method. Our first stop is to learn why Rails creates this cache key and how our resolver uses it.

The Resolvers Cache

As we learned in [code, on page 58](#), `ActionView::Resolver`'s `find_all()` method automatically caches templates using the `cached()` method. The cache is created on initialization and referenced by the instance variable `@cached`. The resolver caches templates only if `Rails.application.config.cache_classes` returns true and it also exposes a `clear_cache()` method to clear it.

Each template must be cached in function of five values: the `cache_key`, its prefix, its name, if it is a partial or not and the set of locals. Given those five keys, we could store the templates in the cache in the different ways:

```
# Nested hash
@cached[key][prefix][name][partial][locals]

# Simple hash with array as key
@cached[[key, prefix, name, partial, locals]]

# Simple hash with hash as key
@cached[key: key, prefix: prefix, name: name, partial: partial, locals: locals]
```

All three cache implementations shown give us the desired behavior. However, there is a difference between them: performance. We need to explore how Ruby does hash lookups to understand this.

Ruby Hash Lookup

Whenever we store a value for a given key in a Hash object, Ruby stores three things: the given key, the given value, and the object hash for the given key. This hash is the result of the `Object#hash()` method called on the object given as the key. There is an easy way to prove that Ruby Hash in fact relies on `Object#hash()`; just start an irb session, and type the following:

```
class NoHash
  undef_method :hash
end

hash = Hash.new
hash[NoHash.new] = 1
# => NoMethodError: undefined method `hash' for #<NoHash:0x101643820>
```

If we undefine the `hash()` method in our object, it can no longer be stored in the hash. Adding an element to the hash is similar to creating a new entry in a table, as seen next:

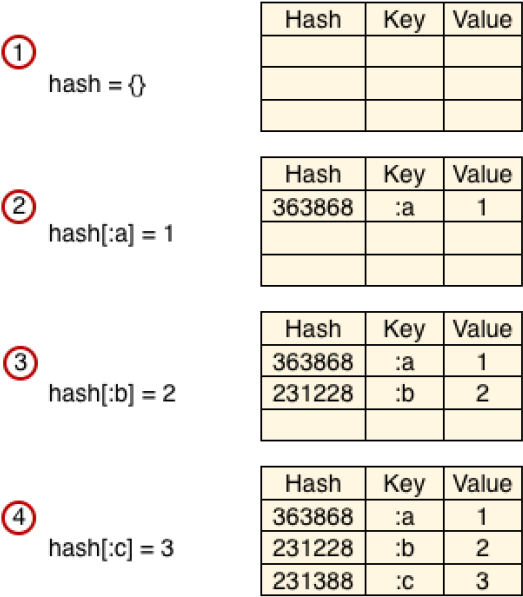


Figure 6—Illustration of what a hash stores for each entry. Keep in mind the Ruby implementation does not use a table structure but uses pointers instead, although the table is an easy way to represent how it works.

Whenever we attempt to retrieve the value for a given key in a Hash object, like `hash[:b]`, Ruby first calculates the `Object#hash()` for this given key and then searches whether one or more entries in the Hash object have this same hash value. For instance, the value returned by `:b.hash` is 231228 in [Figure 6, Illustration of what a hash stores for each entry, on page 50](#). Seeing that there are one or more entries with the value 231228, Ruby then compares whether any of the keys for these entries is equal to the given key, using the equality operator `eq!()`. Since `:b.eq!?(b)` returns true, accessing `hash[:b]` in our example successfully returns 2 as the result.

To prove that Ruby in fact uses `Object#hash()` to localize entries, let's start another `irb` session and do a few experiments.

```
hash = {}
object = Object.new

hash[object] = 1
hash[object] # => 1

def object.hash; 123; end
```

```
hash[object] # => nil
hash        # => {#<Object:0x1016e3de8>=>1}
```

This time, we used an arbitrary Ruby object as a hash key, and we could successfully set and get values. However, after we modified the value returned by `object#hash`, the value could not be found, even though the same object is still in the hash.

The reason Ruby stores `Object#hash()` for each key is to provide faster lookups. Comparing hash values (integers) is much faster than comparing each object stored in the hash.

This implementation implies that the performance hit in finding a value in the hash is not just in the `eq!()` method but also in calculating `Object#hash()` for the given key. Remember, we could choose to implement our resolver cache using a nested hash or a simple hash with arrays as the key or a simple hash with hashes as keys. We should choose the first, because in the nested hash case, the hash keys are strings or booleans, and Ruby knows how to calculate the `Object#hash()` value for these very fast. On the other hand, calculating `Object#hash()` and equality for arrays and hashes is more expensive.

We can demonstrate this in another irb session:

```
require "benchmark"
foo    = "foo"
bar    = "bar"
array  = [foo, bar]
hash   = {a: foo, b: bar}

nested_hash = Hash.new { |h,k| h[k] = {} }
nested_hash[foo][bar] = true

array_hash = { array => true }
hash_hash  = { hash => true }

Benchmark.realtime { 1000.times { nested_hash[foo][bar] } } # => 0.000342
Benchmark.realtime { 1000.times { array_hash[array] } }     # => 0.000779
Benchmark.realtime { 1000.times { hash_hash[hash] } }       # => 0.001645
```

The nested hash implementation yields better results. Although the choice for a nested hash *apparently* does not yield substantial gains, the concepts we learned about Hash lookups in Ruby are fundamental to understand the next section.

The Cache Key

We already know that our resolvers come with a built-in cache. We also know that this cache uses a nested hash to store the templates found and that the cache depends on five values: `@cached[key][prefix][name][partial][locals]`. However, if you recall, the `find_all()` signature receives 6 arguments:

```
def find_all(name, prefix=nil, partial=false, details={}, key=nil, locals=[])
```

The `details` is a hash containing the format, locale and other information useful for the template lookup. This information is held by the *lookup context* and it is essential for retrieving the correct template from the filesystem. So why doesn't the cache use these details?

Remember how we measured that calculating the `Object#hash()` for a hash object is a bit expensive when compared to simpler structures like strings? If we use `details` as the key in the cache's nested hash, it would be slow since `details` is a hash of arrays:

```
details # => {
  formats: [:html],
  locale: [:en, :en],
  handlers: [:erb, :builder, :rjs]
}

# Slow because details is a hash of arrays
@cached[details][prefix][name][partial][locals]
```

Instead, the lookup context generates a simple Ruby object for each `details` hash and sends it as `cache_key` to resolvers. The whole process is similar to the following code:

```
# Generate an object for the details hash
@details_key ||= {}
key = @details_key[details] ||= Object.new

# And send it to each resolver
resolver.find_all(name, prefix, partial, details, key)

# Inside the resolver, the details it not used in the cache
# Instead we use the key which is a simple object and fast
@cached[key][prefix][name][partial][locals]
```

In other words, the `details` is not used directly in the cache but via the `cache_key`. This is important because during a request the `details` hash rarely changes, as the format and locale are usually set before rendering any template. Therefore, regardless of how many templates are rendered and resolvers are involved during a request, the `cache_key` is likely to be calculated just once. In

case a detail happens to change, like the request format, a new `cache_key` would be generated.

Let's fire up `irb` once again and do our last benchmark in this chapter. Our benchmark will show how accessing a hash using a simple Object, like the `cache_key`, compares with using a hash of arrays, like the details hash:

```
require "benchmark"
cache_key = Object.new
details = {
  formats: [:html, :xml, :json],
  locale:  [:en],
  handlers: [:erb, :builder, :rjs]
}

hash_1 = { cache_key => 10 }
hash_2 = { details => 10 }

Benchmark.realtime { 1000.times { hash_1[cache_key] } } # => 0.000202
Benchmark.realtime { 1000.times { hash_2[details] } }   # => 0.003937
```

Twenty times slower is quite a difference! For applications that require high performance, these milliseconds can easily mount up in requests that render several collections and partials, dramatically affecting the response time. In some benchmarks done with Rails, using the details hash *took up to 10 percent of the time spent in the rendering stack*, while using the `cache_key` reduces this to less than 1 percent.

Expiring the Cache

Since the cache inside resolvers is handled automatically by Rails, we only need to worry about expiring the cache using the `Resolver#clear_cache()` method. This cache is stored in the resolver instance, so in order to expire these caches, we would need to track all instances of `SqlTemplate::Resolver` and call `clear_cache()` in each of them whenever we add or update a template in the database.

However, does it really make sense to create several `SqlTemplate::Resolver` instances? If we consider that the cache is in the instance, creating several instances would actually create several caches, reducing their effectiveness. Therefore, we don't want several resolvers instances. We want only one shared across the entire application.

In other words, what we need is a *singleton* class. Luckily, Ruby has a `Singleton` module in its Standard Library, which does all the hard work for us. By including this module in `SqlTemplate::Resolver`, it makes `SqlTemplate::Resolver.new()`

a private method and exposes `SqlTemplate::Resolver.instance()` instead, which always returns the same object.

Also, by having a singleton object, it's very easy to expire the cache. Since we can always access the instantiated resolver with `SqlTemplate::Resolver.instance()`, we just need to call `clear_cache()` on it every time we save a `SqlTemplate` instance.

Let's get started with those changes. The first one is to require and include `Singleton` inside `SqlTemplate::Resolver`:

```
templatier/2_improving/app/models/sql_template.rb
require "singleton"
include Singleton
```

After doing this simple change, we need to update both `app/controllers/users_controller.rb` and `test/models/sql_template_test.rb` to call `SqlTemplate::Resolver.instance()` instead of `SqlTemplate::Resolver.new()`:

```
templatier/2_improving/app/controllers/users_controller.rb
append_view_path SqlTemplate::Resolver.instance
```

```
templatier/2_improving/test/models/sql_template_test.rb
resolver = SqlTemplate::Resolver.instance
```

With our singleton resolver in place, let's write a test in `test/models/sql_template_test.rb`, which asserts that our cache is properly expired. This new test should update the `SqlTemplate` from fixtures and assert our resolver will return the updated template:

```
templatier/2_improving/test/models/sql_template_test.rb
test "sql_template expires the cache on update" do
  cache_key = Object.new
  resolver = SqlTemplate::Resolver.instance
  details = { formats: [:html], locale: [:en], handlers: [:erb] }

  t = resolver.find_all("index", "users", false, details, cache_key).first
  assert_match "Listing users", t.source

  sql_template = sql_templates(:users_index)
  sql_template.update_attributes(body: "New body for template")

  t = resolver.find_all("index", "users", false, details, cache_key).first
  assert_equal "New body for template", t.source
end
```

Notice we generated a fake `cache_key` with `Object.new` to pass to `find_all()` because the cache is activated only if a cache key is supplied.

Finally, to make our test pass, let's add an `after_save` callback to `SqlTemplate`, right after the model validations:


```

templatizer/2_improving/app/models/sql_template.rb
after_save do
  SqlTemplate::Resolver.instance.clear_cache
end

```

Now every time a template is created or updated, the cache is expired allowing the modified template to be picked up and recompiled. Unfortunately, this approach has a severe limitation: it only works for single-instance deployments. For example, if your infra-structure contains more than one server or if you use Passenger or Unicorn with a pool of instances, a request will reach a specific instance, which will have only its own cache cleared up. In other words, this cache is not synchronized in between machines.

Luckily, there are a couple ways we could solve this issue:

- One option is to reimplement the cache using memcached¹ or redis² so it is shared in between machines with an appropriate expiration mechanism
- Another option is to notify each instance whenever the cache is expired, for example through a queue. In this schema, `after_save()` would simply push a message to the queue which would send a notification to all subscribed instances;
- We can also solve this by setting `config.action_view.cache_template_loading` to `false` on production. Previously, we mentioned that the resolver cache is only enabled if `config.cache_classes` is `true`. However, whenever `cache_template_loading()` is set, it has higher preference than the `cache_classes()` one. By disabling the template cache, Rails is going to do a query to the database every time a template lookup happens, but the template is only recompiled if the `updated_at` value we set on template creation is more recent than the one it has cached. So, even though we trigger a new query, the expensive template compilation only happens if needed. This is how Rails behaves by default in development;

Each strategy has different characteristics and may suit better depending on your infra-structure, performance requirements and how frequently templates are actually changed. With these remarks, we have finished our `SqlTemplate::Resolver` implementation!

1. <http://memcached.org/>
 2. <http://redis.io/>

3.4 Serving Templates with Metal

Now that we can create and edit templates from the UI and serve them with our own resolver, we are ready to take it to the next level. In the following sections, let's use this template management system as a simple CMS.

Creating The CMSController

We already can create, update, and delete templates by accessing `/sql_templates`; now we just need to expose them depending on the accessed URL. To achieve this, let's map all requests under `/cms/*` to a controller that will use our resolver to find the template in the database and render them back to the client. A request at `/cms/about` should render a `SqlTemplate` stored in the database with `path` equals to `about`.

We are able to implement this functionality with a few lines of code. Let's start with an integration test that we will write using Capybara. The first step is to setup Capybara as we did in [Section 2.2, Integration tests with Capybara, on page 28](#):

```
templat/2_improving/test/test_helper.rb
require "capybara"
require "capybara/rails"

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

And add it is a dependency to the Gemfile:

```
templat/2_improving/Gemfile
group :test do
  gem 'capybara', '~> 2.0.0'
end
```

And finally write the test that creates a template and then access it:

```
templat/2_improving/test/integration/cms_test.rb
require 'test_helper'

class CmsTest < ActiveSupport::IntegrationCase
  test "can access any page in SqlTemplate" do
    visit "/sql_templates"
    click_link "New Sql template"

    fill_in "Body",    with: "My first CMS template"
    fill_in "Path",    with: "about"
```

```

fill_in "Format", with: "html"
fill_in "Locale", with: "en"
fill_in "Handler", with: "erb"

click_button "Create Sql template"
assert_match "Sql template was successfully created.", page.body

visit "/cms/about"
assert_match "My first CMS template", page.body
end
end

```

To make our new test pass, let's write a route that will map to our CmsController:

```

templatizer/2_improving/config/routes.rb
get "cms/*page", to: "cms#respond"

```

This route maps all requests at /cms/* to the respond() action in the CmsController, which we implement as follows:

```

templatizer/2_improving/app/controllers/cms_controller.rb
class CmsController < ApplicationController
  prepend_view_path SqlTemplate::Resolver.instance

  def respond
    render template: params[:page]
  end
end

```

We simply pass the given route as a template name, which will be looked up in our SqlTemplate::Resolver. Notice we prepend our resolver to the view paths this time, as it is the main source for templates. Finally, our test suite is green once again!

If you want to test it manually, fire up the server, go to /sql_templates, create a new template with path equals to about, and add some content. Then just hit /cms/about and see your new page exhibited!

Playing with Metal

Our CmsController inherits from ApplicationController, which inherits from ActionController::Base, and consequently it comes with all the functionality available in regular Rails controllers. It includes all helpers, adds CSRF protection, allows you to hide actions with hide_action(), supports flash messages, adds the respond_to() method, doing a lot more than required since our controller handles only GET requests. Wouldn't it be nice if we could somehow have a simpler controller with just the behavior we need?

We have already discussed `AbstractController::Base` and how it provides a basic structure that is shared between `Action Mailer` and `Action Controller`. However, `AbstractController::Base` does not know anything about HTTP. On the other hand, `ActionController::Base` comes with the whole package. Isn't there a point in the middle?

Actually, there is! It's called `ActionController::Metal`. `ActionController::Metal` inherits from `AbstractController::Base` and implements the minimum functionality required for our controllers to be a valid Rack application and work with HTTP.

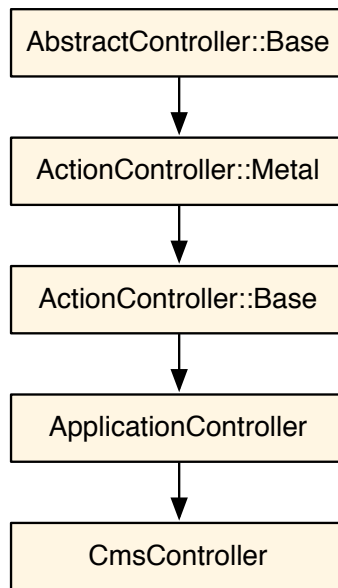


Figure 7—CmsController superclasses

By taking a quick look at `ActionController::Base` in the Rails source code, we notice it inherits from `Metal` and adds a bunch of behavior:

```

rails/actionpack/lib/action_controller/base.rb
module ActionController
  class Base < Metal
    abstract!

    include ActionController::Layouts
    include ActionController::Translation
    include ActionController::AssetPaths
    include Helpers
    include HideActions
    include UrlFor
    include Redirecting
  end
end

```

```

include Rendering
include Renderers::All
include ConditionalGet
include RackDelegation
include Caching
include MimeResponds
include ImplicitRender
include StrongParameters
include Cookies
include Flash
include RequestForgeryProtection
include ForceSSL
include Streaming
include DataStreaming
include RecordIdentifier
include HttpAuthentication::Basic::ControllerMethods
include HttpAuthentication::Digest::ControllerMethods
include HttpAuthentication::Token::ControllerMethods
include ActionController::Callbacks
include Rescue
include Instrumentation
include ParamsWrapper

ActiveSupport.run_load_hooks(:action_controller, self)
end
end

```

Let's reimplement `CmsController`, but this time we'll inherit from `ActionController::Metal` and including only the modules we need, which reduces the overhead in a request:

```

templat3/3_final/app/controllers/cms_controller.rb
class CmsController < ActionController::Metal
  include ActionController::Rendering
  include ActionController::Helpers

  prepend_view_path ::SqlTemplate::Resolver.instance
  helper CmsHelper

  def respond
    render template: params[:page]
  end
end

```

```

templat3/3_final/app/helpers/cms_helper.rb
module CmsHelper
end

```

After these changes, our tests should still be green, showing that our new controller implementation using `ActionController::Metal` works as expected.

If we need more functionality, we just add the required modules. For instance, in case you want to add layouts, just include the `AbstractController::Layouts` module, create a layout in the database with the path `layouts/cms`, and specify layout "cms" in the controller. Try it!

3.5 Wrapping Up

We learned a lot in this chapter. We analyzed Action View's rendering stack, developed a resolver that reads templates from a database with cache expiration mechanisms. Then we created a controller to dynamically access the pages in the resolver and optimized it by making it an `ActionController::Metal` object. If you are eager to see more examples about resolvers, you can check the Rails source code and discover how it implements the filesystem resolver, which retrieves templates from the filesystem.

On the other hand, if you are already familiar with Rails' internals (such as resolvers and metal) and are still looking for a challenge, you can learn more about Ruby hashes by checking Rubinius' source code. Rubinius implements most of the Ruby language in Ruby itself, including the Hash class, so you can learn a lot by looking through its source.

In the next chapter, let's discuss template handlers, such as ERB, Builder, and Haml. We will create our own handler using Markdown and ERB, and we'll hook it up into Rails' generators.

In this chapter, we'll see

- Rails template handler API
- Multipart templates with Action Mailer
- Rails generators and railties

CHAPTER 4

Sending Multipart Emails Using 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 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 43](#), 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 62](#) 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

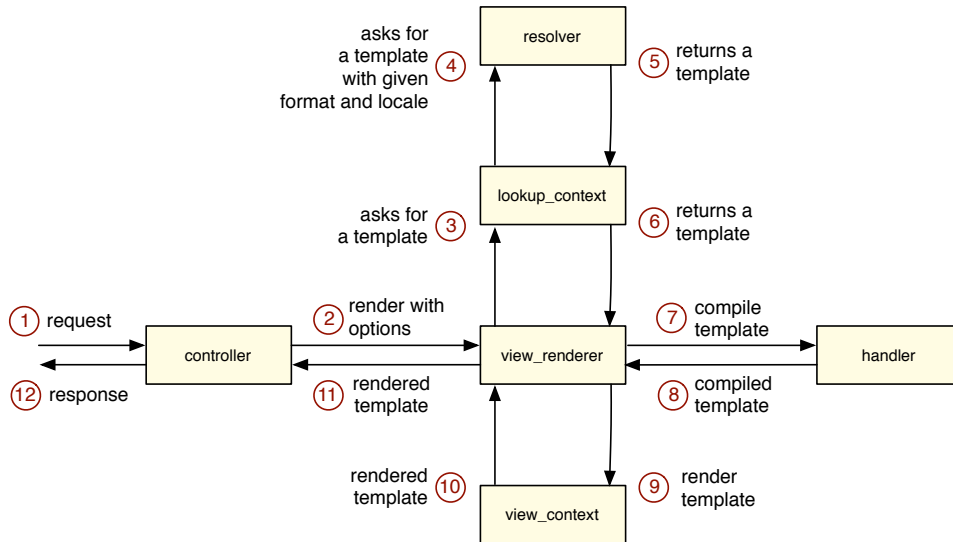


Figure 8—Objects involved in the rendering stack

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. Markdown¹ 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
=====
```

```
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>

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:

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>

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

Regards,

The Team.

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 43](#), 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:

```
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.

