Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

Contents

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I've worked my way through many Rails books, this is the one that finally made me "get" it. Everything is done very much "the Rails way"—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it's like to do a real-world project. The tutorial's code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor Aurelius Prochazka. I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

About the author

Michael Hartl is the author of the *Ruby on Rails Tutorial*, the leading introduction to web development with Ruby on Rails. His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a Ruby Hero Award for his contributions to the Ruby community. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Copyright and license

Chapter 8
Sign in, sign out

Now that new users can sign up for our site (Chapter 7), it's time to give registered users the ability to sign in and sign out. This will allow us to add customizations based on signin status and based on the identity of the current user. For example, in this chapter we'll update the site header with signin/signout links and a profile link. In Chapter 10, we'll use the identity of a signed-in user to create microposts associated with that user, and in Chapter 11 we'll allow the current user to follow other users of the application (thereby receiving a feed of their microposts).

Having users sign in will also allow us to implement a security model, restricting access to particular pages based on the identity of the signed-in user. For instance, as we'll see in Chapter 9, only signed-in users will be able to access the page used to edit user information. The signin system will also make possible special privileges for administrative users, such as the ability (also introduced in Chapter 9) to delete users from the database.

After implementing the core authentication machinery, we'll take a short detour to investigate *Cucumber*, a popular system for behavior-driven development (Section 8.3). In particular, we'll re-implement a couple of the RSpec integration tests in Cucumber to see how the two methods compare.

As in previous chapters, we'll do our work on a topic branch and merge in the changes at the end:

```
$ git checkout -b sign-in-out
```

8.1 Sessions and signin failure

A *session* is a semi-permanent connection between two computers, such as a client computer running a web browser and a server running Rails. We'll be using sessions to implement the common pattern of "signing in", and in this context there are several different models for session behavior common on the web: "forgetting" the session on browser close, using an optional "remember me" checkbox for persistent sessions, and automatically remembering sessions until the user explicitly signs out.[1] We'll opt for the final of these options: when users sign in, we will remember their signin status "forever", clearing the session only when the user explicitly signs out. (We'll see in Section 8.2.1 just how long "forever" is.)

It's convenient to model sessions as a RESTful resource: we'll have a signin page for *new* sessions, signing in will *create* a session, and signing out will *destroy* it. Unlike the Users resource, which uses a database back-end (via the User model) to persist data, the Sessions resource will use a *cookie*, which is a small piece of text placed on the user's browser. Much of the work involved in signin comes from building this cookie-based authentication machinery. In this section and the next, we'll prepare for this work by constructing a Sessions controller, a signin form, and the relevant controller actions. We'll then complete user signin in Section 8.2 by adding the necessary cookie-manipulation code.

8.1.1 Sessions controller

The elements of signing in and out correspond to particular REST actions of the Sessions controller: the signin form is handled by the `new` action (covered in this section), actually signing in is handled by sending a `POST` request to the `create` action (Section 8.1 and Section 8.2), and signing out is handled by sending a `DELETE` request to the `destroy` action (Section 8.2.6). (Recall the association of HTTP verbs with REST actions from Table 7.1.) To get started, we'll generate a Sessions controller and an integration test for the authentication machinery:

```
$ rails generate controller Sessions --no-test-framework
$ rails generate integration_test authentication_pages
```

Following the model from Section 7.2 for the signup page, we'll create a signin form for creating new sessions, as mocked up in Figure 8.1.

Figure 8.1: A mockup of the signin form. (full size)

The signin page will live at the URI given by `signin_path` (defined momentarily), and as usual

we'll start with a minimalist test, as shown in Listing 8.1. (Compare to the analogous code for the signup page in Listing 7.6.)

Listing 8.1. Tests for the `new` session action and view.

spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do

  subject { page }

  describe "signin page" do
    before { visit signin_path }

    it { should have_selector('h1',    text: 'Sign in') }
    it { should have_selector('title', text: 'Sign in') }
  end
end
```

The tests initially fail, as required:

```
$ bundle exec rspec spec/
```

To get the tests in Listing 8.1 to pass, we first need to define routes for the Sessions resource, together with a custom named route for the signin page (which we'll map to the Session controller's `new` action). As with the Users resource, we can use the `resources` method to define the standard RESTful routes:

```
resources :sessions, only: [:new, :create, :destroy]
```

Since we have no need to show or edit sessions, we've restricted the actions to `new`, `create`, and `destroy` using the `:only` option accepted by `resources`. The full result, including named routes for signin and signout, appears in Listing 8.2.

Listing 8.2. Adding a resource to get the standard RESTful actions for sessions.
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users
  resources :sessions, only: [:new, :create, :destroy]

  match '/signup',  to: 'users#new'
  match '/signin',  to: 'sessions#new'
  match '/signout', to: 'sessions#destroy', via: :delete
  .
  .
  .
end
```

Note the use of `via: :delete` for the signout route, which indicates that it should be invoked using an HTTP DELETE request.

