Chapter 4

Rails-flavored Ruby

Grounded in examples from Chapter 3, this chapter explores some elements of the Ruby programming language that are important for Rails. Ruby is a big language, but fortunately the subset needed to be productive as a Rails developer is relatively small. It also differs somewhat from the usual material covered in an introduction to Ruby. This chapter is designed to give you a solid foundation in Rails-flavored Ruby, whether or not you have prior experience in the language. It covers a lot of material, and it's OK not to get it all on the first pass. We'll refer back to it frequently in future chapters.[1]

4.1 Motivation

As we saw in the last chapter, it's possible to develop the skeleton of a Rails application, and even start testing it, with essentially no knowledge of the underlying Ruby language. We did this by relying on the test code provided by the tutorial and addressing each error message until the test suite was passing. This situation can't last forever, though, and we'll open this chapter with an addition to the site that brings us face-to-face with our Ruby limitations.

As in Section 3.2, we'll use a separate topic branch to keep our changes self-contained:

[1]For a more systematic introduction to Ruby, see *Learn Enough Ruby to Be Dangerous*.

```
$ git checkout -b rails-flavored-ruby
```

We'll merge our changes into **`master`** in Section 4.5.

4.1.1 Built-in helpers

When we last saw our new application, we had just updated our mostly static pages to use Rails layouts to eliminate duplication in our views, as shown in Listing 4.1 (which is the same as Listing 3.37).

Listing 4.1: The sample application site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

Let's focus on one particular line in Listing 4.1:

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track': 'reload' %>
```

This uses the built-in Rails function **`stylesheet_link_tag`** (which you can read more about at the Rails API)[2] to include **`application.css`** for all media

[2]An "API" is an Application Programming Interface, which is a set of methods and other conventions that

types (including computer screens and printers). To an experienced Rails developer, this line looks simple, but there are at least four potentially confusing Ruby ideas: built-in Rails methods, method invocation with missing parentheses, symbols, and hashes. We'll cover all of these ideas in this chapter.

4.1.2 Custom helpers

In addition to coming equipped with a large number of built-in functions for use in the views, Rails also allows the creation of new ones. Such functions are called *helpers*; to see how to make a custom helper, let's start by examining the title line from Listing 4.1:

```
<%= yield(:title) %> | Ruby on Rails Tutorial Sample App
```

This relies on the definition of a page title (using **`provide`**) in each view, as in

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

But what if we don't provide a title? It's a good convention to have a *base title* we use on every page, with an optional page title if we want to be more specific. We've *almost* achieved that with our current layout, with one wrinkle: as you can see if you delete the **`provide`** call in one of the views, in the absence of a page-specific title the full title appears as follows:

serves as an abstraction layer for interacting with a software system. The practical effect is that we as developers don't need to understand the program internals; we need only be famliar with the public-facing API. In the present case, this means that, rather than be concerned with how **`stylesheet_link_tag`** is implemented, we need only know how it behaves.

```
| Ruby on Rails Tutorial Sample App
```

In other words, there's a suitable base title, but there's also a leading vertical bar character `|` at the beginning.

To solve the problem of a missing page title, we'll define a custom helper called **`full_title`**. The **`full_title`** helper returns a base title, "Ruby on Rails Tutorial Sample App", if no page title is defined, and adds a vertical bar preceded by the page title if one is defined (Listing 4.2).[3]

Listing 4.2: Defining a **`full_title`** helper.
app/helpers/application_helper.rb

```ruby
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      page_title + " | " + base_title
    end
  end
end
```

Now that we have a helper, we can use it to simplify our layout by replacing

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

with

[3]If a helper is specific to a particular controller, you should put it in the corresponding helper file; for example, helpers for the Static Pages controller generally go in **`app/helpers/static_pages_helper.rb`**. In our case, we expect the **`full_title`** helper to be used on all the site's pages, and Rails has a special helper file for this case: **`app/helpers/application_helper.rb`**.

```
<title><%= full_title(yield(:title)) %></title>
```

as seen in Listing 4.3.

Listing 4.3: The site layout with the `full_title` helper. GREEN
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                               'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

To put our helper to work, we can eliminate the unnecessary word “Home” from the Home page, allowing it to revert to the base title. We do this by first updating our test with the code in Listing 4.4, which updates the previous title test and adds one to test for the absence of the custom `"Home"` string in the title.

Listing 4.4: An updated test for the Home page’s title. RED
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get static_pages_home_url
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
```

```
    get static_pages_help_url
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

Let's run the test suite to verify that one test fails:[4]

Listing 4.5: RED

```
$ rails test
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
```

To get the test suite to pass, we'll remove the **`provide`** line from the Home page's view, as seen in Listing 4.6.

Listing 4.6: The Home page with no custom page title. GREEN
app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

At this point the tests should pass:

Listing 4.7: GREEN

```
$ rails test
```

[4] I'll generally run the test suite explicitly for completeness, but in practice I usually just use Guard as described in Section 3.6.2.