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

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.erb
```

```
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 43](#). Our `.merb` template handler is shown next:

```
handlers/1_first_handlers/lib/handlers.rb
```

```
module Handlers
  module MERB
    def self.erb_handler
      @@erb_handler ||= ActiveSupport::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>rocks</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 43](#), 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 17](#), 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 73](#). 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
```

```

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:

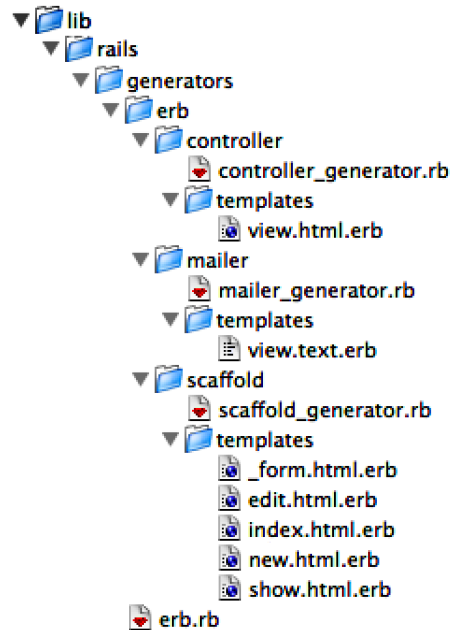


Figure 10—Structure for ERB generators

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.

4.4 Extending Rails with Railties

A `Rails::Railtie` (pronounced “Rails Rail-tie”) allows you to hook into Rails’ initialization and configure some defaults. Such tools allow frameworks like Active Record to tell Rails how Active Record is initialized and configured by providing its own railtie.

You should include a railtie in your plugin only if at least one of the following is true:

- Your plugin needs to perform a given task while or after the Rails application is initialized.
- Your plugin needs to change a configuration value, for instance, setting a generator.
- Your plugin must provide Rake tasks and generators in nondefault locations (the default location for the former is `lib/tasks` and `lib/generators` or `lib/rails/generators` for the latter).
- Your plugin wants to run custom code whenever the Rails console or the Rails runner are started.
- You want your plugin to provide configuration options to the application, such as `config.my_plugin.key = :value`.

Let’s take a look at an excerpt of `ActiveRecord::Railtie` in the Rails source code that contains a few examples of these scenarios:

```
rails/activerecord/lib/active_record/railtie.rb
module ActiveRecord
  class Railtie < Rails::Railtie
    config.active_record = ActiveSupport::OrderedOptions.new
```

```

config.app_generators.orm :active_record, migration: true,
                        timestamps: true

config.app_middleware.insert_after "::ActionDispatch::Callbacks",
  "ActiveRecord::QueryCache"

config.eager_load_namespaces << ActiveRecord

rake_tasks do
  require "active_record/base"
  load "active_record/railties/databases.rake"
end

runner do
  require "active_record/base"
end

initializer "active_record.initialize_timezone" do
  ActiveSupport.on_load(:active_record) do
    self.time_zone_aware_attributes = true
    self.default_timezone = :utc
  end
end

initializer "active_record.migration_error" do
  if config.active_record.delete(:migration_error) == :page_load
    config.app_middleware.insert_after "::ActionDispatch::Callbacks",
      "ActiveRecord::Migration::CheckPending"
  end
end
end
end
end

```

After such examples, we are ready to create our first railtie and configure the mailer generator to use our new template handler by default:

```

handlers/2_final/lib/handlers/railtie.rb
module Handlers
  class Railtie < Rails::Railtie
    config.app_generators.mailer template_engine: :merb
  end
end

```

Since our railtie must be loaded when our plugin is loaded, we need to add a require in lib/handlers.rb:

```

handlers/2_final/lib/handlers.rb
require "handlers/railtie"

```

And that's all! Let's go to the dummy application at `test/dummy` and invoke the generator helper once again with `rails g mailer --help`. Notice the default template engine has changed to *merb*. Therefore, we don't need to pass it as an option when invoking it!

All major Rails generators, such as `model`, `controller`, and `scaffold`, rely on hooks. As we've just seen, this allows us to adapt them to our workflow and preferred tools.

It is important to notice this mechanism only works if our railtie is loaded before our application is initialized. This is the reason why, in a freshly generated Rails application, we require our dependencies before we define the Rails application. You can verify this by opening `config/application.rb` in any Rails application and observing that the `Bundler.require` line comes before we inherit from `Rails::Application`.

Furthermore, even though plugins are allowed to change Rails defaults, the application still has the final word about it. For instance, we changed Rails to use our `:merb` template engine in the mailer generator by default. However, if developers want to set this value back to `:erb`, they can simply do it inside the application definition at `config/application.rb`.

4.5 Wrapping Up

In this chapter, we finished our discussion about Rails' rendering stack by building a few template handlers. Our main template handler with the `.merb` extension mixes Markdown with ERB, allowing it to render both HTML and TEXT parts in an email by using just one template.

At the end of the chapter, we created a generator and customized Rails to use our new template handler by default. There is much more to discover in the Generators API. Besides the methods seen in this chapter, `Thor::Actions` defines `copy_file()`, `inject_into_file()`, `create_file()`, `run()`, and a few more. In addition, Rails has a module called `Rails::Generators::Actions` that provides methods specific to Rails, such as `gem()`, `environment()`, `route()`, and many others. Rails also provides a testing facility to generators called `Rails::Generators::TestCase`, which is useful when testing our generators. Here is an example of how Rails uses `Rails::Generators::TestCase` to test its own mailer generator:

```
rails/railties/test/generators/mailer_generator_test.rb
require "generators/generators_test_helper"
require "rails/generators/mailer/mailer_generator"

class MailerGeneratorTest < Rails::Generators::TestCase
  arguments %w(notifier foo bar)
```

```

def test_mailer_skeleton_is_created
  run_generator
  assert_file "app/mailers/notifier.rb" do |mailer|
    assert_match(/class Notifier < ActionMailer::Base/, mailer)
    assert_match(/default from: "from@example.com"/, mailer)
  end
end

def test_mailer_with_i18n_helper
  run_generator
  assert_file "app/mailers/notifier.rb" do |mailer|
    assert_match(/en\.notifier\.foo\.subject/, mailer)
    assert_match(/en\.notifier\.bar\.subject/, mailer)
  end
end

def test_invokes_default_test_framework
  run_generator
  assert_file "test/mailers/notifier_test.rb" do |test|
    assert_match(/class NotifierTest < ActionMailer::TestCase/, test)
    assert_match(/test "foo"/, test)
    assert_match(/test "bar"/, test)
  end
end
end

```

Be sure to explore all these tools available to you! Next, let's build a Rails engine that uses Rails' streaming functionalities to push updates to the browser as changes happen in our system.

In this chapter, we'll see

- Rails engines
- Rails live streaming functionality
- Ruby threads and queues
- Eager loading of code in Rails

CHAPTER 5

Streaming Server Events to Clients Asynchronously

In the previous chapters, we analyzed the Rails rendering stack inside and out. You learned that when a request reaches a controller, the controller gathers the required information to render a template. The template is retrieved from one of the resolvers, compiled, rendered fully and embedded in a layout. At the end of this process, you have a Ruby string representing this template. The string is set as the HTTP response and sent back to the client.

This approach works fine for the vast majority of applications. However, in some particular cases, we need to send our response in smaller chunks. Sometimes, those smaller chunks may be infinite; we keep on sending chunks to the client until the connection between the server and the client is closed.

Whenever we send a response in chunks, we say the server is *streaming* data to the client. Since Rails was built with the more traditional request-response scenario in mind, streaming support was added and improved in Rails over time and we are going to explore it in this chapter.

To explore how streaming works, let's write a Rails plugin that sends data to the browser whenever one of our stylesheets changes. The browser will use this information to reload the current page stylesheets, allowing developers to see changes in the HTML page as they modify their assets file, without a need to manually refresh the page in the web browser.

Since this plugin is going to have its own controller, assets, routes and more, we will rely on the power provided by Rails engines so we can add those functionalities as if they were part of a Rails application, but then bundle it in a gem to share across different projects.

5.1 Extending Rails with Engines

Rails engines allow our plugin to have its own controllers, models, helpers, views, assets, and routes, just like in a regular Rails application. Let's generate a plugin called `live_assets` using the Rails plugin generator. But this time we'll pass the `--full` flag, which will generate directories for models, controllers, routes and more:

```
$ rails plugin new live_assets --full
```

In addition to the files the generator normally creates for us, the `--full` flag also generates these files:

- an `app` directory with controllers, models and others
- a `config/routes.rb` file for routes
- a `lib/live_assets/engine.rb` file declaring our engine
- an empty `test/integration/navigation_test.rb` file that we can add our integration tests to

The most important file here is `lib/live_assets/engine.rb`, so let's take a closer look at it:

```
live_assets/1_live/lib/live_assets/engine.rb
module LiveAssets
  class Engine < ::Rails::Engine
    end
  end
end
```

In order to create an engine, we need to inherit from `Rails::Engine` and ensure our new engine is loaded as soon as possible. The generator we ran already did this for us by placing this line in `lib/live_assets.rb`:

```
live_assets/1_live/lib/live_assets.rb
require "live_assets/engine"

module LiveAssets
  end
```

Creating a `Rails::Engine` is quite similar to creating a `Rails::Railtie`. This is because a `Rails::Engine` is nothing more than a `Rails::Railtie` with some default initializers and the `Paths` API, which we'll see next.

Paths

A `Rails::Engine` does not have hard-coded paths. This means we are not required to place our models or controllers in `app/`; we can put them anywhere we

choose. For instance, we can configure our engine to load our controllers from `lib/controllers` instead of `app/controllers` as follows:

```
module LiveAssets
  class Engine < Rails::Engine
    paths["app/controllers"] = ["lib/controllers"]
  end
end
```

We can also have Rails load our controllers from both `app/controllers` and `lib/controllers`:

```
module LiveAssets
  class Engine < Rails::Engine
    paths["app/controllers"] << "lib/controllers"
  end
end
```

Those paths have the same semantics as in a Rails application: if you have a controller named `LiveAssetsController` inside `app/controllers/live_assets_controller.rb` or `lib/controllers/live_assets_controller.rb`, the controller will be loaded automatically when you need it, it doesn't need to be explicitly required.

For now, we are going to follow the conventional path and stick our controllers at `app/controllers`, so don't apply the previous changes. We can check all customizable paths for an engine by inspecting the Rails source code:

```
rails/railties/lib/rails/engine/configuration.rb
```

```
def paths
  @paths ||= begin
    paths = Rails::Paths::Root.new(@root)

    paths.add "app",          eager_load: true, glob: "*"
    paths.add "app/assets",   glob: "*"
    paths.add "app/controllers", eager_load: true
    paths.add "app/helpers",  eager_load: true
    paths.add "app/models",   eager_load: true
    paths.add "app/mailers",  eager_load: true
    paths.add "app/views"

    paths.add "app/controllers/concerns", eager_load: true
    paths.add "app/models/concerns",     eager_load: true

    paths.add "lib",          load_path: true
    paths.add "lib/assets",   glob: "*"
    paths.add "lib/tasks",    glob: "**/*.rake"

    paths.add "config"
    paths.add "config/environments", glob: "#{Rails.env}.rb"
    paths.add "config/initializers", glob: "**/*.rb"
```



```

paths.add "config/locales",      glob: "*.rb,*.yml"
paths.add "config/routes.rb"

paths.add "db"
paths.add "db/migrate"
paths.add "db/seeds.rb"

paths.add "vendor",              load_path: true
paths.add "vendor/assets",       glob: "*"

paths
end
end

```

The previous snippet shows the engine also specifies which paths should be eager loaded and which ones should not, besides also listing paths to locales, migrations and more. However, declaring a path is not enough, something has to be done with such paths at some point after all. That's where initializers come in.

Initializers

An engine has several initializers that are responsible for booting the engine. These initializers are more low-level and should not be confused with the ones available inside your application's config/initializers. Let's explore an example:

```
rails/railties/lib/rails/engine.rb
```

```

initializer :add_view_paths do
  views = paths["app/views"].existent
  unless views.empty?
    ActiveSupport.on_load(:action_controller){ prepend_view_path(views) }
    ActiveSupport.on_load(:action_mailer){ prepend_view_path(views) }
  end
end

```

This initializer is responsible for adding our engine views, usually defined in app/views, to ActionController::Base and ActionMailer::Base as soon as they are loaded, allowing a Rails application to use the templates defined in an engine. To see all initializers defined in a Rails::Engine, we can start a new Rails console under test/dummy with rails console and type the following:

```

Rails::Engine.initializers.map(&:name) # =>
[:set_load_path, :set_autoload_paths, :add_routing_paths,
 :add_locales, :add_view_paths, :load_environment_config,
 :append_assets_path, :prepend_helpers_path,
 :load_config_initializers, :engines_blank_point]

```

Working with an engine is pretty much the same as working with a Rails application. Since we know how to build applications, implementing our streaming plugin should feel familiar.

5.2 Live Streaming

In order to show how streaming works, let's create a controller called `LiveAssetsController` at `app/controllers/live_assets_controller.rb` that includes the `ActionController::Live` functionality and streams "hello world" continuously:

```
live_assets/1_live/app/controllers/live_assets_controller.rb
class LiveAssetsController < ActionController::Base
  include ActionController::Live

  def hello
    while true
      response.stream.write "Hello World\n"
      sleep 1
    end
  rescue IOError
    response.stream.close
  end
end
```

Our controller provides an action named `hello()` that streams Hello World every second. If, for any reason, the connection between the server and the client drops, `response.stream.write` will fail with `IOError` which we need to rescue and then properly close our stream.

We also need a route for this endpoint:

```
live_assets/1_live/config/routes.rb
Rails.application.routes.draw do
  get "/live_assets/:action", to: "live_assets"
end
```

We are almost ready to try out our streaming endpoint. However, since a Rails engine cannot run on its own, we need to start it via the application in `test/dummy`. Furthermore, the streaming functionality doesn't work in `WEBrick`, the server that ships with Ruby which is used by default by Rails, since `WEBrick` would first buffer our response before sending it to the client and, given our response is infinite, we would never see anything at all. For this reason, let's add `Puma`¹ as a development dependency to our `gemspec`:

```
live_assets/1_live/live_assets.gemspec
s.add_development_dependency "puma"
```

1. <http://puma.io/>

And finally go into the test/dummy directory and run rails s. Rails now starts Puma instead of WEBrick:

```
=> Booting Puma
=> Rails 4.0.0.rc2 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

Most browsers will also try to buffer the streaming response and it may take a while before they decide to show us anything. So, to test that our streaming endpoint really works, we'll use cURL² which works via the command line. Let's give curl a try:

```
$ curl -v localhost:3000/live_assets/hello
> GET /live_assets/hello HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0)
> Host: localhost:3000
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Frame-Options: SAMEORIGIN
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
< X-UA-Compatible: chrome=1
< Cache-Control: no-cache
< Content-Type: text/html; charset=utf-8
< X-Request-Id: f21f8c0d-d496-4bfa-944c-cd01b44b87ee
< X-Runtime: 0.003120
< Transfer-Encoding: chunked
<
Hello World
Hello World
```

Each second, you will see a new "Hello World" line appear on the screen over and over. This means our streaming end point is working. Hit CTRL+C in your keyboard to stop it as we are ready to move to more complex examples!

Server Sent Events

Developers have always had a need to receive updates from the server in the browser. For a long time polling was the most common technique to solve this problem. In polling, the browser makes frequent requests to the server asking for new data. In case no new information is available, the server returns an empty response and the browser starts a new request. Dependency on the frequency, the browser ends up sending many requests to the server generating a lot of overhead.

2. <http://curl.haxx.se/>

Over time, other techniques like *long polling* appeared. Like polling the browser periodically sends requests the server, except in this case if no updates are available the server waits for some ammount of time before sending an empty response. Although it performs better than traditional polling, it was plagued with cross compatiiblity issues in between browsers. Furthermore, many proxies and servers drop a connection if no communication happens after a while, making this approach less effective.

Based on such needs, the HTML5 specification includes two new APIs: Server Sent Events (SSE) and Websockets. Websockets allow both the client and server to exchange information over the same connection but, since it is a new protocol, it may require changes in your deployment stack to support it. On the other hand, Server Sent Events is a one-way communication channel from the server to the client which can be used with any web server that is able to stream responses. For those reasons, SSE is going to be our tool of choice for this chapter.

The underpinning of Server Sent Events is the *event stream format*, here is a sample response:

```
< HTTP/1.1 200 OK
< Content-Type: text/event-stream
<
< event: some_channel
< data: {"hello":"world"}
<
< event: other_channel
< data: {"another":"message"}
```

Messages are delimited by two new lines. Each message may have an event and an associated data. In this case, we chose the data to be a JSON payload but it could be any text. This is the format we need to return from the server when streaming. Let's create a new action called sse in our LiveAssetsController that streams a reloadCSS event at every one second:

```
live_assets/1_live/app/controllers/live_assets_controller.rb
def sse
  response.headers["Cache-Control"] = "no-cache"
  response.headers["Content-Type"] = "text/event-stream"

  while true
    response.stream.write "event: reloadCSS\ndata: {}\n\n"
    sleep 1
  end
rescue IOError
  response.stream.close
end
```

It is similar to our first action except now we need to set the proper response content type and disable any caching. With the server ready, let's implement the client side with JavaScript:

```
live_assets/1_live/app/assets/javascripts/live_assets/application.js
window.onload = function() {
  // 1. Connect to our event-stream
  var source = new EventSource('/live_assets/sse');

  // 2. This callback will be triggered on every reloadCSS event
  source.addEventListener('reloadCSS', function(e) {

    // 3. Load all CSS entries
    var sheets = document.querySelectorAll("[rel=stylesheet]");
    var forEach = Array.prototype.forEach;

    // 4. For each entry, clone it, add it to the
    //      document and remove the original after
    forEach.call(sheets, function(sheet){
      var clone = sheet.cloneNode();
      clone.addEventListener('load', function() {
        sheet.parentNode.removeChild(sheet);
      });
      document.head.appendChild(clone);
    });

  });
};
```

Our JavaScript file connects to our new endpoint and, on every reloadCSS event, it reloads all the stylesheets in the page. Notice our assets file was defined at `app/assets/live_assets/application.js`, this structure is required because by default Rails only precompiles asset files matching `application.*`. Since they are the only files precompiled, such files usually include all other existing files in the project, that's why they are frequently called *manifests*.

Finally, let's create a helper that will make it easy for applications to load our assets:

```
live_assets/1_live/app/helpers/live_assets_helper.rb
module LiveAssetsHelper
  def live_assets
    javascript_include_tag "live_assets/application"
  end
end
```

With our Server Sent Events mechanism ready, let's try it out. Go to our test/dummy application and create a controller and route:

```
live_assets/1_live/test/dummy/app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    render text: "Hello", layout: true
  end
end
```

```
live_assets/1_live/test/dummy/config/routes.rb
Dummy::Application.routes.draw do
  root to: "home#index"
end
```

And also change our layout to include the engine assets, but only in development:

```
live_assets/1_live/test/dummy/app/views/layouts/application.html.erb
<!DOCTYPE html>
<html>
<head>
  <title>Dummy</title>
  <%= stylesheet_link_tag "application", media: "all" %>
  <%= javascript_include_tag "application" %>
  <%= live_assets if Rails.env.development? %>
  <%= csrf_meta_tags %>
</head>
<body>
  <%= yield %>
</body>
</html>
```

Restart the dummy app and point the browser to *localhost:3000*. If your browser has a network panel that shows all HTTP requests sent by the browser, you might expect each stylesheet to be reloaded every second, but in practice that's not what happens:







Name Path	Method	Status Text	Type
 localhost	GET	200 OK	text/html
 application.css?body=1 /assets	GET	304 Not Modified	text/css
 application.js?body=1 /assets/live_assets	GET	304 Not Modified	application/javascript
 application.js?body=1 /assets	GET	304 Not Modified	application/javascript
 sse /live_assets	GET	200 OK	text/event-stream
 application.css?body=1 /assets	GET	(pending)	Pending

Figure 11—Pending application.css request on Google Chrome’s Network Panel

This happens because, even though Puma is a threaded web server, Rails only allows one thread to run at the same time. Let’s work around this issue by changing the dummy application to allow concurrency:

```
live_assets/1_live/test/dummy/config/application.rb
config.allow_concurrency = true
```

Since the browser is connected to the web server, waiting on a request, we need to first close the browser in order to restart the web server. Close your browser, restart the server, then reopen *localhost:3000* and we can finally see the stylesheet files reloading every second. We can verify that our stylesheets reload by editing *test/dummy/app/assets/stylesheets/application.css* and observing the changes happen live, without refreshing the page. For example, try setting the text color to red:

```
body { color: red; }
```

As you can see, our streaming of server sent events works! However, we can make a few improvements. First of all, we want to reload the stylesheets only when changes happen in the file system and not every second. Furthermore, watching for those changes should be efficient. In particular, if we have five pages open, we don’t want to query our file system for each open page; ideally, we should have one main file system listener entity that each request can subscribe to.

The second problem in our code so far is that we haven’t written any tests. This feature is particularly hard to test because we are streaming an infinite

amount of data, so instead of testing it directly from the controller, we need to break all the existing components into smaller, testable chunks.

Finally, since we have enabled `config.allow_concurrency`, we need to understand exactly the implications of such configuration and how it will affect the deployment of applications that rely on streaming. So don't go anywhere yet, we still have a lot to do!

5.3 File system notifications with Threads

A Rails application is generated with three assets directories by default: `app/assets`, `lib/assets`, and `vendor/assets`. Our assets should be split in between those directories in the same way we'd split our code: the `app` directory should contain assets related directly to our application, the `lib` directory holds isolated JavaScript or style sheet components that would be useful beyond our application and the `vendor` directory should contain third party files.

We would like to watch for file system changes in each of these directories. One option would be to manually check the modification time of each file in those directories every second or less. This is in general known as file system polling. Polling may be a good starting point but, as the number of assets grows, it can become very CPU intensive.

Luckily, most operating systems provide a notification mechanism for file system changes. All we need to do is pass all the directories we want to watch and, if any file is added, removed or changed, our code will be notified. And the *listen gem*³ exposes all major operating systems notifications mechanisms under a single, easy to use API.