The resources defined in Listing 8.2 provide URIs and actions similar to those for users (Table 7.1), as shown in Table 8.1. Note that the routes for signin and signout are custom, but the route for creating a session is simply the default (i.e., `[resource name]_path`).

HTTP request	URI	Named route	Action	Purpose
GET	/signin	`signin_path`	`new`	page for a new session (signin)
POST	/sessions	`sessions_path`	`create`	create a new session
DELETE	/signout	`signout_path`	`destroy`	delete a session (sign out)

Table 8.1: RESTful routes provided by the sessions rules in Listing 8.2.

The next step to get the tests in Listing 8.1 to pass is to add a `new` action to the Sessions controller, as shown in Listing 8.3 (which also defines the `create` and `destroy` actions for future reference).

Listing 8.3. The initial Sessions controller.

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
  end

  def destroy
  end
end
```

The final step is to define the initial version of the signin page. Note that, since it is the page for a new session, the signin page lives in the file `app/views/sessions/new.html.erb`, which we have to create. The contents, which for now only define the page title and top-level heading, appear as in Listing 8.4.

Listing 8.4. The initial signin view.

`app/views/sessions/new.html.erb`

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>
```

With that, the tests in Listing 8.1 should be passing, and we're ready to make the actual signin form.

```
$ bundle exec rspec spec/
```

8.1.2 Signin tests

Comparing Figure 8.1 with Figure 7.11, we see that the signin form (or, equivalently, the new session form) is similar in appearance to the signup form, except with two fields (email and password) in place of four. As with the signup form, we can test the signin form by using Capybara to fill in the form values and then click the button.

In the process of writing the tests, we'll be forced to design aspects of the application, which is one of the nice side-effects of test-driven development. We'll start with invalid signin, as mocked up in Figure 8.2.

Figure 8.2: A mockup of signin failure. (full size)

As seen in Figure 8.2, when the signin information is invalid we want to re-render the signin page

and display an error message. We'll render the error as a flash message, which we can test for as follows:

```
it { should have_selector('div.alert.alert-error', text: 'Invalid') }
```

(We saw similar code in Listing 7.32 from the exercises in Chapter 7.) Here the selector element (i.e., the tag) we're looking for is

```
div.alert.alert-error
```

Recalling that the dot means "class" in CSS (Section 5.1.2), you might be able to guess that this tests for a `div` tag with the classes `"alert"` and `"alert-error"`. We also test that the error message contains the text `"Invalid"`. Putting these together, the test looks for an element of the following form:

```
<div class="alert alert-error">Invalid...</div>
```

Combining the title and flash tests gives the code in Listing 8.5. As we'll see, these tests miss an important subtlety, which we'll address in Section 8.1.5.

Listing 8.5. The tests for signin failure.
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
```