Also, given our requirement of having one main entity watching the file system that our requests can subscribe to, let's wrap all the listening functionality inside a thread, which will run alongside our requests concurrently. Let's open up `lib/live_assets.rb` and implement it:

```
live_assets/2_listener/lib/live_assets.rb
require "live_assets/engine"
require "thread"
require "listen"

module LiveAssets
  attr_reader :subscribers
  @@subscribers = []

  # Subscribe to all published events.
```

3. <https://github.com/guard/listen>


```

def self.subscribe(subscriber)
  subscribers << subscriber
end

# Unsubscribe an existing subscriber.
def self.unsubscribe(subscriber)
  subscribers.delete(subscriber)
end

# Start a listener for the following directories.
# Every time a change happens, publish the given
# event to all subscribers available.
def self.start_listener(event, directories)
  Thread.new do
    Listen.to(*directories, latency: 0.5) do |_modified, _added, _removed|
      subscribers.each { |s| s << event }
    end
  end
end
end

```

Our code provides a mechanism for starting listeners inside a thread. These listeners watch a given set of directories and, every time a change happens, it pushes the registered event to each of the subscribers. Since we are using the `listen` gem, let's add it to the `gemspec` too:

```

live_assets/2_listener/live_assets.gemspec
s.add_dependency "listen"

```

Although we can't write integration tests for an action that streams file system updates infinitely, our listener functionality is decoupled from the streaming system, allowing us to test it in isolation. So let's write a test that starts a listener and verifies an event will be pushed to our subscriber whenever a change happens in the `test/tmp` directory:

```

live_assets/2_listener/test/live_assets_test.rb
require "test_helper"
require "fileutils"

class LiveAssetsTest < ActiveSupport::TestCase
  setup do
    FileUtils.mkdir_p "test/tmp"
  end

  teardown do
    FileUtils.rm_rf "test/tmp"
  end

  test "can subscribe to listener events" do
    # Create a listener

```

```

l = LiveAssets.start_listener(:reload, ["test/tmp"])

# Our subscriber is a simple array
subscriber = []
LiveAssets.subscribe(subscriber)

begin
  while subscriber.empty?
    # Trigger changes in a file until we get an event
    File.write("test/tmp/sample", SecureRandom.hex(20))
  end

  # Assert we got the event
  assert_includes subscriber, :reload
ensure
  # Clean up
  LiveAssets.unsubscribe(subscriber)
  l.kill
end
end
end

```

Excellent! It seems our listener works as expected. As you run the test suite, you may get some warnings from the listen gem. This is because by default it uses file system polling, unless you install a gem specific to your operating system that uses the file system notifications. Feel free to add the mentioned gem to your Gemfile (and not to the gems spec since it is not a strict dependency of our plugin).

Finally, we need to ensure a listener that watches over the assets directories starts whenever our application boots and pushes a :reloadCSS whenever there is a change. Let's write a test:

```

live_assets/2_listener/test/live_assets_test.rb
test "can subscribe to existing reloadCSS events" do
  subscriber = []
  LiveAssets.subscribe(subscriber)

  begin
    while subscriber.empty?
      FileUtils.touch("test/dummy/app/assets/stylesheets/application.css")
    end

    assert_includes subscriber, :reloadCSS
  ensure
    LiveAssets.unsubscribe(subscriber)
  end
end
end

```

Our test assumes the listener is already available by the time it runs. To make it pass, let's define an initializer inside our engine, similar to the ones we have seen earlier in this chapter, that starts the listener passing all the assets directories as argument:

```
live_assets/2_listener/lib/live_assets/engine.rb
module LiveAssets
  class Engine < ::Rails::Engine
    initializer "live_assets.start_listener" do |app|
      paths = app.paths["app/assets"].existent +
              app.paths["lib/assets"].existent +
              app.paths["vendor/assets"].existent

      paths = paths.select { |p| p =~ /stylesheets/ }

      if app.config.assets.compile
        LiveAssets.start_listener :reloadCSS, paths
      end
    end
  end
end
```

Notice we only start the listener if our assets are being dynamically compiled, this avoids starting the listener in production where assets are usually pre-compiled and the compile configuration is set to false by default.

Now that our listener is started by default and is ready to push events to subscribers, every time a new request is done to `/live_assets/sse`, we need to create a new subscriber, add it to the subscribers list and *wait* until a new event is pushed to our subscriber. Once the event arrives, we stream a server sent event to the browser as shown in the following figure:

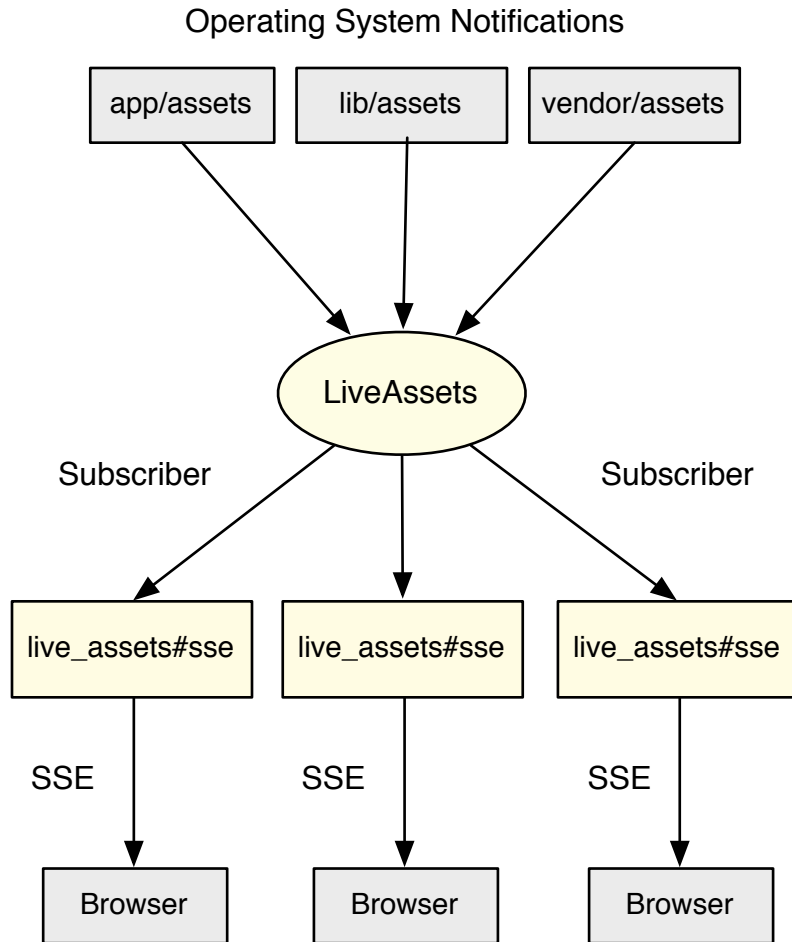


Figure 12—Visualization of file system notifications stack

The tricky part in this schema is waiting: we want each request to be idle until an event arrives. Checking for a new event in a loop, as we did in the test, is not an option since it will cause the CPU to spike. We could work around this by sleeping for a specific amount of time, like half second, and then checking for an event but this is also suboptimal. Ideally, what we want is to sleep for whatever time is necessary and automatically wake up as soon as an event arrives.

It happens that Ruby already ships with a perfect solution for this in its standard library: the `Queue` class. Let's take a look at it.

Threads and Queues

A queue is first-in first-out data structure. A queue implementation is accessible to any Ruby code by requiring `thread` and it provides a very simple API:

```
require "thread"
q = Queue.new

t = Thread.new do
  while last = q.pop
    sleep(1) # simulate cost
    puts last
  end
end

q << :foo
sleep(1)
$stdout.flush
```

This code creates a new Queue and a new Thread. Inside the thread there is a loop that calls `Queue#pop()`. If there is no item in the queue, the thread will block until an item is pushed to the queue. In the last three lines, we push a symbol to the queue, which will wake up the thread. After one second, if we flush what was written to `$stdout`, we'll see "foo" printed.

This means queues are the perfect structure for us to use as subscribers! If the queue is empty, the request is going to sleep until a new event arrives, then we stream this new event and go back to sleep. Let's create a class named `LiveAssets::SSESubscriber` that will receive those events and output them in the server sent event stream format, as shown in the following test:

```
live_assets/2_listener/test/live_assets/subscriber_test.rb
require "test_helper"
require "thread"

class LiveAssets::SubscriberTest < ActiveSupport::TestCase
  test "yields server sent events from the queue" do
    # Let's start our queue with some events
    queue = Queue.new
    queue << :reloadCSS
    queue << :ping
    queue << nil

    # And create a subscriber on top of it
    subscriber = LiveAssets::SSESubscriber.new(queue)

    stream = []
    subscriber.each do |msg|
```

```

    stream << msg
  end

  assert_equal 2, stream.length
  assert_includes stream, "event: reloadCSS\ndata: {}\n\n"
  assert_includes stream, "event: ping\ndata: {}\n\n"
end
end

```

Our test creates a queue, passes it to a subscriber and then consumes all events emitted by the subscriber. Notice we push nil to the queue as a way to signal we have no more events to consume. Let's implement the subscriber:

```

live_assets/2_listener/lib/live_assets/sse_subscriber.rb
require "thread"

module LiveAssets
  class SSESubscriber
    def initialize(queue = Queue.new)
      @queue = queue
      LiveAssets.subscribe(@queue)
    end

    def each
      while event = @queue.pop
        yield "event: #{event}\ndata: {}\n\n"
      end
    end

    def close
      LiveAssets.unsubscribe(@queue)
    end
  end
end

```

And autoload it:

```

live_assets/2_listener/lib/live_assets.rb
module LiveAssets
  autoload :SSESubscriber, "live_assets/sse_subscriber"
end

```

Finally, let's rewrite our live_assets#sse action to make use of our new subscriber:

```

live_assets/2_listener/app/controllers/live_assets_controller.rb
def sse
  response.headers["Cache-Control"] = "no-cache"
  response.headers["Content-Type"] = "text/event-stream"

  sse = LiveAssets::SSESubscriber.new
  sse.each { |msg| response.stream.write msg }
end

```

```
rescue IOError
  sse.close
  response.stream.close
end
```

Once again, restart the Puma server running inside test/dummy and verify that event streams only come after you edit `app/assets/stylesheets/application.css`, reflecting on immediate changes on the page. This time, let's make the font bigger:

```
body { font-size: 32px; }
```

We are almost done with our implementation, there is one last problem we have to tackle. In case no changes happen in a stylesheet for some time, we can remain a long period without streaming any data to the browser. This may cause the browser, the server or even a proxy in between to close the connection.

Timer

To ensure the connection won't be closed for being idle for long periods, we need a timer whose sole responsibility is to push a ping event down to subscribers every 10 seconds. Let's start by writing a test:

```
live_assets/3_final/test/live_assets_test.rb
test "receives timer notifications" do
  # Create a timer
  t = LiveAssets.start_timer(:ping, 0.5)

  # Our subscriber is a simple array
  subscriber = []
  LiveAssets.subscribe(subscriber)

  begin
    # Wait until we get an event
    true while subscriber.empty?
    assert_includes subscriber, :ping
  ensure
    # Clean up
    LiveAssets.unsubscribe(subscriber)
  end
end
```

Our timer will also run on its own thread and push events to the subscribers synchronously:

```
live_assets/3_final/lib/live_assets.rb
def self.start_timer(event, time)
  Thread.new do
    while true
```

```

    subscribers.each { |s| s << event }
    sleep(time)
  end
end
end

```

This implementation is enough to make our test pass! Finally, let's add another initializer to our engine responsible for starting the timer:

```

live_assets/3_final/lib/live_assets/engine.rb
initializer "live_assets.start_timer" do |app|
  if app.config.assets.compile
    LiveAssets.start_timer :ping, 10
  end
end

```

Restart the Puma web server and now ping events should be sent every 10 seconds. We have not registered any callback for such events on the JavaScript side, but we could if desired. Furthermore, the JavaScript EventSource object also emits open and close events for when the connection is opened and closed. Mozilla's Developer Network has more information on Server Sent Events that you can explore⁴.

Throughout our implementation, one of the details we just glanced over was the need for setting `config.allow_concurrency` to true. Now, with the live assets implementation out of the way, we have the perfect opportunity to discuss it.

5.4 Code loading techniques

In order to understand why we need to explicitly turn on `allow_concurrency`, we need to analyze the mechanisms available in Ruby and Rails to load code.

The most common form of loading code is Ruby's `require()` method:

```
require "live_assets"
```

Some libraries work great by simply using `requires`, but as they grow, some of them tend to rely on autoload techniques to avoid loading all their files upfront. Autoload is particularly important in Rails plugins because it helps applications boot time to stay low in development and test environments, since we only load modules when we first need them.

Autoload techniques

We have used *Ruby's autoload* in this chapter with the `LiveAssets::SSESubscriber` class:

4. https://developer.mozilla.org/en-US/docs/Server-sent_events/Using_server-sent_events


```

module LiveAssets
  autoload :SSESubscriber, "live_assets/sse_subscriber"
end

```

Now, the first time `LiveAssets::SSESubscriber` is accessed, it will be automatically loaded. Rails plugins and applications have another code loading technique, which is *Rails' autoload*. For example, our `LiveAssetsController` is also automatically loaded when we first need it. But this case is not handled by Ruby, but by `ActiveSupport::Dependencies` that ships with Rails.

The issue with both Ruby and Rails approaches is that loading code in Ruby is not atomic. If you have a request happening inside a Thread A and this thread starts loading `LiveAssetsController`, the `LiveAssetsController` class can be visible from a Thread B in another request *before* Thread A has finished loading the `app/controllers/live_assets_controller.rb` file. In this scenario, the Thread B has a partial implementation of the controller which, for example, could contain only the `hello()` action but not the `sse()` one, leading to a failure.

Although some Ruby implementations have been working towards making Ruby's autoload thread-safe (so the scenario previously described won't happen), Rails' autoload is not thread-safe. To work around this fact, whenever Rails needs to autoload code, Rails allows only one thread to run by default, which means it can only serve one request at a time. That's why we could not serve assets at the same time the Server Sent Events connection was open. To work around this limitation, we have set `config.allow_concurrency` to `true` at our own risk.

However, what does it mean for production? Do we also need to allow concurrency explicitly in production? What are our options to deploy this application?

Eager load techniques

In production, Rails *eager loads* your code: all of your models, controllers, helpers and more are loaded on boot. Since all Rails code is loaded up front and there is no code reloading happening, there is no autoload and allow concurrency defaults to `true`.

However, Rails is only going to eager load the code defined inside the `app` directory. If we are relying on Ruby autoload, we need to eager load the code ourselves! Otherwise we are possibly autoloading code in the middle of a request. This is what could happen with `LiveAssets::SSESubscriber`. Imagine this scenario:

On the first request to `/live_assets/sse`, Ruby will start to load `LiveAssets::SSESubscriber`. If many requests happen to this endpoint at the same time, the first

request may not have finished loading the subscriber, leading the following requests to see a partial definition of the subscriber. The solution is to also ensure `LiveAssets::SSESubscriber` is eager loaded when the Rails application boots. Since this is a common need in Rails plugins and in Rails codebase itself, Rails provides some conveniences for us.

The first convenience is the `config.eager_load_namespaces` configuration option, available in any Railtie or Engine, that keeps a list of namespaces to eager load. Let's add `LiveAssets` to this list in our engine definition:

```
live_assets/3_final/lib/live_assets/engine.rb
config.eager_load_namespaces << LiveAssets
```

Now, every time Rails is eager loading code, it will call `LiveAssets.eager_load!` to explicitly eager load our code. However we haven't implemented the `eager_load!` yet. Let's define it with the help of `ActiveSupport::Autoload`:

```
live_assets/3_final/lib/live_assets.rb
module LiveAssets
  extend ActiveSupport::Autoload

  eager_autoload do
    autoload :SSESubscriber
  end
end
```

By extending our module with `ActiveSupport::Autoload`, we automatically get a `LiveAssets.eager_load!` method that eager loads everything defined inside the `eager_autoload()` block. Also notice that we no longer need to pass a path to `autoload()`, Rails does its best to guess it based on the constant name.

That's all we need to do for Rails to eager load the rest of our code. Remember we need to resort to this technique every time we have code that is not automatically loaded by Rails, usually set up via Ruby autoloading. We can check all namespaces eager loaded by Rails by opening a console in the `test/dummy` directory:

```
Rails.application.config.eager_load_namespaces # =>
[ ActiveSupport, ActionDispatch, ActiveModel, ActionView,
  ActionController, ActiveRecord, ActionMailer, LiveAssets::Engine,
  LiveAssets, Dummy::Application ]
```

Keep in mind that eager loading is not only beneficial for threaded web servers like Puma but also for fork-based servers like Unicorn⁵. Unicorn operates by taking a snapshot of our Rails application just after it boots. By eager loading

5. <http://unicorn.bogomips.org/>

our code, we guarantee that this snapshot contains all of our code loaded up front and we don't need to spend time autoloading the code on every request.

Therefore, the decision of which web server to use when streaming is involved (and in general long-lived requests) usually boils down to the ability of your web server to handle many concurrent connections. For example, Unicorn works by starting a pool of web server processes and each web server is able to handle only one request at a time (the so-called single-threaded multi-process model). If one web server is streaming data or receiving a huge uploaded file, it is unable to serve other requests, even if data is streamed though the connection only every 10 seconds! On the other hand, the Puma web server we used in this chapter was able to handle other requests, even while we are streaming.

Unfortunately, there is no silver bullet and the best option when it comes to deployment is to benchmark the different web servers available. Multi-threaded web servers like Puma can handle multiple requests, the same for evented servers like Thin⁶. Servers like Passenger⁷ and Rainbows!⁸ allow you to mix different concurrency styles, so you have a hybrid multi-threaded multi-process deployment option. To add more options to the mix, you may even get better results by deploying on other platforms like JRuby⁹ and Rubinius¹⁰.

Often, different platforms are going to offer developers different thread-safety guarantees. For instance, array operations in JRuby are not guaranteed to be safe, which is an issue in our plugin. The `LiveAssets.subscribers` array is a global data structure and it could happen that two requests try to subscribe at the exactly same time, corrupting our array. That said, we need to wrap our subscribers operations in a mutex, that guarantees just one thread can execute that code at a given moment:

```
live_assets/3_final/lib/live_assets.rb
```

```
@mutex = Mutex.new
```

```
def self.subscribe(subscriber)
  @mutex.synchronize do
    subscribers << subscriber
  end
end
```

6. <http://code.macourmoyer.com/thin/>, however, as of version 1.5, Thin does not support streaming yet

7. <https://www.phusionpassenger.com/>

8. <http://rainbows.rubyforge.org>

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

10. <http://rubini.us/>

```

end

def self.unsubscribe(subscriber)
  @@mutex.synchronize do
    subscribers.delete(subscriber)
  end
end

```

By running the tests again, our test suite should remain green and our code is now also thread safe on JRuby! This is something important to keep in mind: every time we have global state manipulated in the middle of a request, we need to check across different Ruby implementations which thread safety guarantees that specific implementation gives us and act accordingly. Only by writing thread-safe code we can reliably have all the different deployment options available to ourselves.

5.5 Wrapping up

In this chapter, we have relied on Rails live streaming functionalities to push server sent events to the browser. Our implementation had a central entity listening for file system changes which were pushed to a group of subscribers. We have used Ruby's built-in threads and queues to control the information flow throughout our system.

The topics discussed in this chapter were used as basis to introduce code loading techniques in Rails and how those techniques affect the deployment of our applications. We have briefly glanced over thread-safety and the subtleties involved in different web servers and Ruby implementations.

Next, let's learn how to encapsulate our controllers' behavior in an object called ActionController::Responder and customize it to suit our needs! Then we'll discuss Rails generators and learn other ways to customize them.

In this chapter, we'll see

- Rails responders and the `respond_with()` method
- Rails generators' templates customization

CHAPTER 6

Writing DRY Controllers with Responders

Rails' scaffold generator is a great tool to help us prototype a new application. Its flexibility lets us swap the default template engine, test framework and ORM for our favorite options, ensuring we are productive regardless of the tools we choose. The only problem with scaffolding is that the generated controllers are still a little bit verbose, and we end up with a lot of behavior duplicated across different controllers. For example, here are the `create()`, and `destroy()` actions similar to what would be generated by the scaffold generator when invoked with a name attribute:

```
class UsersController < ApplicationController
  def create
    @user = User.new(user_params)
    respond_to do |format|
      if @user.save
        format.html { redirect_to @user, notice: 'User was successfully created.' }
        format.json { render action: 'show', status: :created, location: @user }
      else
        format.html { render action: 'new' }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end

  def destroy
    @user = User.find(params[:id])
    @user.destroy
    respond_to do |format|
      format.html { redirect_to users_url }
      format.json { head :no_content }
    end
  end

  private
end
```

```

def user_params
  params.require(:user).permit(:name)
end
end

```

All of these `respond_to()` blocks are very similar from one controller to another. To solve this issue, Rails provides a method called `respond_with()`, which uses an `ActionController::Responder` to abstract how our controllers respond. Using this new API, these actions are reduced to the following:

```

class UsersController < ApplicationController
  respond_to :html, :json

  def create
    @user = User.new(user_params)
    flash[:notice] = 'User was successfully created.' if @user.save
    respond_with(@user)
  end

  def destroy
    @user = User.find(params[:id])
    @user.destroy
    respond_with(@user)
  end

  private

  def user_params
    params.require(:user).permit(:name)
  end
end

```

We declare at the top which formats our controller responds to and delegate all the hard work to `respond_with()`. We could rewrite all of our actions using this cleaner API.

In this chapter, we'll learn how responders work, customize them to handle HTTP caching and flash messages automatically, and finally customize the scaffold generator to use `respond_with()` by default.

6.1 Understanding Responders

To understand the concepts behind responders, we must understand the three variables that affect how controllers respond: request type, HTTP verb, and resource status.

Navigational and API Requests

A controller generated by the scaffold generator responds to two formats by default: HTML and JSON. The reason why scaffold uses these two formats is because they represent two types of requests: navigational and API requests. The former is handled by a browser and holds formats like HTML and IPHONE, while the latter is used by machines and represents formats like XML and JSON.

```
def index
  @users = User.all
  respond_to do |format|
    format.html # index.html.erb
    format.json { render json: @users }
  end
end
```

Let's analyze this `index()` action, common to many Rails applications. The HTML format receives no block, so it renders a template, while the JSON format renders the JSON representation of the resource with `render json: @users`.

This means that, depending on the request type, controllers behave in one way or another. Consequently, in order to abstract how controllers work, responders should take the request type into account.

HTTP Verb

If we look at the `show()` and `new()` actions in a Rails controller, they respond similarly to `index()`, by rendering a template or a representation of the requested object. And what all these three actions have in common? The HTTP verb.

If we look at the remaining actions like `create()` and `destroy()`, they are triggered by the POST and DELETE verbs respectively, and they respond in a different fashion, redirecting to distinct places, returning distinct status codes and HTTP headers. In other words, the HTTP verb is another variable that affects how a controller responds.

Given the ammount of possible outcomes from a request, let's look closely at the scaffold controller generated by Rails and create a table representing how it responds depending on the request type and HTTP verb.