```
  .
  describe "signin" do
    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_selector('title', text: 'Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }
    end
  end
end
```

Having written tests for signin failure, we now turn to signin success. The changes we'll test for are the rendering of the user's profile page (as determined by the page title, which should be the user's name), together with three planned changes to the site navigation:

1. The appearance of a link to the profile page
2. The appearance of a "Sign out" link
3. The disappearance of the "Sign in" link

(We'll defer the test for the "Settings" link to Section 9.1 and for the "Users" link to Section 9.3.) A mockup of these changes appears in Figure 8.3.[2] Note that the signout and profile links appear in a dropdown "Account" menu; in Section 8.2.4, we'll see how to make such a menu with Bootstrap.

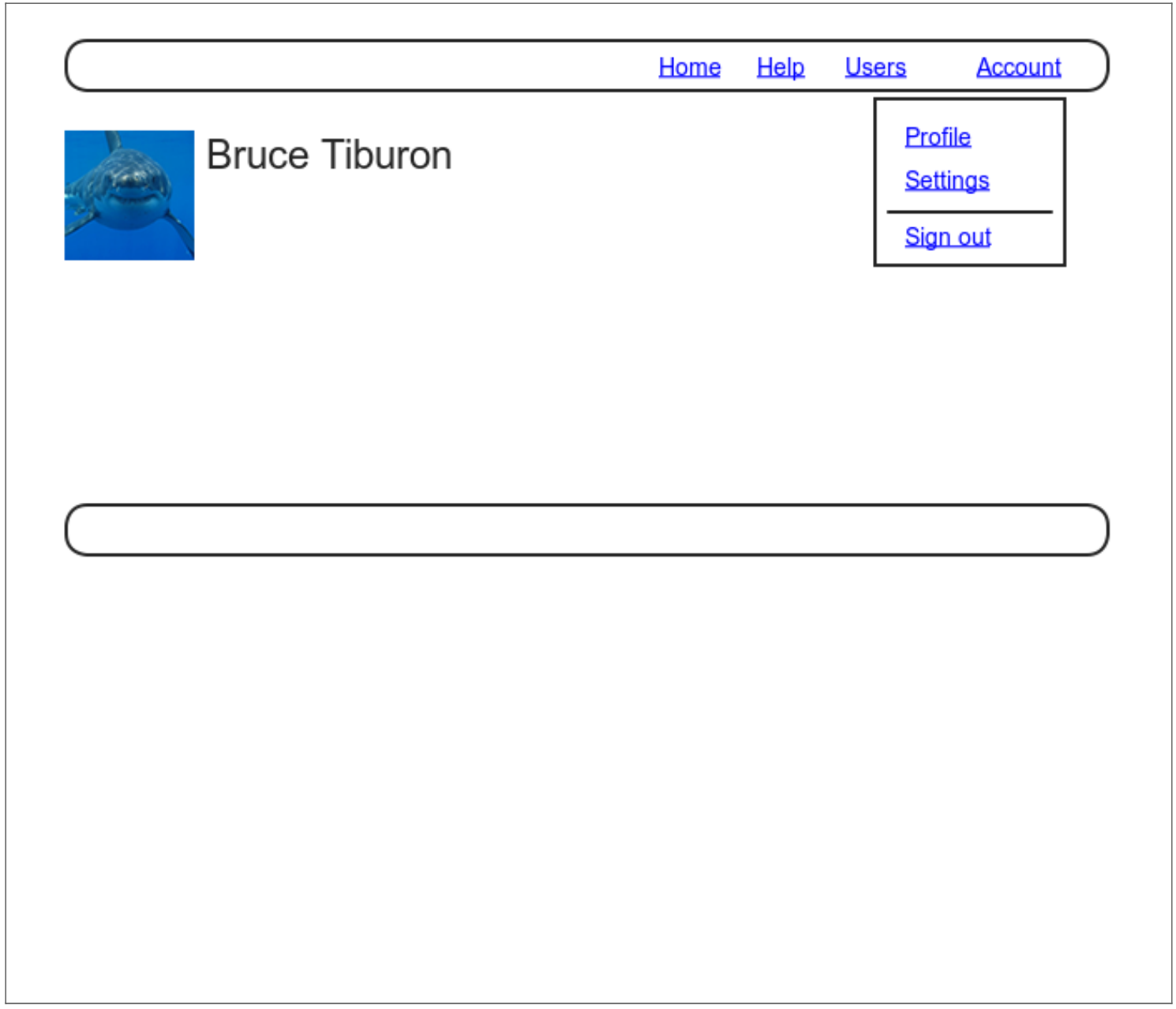


Figure 8.3: A mockup of the user profile after a successful signin. (full size)

The test code for signin success appears in Listing 8.6.

Listing 8.6. Test for signin success.

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do
    before { visit signin_path }
    .
    .
    .
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        fill_in "Email",    with: user.email
        fill_in "Password", with: user.password
        click_button "Sign in"
      end

      it { should have_selector('title', text: user.name) }
      it { should have_link('Profile', href: user_path(user)) }
      it { should have_link('Sign out', href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
    end
  end
end
```

Here we've used the `have_link` method. It takes as arguments the text of the link and an optional `:href` parameter, so that

```
it { should have_link('Profile', href: user_path(user)) }
```

ensures that the anchor tag `a` has the right `href` (URI) attribute—in this case, a link to the user's profile page.

8.1.3 Signin form

With our tests in place, we're ready to start developing the signin form. Recall from Listing 7.17 that the signup form uses the `form_for` helper, taking as an argument the user instance variable `@user`:

```
<%= form_for(@user) do |f| %>
  .
  .
  .
<% end %>
```

The main difference between this and the signin form is that we have no Session model, and hence no analogue for the `@user` variable. This means that, in constructing the new session form, we have to give `form_for` slightly more information; in particular, whereas

```
form_for(@user)
```

allows Rails to infer that the `action` of the form should be to `POST` to the URI `/users`, in the case of sessions we need to indicate the *name* of the resource and the corresponding URI:

```
form_for(:session, url: sessions_path)
```

(A second option is to use `form_tag` in place of `form_for`; this might be more even idiomatically correct Rails, but it has less in common with the signup form, and at this stage I want to emphasize

the parallel structure. Making a working form with `form_tag` is left as an exercise (Section 8.5).)

With the proper `form_for` in hand, it's easy to make a signin form to match the mockup in Figure 8.1 using the signup form (Listing 7.17) as a model, as shown in Listing 8.7.

Listing 8.7. Code for the signin form.