By default all GET requests, usually handled by actions like `index()`, `show()` and `new()`, render a template for navigational requests. If we have an API request, we may either render a template (like a `.builder` template) or render the resource

representation (for example, by calling `to_json()` on it). This information is summarized in the following table:

	Navigational	API
GET	render template	render template or render resource.to_format
POST		
PUT		
DELETE		

So far, we know how an scaffolded controller responds to GET in both request types. Now let's run through these other HTTP verbs and fill this whole table.

Resource Status

If we analyze the create() action, which represents a POST request, we realize that it has two branches: one if the resource is saved with success and the other if not. Each of these branches responds in a different way:

```
def create
  @user = User.new(user_params)

  respond_to do |format|
    if @user.save
      format.html { redirect_to @user, notice: 'User was successfully created.' }
      format.json { render action: 'show', status: :created, location: @user }
    else
      format.html { render action: "new" }
      format.json { render json: @user.errors, status: :unprocessable_entity }
    end
  end
end
```

The status of the resource determines how the scaffolded controller responds. In this case, we redirect if the resource is successfully created but render a page with errors if creation fails. We can also see this pattern in the update() action, which is invoked by PATCH and PUT requests.

Although the destroy() action generated by the scaffolding does not seem to depend on the resource status, we may eventually need to change the destroy() action to handle cases where resource.destroy returns false. For example, imagine a setup where a group has several managers. Because a group needs to have at least one manager, we implement a before_destroy() callback that checks for this condition every time we try to remove a manager. If the condition isn't met, both the callback and the destroy() method return false. This new scenario needs to be handled in the controller, usually by changing the destroy() action to show a flash message and redirect to the group page. In other words, even though the destroy() action generated by the scaffold does not depend on the resource status, DELETE requests may depend on it.

That said, the controller needs to know the resource status in order to respond to POST, PUT, and DELETE requests. Our table is modified to represent this new scenario and filled in accordingly for each request type, HTTP verb, and resource status:

		Navigational	API
GET		render template	render template or resource.to_format
POST	Success	redirect_to resource	render template or resource.to_format
POST	Failure	render :new	render resource.errors
PUT	Success	redirect_to resource	head :no_content
PUT	Failure	render :edit	render resource.errors
DELETE	Success	redirect_to collection	head :no_content
DELETE	Failure	redirect_to collection	render resource.errors

Table 1—Scaffold Resource Behaviour

Whenever you invoke `respond_with()` in your controllers, it calls the `ActionController::Responder` class, which is nothing more than this whole table written in Ruby code. Let's explore how `ActionController::Responder` is implemented and how we can modify it to behave in a custom way.

6.2 Exploring ActionController::Responder

Anything that responds to `call()`, accepting three arguments, can be a responder. The three arguments given to `call()` are the current controller, the resource (or a nested resource or an array of resources), and a hash of options. All the options given to `respond_with()` are forwarded to the responder as the third argument.

`ActionController::Responder` implements the `call()` method in a single line of code, as we can see in Rails source code:

```
rails/actionpack/lib/action_controller/metal/responder.rb
def self.call(*args)
  new(*args).respond
end
```

The `call()` method just forwards these three arguments to the `ActionController::Responder` initialization and then calls `respond()`:

```
rails/actionpack/lib/action_controller/metal/responder.rb
# Main entry point for responder responsible
# for dispatching to the proper format.
```

```

def respond
  method = "to_#{format}"
  respond_to?(method) ? send(method) : to_format
end

# HTML format does not render the resource,
# it always attempt to render a template.
def to_html
  default_render
rescue ActionController::MissingTemplate => e
  navigation_behavior(e)
end

# to_js simply tries to render a template.
# If no template is found, raises the error.
def to_js
  default_render
end

# All other formats follow the procedure below. First we
# try to render a template, if the template is not available,
# we verify if the resource responds to :to_format and display it.
def to_format
  if get? || !has_errors? || response_overridden?
    default_render
  else
    display_errors
  end
rescue ActionController::MissingTemplate => e
  api_behavior(e)
end

```

The `respond()` method checks whether the responder handles the current request format. If positive, it calls the specific method for this format; otherwise, it calls `to_format()`. Since `ActionController::Responder` defines only `to_html()` and `to_js()`, only HTML and JS requests have a custom behavior, and all others fall back to the `to_format()` case.

By analyzing both `to_html()` and `to_format()` implementations, we see that the former responds with `navigation_behavior()` and the latter with `api_behavior()`. If we add a new navigational format to an application, like `IPHONE`, the responder will treat it as an API format and not navigational. Luckily, because of how responders work, we can make `IPHONE` use the navigational behavior by simply aliasing the `:to_iphone` method to `:to_html` in an initializer.

```

ActionController::Responder.class_eval do
  alias :to_iphone :to_html
end

```

Additionally, note that a responder always invokes the `default_render()` method before falling back to the API or navigational behavior:

```

rails/actionpack/lib/action_controller/metal/responder.rb
def to_html
  default_render
rescue ActionController::MissingTemplate => e
  navigation_behavior(e)
end

```

`default_render()` simply tries to render a template in case none was rendered yet (performed?() must return The code tag should not be here.), and in case the template is not found, it raises an `ActionView::MissingTemplate`, which is properly rescued, allowing responders behavior to kick in.

Here's how Rails implements the `navigational_behavior()` and `api_behavior()` methods:

```

rails/actionpack/lib/action_controller/metal/responder.rb
DEFAULT_ACTIONS_FOR_VERBS = {
  post: :new,
  patch: :edit,
  put: :edit
}
# This is the common behavior for formats associated
# with browsing, like :html, :iphone and so forth.
def navigation_behavior(error)
  if get?
    raise error
  elsif has_errors? && default_action
    render :action => default_action
  else
    redirect_to navigation_location
  end
end

# This is the common behavior for formats associated
# with APIs, such as :xml and :json.
def api_behavior(error)
  raise error unless resourceful?
  if get?
    display resource
  elsif post?
    display resource, :status => :created, :location => api_location
  else
    head :no_content
  end
end

```

```

end
def resourceful?
  resource.respond_to?("to_#{format}")
end

def has_errors?
  resource.respond_to?(:errors) && !resource.errors.empty?
end

def resource_location
  options[:location] || resources
end
alias :navigation_location :resource_location
alias :api_location :resource_location

# Display is just a shortcut to render a resource with the current format.
#
#   display @user, status: :ok
#
# For XML requests it's equivalent to:
#
#   render xml: @user, status: :ok
#
# Options sent by the user are also used:
#
#   respond_with(@user, status: :created)
#   display(@user, status: :ok)
#
# Results in:
#
#   render xml: @user, status: :created
#
def display(resource, given_options={})
  controller.render given_options.merge!(options).merge!(format => resource)
end

```

The `navigational_behavior()` implementation maps straight to the table in [Resource Status, on page 113](#). For a GET request, it raises a missing template error, because the only option for GET requests is to render a template, which we already tried and did not succeed.

For other HTTP verbs, the navigational behavior checks whether the resource has errors. If positive and a default action is given, it renders the default action specified by the `DEFAULT_ACTIONS_FOR_VERBS` hash. Finally, if the resource does not have errors, it redirects to the resource, which is what we expect in success cases.

The `api_behavior()` implementation goes through a different path. Notice that it makes use of the `display()` method, which merges the options given to

`respond_with()` and adds a format before calling `render`. In other words, when we call `respond_with()` like this:

```
respond_with @user, status: :created
```

on GET requests for JSON format, the controller responds as follows:

```
render json: @user, status: :created
```

It's important to realize Rails responders do not call `@user.to_json`. They simply delegate this responsibility to the `render()` method and consequently to the `:json` renderer, as we saw in [Section 1.2, Writing the Renderer, on page 5](#)! This is important because people can add new renderers, and they work in responders without adding any other line of code.

Finally, the last customization available in responders can be done in our own controller. Imagine that we have a responder that works great in all cases, except for one specific action and format where we want it to behave differently. We can customize the responder for this scenario using the same block API as in `respond_to()`:

```
def index
  @users = User.all
  respond_with(@users) do |format|
    format.json { render json: @users.to_json(some_specific_option: true) }
  end
end
```

And this all works because `respond_with()` forwards the block given to `format.json` to the responder when the request format is JSON. This block, whenever available, is called by the `default_render()` seen in the previous responder snippets.

The great advantage in using `ActionController::Responder` is that it centralizes how our application should behave per format. That said, if we want to change how all controllers behave at once, we just need to create our own responder and configure Rails to use it, as shown here:

```
ApplicationController.responder = MyAppResponder
```

Furthermore, we can even set custom responders for specific controllers in our application:

```
class UsersController < ApplicationController
  self.responder = MyCustomUsersResponder
end
```

Let's create a responder with some extra behavior and configure Rails to use it.

6.3 The Flash Responder

The scaffolded controller uses flash messages in both `create()` and `update()` actions. These messages are quite similar across different controllers. Wouldn't it be nice then if we could set these flash messages by default inside responders but still provide a nice API to change them?

Let's implement this feature using I18n. By using I18n, we can easily lookup flash messages from YAML files, configure default values, while making it possible to translate such messages in the future. Let's use rails plugin to create a new project called responders:

```
$ rails plugin new responders
```

Let's start by writing some tests that access `create()`, `update()` and `destroy()` actions and ensure a flash message is being exhibited to the client:

```
responders/1_flash/test/responders/flash_test.rb
require "test_helper"

class FlashTest < ActionController::TestCase
  tests UsersController

  test "sets notice message on successful creation" do
    post :create, user: { name: "John Doe" }
    assert_equal "User was successfully created.", flash[:notice]
  end

  test "sets notice message on successful update" do
    user = User.create!(name: "John Doe")
    put :update, id: user.id, user: { name: "Another John Doe" }
    assert_equal "User was successfully updated.", flash[:notice]
  end

  test "sets notice message on successful destroy" do
    user = User.create!(name: "John Doe")
    delete :destroy, id: user.id
    assert_equal "User was successfully destroyed.", flash[:notice]
  end
end
```

The test relies on the existence of a `UsersController`, which can be generated by invoking the scaffold generator inside the dummy application at `test/dummy`. When invoking the generator, let's skip the test files, guaranteeing they won't conflict with our plugin tests:

```
$ rails g scaffold User name:string --no-test-framework
```

Next run migrations and set up our test database:

```
$ rake db:migrate db:test:clone
```

Notice however the scaffold generator does not use the responder API. Let's change the generated controller to use `respond_with()` and then remove all flash messages, as they will be automatically set by our responders. This is our `UserController` after these changes:

```
responders/1_flash/test/dummy/app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  respond_to :html, :json
  before_action :set_user, only: [:show, :edit, :update, :destroy]

  def index
    @users = User.all
    respond_with(@users)
  end

  def show
    respond_with(@user)
  end

  def new
    @user = User.new
    respond_with(@user)
  end

  def edit
  end

  def create
    @user = User.new(user_params)
    @user.save
    respond_with(@user)
  end

  def update
    @user.update(user_params)
    respond_with(@user)
  end

  def destroy
    @user.destroy
    respond_with(@user)
  end

  private

  # Use callbacks to share common setup or constraints between actions.
  def set_user
    @user = User.find(params[:id])
  end
end
```



```

end

# Only allow a trusted parameter "white list" through.
def user_params
  params.require(:user).permit(:name)
end
end

```

All the actions in this controller should invoke `respond_with()` with the exception of the `edit()` action. This is because the `edit()` action is only used by navigational requests, as its main responsibility is to show the form to edit the resource.

When we run the test suite, it fails with the following message:

```

1) Failure:
test_sets_notice_message_on_successful_creation(FlashTest):
Expected: "User was successfully created."
Actual: nil

```

The failure is expected because we haven't implemented our responders yet. Since we will develop two responders extensions in this chapter, let's write each extension as a module, allowing developers to include such functionality wherever they desire to. Our first module is called `Responders::Flash` and it looks flash messages up via the I18n framework.

Imagine a request with valid parameters at the `create()` action in the `UsersController`. When `respond_with()` is called and no flash message is set, the responder should try to find an I18n message under the controller namespace and action, which in this case is `"flash.users.create.notice"`. If an I18n message is found, the responder should set it in `flash[:notice]`, and it will be properly exhibited in the next request.

Alternatively, if the request at `UserController#create` does not have valid parameters (that is, the created user is invalid), the responder should search for a message at `"flash.users.create.alert"` and set `flash[:alert]` instead.

With these requirements in mind, let's write our `Responders::Flash` module:

```

responders/1_flash/lib/responders/flash.rb
module Responders
  module Flash
    def to_html
      set_flash_message! unless get?
      super
    end

    private

    def set_flash_message!

```

```

status = has_errors? ? :alert : :notice
return if controller.flash[status].present?

message = i18n_lookup(status)
controller.flash[status] = message if message.present?
end

def i18n_lookup(status)
  namespace = controller.controller_path.gsub("/", ".")
  action     = controller.action_name

  lookup = [namespace, action, status].join(".").to_sym
  default = ["actions", action, status].join(".").to_sym

  I18n.t(lookup, scope: :flash, default: default,
        resource_name: resource.class.model_name.human)
end
end
end

```

Our module overwrites the `to_html()` behavior to set flash messages for non-GET requests and then calls `super`, allowing the responder behavior and other extensions to kick in.

Besides setting flash messages based in the controller namespace, our implementation gives an "actions" namespace as a `:default` option to `I18n.t`. This allows `I18n` to fall back to "flash.actions.create.notice" if a message cannot be found at "flash.users.create.notice".

This fallback mechanism lets us provide application-wide default messages so we don't need to repeat ourselves in each controller. Let's set the default scaffold messages inside our plugin by simply creating a YAML file with the following:

```
responders/1_flash/lib/responders/locales/en.yml
```

```

en:
  flash:
    actions:
      create:
        notice: "%{resource_name} was successfully created."
        alert: ""
      update:
        notice: "%{resource_name} was successfully updated."
        alert: ""
      destroy:
        notice: "%{resource_name} was successfully destroyed."
        alert: "%{resource_name} could not be destroyed."

```

Now any controller will use the flash messages configured in this YAML file, unless a specific key for the controller is given. To achieve this, we use `I18n` interpolation, which allows us to use `%{resource_name}` in our messages, and it will be properly replaced by the resource human name given to `:resource_name` when `I18n.t` is invoked.

To finally make our tests pass, we need to make our `Responders::Flash` available. Instead of monkey-patching Rails default responder, let's inherit from it and add our own customizations. We will set our new responders as the default one and also add our default yml file with translations to `I18n`'s load path:

```
responders/1_flash/lib/responders.rb
```

```
require "action_controller"
require "responders/flash"
```

```
module Responders
```

```
  class AppResponder < ActionController::Responder
    include Flash
```

```
  end
```

```
end
```

```
ActionController::Base.responder = Responders::AppResponder
```

```
require "active_support/i18n"
```

```
I18n.load_path << File.expand_path("../responders/locales/en.yml", __FILE__)
```

Run the test suite and you'll see that our responder is properly triggered and is using the default flash messages in the YAML file! Since our tests only assert for "notice" messages, let's write one extra test asserting that "alert" messages will be shown in case of failures:

```
responders/1_flash/test/responders/flash_test.rb
```

```
test "sets alert messages from the controller scope" do
```

```
  begin
```

```
    I18n.backend.store_translations :en,
```

```
      flash: { users: { destroy: { alert: "Cannot destroy!" } } }
```

```
    user = User.create!(name: "Undestroyable")
```

```
    delete :destroy, id: user.id
```

```
    assert_equal "Cannot destroy!", flash[:alert]
```

```
  ensure
```

```
    I18n.reload!
```

```
  end
```

```
end
```

The test creates a resource and tries to destroy it but fails, showing a message that the resource cannot be destroyed. As we did in [Aiming for an Active](#)

[Model-Compliant API, on page 21](#), we're using the I18n API to store translations on the fly for the failure scenario.

To make the test pass, let's add a `before_destroy()` callback that adds error messages to `@user.errors` and returns false if the username is "Undestroyable":

```
responders/1_flash/test/dummy/app/models/user.rb
class User < ActiveRecord::Base
  before_destroy do
    if name == "Undestroyable"
      errors.add(:base, "is undestroyable")
      false
    end
  end
end
```

We need to attach errors to the model in order to signal to the responder that something went wrong. With this final change, our tests pass again! There are other features we could add to our flash responder, but let's move on and make our responder a better HTTP citizen.

6.4 HTTP Cache Responder

Rails has embraced REST since version 1.2 and since then, developing APIs has become easier and easier. However, as your application grows, you may have to focus more on your API implementation and find ways to optimize the number of requests it can handle.

When you expose an API, it's common that a client requests a resource to the server several times, and the client always gets the same response back since the requested resource has not changed. In these cases, the server is wasting time rendering the same resource all over again, and the client is parsing the same response just to find out that nothing has changed.

Luckily, the HTTP 1.1 specification has a section dedicated to caching. The previous problem could be easily solved if the server appends a Last-Modified header to the response with a timestamp representing when the resource was last modified. For subsequent requests, the client should add an If-Modified-Since header with this timestamp, and if the resource has not changed, the server should return a 304 Not Modified status and does not need to render the resource again. Upon receiving a 304 status, the client knows that nothing has changed. This scenario is exhibited in the following figure:

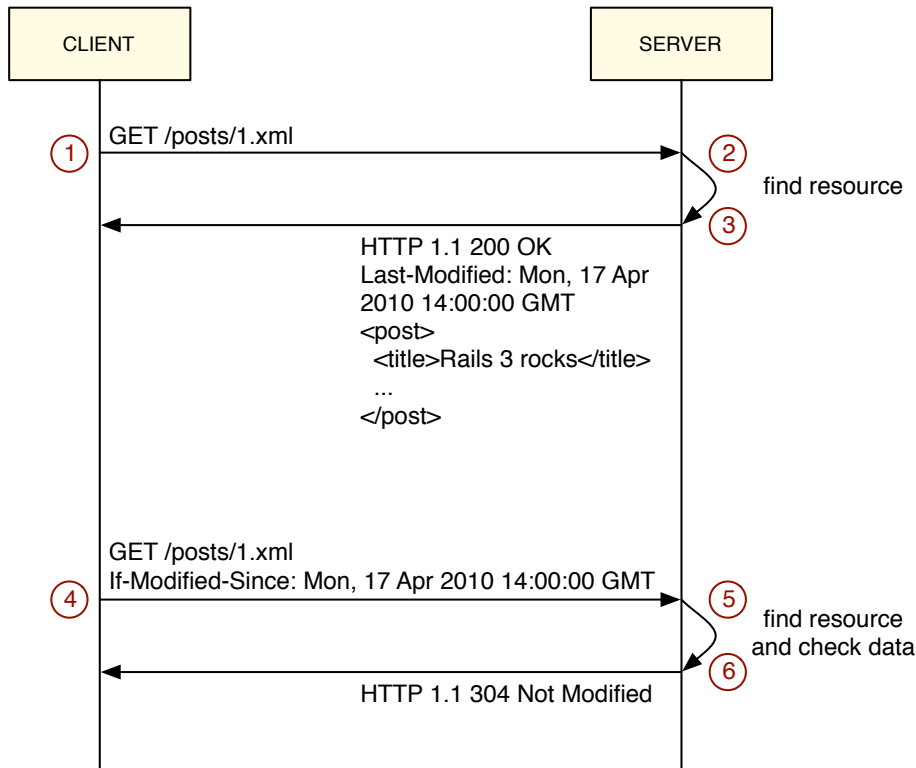


Figure 13—Client and server interaction with HTTP Cache

As usual, let's start our implementation by writing tests first. There are at least three scenarios to take into account:

- When `If-Modified-Since` is not provided, our controller responds normally but adds a `Last-Modified` header.
- When `If-Modified-Since` is provided and fresh, our controller responds with a status of 304 and a blank body.
- When `If-Modified-Since` is provided and not fresh, our controller responds normally but adds a new `Last-Modified` header.

To write these tests, we need to modify some request headers and verify that a few response headers are being properly set. Let's once again use the existing `UserController`, available from the dummy app, to support our tests:

```

responders/2_http_cache/test/responders/http_cache_test.rb
require "test_helper"

class HttpCacheTest < ActionController::TestCase
  tests UsersController

  setup do
    @request.accept = "application/json"
    ActionController::Base.perform_caching = true

    User.create(name: "First", updated_at: Time.utc(2009))
    User.create(name: "Second", updated_at: Time.utc(2008))
  end

  test "responds with last modified using the latest timestamp" do
    get :index
    assert_equal Time.utc(2009).httpdate, @response.headers["Last-Modified"]
    assert_match '"name":"First"', @response.body
    assert_equal 200, @response.status
  end

  test "responds with not modified if request is still fresh" do
    @request.env["HTTP_IF_MODIFIED_SINCE"] = Time.utc(2009, 6).httpdate
    get :index
    assert_equal 304, @response.status
    assert @response.body.blank?
  end

  test "responds with last modified if request is not fresh" do
    @request.env["HTTP_IF_MODIFIED_SINCE"] = Time.utc(2008, 6).httpdate
    get :index
    assert_equal Time.utc(2009).httpdate, @response.headers["Last-Modified"]
    assert_match '"name":"First"', @response.body
    assert_equal 200, @response.status
  end
end

```

Rails provides several helpers on top of the HTTP Cache specification, and we are simply going to use them to create a new module called `Responders::HttpCache` that automatically adds HTTP Cache to all GET requests:

```

responders/2_http_cache/lib/responders/http_cache.rb
module Responders
  module HttpCache
    delegate :response, to: :controller

    def to_format
      return if do_http_cache? && do_http_cache!
      super
    end
  end
end

```

```

private

def do_http_cache!
  response.last_modified ||= max_timestamp if max_timestamp
  head :not_modified if fresh = request.fresh?(response)
  fresh
end

# Iterate through all resources and find the last updated.
def max_timestamp
  @max_timestamp ||= resources.flatten.map do |resource|
    resource.updated_at.try(:utc) if resource.respond_to?(:updated_at)
  end.compact.max
end

# Just trigger the cache if it's a GET request and
# perform caching is enabled.
def do_http_cache?
  get? && ActionController::Base.perform_caching
end
end
end

```

Our implementation mainly loops through all given resources and retrieves the timestamp of the last updated one. We then update the response object, and if the request is fresh (that is, if the resource was not modified in between requests), we return a 304 status to the client and do not render any resource, since `to_format()` returns before calling `super`.

Before running our new tests, let's require `responders/http_cache` and include `Responders::HttpCache` in our `AppResponder`, modifying the top of the `lib/responders.rb` file:

```

responders/2_http_cache/lib/responders.rb
require "responders/flash"
require "responders/http_cache"

module Responders
  class AppResponder < ActionController::Responder
    include Flash
    include HttpCache
  end
end

```

And that's it! Our test suite is green again! We extracted the Flash and HTTP Cache responsibility from our controllers, and now it's handled automatically by our responder!

6.5 More Ways to Customize Generators

Now that we understand how responders really work and how to adapt them to our needs, we can feel confident about using them more and more in our controllers. The only issue is that the scaffold generator uses `respond_to()` by default and not `respond_with()`.

On the other hand, we learned in [Generators' Hooks, on page 75](#) how to customize generators, and there must be a hook to customize the controller generated in scaffold, right? To confirm our thoughts, we just need to take a look at the output generated by the scaffold:

```
invoke active_record
create db/migrate/20130415031520_create_users.rb
create app/models/user.rb
invoke resource_route
route resources :users
invoke scaffold_controller
create app/controllers/users_controller.rb
invoke erb
create app/views/users
create app/views/users/index.html.erb
create app/views/users/edit.html.erb
create app/views/users/show.html.erb
create app/views/users/new.html.erb
create app/views/users/_form.html.erb
invoke helper
create app/helpers/users_helper.rb
invoke assets
invoke js
create app/assets/javascripts/users.js
invoke css
create app/assets/stylesheets/users.css
invoke css
create app/assets/stylesheets/scaffold.css
```

Each invoke in the output is a hook that we can overwrite. This means we can indeed replace the `scaffold_controller` generator by another one that fits our needs.

However, this is not how we will solve this problem. Instead, let's use another feature provided by Rails generators that allows us to customize generator templates without a need to use generator hooks.

Generators' Source Path

Consider the following line in a Rails generator:

```
copy_file "controller.rb", "app/controller/#{file_name}_controller.rb"
```


It simply copies the `controller.rb` file from the generators' source to the given destination, which for a `UserController` would be `app/controllers/users_controller.rb`.

However, a generator can have more than one source! Before copying a file to a given location, the generator searches for this source file in several locations, called *source paths*. The `source_root()` class method we specified in [Creating Our First Generator, on page 79](#) is actually the last place a generator searches for a template.

This behavior is actually built into Thor¹, but Rails wraps the behavior nicely by automatically adding the `lib/templates` directory inside your application to all generators' source paths. This means that the `Rails::Generators::ScaffoldControllerGenerator` used in `scaffold` will always try to find a template at `lib/templates/rails/scaffold_controller` before using the one provided by Rails.

When we look at the implementation of `Rails::Generators::ScaffoldControllerGenerator` in the Rails source code, we can easily see the logic that copies the controller template:

```
rails/railties/lib/rails/generators/rails/scaffold_controller/scaffold_controller_generator.rb
module Rails
  module Generators
    class ScaffoldControllerGenerator < NamedBase
      def create_controller_files
        template "controller.rb", File.join("app/controllers", class_path,
      "#{controller_file_name}_controller.rb")
      end
    end
  end
end
```

It uses a template named `controller.rb`, which is available at `railties/lib/rails/generators/rails/scaffold_controller/templates/controller.rb`. According to the source paths, if we place a file at `lib/templates/rails/scaffold_controller/controller.rb` inside our application, Rails will use this application file instead of the one that ships with Rails!

You can easily try this by creating a new Rails application, placing an empty file at `lib/templates/rails/scaffold_controller/controller.rb` inside your application, and running the `scaffold` command. When you check the controller created by the `scaffold`, it's empty as well!

Next, let's use this awesome feature to customize `scaffold` to use `respond_with()` by default.

1. <https://github.com/wycats/thor>

Using respond_with by Default

To use `respond_with()` by default in the scaffold, let's place a template inside our application's `lib/templates`. However, to avoid doing this in each new application, we are going to create a generator that copies a file to the proper location.

Let's call this generator `Responders::Generators::InstallGenerator` and implement it as follows:

```
responders/3_final/lib/generators/responders/install/install_generator.rb
module Responders
  module Generators
    class InstallGenerator < Rails::Generators::Base
      source_root File.expand_path("../templates", __FILE__)
      def copy_template_file
        copy_file "controller.rb",
          "lib/templates/rails/scaffold_controller/controller.rb"
      end
    end
  end
end
```

The template used by our generator is exhibited next:

```
responders/3_final/lib/generators/responders/install/templates/controller.rb
<% module_namespacing do -%>
class <%= controller_class_name %>Controller < ApplicationController
  before_action :set_<%= singular_table_name %>,
    only: [:show, :edit, :update, :destroy]

  # GET <%= route_url %>
  def index
    @<%= plural_table_name %> = <%= orm_class.all(class_name) %>
    respond_with(@<%= plural_table_name %>)
  end

  # GET <%= route_url %>/1
  def show
    respond_with(@<%= singular_table_name %>)
  end

  # GET <%= route_url %>/new
  def new
    @<%= singular_table_name %> = <%= orm_class.build(class_name) %>
    respond_with(@<%= singular_table_name %>)
  end

  # GET <%= route_url %>/1/edit
  def edit
  end
end
```

```

# POST <%= route_url %>
def create
  @<%= singular_table_name %> = <%= orm_class.build(class_name,
                                                    "#{singular_table_name}_params") %>

  @<%= orm_instance.save %>
  respond_with(@<%= singular_table_name %>)
end

# PATCH/PUT <%= route_url %>/1
def update
  @<%= orm_instance.update("#{singular_table_name}_params") %>
  respond_with(@<%= singular_table_name %>)
end

# DELETE <%= route_url %>/1
def destroy
  @<%= orm_instance.destroy %>
  respond_with(@<%= singular_table_name %>)
end

private

# Use callbacks to share common setup or constraints between actions.
def set_<%= singular_table_name %>
  @<%= singular_table_name %> = <%= orm_class.find(class_name, "params[:id]") %>
end

# Only allow a trusted parameter "white list" through.
def <%= "#{singular_table_name}_params" %>
  <%- if attributes_names.empty? -%>
    params[<%= ":%#{singular_table_name}" %>]
  <%- else -%>
    params.require(<%= ":%#{singular_table_name}" %>).
      permit(<%= attributes_names.map { |name| ":%#{name}" }.join(', ') %>)
  <%- end -%>
end
end
<% end -%>

```

The previous template is based on the one that ships with Rails, but it replaces all `respond_to()` calls with `respond_with()`. Note it also uses several methods we have already discussed, except `orm_class()` and `orm_instance()`, which we are going to discuss soon.

To try the new generator, you can simply move to the dummy application and invoke it:

```
$ rails g responders:install
```

Now when you scaffold any new resource, it will use the template we just installed! This means the Rails scaffold is flexible not only for Rails extensions like Haml or RSpec but also for application developers because they can customize scaffold to fit their workflow, application structure and markup.

Generators and ORM Agnosticism

We already discussed Active Model and its role in ORM agnosticism. We also talked about generator hooks, which provide a way for other ORMs to hook into model and scaffold generators. Whenever we use the scaffold generator, those two roles intersect and Rails provides a custom API to allow ORMs to customize the generated code.

Rails controllers are responsible for interacting with the model and passing the desired objects to the view. In other words, controllers should interact with the current ORM and retrieve the required information from it. The controller generated by scaffolding should change depending on the ORM used. Rails solves this problem by creating an object responsible to tell the scaffold generator how the interaction with the ORM happens. The basic implementation for this object is available in the Rails source code, and it looks like this:

```
rails/railties/lib/rails/generators/active_model.rb
module Rails
  module Generators
    class ActiveModel
      attr_reader :name

      def initialize(name)
        @name = name
      end

      # GET index
      def self.all(klass)
        "#{klass}.all"
      end

      # GET show
      # GET edit
      # PATCH/PUT update
      # DELETE destroy
      def self.find(klass, params=nil)
        "#{klass}.find(#{params})"
      end

      # GET new
      # POST create
      def self.build(klass, params=nil)
```

```

    if params
      "#{klass}.new(#{params})"
    else
      "#{klass}.new"
    end
  end

  # POST create
  def save
    "#{name}.save"
  end

  # PATCH/PUT update
  def update(params=nil)
    "#{name}.update(#{params})"
  end

  # POST create
  # PATCH/PUT update
  def errors
    "#{name}.errors"
  end

  # DELETE destroy
  def destroy
    "#{name}.destroy"
  end
end
end
end
end

```

The `orm_class()` simply points to `Rails::Generators::ActiveModel`, and `orm_instance()` points to an instance of this same class. So, whenever we invoke `orm_class.all("User")` in the template, it invokes `Rails::Generators::ActiveModel.all("User")` and returns `User.all`, which is the normal Active Record behavior.

The `orm_instance()` behaves similarly, except we don't need to pass the resource name as an argument, since we already did it in initialization. For this reason, `orm_instance.save` successfully returns `user.save` for Active Record.

All interaction between the controller and the ORM is specified in `Rails::Generators::ActiveModel`. The agnosticism comes from the fact that any ORM can provide its own implementation of this class. All we need to do is to define an Active Model class inside the ORM's generator namespace.

For example, `DataMapper` has different syntax for finding and updating records. So, it needs to inherit from `Rails::Generators::ActiveModel` and implement the new API:

```

module DataMapper
  module Generators
    class ActiveModel < ::Rails::Generators::ActiveModel
      def self.find(klass, params=nil)
        "#{klass}.get(#{params})"
      end
    end
  end
end

```

This structure provided by generators plus the Active Model API is what makes agnosticism possible in Rails and allows any developer to choose the ORM they find most appropriate for their work.

6.6 Wrapping Up

In this chapter, we looked into responders to understand how they work and how to customize them. As a proof of concept, we developed two extensions for responders, one to handle flash messages and another to handle HTTP caching.

There is much more we could delegate to responders. In the HTTP layer, we could use the If-Unmodified-Since request-header to provide conditional PUT requests, wherein the resource is updated only if not modified after the given date; otherwise, we return a 409 Conflict status. This scenario is exhibited in [Figure 14, Client and server interaction with HTTP conditional requests, on page 135](#).

We also took another look at Rails generators and learned more about ORM agnosticism.

If you want to bring responders and `respond_with()` to your workflow, you may want to try the Responders gem by Plataformatec,² which implements both extensions developed in this chapter with some extra features, such as the ability to change responders to redirect to the `index()` action instead of the `show()` action when a user is created or updated.

Next, let's hook into Rails' Notifications API to store all queries Rails sends to the database and use a Rails' mountable engine to expose those queries through a web interface isolated from our application!

2. <https://github.com/plataformatec/responders>

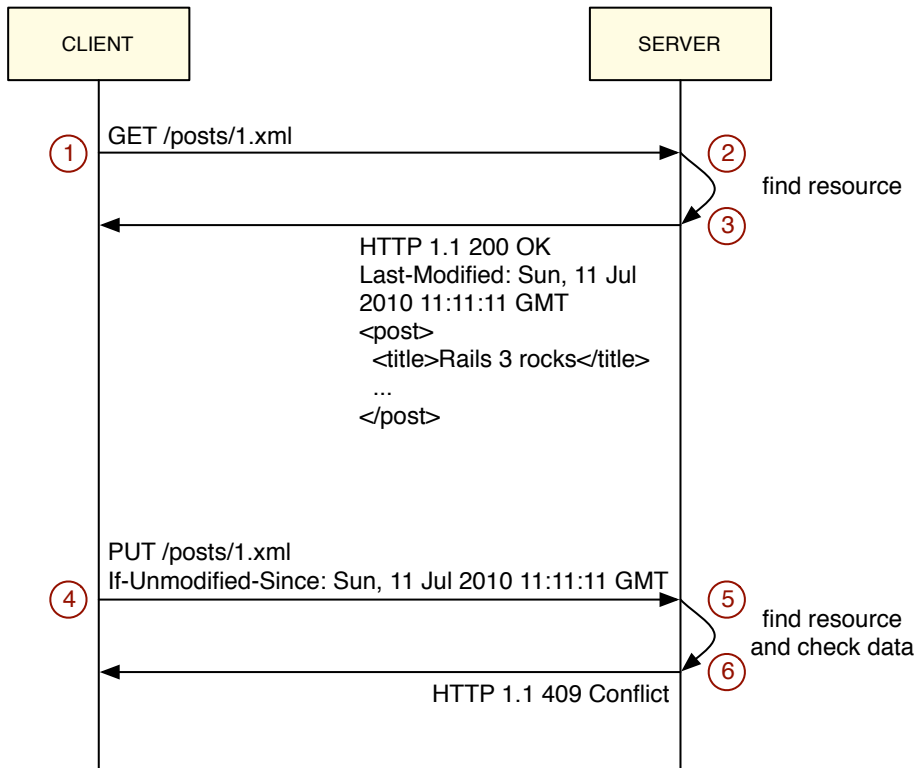


Figure 14—Client and server interaction with HTTP conditional requests

In this chapter, we'll see

- ActiveSupport::Notifications API
- Rails mountable and isolated engines
- Rack, middleware stacks, and custom middleware

CHAPTER 7

Managing Application Events with Mountable Engines

Since Ruby on Rails' early days, people have wondered what happens inside of their applications. How many queries were performed in this request? How long did this request take?

To address this common concern, a few open source projects (such as Footnotes¹ and Bullet²) and services (such as NewRelic's RPM³ and Scout⁴) were built. Since all those different tools needed to extract this information from Rails, Rails evolved to provide a centralized way to publish and subscribe to events happening inside an application with the ActiveSupport::Notifications API.

In this chapter, we'll use this API to subscribe to all actions processed by our application and store them in a MongoDB database. Then we will use a Rails engine to create a set of routes, controllers and views to navigate through the stored data. This engine can then be shared in between Rails applications and mounted at specific endpoints.

7.1 Mountable and Isolated Engines

In [Chapter 5, *Streaming Server Events to Clients Asynchronously*, on page 85](#), we created a Rails engine that streams data to our application. In addition to providing a controller, the engine added new routes to our application, and helper methods like `live_assets`. In a way, that engine was directly extending

1. <https://github.com/josevalim/rails-footnotes>

2. <https://github.com/flyerhzm/bullet>

3. <http://www.newrelic.com/features.html>

4. <http://scoutapp.com/>

our Rails application with its own components. However this behavior is not always desirable.

Take for instance the plugin we'll build in this chapter. It is going to provide its own models, controllers and views. As the plugin grows, the number of routes will start to multiply, as will the number of helper methods. If our plugin implements a `show_paginated_results()` helper and a Rails application uses our plugin, we don't want our helper to be available inside the Rails application, since the helper is internal to our plugin. Even worse, if the application has its own `show_paginated_results()` helper, it would override the helper defined in our plugin, which can lead to failures!

Rails solves those issues by providing mountable and isolated engines. A mountable engine uses its own router instead of adding routes directly to the application router. An isolated engine is built inside its own namespace, with its own models, controllers, views, assets and helpers. Let's generate our first mountable engine with the rails plugin command passing `--mountable` as an option:

```
$ rails plugin new mongo_metrics --mountable
```

The `--mountable` option generates a mountable and isolated engine. This can be observed by simply checking a couple of different files. First, let's open up the plugin's `config/routes.rb`:

```
mongo_metrics/1_engine/config/routes.rb
MongoMetrics::Engine.routes.draw do
end
```

Notice how the routes are drawn through the engine. Compare them to the `config/routes.rb` generated in rake [Section 5.1, Extending Rails with Engines, on page 86](#), which are drawn directly on the application:

```
live_assets/1_live/config/routes.rb
Rails.application.routes.draw do
  get "/live_assets/:action", to: "live_assets"
end
```

Since the routes are no longer added to the application, the engine needs to be explicitly mounted in the application router, which is done automatically by the rails plugin command in the test/dummy application:

```
mongo_metrics/1_engine/test/dummy/config/routes.rb
Rails.application.routes.draw do
  mount MongoMetrics::Engine => "/mongo_metrics"
end
```

That's all it takes to make our engine mountable. To have an isolated engine, we need to explicitly declare it as isolated and choose a namespace. The

--mountable option automatically generated an engine definition with MongoMetrics as the isolated namespace:

```

mongo_metrics/1_engine/lib/mongo_metrics/engine.rb
module MongoMetrics
  class Engine < ::Rails::Engine
    isolate_namespace MongoMetrics
  end
end

```

By declaring an isolated namespace, all of our controllers, models and helpers should be defined *inside* this namespace, guaranteeing they will be isolated from the rest of the application. Now helper files defined in the engine won't be automatically included by the application and vice-versa. It also sets up many conveniences, for example, if we were using Active Record, it would prefix all model table names with `mongo_metrics_` and ensure Rails generates namespaced models, controllers and helpers.

Besides those changes, the rails plugin command with the --mountable option generates a couple extra files, like assets manifests and a `MongoMetrics::ApplicationController` at `app/controllers/mongo_metrics/application_controller.rb`, making it closer to how a brand new Rails application would look .

With our plugin setup ready, it is time to explore Rails' ActiveSupport::Notifications API and start storing those notifications in the database.

7.2 Storing Notifications in the Database

Before we implement the logic to store notifications in the database, let's take a look at the Notifications API.

The Notifications API

The Notifications API consists of just two methods: `instrument()` and `subscribe()`. The former is called when we want to instrument and publish an event, and for Action Controller processing, it looks like this:

```

ActiveSupport::Notifications.instrument("process_action.action_controller",
  format: :html, path: "/", action: "index") do
  process_action("index")
end

```

The first argument is the name of the event published, which in this case is `process_action.action_controller`, and the second is a hash with information about the event, called payload. To subscribe to those notifications, we just need to pass the event name and a block to `subscribe()`:

```

event = "process_action.action_controller"

```

```
ActiveSupport::Notifications.subscribe(event) do |*args|
  # do something
end
```

where `args` is an array with five items:

- `name`: A String with the name of the event
- `started_at`: A Time object representing when the event started
- `ended_at`: A Time object representing when the event ended
- `instrumenter_id`: A String containing the unique ID of the object instrumenting the event
- `payload`: A Hash with the information given as payload to the `instrument()` method

And that's all we need to know. Next let's take a look at the database where we are going to store the notifications.

Using MongoDB

MongoDB is a fast and document-oriented database perfectly suited to storing notifications since it's high-volume, low-value data. You can read more about MongoDB at its website⁵, which also includes installation instructions for different operating systems.

Currently, there are several ORMs to interact with MongoDB, and we are going to use Mongoid⁶ for this project. We won't cover installation instructions, so if you don't have MongoDB installed, please do it now! After MongoDB is installed and running, let's add Mongoid as a dependency to our plugin:

```
mongo_metrics/1_engine/mongo_metrics.gemspec
s.add_dependency "mongoid", "~> 4.0.0"
```

Let's also generate the Mongoid configuration inside our test/dummy application:

```
$ rails g mongoid:config
```

And require mongoid as soon as we load up our metrics plugin:

```
mongo_metrics/1_engine/lib/mongo_metrics.rb
require "mongoid"
require "mongo_metrics/engine"
```

```
module MongoMetrics
end
```

5. <http://www.mongodb.org/>

6. <http://mongoid.org/>

With Mongoid configured, let's create our first model, properly namespaced at `app/models/mongo_metrics/metric.rb`:

```
mongo_metrics/1_engine/app/models/mongo_metrics/metric.rb
module MongoMetrics
  class Metric
    include Mongoid::Document
  end
end
```

Before writing any logic that actually stores documents in MongoDB, let's write a test in `test/mongo_metrics_test.rb` first. The test instruments an event with the name `process_action.action_controller` and asserts a metric was stored in MongoDB with all relevant fields:

```
mongo_metrics/1_engine/test/mongo_metrics_test.rb
require "test_helper"

class MongoMetricsTest < ActiveSupport::TestCase
  setup { MongoMetrics::Metric.delete_all }

  test "process_action notification is saved in the mongo database" do
    event = "process_action.action_controller"
    payload = { "path" => "/" }

    ActiveSupport::Notifications.instrument event, payload do
      sleep(0.001) # simulate work
    end

    metric = MongoMetrics::Metric.first
    assert_equal 1, MongoMetrics::Metric.count
    assert_equal event, metric.name
    assert_equal "/", metric.payload["path"]

    assert metric.duration
    assert metric.instrumenter_id
    assert metric.started_at
    assert metric.created_at
  end
end
```

When we run the test, it fails since we are not storing anything yet:

```
1) Failure:
test_process_action_notification_is_saved_in_the_mongo_database(MongoMetricsTest)
Expected: 1
Actual: 0
```

To make the test pass, let's first subscribe to `ActiveSupport::Notifications` at the end of `lib/mongo_metrics.rb`:

```

mongo_metrics/2_metrics/lib/mongo_metrics.rb
require "mongoid"
require "mongo_metrics/engine"
require "active_support/notifications"

module MongoMetrics
  EVENT = "process_action.action_controller"

  ActiveSupport::Notifications.subscribe EVENT do |*args|
    MongoMetrics::Metric.store!(args)
  end
end

```

Our notification hook is simply calling the `store!()` method in our `MongoMetrics::Metric`, which will be responsible for parsing the arguments and creating a record in the database as follows:

```

mongo_metrics/2_metrics/app/models/mongo_metrics/metric.rb
module MongoMetrics
  class Metric
    include Mongoid::Document

    field :name, type: String
    field :duration, type: Integer
    field :instrumenter_id, type: String
    field :payload, type: Hash
    field :started_at, type: DateTime
    field :created_at, type: DateTime

    def self.store!(args)
      metric = new
      metric.parse(args)
      metric.save!
    end

    def parse(args)
      self.name           = args[0]
      self.started_at     = args[1]
      self.duration       = (args[2] - args[1]) * 1000000
      self.instrumenter_id = args[3]
      self.payload        = args[4]
      self.created_at     = Time.now.utc
    end
  end
end

```

After this change, our test suite is green! To take a look at how our plugin works outside the test environment, let's create a controller `HomeController` with three actions inside `test/dummy` and then boot the dummy application:

```
$ rails g controller Home foo bar baz
$ rails s
```

Now make a few requests over /home/foo, /home/bar and /home/baz to generate some data. When you are done, start a new console session with rails console and see all notifications created by these requests by typing `MongoMetrics::Metric.all.to_a` in the command line.