`app/views/sessions/new.html.erb`

```
<% provide(:title, "Sign in") %>
<h1>Sign in</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(:session, url: sessions_path) do |f| %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.submit "Sign in", class: "btn btn-large btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

Note that we've added a link to the signup page for convenience. With the code in Listing 8.7, the signin form appears as in Figure 8.4.

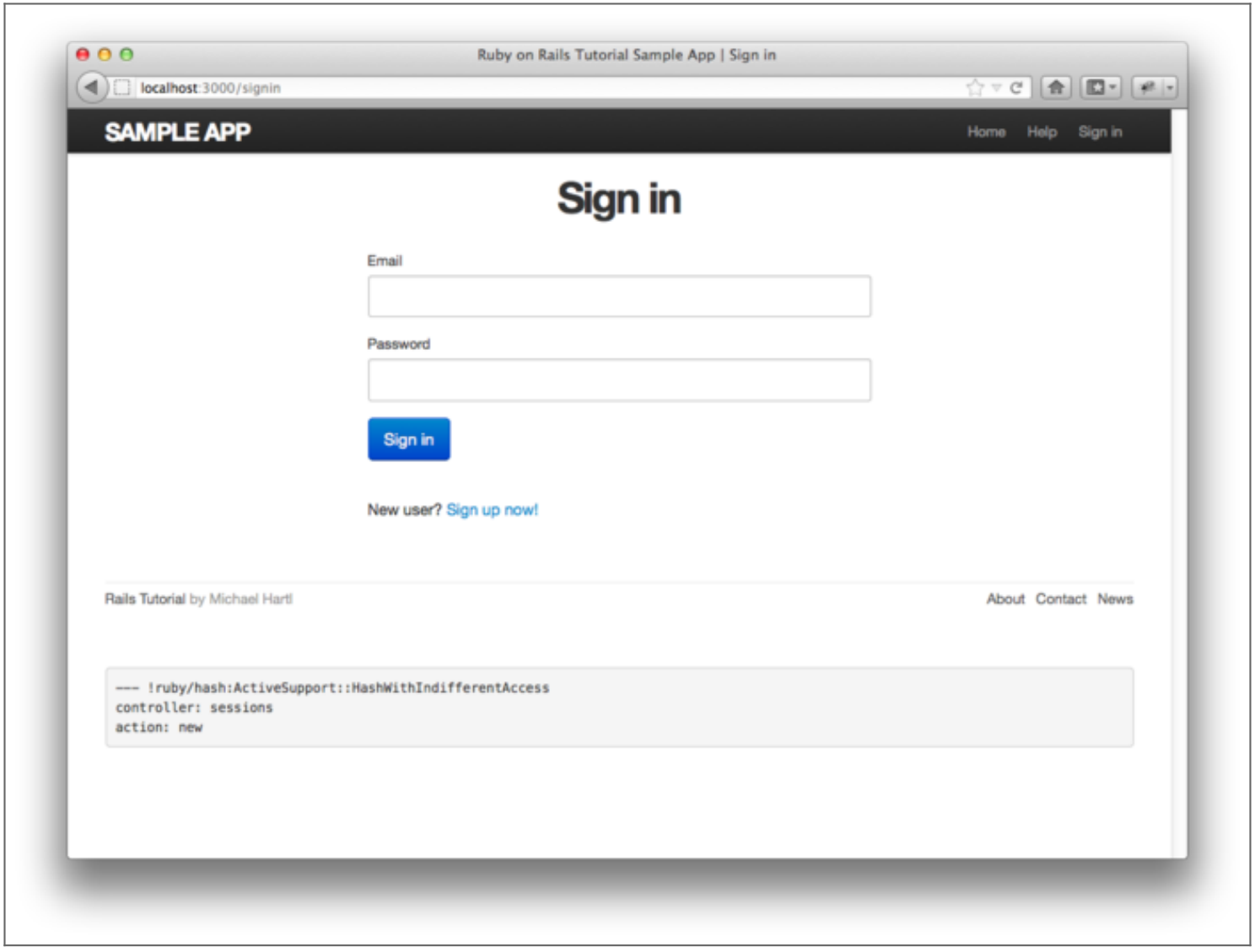


Figure 8.4: The signin form (/signin). (full size)

Though you'll soon get out of the habit of looking at the HTML generated by Rails (instead trusting the helpers to do their job), for now let's take a look at it (Listing 8.8).

Listing 8.8. HTML for the signin form produced by Listing 8.7.