Although our subscriber works as expected, wouldn't it be nice if we had a page where we could see these notifications instead of using the Rails console? So let's harness the power provided by Rails engines once again!

The Notifications Page

To create our notifications page, let's create a controller, a view, and routes inside our engine. Let's start with the controller:

```
mongo_metrics/2_metrics/app/controllers/mongo_metrics/metrics_controller.rb
```

```
module MongoMetrics
  class MetricsController < ApplicationController
    respond_to :html, :json

    def index
      @metrics = Metric.all
      respond_with(@metrics)
    end

    def destroy
      @metric = Metric.find(params[:id])
      @metric.destroy
      respond_with(@metric)
    end
  end
end
```

Our controller has two actions: `index()` and `destroy()`. For the first one, we need to create a view:

```
mongo_metrics/2_metrics/app/views/mongo_metrics/metrics/index.html.erb
```

```
<h1>Listing Metrics</h1>
```

```
<table>
  <tr>
    <th>Name</th>
    <th>Duration</th>
    <th>Started at</th>
    <th>Payload</th>
    <th></th>
  </tr>
  <%= content_tag_for :tr, @metrics do |metric| %>
```

```

<td><%= metric.name %></td>
<td><%= metric.duration / 1000 %>ms</td>
<td><%= time_ago_in_words metric.started_at %> ago</td>
<td>
  <ul>
    <% metric.payload.each do |k, v| %>
      <li><%= k.humanize %>: <%= v %></li>
    <% end %>
  </ul>
</td>
<td><%= link_to 'Destroy', metric_path(metric),
  method: :delete, data: { confirm: 'Are you sure?' } %></td>
<% end %>
</table>

```

For the destroy links in our view to work in the browser, we need to add the The code tag should not be here. gem as a dependency:

```

mongo_metrics/2_metrics/mongo_metrics.gemspec
s.add_dependency "jquery-rails", "~> 3.0.1"

```

And require both The code tag should not be here. and The code tag should not be here. in the javascript manifest file:

```

mongo_metrics/2_metrics/app/assets/javascripts/mongo_metrics/application.js
//= require jquery
//= require jquery_ujs
//= require_tree .

```

Finally, let's add some routes at config/routes.rb:

```

mongo_metrics/2_metrics/config/routes.rb
MongoMetrics::Engine.routes.draw do
  root to: "metrics#index"
  resources :metrics, only: [:index, :destroy]
end

```

Notice how we declared the routes without worrying about namespaces. Since our engine is mountable and isolated, Rails saves us the trouble from specifying the namespace on every route. Furthermore, in our view, we simply called `metric_path()` and Rails automatically looked up this route in the engine router and not in the application one. Even in the application had a `metric_path()` named route, they wouldn't conflict!

However, this begs the question: what if we want to access an application route from the engine or an engine route from the application?

To illustrate the issue, we are going to write a couple integration tests. Our tests need to access a couple pages in the dummy application and then go

over our plugin pages to guarantee the notifications are being exhibited properly:

```
mongo_metrics/2_metrics/test/integration/navigation_test.rb
require "test_helper"

class NavigationTest < ActionDispatch::IntegrationTest
  setup { MongoMetrics::Metric.delete_all }

  test "can visualize notifications" do
    get main_app.home_foo_path
    get main_app.home_bar_path
    get main_app.home_baz_path

    get mongo_metrics.root_path
    assert_match "Path: /home/foo", response.body
    assert_match "Path: /home/bar", response.body
    assert_match "Path: /home/baz", response.body
  end

  test "can destroy notifications" do
    get main_app.home_foo_path
    metric = MongoMetrics::Metric.first

    delete mongo_metrics.metric_path(metric)
    assert_empty MongoMetrics::Metric.where(id: metric.id)
  end
end
```

Our new tests should pass right off the bat. Notice that every time we want to access an application page, we use the `main_app()` proxy and the `mongo_metrics()` proxy for the mountable engine pages. The Rails application will always be accessible through `main_app()`, the name of the engine one can be found by running `rake routes` in the dummy application:

Prefix	Verb	URI Pattern	Controller#Action
home_foo	GET	/home/foo(.:format)	home#foo
home_bar	GET	/home/bar(.:format)	home#bar
home_baz	GET	/home/baz(.:format)	home#baz
mongo_metrics		/mongo_metrics	MongoMetrics::Engine


```
Routes for MongoMetrics::Engine:
  root GET    /                  mongo_metrics/metrics#index
  metrics GET   /metrics(.:format) mongo_metrics/metrics#index
  metric DELETE /metrics/:id(.:format) mongo_metrics/metrics#destroy
```

The name of the proxy is under the Prefix column. Notice `rake routes` conveniently shows the routes of the mounted engine as well! If desired, you can also boot up the dummy app and access the notifications page in your browser.

Now that we are properly storing and exhibiting information from MongoDB, we can notice that every time we access our engine pages, it also stores data in MongoDB. It would be practical if we turned off the metrics storage every time we access the plugin itself. To solve this problem, let's provide a mechanism to mute notifications in some selected places. In order to implement it, we need to understand how Rails integrates with Rack.

7.3 Rails and Rack

Rack provides a minimal, modular, and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distills the API for web servers, web frameworks, and software in between (the so-called middleware) into a single method call.

—Rack documentation⁷

Rails applications need a web server in order to interact through the HTTP protocol. And since the early days, the Rails community saw a huge range of web servers available to deploy their applications.

At the time, Rails was responsible for providing an adapter to each different web server it supported: one for Mongrel, another for WEBrick, another for Thin, and so on. Similarly, other frameworks had to provide different adapters for the same web servers since they had a different API than Rails.

This quickly proved to be a duplication of efforts, and at the beginning of 2007, Rack was released with the goal of unifying the APIs used by web servers and web frameworks. By following the Rack API, a web framework could use Rack web servers adapters instead of providing its own, removing the duplication of effort existing in the Ruby community.

While Rails 2.2 already provided a simple Rack interface, Rails more closely embraced Rack and its API in version 2.3. However, the Rack revolution really happened in Rails 3, where several parts of Rails became Rack end points, and you could easily mount different Rack applications in the same process. For example, we can easily mount a Sinatra application inside the Rails router, similar to how we mounted an engine inside the dummy application in this chapter.

Hello, Rack!

The Rack specification clearly outlines the API used by Rack applications to communicate with a web server and between themselves:

7. <http://rack.rubyforge.org/doc/>

A Rack application is any Ruby object that responds to `call`. It takes exactly one argument, the environment, and returns an array of exactly three values: the status, the headers, and the body.

—Rack specification⁸

Rack's minimal API allows us to write a simple web application in just a few lines of code:

```
require 'rack'

class HelloRack
  def call(env)
    [200, { 'Content-Type' => 'text/html' }, ['Hello Rack!']]
  end
end

run HelloRack.new
```

By creating the previous `config.ru` file in a directory and invoking the `rackup` command inside this same directory, Rack starts a web server and invokes our `HelloRack` application in each request. When you fire up a browser and hit `http://localhost:9292/`, you can see “Hello Rack!” returned as the response body.

All Rails applications ship with a `config.ru` file, as we can see in the dummy application inside `test/dummy` with the following contents:

```
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment', __FILE__)
run Rails.application
```

In other words, every Rails application is also a Rack application: it implements the `call()` method which receives the environment and returns an array with three elements. By default, its implementation sends the request to the application router (the one defined in `config/routes.rb`) which dispatches the request to another Rack application if any route matches. Let's explore further how it works.

Understanding the Rails Router

The Rails router can dispatch to any Rack application:

```
Rails.application.routes.draw do
  match "/hello", to:
    lambda { |env| [200, { "Content-Type" => "text/plain" }, ["World"]] }
```

8. <http://rack.rubyforge.org/doc/SPEC.html>

end

When we add this route to any Rails application and hit `/hello` in the browser, we get “World” as the response from the server. In fact, when we have a route like this:

```
Rails.application.routes.draw do
  match "/hello", to: "posts#index"
end
```

Rails, before a request, automatically converts `controller#action` to a Rack application. You can retrieve any action from a controller as a Rack application by simply doing this:

```
PostsController.action(:index)
PostsController.action(:index).responds_to?(:call) # => true
```

Whenever you call `get()`, `post()`, `put()`, `delete()`, `resources()`, or `resource()` in the router DSL, at the end they always call the `match()` method. The only method that has different semantics is `mount()`, used in the dummy application to mount our engine.

The `match()` method works by matching the full route. If we think in terms of regular expressions, when we say `match "/mongo_metrics"`, it only matches paths as `%r"\A/mongo_metrics\b"` (the query string is not considered in matches). However, when mounting an engine or any other Rack application, we don't match just requests to `/mongo_metrics` but also to `/mongo_metrics/metrics`, `/mongo_metrics/other`, and so on. In other words, the equivalent regular expression would be `%r"\A/mongo_metrics"`, without the `\b` anchor.

You may have noticed there is something more going on. In order to access the mounted engine, we issue requests to `/mongo_metrics/metrics` but the engine router matches only on `/metrics`, ignoring the `/mongo_metrics` prefix. How so?

Whenever a request hits a Rack server, the server gets the request path, stores it in the environment hash as `env["PATH_INFO"]`, and pass it down to the underlying Rack application. Mounting works because, when dispatching to a mounted engine, Rails removes `/mongo_metrics` from `env["PATH_INFO"]`, so the engine sees only `/metrics` (as if a browser was accessing `/metrics` straight in the engine).

This mechanism works not only with engines but with any Rack application, since this is outlined in the Rack specification itself. The specification also dictates that Rails should also set `env["SCRIPT_NAME"] = "/mongo_metrics"` before calling the mounted engine, in order to tell the engine it is mounted in a specific point, allowing the engine to still generate full urls.

To sum it up, we have a Rails application, which is a Rack application, that invokes our router, which is another Rack application, that finally dispatches to another Rack application, may it be a controller and action, an engine or even a Sinatra application. It is Rack applications all the way down! And to make things even more interesting, Rack provides the concept of middleware, which allows us to add custom code in between those Rack applications, giving us even more flexibility. That's what we will explore next.

7.4 Middleware Stacks

Although web server adapters and the Rack application API revolutionized the way that Ruby web frameworks are developed, you are probably more familiar with another term related to Rack: *middleware*.

A middleware wraps around a Rack application. It lets us manipulate both the request sent down to the application and the response returned by the application. By piling up many middleware before an application, we create a so-called *middleware stack*. Any request to an action in a Rails controller passes at least through three middleware stacks, as shown in the following figure:

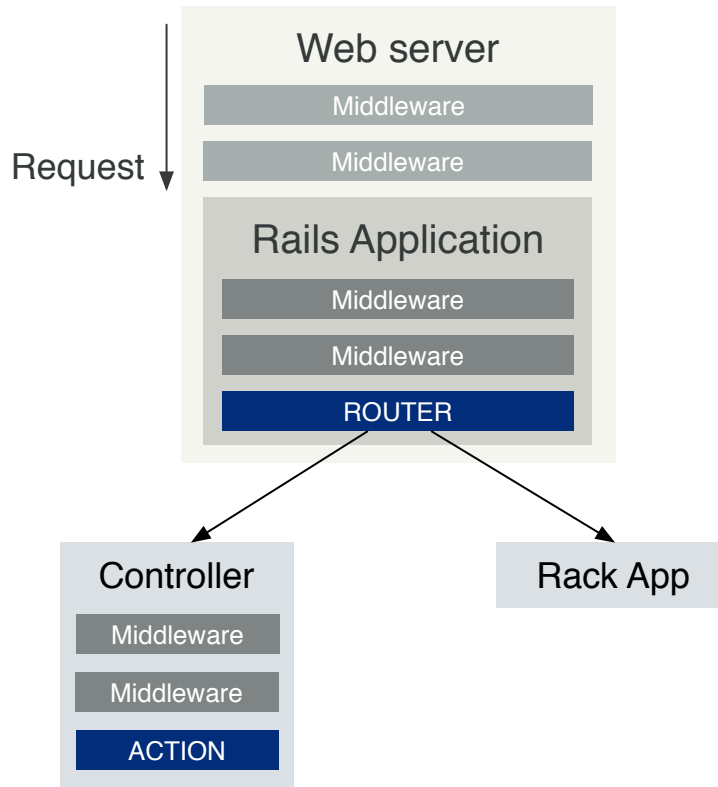


Figure 15—All the middleware stacks involved in a request to an action in a Rails controller

The first of these three middleware stacks is hidden by Rails. It sits between the web server and the `Rails::Application` object and contains only two middleware components:

- `Rails::Rack::LogTailer`: It parses the log file and prints it on console
- `Rails::Rack::Debugger`: It requires and enables debugger

After passing through this middleware stack, the request hits a `Rails::Application` (for example our `Dummy::Application` inside `test/dummy`), which is nothing more than another middleware stack with the router sitting at the end.

The stack contained by the `Rails::Application` is the most known middleware stack in Rails. You can add or remove middleware from this stack through `config.middleware`s available inside `config/application.rb`. To see all available middleware in the stack, we need to invoke `rake middleware` from the command line at the application root.

For our dummy application inside test/dummy, this command returns the following:

```
use ActionDispatch::Static
use Rack::Lock
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007fed3d5eddf0>
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::EncryptedCookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
use Rack::Mongoid::Middleware::IdentityMap
run Dummy::Application.routes
```

The middleware stack will change depending on your dependencies and your Rails environment. For example, Rack::Mongoid::Middleware::IdentityMap is a Mongoid dependency while ActionDispatch::Reloader is usually available only in development. Let's give this list a rundown:

- ActionDispatch::Static: Serves public assets in development
- Rack::Lock: Wraps a Mutex around the application so just one thread can access it at the same time. When we set config.allow_concurrency to true in [Section 5.3, File system notifications with Threads, on page 95](#), we instructed Rails to remove this middleware
- ActiveSupport::Cache::Strategy::LocalCache: Uses an in-memory cache store to provide a local cache during requests
- Rack::Runtime: Measures the request time and returns it as an X-Runtime header
- Rack::MethodOverride: Checks POST requests and converts them to PUT or DELETE if _method is present in parameters

- `ActionDispatch::RequestId`: Sets the request id information which is shown in logs
- `Rails::Rack::Logger`: Logs each request
- `ActionDispatch::DebugExceptions` and `ActionDispatch::ShowExceptions`: Is responsible for showing helpful error pages in development and rendering status pages in production from the public directory
- `ActionDispatch::RemoteIp`: Handles IP spoof checking
- `ActionDispatch::Callbacks` and `ActionDispatch::Reloader`: Runs the so-called `to_prepare` callbacks, which happens once on application boot in production and before each request in development (such as I18n and routes reloading)

The next three middleware are all related to Active Record followed by middleware responsible for managing cookies, session and flash messages. The last few middleware are:

- `ActionDispatch::ParamsParser`: Parses the parameters given in the request, both from query string or in the POST body
- `Rack::Head`: Converts HEAD requests to GET requests
- `Rack::ConditionalGet`: Returns a 304 status code in case the appropriate HTTP Cache headers match
- `Rack::ETag`: Calculate the MD5 of the response body and sets it as the ETag header

The last stop in the stack is the application router, which is yet another Rack application. If the router dispatches the request to a specific action in a controller, it will also pass through another middleware stack since each controller also has its own middleware stack. We can add a middleware to a controller as follow:

```
class UsersController < ApplicationController
  use MyMiddleware
  use AnotherMiddleware
end
```

The controller middleware are invoked before the action is processed and before any filters, so they sit quite high in the controller stack. It seems that Rails provides a lot of flexibility for developers to hook their own middleware, which make them a good choice to implement the mechanism that turns off our metrics store, so let's write our first middleware!

MuteMiddleware

Before writing our middleware, we need to ensure MongoMetrics provides a method to mute notifications for a specific block of code. Let's call this method `mute!()` and also provide a `mute?()` that returns true whenever notifications are muted. Let's write a test:

```
mongo_metrics/3_final/test/mongo_metrics_test.rb
test 'can ignore notifications when specified' do
  MongoMetrics.mute! do
    assert MongoMetrics.mute?

    event = "process_action.action_controller"
    ActiveSupport::Notifications.instrument event do
      sleep(0.001) # simulate work
    end
  end

  assert !MongoMetrics.mute?
  assert_equal 0, MongoMetrics::Metric.count
end
```

To make our test pass, let's implement these two methods and change our subscribe block to respect the mute condition:

```
mongo_metrics/3_final/lib/mongo_metrics.rb
require "mongoid"
require "mongo_metrics/engine"
require "active_support/notifications"

module MongoMetrics
  EVENT = "process_action.action_controller"

  ActiveSupport::Notifications.subscribe EVENT do |*args|
    MongoMetrics::Metric.store!(args) unless mute?
  end

  def self.mute!
    Thread.current["sql_metrics.mute"] = true
    yield
  ensure
    Thread.current["sql_metrics.mute"] = false
  end

  def self.mute?
    Thread.current["sql_metrics.mute"] || false
  end
end
```


Notice we used thread variables to ensure that muting a request in a thread is not going to affect other threads in a threaded environment. Also, we need to wrap the `yield()` call in an `ensure` block, allowing the mute status to be reverted even if an exception happens while executing the block. Let's write a test for it too:

```
mongo_metrics/3_final/test/mongo_metrics_test.rb
test 'does not leak mute state on failures' do
  MongoMetrics.mute! do
    assert MongoMetrics.mute?
    raise "oops"
  end rescue nil

  assert !MongoMetrics.mute?
end
```

With our `mute!()` implementation ready, we are ready to write the `MuteMiddleware`. In order to guarantee our middleware works as expected, let's write an integration test that asserts that accessing our metrics controller does not generate any event:

```
mongo_metrics/3_final/test/integration/navigation_test.rb
test "does not log engine actions" do
  get mongo_metrics.root_path
  assert 0, MongoMetrics::Metric.count
end
```

Any Rack middleware is initialized with the application or the middleware it should call next in the stack. Whenever our `MuteMiddleware` middleware is invoked, all it has to do is invoke the underlying application inside a `mute!()` block, effectively muting everything happening downstream. Let's implement it:

```
mongo_metrics/3_final/lib/mongo_metrics/mute_middleware.rb
module MongoMetrics
  class MuteMiddleware
    def initialize(app)
      @app = app
    end

    def call(env)
      MongoMetrics.mute! { @app.call(env) }
    end
  end
end
```

Since a usual Rails request goes through three different middleware stacks, we need to evaluate the best place to use our middleware:

- a stack in between the web server and the Rails application, which we cannot access so it is out of option
- a stack inside the application that finishes with the router, which is not appropriate since adding our middleware to this stack would mute *all* requests
- a stack inside each controller, which is our best option so far since we can add it directly to `MongoMetrics::ApplicationController`

However, when a request goes to an engine, there is another middleware stack sitting in between the application router and the engine router, which is the engine's own middleware stack, as shown in [Figure 16, All the middleware stacks involved in a request that includes a Rails engine, on page 156](#)

This middleware stack is similar to a Rails application stack but empty by default. If we add our middleware to the engine stack, any request that goes to our metrics plugin, regardless of the controller, will be automatically muted. That seems handy, let's add our middleware to this stack:

```
mongo_metrics/3_final/lib/mongo_metrics/engine.rb
require "mongo_metrics/mute_middleware"
```

```
module MongoMetrics
  class Engine < ::Rails::Engine
    isolate_namespace MongoMetrics
    config.middleware.use MuteMiddleware
  end
end
```

We can configure the engine middleware stack via `config.middleware`. If we were inside a Rails application, we would have used `config.middleware` too. `config.middleware` always point to the middleware stack of the current engine or application. In case we wanted to modify the application middleware stack from the engine, we could access it via `config.app_middleware`.

After those changes, our tests will be green once again. By using a middleware, we can easily disable metrics storage for specific sections of our plugins and applications. In particular, in case you want to disable it for a specific controller in your application, or even for specific actions, it can be achieved as follow:

```
class AdminController < ApplicationController
  # You could also use :only and :except options.
  # use MuteControllerMiddleware, only: :index
  use MuteControllerMiddleware
end
```

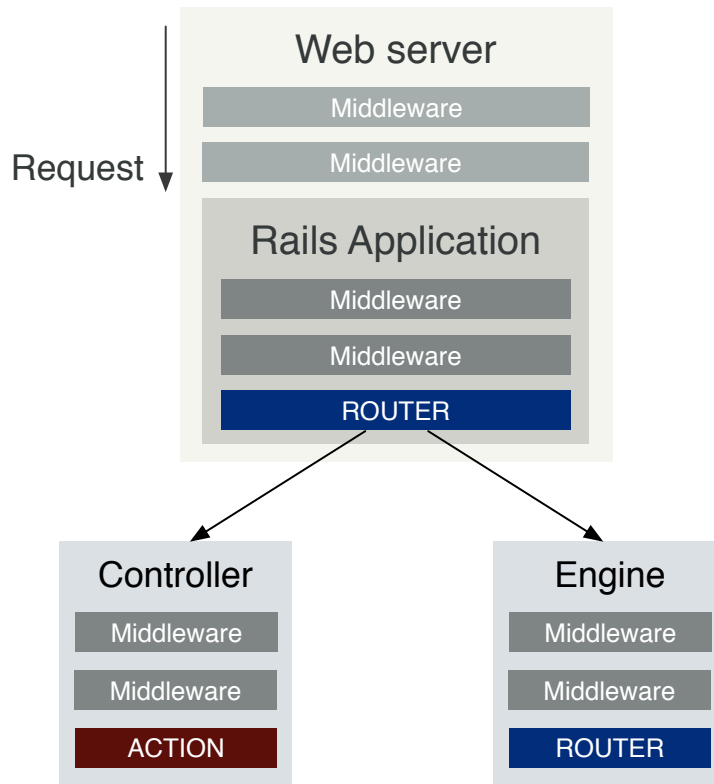


Figure 16—All the middleware stacks involved in a request that includes a Rails engine

Every time a Rack application is involved, it is particularly trivial to add a middleware stack around it and Rails does it often in order to provide different points for us to extend our applications.