```
<form accept-charset="UTF-8" action="/sessions" method="post">
  <div>
    <label for="session_email">Email</label>
    <input id="session_email" name="session[email]" size="30" type="text" />
  </div>
  <div>
    <label for="session_password">Password</label>
    <input id="session_password" name="session[password]" size="30"
           type="password" />
  </div>
  <input class="btn btn-large btn-primary" name="commit" type="submit"
       value="Sign in" />
</form>
```

Comparing Listing 8.8 with Listing 7.20, you might be able to guess that submitting this form will result in a `params` hash where `params[:session][:email]` and `params[:session][:password]` correspond to the email and password fields.

8.1.4 Reviewing form submission

As in the case of creating users (signup), the first step in creating sessions (signin) is to handle *invalid* input. We already have tests for the signup failure (Listing 8.5), and the application code is simple apart from a couple of subtleties. We'll start by reviewing what happens when a form gets submitted, and then arrange for helpful error messages to appear in the case of signin failure (as mocked up in Figure 8.2.) Then we'll lay the foundation for successful signin (Section 8.2) by evaluating each signin submission based on the validity of its email/password combination.

Let's start by defining a minimalist `create` action for the Sessions controller (Listing 8.9), which does nothing but render the `new` view. Submitting the /sessions/new form with blank fields then yields the result shown in Figure 8.5.

Listing 8.9. A preliminary version of the Sessions `create` action.

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    render 'new'
  end
  .
  .
  .
end
```

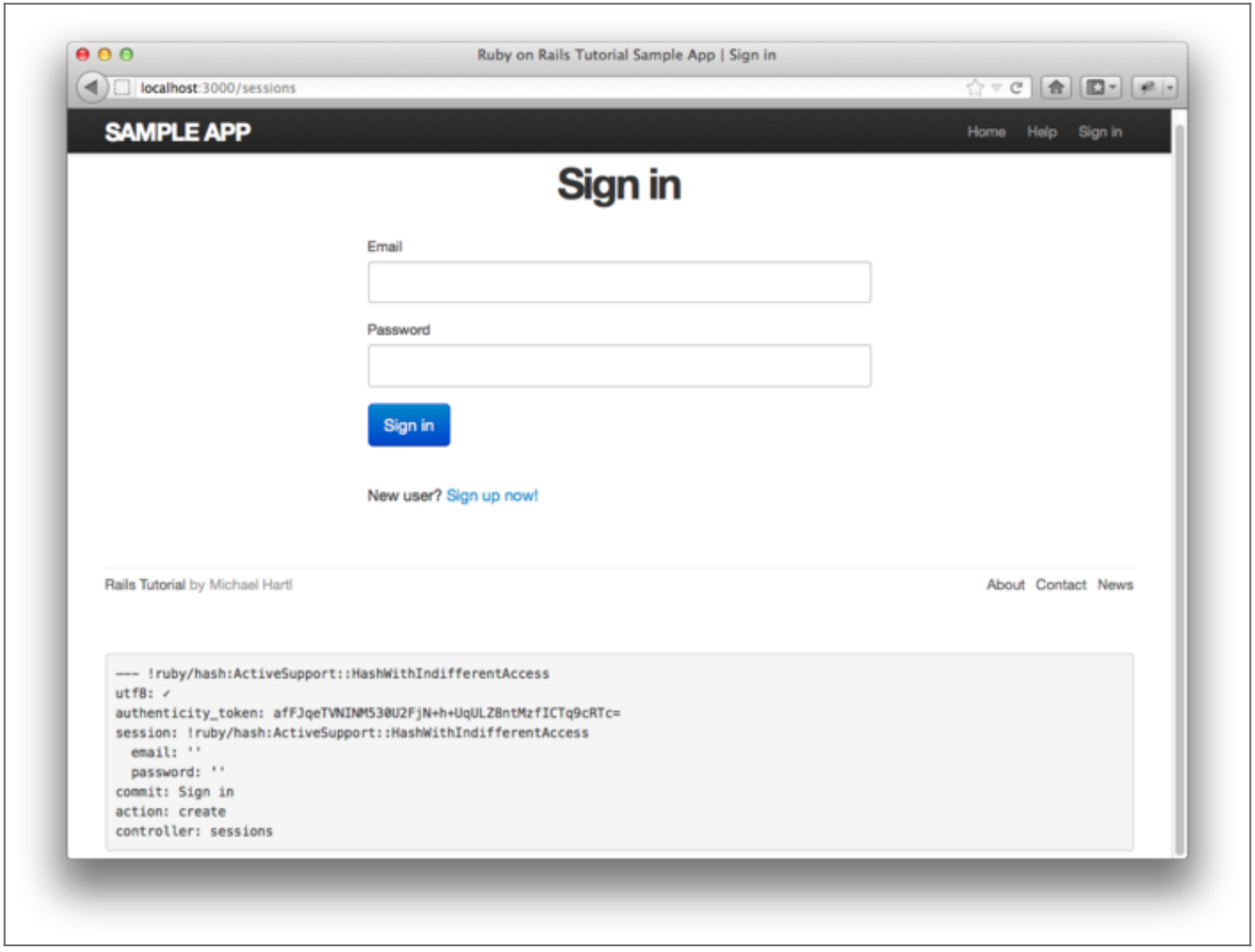


Figure 8.5: The initial failed signin, with `create` as in Listing 8.9. (full size)

Carefully inspecting the debug information in Figure 8.5 shows that, as hinted at the end of Section 8.1.3, the submission results in a `params` hash containing the email and password under the key `:session`:

```
---
session:
  email: ''
  password: ''
commit: Sign in
action: create
controller: sessions
```

As with the case of user signup (Figure 7.15) these parameters form a *nested* hash like the one we saw in Listing 4.6. In particular, **`params`** contains a nested hash of the form

```
{ session: { password: "", email: "" } }
```

This means that

```
params[:session]
```

is itself a hash:

```
{ password: "", email: "" }
```

As a result,

```
params[:session][:email]
```

is the submitted email address and

```
params[:session][:password]
```

is the submitted password.

In other words, inside the **`create`** action the **`params`** hash has all the information needed to authenticate users by email and password. Not coincidentally, we already have exactly the methods we need: the **`User.find_by_email`** method provided by Active Record (Section 6.1.4) and the **`authenticate`** method provided by **`has_secure_password`** (Section 6.3.3). Recalling that **`authenticate`** returns **`false`** for an invalid authentication, our strategy for user signin can be summarized as follows:

```
def create
  user = User.find_by_email(params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    # Sign the user in and redirect to the user's show page.
  else
    # Create an error message and re-render the signin form.
  end
end
```

The first line here pulls the user out of the database using the submitted email address. (Recall from Section 6.2.5 that email addresses are saved as all lower-case, so here we use the **`downcase`** method to ensure a match when the submitted address is valid.) The next line can be a bit confusing but is fairly common in idiomatic Rails programming:

```
user && user.authenticate(params[:session][:password])
```

This uses **`&&`** (logical *and*) to determine if the resulting user is valid. Taking into account that any

object other than `nil` and `false` itself is `true` in a boolean context (Section 4.2.3), the possibilities appear as in Table 8.2. We see from Table 8.2 that the `if` statement is `true` only if a user with the given email both exists in the database and has the given password, exactly as required.

User	Password	a && b
nonexistent	*anything*	`nil && [anything] == false`
valid user	wrong password	`true && false == false`
valid user	right password	`true && true == true`

Table 8.2: Possible results of `user && user.authenticate(...)`.

8.1.5 Rendering with a flash message

Recall from Section 7.3.2 that we displayed signup errors using the User model error messages. These errors are associated with a particular Active Record object, but this strategy won't work here because the session isn't an Active Record model. Instead, we'll put a message in the flash to be displayed upon failed signin. A first, slightly incorrect, attempt appears in Listing 8.10.

Listing 8.10. An (unsuccessful) attempt at handling failed signin.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by_email(params[:session][:email].downcase)
```

```
    if user && user.authenticate(params[:session][:password])
      # Sign the user in and redirect to the user's show page.
    else
      flash[:error] = 'Invalid email/password combination' # Not quite right!
      render 'new'
    end
  end