Trimming down the middleware stack

When we executed `rake middleware` in the dummy application, it contained a bunch of middleware related to Active Record. In fact, our plugin has a dependency on the rails gem, which depends on Active Record and other gems, like Action Mailer, that we are not using. So let's break our rails dependency in the gemspec into two smaller ones:

```

mongo_metrics/3_final/mongo_metrics.gemspec
s.add_dependency "railties", "~> 4.0.0.rc2"
s.add_dependency "sprockets-rails", "~> 2.0.0.rc3"

```

The rails gem is a "meta gem". It doesn't ship with any code, instead it simply contains Rails' default stack as a dependency, that's why it depends on Active

Record, Action Mailer, Sprockets and more. In our case, we have replaced the rails gem by the railties gem, which is Rails' backbone containing the booting processes, railties, engine and application definitions, and the sprockets-rails, which allows us to serve our assets. The railties gem already depends on actionpack (for routes, controllers and views) and on activesupport (Ruby extensions), so we don't need to add them explicitly.

After doing those changes, our application will no longer boot since our application configuration files were generated expecting some of the removed dependencies to be present. That said, make sure to remove any `config.active_record` and `config.action_mailer` from the environment files at `test/dummy/config/environments`. Also, check your test files to see we don't have any call related to fixtures, like `fixtures :all` in the `test/integration/navigation_test.rb`.

If you run `rake middleware` once again, Active Record middleware will no longer be in our stack. The process we have just done is extremely similar to how we would have trimmed down dependencies in a Rails application. The main step is to replace the rails gem in the Gemfile by its smaller dependencies and then clean up the application configuration files. In fact, some components like Sprockets and Active Record can even be removed when generating the application or the plugin by simply passing `--skip-sprockets` or `--skip-active-record` when invoking `rails new` or `rails plugin new`.

7.5 Streaming with Rack

Although we could improve our metrics application to show beautiful graphics and reports, there are many tools and web services available out there that could do the hard work for us, as long as we give them the metrics we stored in the database.

The default approach to send data from the server to the client in Rails is by using the `send_data()` method. However, this approach requires us to generate the whole string we want to send upfront, which may take time and require a lot of memory as we store more metrics in our database. To solve this problem, we are going to stream data, allowing us to send data to the client in chunks, without allocating a huge blob in memory.

In [Chapter 5, Streaming Server Events to Clients Asynchronously, on page 85](#), we used Rails' live streaming facilities to stream data from the server. This time however we'll rely solely on the flexibility given by the Rack specification to implement this feature. The streaming format of choice is going to be CSV, since a client can consume the data row-by-row, and because Ruby has built-in tools to convert data to CSV.

Streaming redux

The Rack specification outlines that a valid response body is any Ruby object that responds to the method `each()`. That's why we commonly use arrays in Rack applications to hold the response body:

```
class HelloRack
  def call(env)
    [200, { 'Content-Type' => 'text/html' }, ['Hello Rack!']]
  end
end
```

The Rack web server is going to loop over the response body, using the `each()` method, and outputting the data yielded. Since the array responds to `each()`, it is going to yield once with value “Hello Rack!”.

This API contract means we can implement streaming in Rack by simply having a custom response body that implements a custom iteration mechanism via `each()`. Here's how we could implement streaming using only Rack:

```
require 'rack'
```

```
class StreamingRack
  def call(env)
    [200, { 'Content-Type' => 'text/html' }, self]
  end
  def each
    while true
      yield 'Hello Rack!\n'
    end
  end
end
```

```
run StreamingRack.new
```

Write the previous code to a `config.ru` file and then run `rackup -s puma` to boot the Rack application. Let's once again use `cURL`⁹ through the command line and see data streaming through:

```
$ curl -v localhost:9292/
```

And that is it! Streaming data with Rack is quite accessible. Hit `CTRL+C` to shutdown the Rack application and the `cURL` process.

9. <http://curl.haxx.se/>

CSV Streamer

Let's start our CSV Streamer implementation by writing a test that accesses the metrics path with .csv and get CSV document back:

```
mongo_metrics/3_final/test/integration/navigation_test.rb
```

```
test "exports data to csv" do
  get main_app.home_foo_path

  get mongo_metrics.metrics_path(format: :csv)
  assert_match "process_action.action_controller", response.body
end
```

As the route which handles the CSV request is already defined, we just need to change the MongoMetrics::MetricsController controller to respond to CSV requests on the index action. As our index() already uses respond_with(), the following should be enough to get a CSV response back:

```
mongo_metrics/3_final/app/controllers/mongo_metrics/metrics_controller.rb
```

```
module MongoMetrics
  class MetricsController < ApplicationController
    respond_to :html, :json
    respond_to :csv, only: :index
  end
end
```

However, if we recall the discussion from [Section 6.2, Exploring ActionController::Responder, on page 114](#), Rails will first try to render a template, which does not exist, and then fallback to calling render csv: @metrics. The issue here is that Rails does not implement a CSV renderer, so we need to write it ourselves. Luckily, we have already learned how to write renderers in [Chapter 1, Creating Our Own Renderer, on page 1](#).

We want our CSV renderer to send a file back to the user, streaming each chunk little by little. The renderer should also set the response body to be a custom object, which we will call CSVStreamer. Let's implement it:

```
mongo_metrics/3_final/lib/mongo_metrics/csv_streamer.rb
```

```
module MongoMetrics
  ActionController::Renderers.add :csv do |model, options|
    headers = self.response.headers
    headers["Content-Disposition"] =
      %(attachment; filename="#{controller_name}.csv")
    headers["Cache-Control"] = "no-cache"
    headers.delete "Content-Length"
    self.content_type ||= Mime::CSV
    self.response_body = CSVStreamer.new(model)
  end

  class CSVStreamer
    def initialize(scope)

```

```

    @scope = scope
  end

  def each
    @scope.each do |record|
      yield record.to_csv
    end
  end
end
end
end

```

First, we set the “Content-Disposition” header to tell the browser we are sending an attachment. Second, we turn off the cache and remove the “Content-Length”, notifying the underlying Rack server we want to stream the data. Finally, we assign the proper content type and set the response body to our custom CSVStreamer object.

Notice our streamer object calls `to_csv()` on each metrics record, so we need to implement it too. Since Ruby provides a library named `csv` to generate and consume CSV in its standard library, let’s require it and use it to convert an array with metrics information to csv:

```

mongo_metrics/3_final/app/models/mongo_metrics/metric.rb
require "csv"

```

```

def to_csv
  [name, started_at, duration, instrumenter_id, created_at].to_csv
end

```

The last step to make our test pass is to require `mongo_metrics/csv_streamer`, the file where we defined our custom CSV renderer, making it available to our controllers:

```

mongo_metrics/3_final/lib/mongo_metrics.rb
require "mongo_metrics/csv_streamer"

```

Our new test now passes! Although the implementation of CSV streaming with Rack is a bit different of how we would implement it using `ActionController::Live`, it suffers from the same limitations and requires the same care when deployed. This time we were able to test our endpoint, differently from what we have built on [Chapter 5, Streaming Server Events to Clients Asynchronously, on page 85](#), but it was only possible because we were not streaming infinitely. With Rack, you could also stream indefinitely, and the same testing difficulties would arise.

That said, which option should we choose to implement streaming: `ActionController::Live` or Rack streaming? In general, live streaming should be preferred.

Because Rack streaming relies on Rack's basic API, a web server does not know upfront which kind of response it has, long (streaming) one or a short (regular) one. This makes it hard for web servers to optimize such scenarios. Live streaming, on the other hand, is a Rails abstraction and, as Rack provides better and dedicated streaming facilities, Rails can adapt accordingly, transparently from the developer code. In any case, streaming with Rack may be convenient for smaller Rack applications that do not have access to all the conveniences provided by Rails.

Finally, this particular feature could even be implemented without streaming, simply by using one of the many background processing tools available to Rails, like Delayed Job¹⁰ and Resque¹¹, to generate the CSV file in the background and send an e-mail once it is done.

7.6 Wrapping Up

In this chapter, we developed a Rails engine that listens to all actions processed by an application and stores them in MongoDB. These notifications can be seen by accessing `/mongo_metrics` in the browser.

Our implementation was built atop a mountable and isolated engine, which allows us to build our features isolated from a Rails application, guaranteeing we won't have conflicts on the most likely places like routes and helpers. At the end, we have explored Rack by creating a middleware that allows us to mute the metrics storage on particular areas of our plugin and by using a custom Rack body to stream data.

Finally, there is still a lot to be done in our plugin when it comes to the visualization part. We could, for instance, allow the developer to sort these metrics by duration and provide charts. Even more interestingly, we could use the streaming techniques we have learned so far to stream events to the visualization page as they happen.

Next, let's create a Rails application that allows us to translate I18n messages through a Sinatra app authenticated with Devise, a popular authentication library.

10. https://github.com/collectiveidea/delayed_job

11. <https://github.com/resque/resque>

- In this chapter, we'll see
- The I18n framework
 - The Sinatra web framework
 - The Rails router
 - Devise (for authentication) and Capybara (for integration testing) gems

CHAPTER 8

Translating Applications Using Key-Value Backends

The internationalization framework (I18n) added in Rails 2.2 definitely played a key role in increasing Rails adoption around the world. Although we can easily make an application available in different languages, the biggest issue is maintaining this translation data. Some companies have a team of translators available, while others choose a collaborative approach and allow their own users to translate the web app. In both cases, it is common to develop a web interface to aid with the translation process.

By default I18n stores translation data into YAML files, which can be difficult to manipulate through the web interface. In fact, using YAML would require a mechanism to tell all servers to sync and reload these YAML files once they are updated. As you can imagine, such a solution could grow in complexity quickly.

Luckily, the I18n framework comes with different backends that allow us to store translations in other places than YAML files. This makes it much easier to manipulate the translations table through a web interface and update the site translations on demand. There is no need to synchronize YAML files between web servers. On the downside, retrieving translations from the database instead of an in-memory hash causes a huge impact in performance.

A solution that can comply with both these requirements (simplicity and performance) is a key-value store. In this chapter, we will store translations in a Redis store and use a Key Value backend to retrieve them. Additionally,

we will build a simple Sinatra¹ application to expose a web interface to read, create, and update these translations on the fly.

Unlike previous chapters, we'll develop all this functionality as a Rails application instead of a plugin. After studying and analyzing Railties and engines, we can now build Rails applications with a different perspective.

8.1 Revisiting Rails::Application

In previous chapters, we discussed Rails::Engine and how it exhibits several behaviors similar to a Rails application. When we look at the Rails source code, we find the following:

```
module Rails
  class Application < Engine
    # ...
  end
end
```

The Rails::Application class inherits from Rails::Engine! This means an application can do everything an engine does, plus some specific behavior:

- An application is responsible for all bootstrapping (for example, loading Active Support, setting up load paths, configuring the logger, and so on).
- An application has its own router and middleware stack (as we discussed in [Section 7.4, Middleware Stacks, on page 149](#)).
- An application should load and initialize all plugins.
- An application is responsible for reloading code and routes between requests if they changed.
- An application is responsible for loading tasks and generators when appropriate.

To take a closer look at these responsibilities, let's start developing our Translator app:

```
$ rails new translator
```

When we studied the dummy application in [Chapter 1, Creating Our Own Renderer, on page 1](#), we discussed the responsibilities of the config/boot.rb, config/application.rb and config/environment.tb files. In particular, the boot file is responsible for setting our load paths, the application file defines our Rails

1. <http://www.sinatrarb.com/>

application while the environment file finally initializes the app by calling the `initialize!()` method:

```
translator/1_app/config/environment.rb
# Load the rails application.
require File.expand_path('../application', __FILE__)

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

In [Initializers, on page 88](#), we showed how engines provide a set of initializers that drive how the engine boots. It is no surprise that a Rails application provides such initializers too:

```
module Translator
  class Application < Rails::Application
    initializer "translator.say_hello" do
      puts "hello on initialization"
    end
  end
end
```

To see all initializers available in a Rails application, open up a Rails console on our newly generated app and type:

```
Rails.application.initializers.map(&:name)
```

The difference here is that the application does not contain only its own initializers, but the initializers defined on all railties and engines. To initialize a Rails application is just a matter of executing those initializers one by one.

Everything else in a Rails application is built around those three main files. If we open up the Rakefile, we will see the following:

```
translator/1_app/Rakefile
require File.expand_path('../config/application', __FILE__)
Translator::Application.load_tasks
```

First, the application file is required, defining the Rails application. Next, `load_tasks()` is invoked, loading all Rake tasks provided by the application, plugins and by Rails itself. Note that we don't require the environment file at any point. This allows basic rake commands to run fast since they don't initialize the application, just define it.

However, there are many tasks that need the application to be initialized. For example, `rake db:migrate` only works if the database is configured. That's why Rails provides a Rake task called `:environment` and all this task does is to require `config/environment.rb` to initialize our application. Whenever you need to access

the database or any of your application classes in a Rake task, you need to depend on the `:environment` task.

Finally, let's look at `config.ru` in the root of our application. It simply requires the environment file, effectively initializing the application, and runs the current Rails application as a Rack application:

```
translator/1_app/config.ru
```

```
# This file is used by Rack-based servers to start the application.
```

```
require ::File.expand_path('../config/environment', __FILE__)
run Rails.application
```

Even though the application initialization process is broken into many files, the only reason for such is because we need to hook at different points. The `config.ru` file needs the whole environment up-front, while Rakefile loads it in steps. However, at the end of the day, there is nothing stopping us from merging all these files together into a single file Rails application!

The single-file Rails application

Bundling a Rails application into a single file really helps us understand how to setup and initialize Rails. Let's take a look at a sample single-file Rails application and discuss it next. Create an empty directory and add a `config.ru` file with the following contents:

```
translator/config.ru
```

```
# We will simply use rubygems to set our load paths
```

```
require "rubygems"
```

```
# Require our dependencies
```

```
require "rails"
```

```
require "active_support/railtie"
```

```
require "action_dispatch/railtie"
```

```
require "action_controller/railtie"
```

```
class SingleFile < Rails::Application
```

```
  # Set up production configuration
```

```
  config.eager_load = true
```

```
  config.cache_classes = true
```

```
  # A key base is required for our app to boot
```

```
  config.secret_key_base = "pa34u13hsleuowilaisejkez12u39201pluaep2ejlwhkj"
```

```
  # Define a basic route
```

```
  routes.append do
```

```
    root to: lambda { |env|
```

```
      [200, { "Content-Type" => "text/plain" }, ["Hello world"]]
```

```
    }
```

```

    end
  end

  SingleFile.initialize!
  run Rails.application

```

This application can be initialized by running rackup in the same directory as the config.ru. Open up your browser at *localhost:9292* and you should get “Hello world” back!

A single-file Rails application is not much different from a regular Rails application. It sets up the load path, in this case by simply using RubyGems instead of Bundler, then it loads all dependencies one by one, instead of using require “rails/all” usually found at the top of config/application.rb, and then it finally defines, initializes and runs the application.

Notice that there are some configuration options that Rails requires us to define, like the config.secret_key_base. All this API is already familiar to us, the only new method used in this file is routes.append.

In general, the only method Rails developers access on the router is the draw() method, as found in a config/routes.rb file:

```

Translator::Application.routes.draw do
  # ...
end

```

The draw() method is meant to work with code reloading. Every time a route file changes, all previously drawn routes are cleared and they are redrawn from scratch by reloading all config/routes.rb entries in the application and plugins. However, in some cases, some routes may be defined during initialization or inside a file that is never reloaded. For such scenarios, Rails provides both routes.prepend and routes.append to define sticky routes.

For example, if you use routes.draw to define routes inside config/application.rb, which is not reloaded in development, as soon as your routes are reloaded because you changed something in config/routes.rb, the routes defined in the application are going to be effectively lost.

A request to this single-file Rails application is going to work as in any other Rails application. The web server is going to invoke the SingleFile#call() method, passing through a middleware stack which ends with the router. In our case, the router simply matches on the root action to a custom Rack application.

Since we now understand the application responsibilities and how it is built on top of railties and engines, it is time to move back to the Translator app and create our translation backend using the I18n API.

8.2 I18n Backends and Extensions

Whenever we invoke `I18n.translate()` (also aliased as `I18n.t()`) or `I18n.localize()` (also aliased as `I18n.l()`) in our application, it is in fact just delegating these methods to the I18n backend stored in `I18n.backend()`. By replacing this backend, you can completely modify how the I18n library works. The I18n framework ships with three different backends:

- `I18n::Backend::Simple`: Is the default backend and keeps translations in an in-memory hash populated from YAML files
- `I18n::Backend::KeyValue`: Uses any key value store as backend, as long it complies with a minimum API
- `I18n::Backend::Chain`: Allows you to chain several backends; in other words, if a translation cannot be found in one backend, it searches for it in the next backend in the chain

There are many features provided by I18n that Rails relies on. For example, in our translator app, we can see in `config/environments/production.rb` the following line:

```
config.i18n.fallbacks = true
```

Whenever this configuration option is set to true, Rails configures the I18n framework to *include* the fallbacks functionality in the current backend, allowing any translation to fall back to the default locale if one cannot be found in the current locale. If you are using I18n outside of a Rails application, you can also use the fallbacks behavior with one line of code:

```
I18n.backend.class.send(:include, I18n::Backend::Fallbacks)
```

Another I18n feature used by Rails is transliteration support. The transliteration that ships with Rails allows you to replace accented Latin characters with their correspondent unaccented ones, as shown here:

```
I18n.transliterate("dziękuję") # => "dziekuje"
```

If you need to transliterate Hebraic, Cyrillic, Chinese characters, and so on, you can add new transliteration rules on demand. Keep in mind that fallbacks and transliterations are not a backend but one of the several extensions provided by the I18n library listed here:

- `I18n::Backend::Cache`: Uses a cache store in front of `I18n.t` to store translation results, that is, the string after lookup, interpolation, and pluralization took place

- `I18n::Backend::Cascade`: Cascades lookups by removing nested scopes from the lookup key; in other words, if `:"foo.bar.baz"` cannot be found, it automatically searches for `:"foo.bar"`;
- `I18n::Backend::Fallbacks`: Provides locale fallbacks, falling back to the default locale if a translation cannot be found in the current one
- `I18n::Backend::Gettext`: Provides support to gettext and `.po` files
- `I18n::Backend::InterpolationCompiler`: Compiles interpolation keys (like `%{model}`) into translation data to speed up performance
- `I18n::Backend::Memoize`: Memoizes lookup results; opposed to `I18n::Backend::Cache`, it uses an in-memory hash and is useful if you are using the Key Value backend
- `I18n::Backend::Metadata`: Adds metadata (such as pluralization count and interpolation values) to translation results
- `I18n::Backend::Pluralization`: Adds support to pluralization rules under `:"i18n.plural.rule"`
- `I18n::Backend::Transliterator`: Adds support to transliteration rules (as discussed earlier) under `:"i18n.transliterate.rule"`;

The I18n library provides several backends and extensions for different areas such as improving performance or adding more flexibility for languages with specific needs. In this chapter, though, we are going to use just two of them: `I18n::Backend::KeyValue` and `I18n::Backend::Memoize`.

The Key Value backend for I18n can accept any object as a store as long as it complies with the following API:

- `@store[]`: A method to read a value given a key
- `@store[]=`: A method to set a value given a key
- `@store.keys`: A method to retrieve all stored keys

Since providing a compliant API is trivial, almost all key-value stores out there can be used with this backend. In this chapter, let's use Redis² since it is generally available and used widely in production.

After Redis is installed and running, let's integrate it with our Rails application by adding the `redis`³ gem, a pure-Ruby client library for Redis, to our Gemfile:

2. <http://redis.io>

3. <https://github.com/redis/redis-rb>

```
translator/1_app/Gemfile
gem 'redis', '~> 3.0.3'
```

And then install it:

```
bundle install
```

Now let's fire up a Rails console with `rails console` and check that Redis conforms with the API expected by I18n:

```
db = Redis.new
db["foo"] = "bar"
db["foo"] # => bar
db.keys   # => ["foo"]
```

Going back to our I18n setup, let's create a file called `lib/translator.rb`, which will be responsible for setting up a Redis instance pointing to the appropriate database (the database is referenced as an integer in Redis). Let's also create a customized Key Value backend that includes the `I18n::Backend::Memoize` module to cache lookups and uses the Redis store on initialization:

```
translator/1_app/lib/translator.rb
module Translator
  DATABASES = {
    "development" => 0,
    "test" => 1,
    "production" => 2
  }

  def self.store
    @store ||= Redis.new(db: DATABASES[Rails.env.to_s])
  end

  class Backend < I18n::Backend::KeyValue
    include I18n::Backend::Memoize

    def initialize
      super(Translator.store)
    end
  end
end
```

Next, let's configure the I18n framework to use our new backend at `Translator::Application`:

```
translator/1_app/config/application.rb
module Translator
  class Application < Rails::Application
    # Set translator backend for I18n
    require "translator"
    config.i18n.backend = Translator::Backend.new
  end
end
```


Opposed to the default I18n backend, the Key Value backend does not load translations from YAML files before each request but just on demand (since it would be slow). That said, to store all default translations in our Redis store, we just need to execute the following command in a terminal:

```
$ rails runner "I18n.backend.load_translations"
```

When we start the Rails console once again, we can access all new translations stored in our Redis store:

```
db = Translator.store
db.keys
db["en.errors.messages.blank"] # => "can't be blank"
db["en.number.precision"] # => "{\"format\":{\"delimiter\":\"|\"}}"
```

Notice that the Key Value store automatically encodes the values to JSON.

8.3 Rails and Sinatra

With translations properly stored, we can now write our Translator app using Sinatra. Sinatra is a DSL for quickly creating web applications in Ruby with minimal effort. The “Hello world” is just a few lines of code:

```
# myapp.rb
require 'sinatra'

get '/' do
  'Hello world!'
end
```

In our case, we won’t access the Sinatra application directly, but we’ll integrate it with our Rails app. This allows us to reuse all the structure we already have in the Rails ecosystem, such as tests, session, authentication, and so on. Before we develop our Sinatra application, let’s write an integration test once again using Capybara to make our tests more robust and readable. First, let’s define ActiveSupport::IntegrationCase inside our test/test_helper.rb which includes Capybara’s DSL:

```
translator/1_app/test/test_helper.rb
require "capybara"
require "capybara/rails"

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

Our test case definition is exactly the same as we did back in [Section 2.2, Integration tests with Capybara, on page 28](#). Now add Capybara to the Gemfile:

```
translator/1_app/Gemfile
group :test do
  gem 'capybara', '~> 2.0.0'
end
```

Our test attempts to localize a date using the Polish locale but will fail because we don't have any translation data for this locale. Next, we should visit the translation URL `/translator/en/pl`, meaning we want to translate messages from English to Polish, fill in the appropriate translation field, and store this new translation. After that, we can assert our translation was successfully stored by being able to localize the same date. The implementation goes like this:

```
translator/1_app/test/integration/translator_app_test.rb
require "test_helper"

class TranslatorAppTest < ActiveSupport::IntegrationCase
  # Set up store and load default translations
  setup { Translator.reload! }

  test "can translate messages from a given locale to another" do
    assert_raise I18n::MissingTranslationData do
      I18n.l(Date.new(2010, 4, 17), locale: :pl)
    end

    visit "/translator/en/pl"
    fill_in "date.formats.default", with: %{"%d-%m-%Y"}
    click_button "Store translations"

    assert_match "Translations stored with success!", page.body
    assert_equal "17-04-2010", I18n.l(Date.new(2010, 4, 17), locale: :pl)
  end
end
```

Notice our test setup invokes a method called `Translator.reload!()`. This method will be responsible for removing all keys from the database and reloading the translation data. Let's implement it next:

```
translator/1_app/lib/translator.rb
def self.reload!
  store.flushdb
  I18n.backend.load_translations
end
```

Our tests are ready to run with `rake test`, but they fail because our Sinatra application is not built yet. So, let's add both Sinatra and Haml to our project Gemfile (and install these new dependencies with `bundle install`):

translator/1_app/Gemfile

```
gem 'sinatra', '~> 1.4.2', require: 'sinatra/base'
gem 'haml', '~> 4.0.2'
```

Our Sinatra application should define a route as `/:from/:to`, which, when accessed, renders a view with all translation data available in `:from` locale, ready to be translated to the `:to` locale. Our first code iteration for our Sinatra application is shown here:

translator/1_app/lib/translator/app.rb

```
module Translator
  class App < Sinatra::Base
    set :environment, Rails.env
    enable :inline_templates

    get "/:from/:to" do |from, to|
      exhibit_translations(from, to)
    end

    protected

    # Store from and to locales in variables and retrieve
    # all keys available for translation.
    def exhibit_translations(from, to)
      @from, @to, @keys = from, to, available_keys(from)
      haml :index
    end

    # Get all keys for a locale. Remove the locale from the key and sort them.
    # If a key is named "en.foo.bar", this method will return it as "foo.bar".
    def available_keys(locale)
      keys = Translator.store.keys("#{locale}.*")
      range = Range.new(locale.size + 1, -1)
      keys.map { |k| k.slice(range) }.sort!
    end

    # Get the value in the translator store for a given locale. This method
    # decodes values and checks if they are a hash, as we don't want subtrees
    # available for translation since they are managed automatically by I18n.
    def locale_value(locale, key)
      value = Translator.store["#{locale}.#{key}"]
      value if value && !ActiveSupport::JSON.decode(value).is_a?(Hash)
    end
  end
end

__END__

@@ index
!!!
%html
```

```

%head
  %title
    Translator::App
%body
  %h2= "From #{@from} to #{@to}"

  %p(style="color:green")= @message

  - if @keys.empty?
    No translations available for #{@from}
  - else
    %form(method="post" action="")
      - @keys.each do |key|
        - from_value = locale_value(@from, key)
        - next unless from_value
        - to_value = locale_value(@to, key) || from_value
      %p
        %label(for=key)
          %small= key
          = from_value
        %br
        %input(id=key name=key type="text" value=to_value size="120")
      %p
        %input(type="submit" value="Store translations")

```

There are a few things to discuss in this implementation. First, notice we explicitly forward the Rails environment to the Sinatra application one. Next, we define the `/:from/:to` route available through the request method GET. If a route matches, Sinatra will yield both parameters to the block, which will be executed. The block simply invokes `exhibit_translations()`, which assigns these parameters to instance variables, gets all locale keys available for translation, and renders the index template.

In this case, we chose to use `Haml`⁴ as template markup for the index page. The template is just a few lines of code and is in the same file as the application by using Sinatra's inline templates feature, which we enabled at the top of the application. However, it is important to notice that templates are evaluated in the same context as the application. This means any method defined in our Sinatra application, as well as its instance variables, are also available in the template. This approach is different from Rails, because Rails templates are not evaluated in the same context as controllers, but in a specific view context, so Rails needs to copy all instance variables from controllers to views behind the scenes, as we saw in [Section 1.3, Understanding The Rails Render-](#)

4. Haml stands for HTML Abstraction Markup Language, and you can find some examples at <http://haml-lang.com/>.

[ing Stack, on page 9](#), and controller methods should be called explicitly as `controller.method()`.

Finally, notice our template calls the `locale_value()` method. This method receives a locale and a key and returns the value stored in Redis. This method should also handle hashes, which are created and stored by default by the I18n framework, to allow you to retrieve subtrees from backends.

In I18n, whenever you store a translation `{ "foo.bar" => "baz" }`, it decomposes the `"foo.bar"` key and also stores `{ "foo" => { "bar" => "baz" } }` as the translation. This allows you to retrieve either the specific translation with `I18n.t("foo.bar") #=> "baz"` or a subtree hash with `I18n.t("foo") #=> { "bar" => "baz" }`. That said, if we show hashes in our Sinatra interface, several translations would be duplicated, because they would appear either in the subtree hash in the `foo` key or in the full key `foo.bar`.

Before we try our Sinatra application, autoload it from `lib/translator.rb`, as shown here:

```
translator/1_app/lib/translator.rb
autoload :App, "translator/app"
```

And finally mount it in the router at `"translator"`:

```
translator/1_app/config/routes.rb
Translator::Application.routes.draw do
  mount Translator::App, at: "/translator"
end
```

Let's verify this now works by starting the server using rails server as usual and accessing `/translator/en/pl` in the browser. You get a translation page similar to the following figure:

From en to pl

```

activerecord.errors.messages.record_invalid "Validation failed: %{errors}"
"Validation failed: %{errors}"

activerecord.errors.messages.taken "has already been taken"
"has already been taken"

date.abbr_day_names ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

date.abbr_month_names [null, "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
[null, "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

date.day_names ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

```

Figure 17—Translator app

This page automatically sets up a page to translate message from English to Polish, but don't click the submit button yet, since we still haven't implemented the POST behavior. In fact, when we run the tests again, they fail exactly for this reason. Clicking the button in integration tests returns a "No routes matches" error:

```

1) Error:
test_can_translate_messages_from_a_given_locale_to_another(TranslatorAppTest)
  ActionController::RoutingError: No route matches [POST] "/translator/en/pl"

```

To make the test pass, let's add a new route to Sinatra for POST requests. This new route should store the translation in the I18n backend, passing the destination locale and the translations decoded from JSON to Ruby; call `save()` in the Redis store, forcing it to be dumped to the filesystem; and exhibit the translation page once again:

```

translator/2_final/lib/translator/app.rb
post "(/:from/:to)" do |from, to|
  I18n.backend.store_translations to, decoded_translations, escape: false
  Translator.store.save
  @message = "Translations stored with success!"
  exhibit_translations(from, to)
end

protected
# Get all translations sent through the form and decode
# their JSON values to check validity.
def decoded_translations
  translations = params.except("from", "to")
  translations.each do |key, value|

```

```
translations[key] = ActiveSupport::JSON.decode(value) rescue nil
end
end
```

Notice we set `:escape` to `false` when storing translations so `I18n` can properly generate subtrees. By default, if you give a translation as `{ "foo.bar" => "baz" }`, `I18n` will treat it as a single key, escaped as `{ "foo\000.bar" => "baz" }`. When it's stored this way, we cannot retrieve its subtree as `I18n.t("foo")`. However, if we turn escaping to `false`, `I18n` will break the key apart, converting it to `{ "foo" => { "bar" => "baz" } }` and allow us to retrieve it as `I18n.t("foo")` or as `I18n.t("foo.bar")`.

Feel free to restart the server and translate all data from any locale to another! Notice we choose to represent the data as JSON in the interface, because we can easily represent arrays, strings, numbers, or booleans. At this point, all our tests are green as well!

Our translator application is almost ready, now it is time to add authentication and improve the robustness of our tests with Capybara!

8.4 Taking It to the Next Level with Devise and Capybara

If any of our applications are going to provide an interface for translations, we should make sure this interface is password-protected and that we can properly test its functionality. In this section, let's take a look at Devise,⁵ a full-stack authentication solution based on Rack, and a deeper look at how Capybara⁶ can be used to test Rack applications.

Adding Cross-Application Authentication

Devise is an interesting solution for authentication because it provides, in very few lines of code, a whole authentication stack, with sign-in, sign-up, password recovery, and more. It uses Warden⁷ to move the authentication handling to the middleware stack, allowing any application, regardless if it is Sinatra or a Rails controller, to use the same authentication rules.

To add Devise to our Translator app, we first need to add it to our Gemfile and run `bundle install` to install it:

```
translator/2_final/Gemfile
gem 'devise', '~> 3.0.0.rc'
```

With the gem installed in our machine, we need to invoke the `devise:install` generator:

5. <http://devise.plataformatec.com.br/>
6. <https://github.com/jnicklas/capybara>
7. <https://github.com/hassox/warden>

```
$ rails g devise:install
```

The generator copies a locale file and an initializer with several configuration options to our application. At the end, it also prints some steps that we need to do manually.

The first step is to configure Action Mailer for development:

```
translator/2_final/config/environments/development.rb
config.action_mailer.default_url_options = { host: 'localhost:3000' }
```

Then we add flash messages to our layout:

```
translator/2_final/app/views/layouts/application.html.erb
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

And finally we add a root route:

```
translator/2_final/config/routes.rb
root to: "home#index"
```

Since our root route points to a HomeController, let's implement that controller. For now, the index action just renders a link to the mounted Sinatra application:

```
translator/2_final/app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    render inline:
      "<%= link_to 'Translate from English to Polish', '/translator/en/pl' %>"
  end
end
```

With all of the setup done, we are ready to create our first Devise model, called Admin:

```
$ rails g devise Admin
```

And then run the migration added by the generator:

```
$ bundle exec rake db:migrate
```

At this point, we didn't make any significant change to our application, but if we run our integration tests, they will fail. This is because the fixtures generated for the Admin have to be properly filled in. However, since we won't need those fixtures for now, let's just delete the fixtures file at test/fixtures/admins.yml to make our tests pass again.

To see how Devise works with our application, feel free to fire up a new server, visit /admins/sign_up, create a new admin account, and sign in. You can also

access `/admins/edit` if you want to change your account (although you may want to disable this sign-up ability before deploying the app).

Devise provides several helpers to restrict access to Rails controllers. Since we created a model called `Admin`, we can use `authenticate_admin!` as a before filter, and the request proceeds only if an admin model is authenticated:

```
class PostsController < ApplicationController
  before_filter :authenticate_admin!
end
```

However, we want to add authentication to our Sinatra app, where Devise does not include any helpers. Fortunately, this is still trivial to achieve with Devise because of Warden. Whenever we invoke `authenticate_admin!` in a Rails controller, it simply executes the following:

```
env["warden"].authenticate!(scope: "admin")
```

The `env["warden"]` object is a proxy created by the Warden middleware, and this middleware is added by Devise to the Rails middleware stack via a `Rails::Engine`. Since this middleware is executed before hitting the router, the proxy object is also available in Sinatra, and we can easily add authentication to `Translator::App` in a before callback:

```
translator/2_final/lib/translator/app.rb
before do
  env["warden"].authenticate!(scope: "admin")
end
```

Overall, our request goes through our application and middleware stack as shown in the following figure:

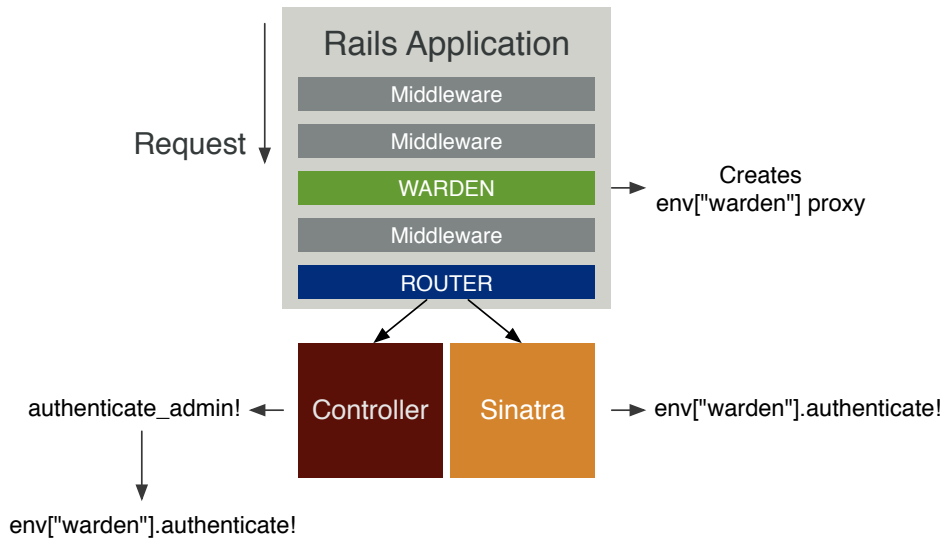


Figure 18—The middleware stack and Rack applications involved in a request to a Rails app with Sinatra, Warden, and Devise

Now when you request the Sinatra application without an admin signed in, the previous before filter will throw an error. This error is caught by the Warden middleware using Ruby's *throw/catch*, allowing Warden to redirect you to the sign-in page inside Devise. Once you sign in, the previous code will simply return the current admin in session, proceeding with Sinatra request.

Although this approach allows us to use the same authentication mechanism across different Rack applications, it has one issue: it requires us to change the Sinatra application by adding a before filter. That said, if we are using a third-party Sinatra application, like the one provided in the Resque⁸ gem, we won't be able to modify it.

In this case, we could ensure authentication at the router level, without a need to change the Sinatra application, as shown here:

```
authenticate "admin" do
  mount Translator::App, at: "/translator"
end
```

8. <https://github.com/resque/resque>

The `authenticate()` shown previously is added by Devise to the Rails router, and it simply uses the router's constraint API to ensure the "admin" role is authenticated. Let's check its implementation in Devise source code:

```
def authenticate(scope)
  constraint = lambda do |request|
    request.env["warden"].authenticate!(:scope => scope)
  end
  constraints(constraint) do
    yield
  end
end
```

Regardless if we choose a before filter or a router constraint to require authentication in our Sinatra application, we can check that the translator backend is now secure by rerunning our test suite and watching it fail.

1) Error:

```
test_can_translate_messages_from_a_given_locale_to_another(TranslatorAppTest)
  Capybara::ElementNotFound: Unable to find field "date.formats.default"
```

The test cannot find the "date.formats.default" label given to `fill_in()` because it is showing the `/admin/sign_in` page instead of the translations page. To fix it, let's authenticate an admin in our integration test using a setup hook:

```
translator/2_final/test/integration/translator_app_test.rb
```

```
setup { sign_in(admin) }
```

```
def admin
  @admin ||= Admin.create!(
    email: "admin_#{Admin.count}@example.org",
    password: "12345678"
  )
end
```

```
def sign_in(admin)
  visit "/admins/sign_in"
  fill_in "Email", with: admin.email
  fill_in "Password", with: admin.password
  click_button "Sign in"
end
```

With this code in place, let's run our test suite again and watch it pass! Notice we decided to manually sign the admin in by filling out the form, instead of using a hack in our tests that modifies the session or passes in a cookie. In fact, even if we wanted to modify the session or a cookie, Capybara would not allow us to do that. And, as we will see next, it actually has a great reason to not allow this.

Adding Cross-Browser Testing

Every time we used Capybara throughout the book, we created our own test case called `ActiveSupport::IntegrationCase` class instead of using `ActionController::IntegrationTest`:

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

While writing Rails integration tests using `ActionController::IntegrationTest`, we have full access to the raw request and response objects, allowing us to check and manipulate cookies, sessions, headers, and so on. Capybara, on the other hand, has a very closed API that does not expose these. That said, if we simply included Capybara in `ActionController::IntegrationTest`, we would be tempted to access and manipulate these objects, leading to both conceptual and practical issues.

Let's discuss the conceptual issue a bit. Capybara was designed to let us write integration tests from the mind-set of an end user. For example, imagine we are building an ecommerce site that keeps the last five products we viewed in the footer. If our implementation is simply storing these product IDs in the session, a naive integration test would simply assert that, after accessing a product page, the product ID was added to the session.

The issue with this kind of test is that the ecommerce user does not care at all if something was stored in the session. They just want to see the last-visited products in the footer and be able to click them, something we did not assert in our tests.

Besides, the fact we store this information in the session is an implementation detail. If at some point we decide to keep this data in a cookie, our naive test will fail, while it should still pass since the user interface has not changed at all. This is a common symptom in tests too coupled to their implementation.

For this reason, Capybara hides all these internals from you, which works out well considering that one of the most important features in Capybara is that it supports different drivers. Capybara drivers manipulate a browser, which then accesses our application through a web server, as shown in the following figure:

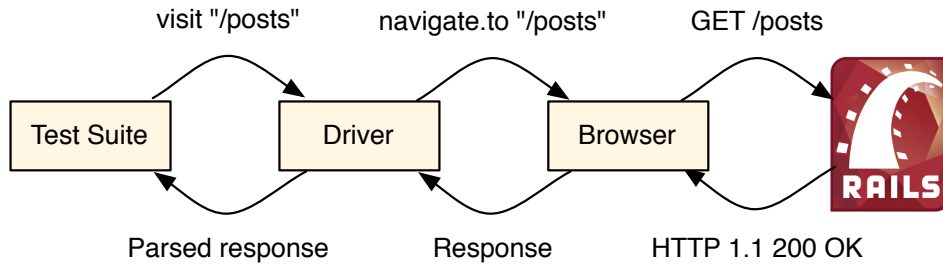


Figure 19—Call trace when using Capybara with Selenium driver and Firefox

Some drivers such as Selenium use common browsers as Mozilla Firefox, Internet Explorer, and Google Chrome, while others interact with a headless browser like PhantomJS.

As you may expect, each browser supported by Selenium must expose a limited API. Some may expose access to cookies; others may not. Some headless browsers may give you full control on request headers, but others do not. To allow you to switch drivers and browsers without a need to rewrite a huge part of your integration tests, Capybara focuses on the common set supported by most of them.

By default, Capybara uses the Rack test driver, which bypasses the whole browser and accesses the Rack application directly. This is very convenient in terms of performance but it is also very limited. For example, any feature that relies on JavaScript can't be tested with the default driver. Luckily, we can easily change our applications to use another driver. Let's give Selenium⁹ a try in our application by adding the following lines to our test helper:

```

translator/2_final/test/test_helper.rb
require "selenium-webdriver"

# Can be :chrome, :firefox or :ie
Selenium::WebDriver.for :firefox
Capybara.default_driver = :selenium

class ActiveSupport::TestCase
  # Disable transactional fixtures for integration testing
  self.use_transactional_fixtures = false
end

```

9. <http://seleniumhq.org/>

Selenium uses real browsers to test our application. By default it uses Firefox,¹⁰ which you need to have installed before running tests again. After you install it, run our integration tests again, and notice that Selenium is automatically starting Firefox and driving it against our website! At the end, our tests still pass!

Because Selenium needs to access a web server for each request, Capybara automatically starts one up. Since Capybara fires this new web server in a thread, the database connection used in tests is not the same used by the server in each request. That said, if we use transactional fixtures to wrap each test in a database transaction, the data created in tests won't be available in the server since transactional data is not shared between database connections until it is committed. This is the reason we need to disable transactional fixtures in our `test/test_helper.rb` file, even though it reduces performance.

Another consequence of disabling transactional fixtures is that the data stored in our database is not cleaned up between tests, and this will definitely get in the way as we add new tests to our suite. Fortunately, a few solutions are already available today that do all this work for us. One that stands out is Database Cleaner¹¹, since it supports different ORMs and databases.

8.5 Wrapping Up

In this chapter, we created another Rails application and used the opportunity to better describe how Rails applications are structured and designed. We have once again seen the importance of the Rack specification and how it makes easy for different frameworks (such as Rails and Sinatra) to play along without hassle. We also learned more about I18n, its backends, and its extensions. Plus, we found and implemented a good case to use a simple key-value store as Redis.

At the end, we discussed two gems that are widely used in the Rails community: Devise and Capybara. Even though it was simply an introduction, I advise you to take the next step and play with them a bit more. And by playing, I mean not only using them but checking out their source code. You will see how Devise uses ActionController::Metal, as we saw in *Playing with Metal*, on page 57, to define a bare-bones controller and learn how Capybara makes use of Rack applications and handlers to automatically start up servers.

That finishes our tour of Rails. Everything we discussed (the rendering stack, railties, engines, generators, Active Model, and so on) are not only powerful

10. <http://www.mozilla.com/firefox/>

11. https://github.com/bmabey/database_cleaner

APIs for the development of Rails extensions and applications but also for the development of Rails itself.

While developing your next web application or maintaining an existing one, remember all the tools available to make your code cleaner. You can use responders to DRY up your controller and use generators to keep you and your team productive. You are also able to analyze other Rails extensions source code, submit changes, and debug problems with more ease.

Finally, you understand Rails better. You can explore other areas of the source code, study other Action Controller and Active Models modules, check other generators implementations, or read the source of railties, engines, and applications with detail! Rails also has detailed guides on how to contribute to Rails,¹² so if you haven't yet, this is the high time to propose improvements or fix some bugs that may have bothered you some time ago.

I hope this book has taught you new ways to improve your Ruby code and Rails applications. Most of all, I hope you had fun.

—José Valim

12. http://guides.rubyonrails.org/contributing_to_rails.html

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

This Book's Home Page

<http://pragprog.com/book/jvrails2>

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: <http://pragprog.com/book/jvrails2>

Contact Us

Online Orders: <http://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://pragprog.com/write-for-us>

Or Call: +1 800-699-7764