  def destroy
  end
end
```

Because of the flash message display in the site layout (Listing 7.26), the `flash[:error]` message automatically gets displayed; because of the Bootstrap CSS, it automatically gets nice styling (Figure 8.6).

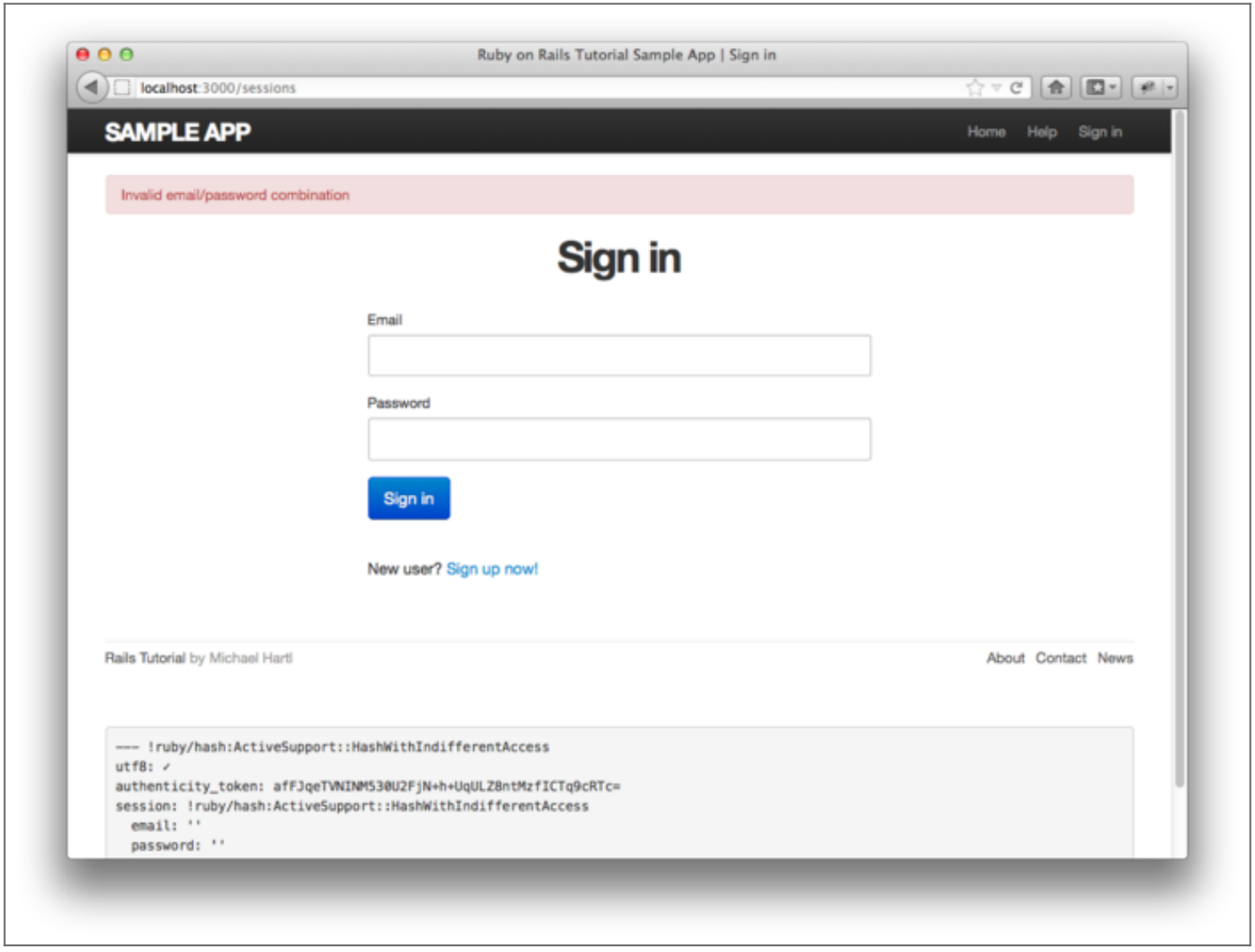


Figure 8.6: The flash message for a failed signin. (full size)

Unfortunately, as noted in the text and in the comment in Listing 8.10, this code isn't quite right. The page looks fine, though, so what's the problem? The issue is that the contents of the flash persist for one *request*, but—unlike a redirect, which we used in Listing 7.27—re-rendering a template with `render` doesn't count as a request. The result is that the flash message persists one

request longer than we want. For example, if we submit invalid information, the flash is set and gets displayed on the signin page (Figure 8.6); if we then click on another page, such as the Home page, that's the first request since the form submission, and the flash gets displayed again (Figure 8.7).

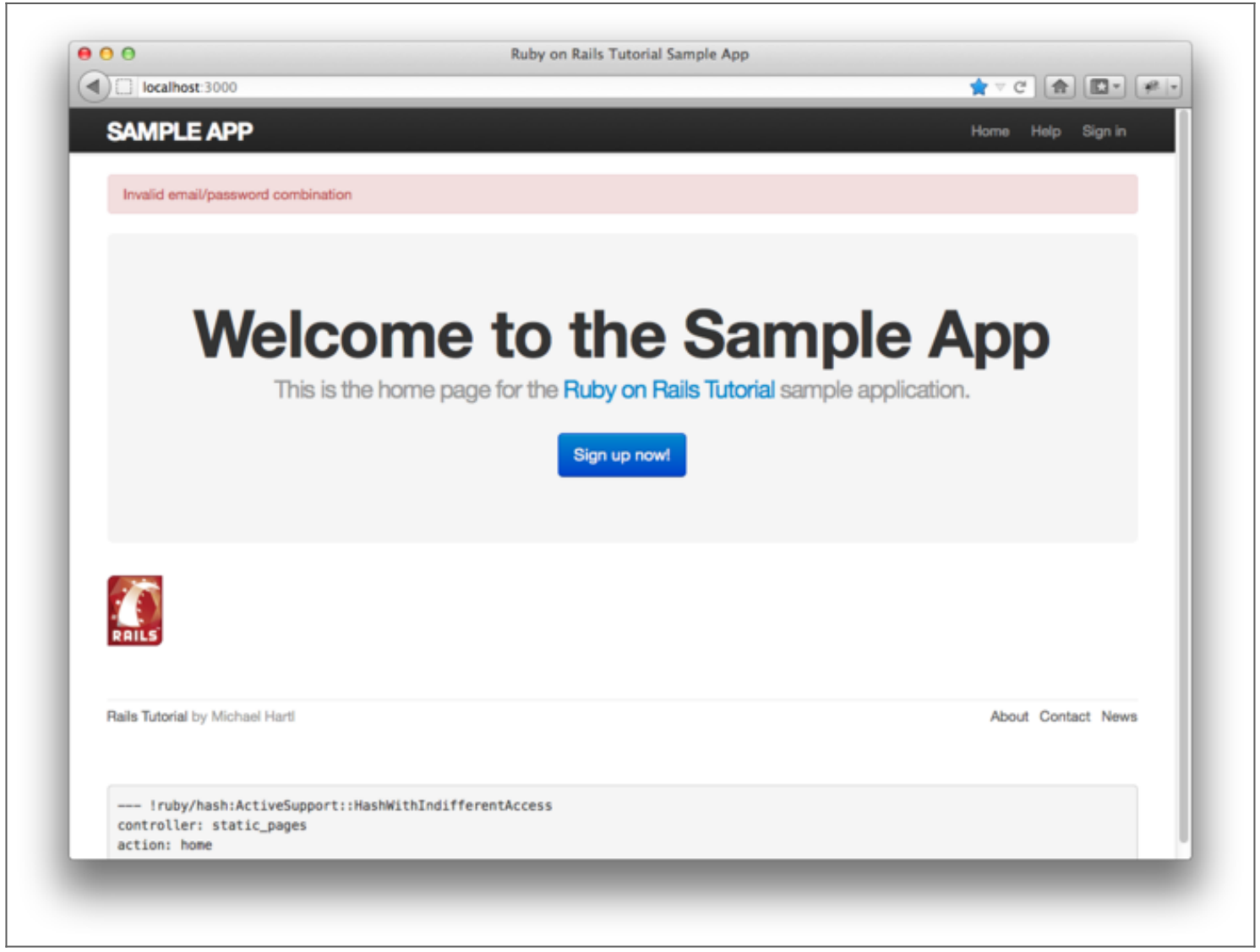


Figure 8.7: An example of the flash persisting. (full size)

This flash persistence is a bug in our application, and before proceeding with a fix it is a good idea to write a test catching the error. In particular, the signin failure tests are currently passing:

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "signin with invalid information"
```

But the tests should never pass when there is a known bug, so we should add a failing test to catch it. Fortunately, dealing with a problem like flash persistence is one of many areas where integration tests really shine; they let us say exactly what we mean:

```
describe "after visiting another page" do
  before { click_link "Home" }
  it { should_not have_selector('div.alert.alert-error') }
end
```

After submitting invalid signin data, this test follows the Home link in the site layout and then requires that the flash error message not appear. The updated code, with the modified flash test, is shown in Listing 8.11.

Listing 8.11. Correct tests for signin failure.
spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "signin" do

    before { visit signin_path }
```

```
    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_selector('title', text: 'Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }

      describe "after visiting another page" do
        before { click_link "Home" }
        it { should_not have_selector('div.alert.alert-error') }
      end
    end
    .
    .
    .
  end
end
```

The new test fails, as required:

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "signin with invalid information"
```

To get the failing test to pass, instead of `flash` we use `flash.now`, which is specifically designed for displaying flash messages on rendered pages; unlike the contents of `flash`, its contents disappear as soon as there is an additional request. The corrected application code appears in Listing 8.12.

Listing 8.12. Correct code for failed signin.

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end
```

```
  def create
    user = User.find_by_email(params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Sign the user in and redirect to the user's show page.
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Now the test suite for users with invalid information should be green:

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb \
> -e "with invalid information"
```

8.2 Signin success

Having handled a failed signin, we now need to actually sign a user in. Getting there will require some of the most challenging Ruby programming so far in this tutorial, so hang in there through the end and be prepared for a little heavy lifting. Happily, the first step is easy—completing the Sessions controller `create` action is a snap. Unfortunately, it's also a cheat.

Filling in the area now occupied by the signin comment (Listing 8.12) is simple: upon successful signin, we sign the user in using the `sign_in` function, and then redirect to the profile page (Listing 8.13). We see now why this is a cheat: alas, `sign_in` doesn't currently exist. Writing it will occupy the rest of this section.

Listing 8.13. The completed Sessions controller `create` action (not yet working).

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by_email(params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      sign_in user
      redirect_to user
    else
      flash.now[:error] = 'Invalid email/password combination'
      render 'new'
    end
  end
  .
  .
  .
end
```

8.2.1 Remember me

We're now in a position to start implementing our signin model, namely, remembering user signin status "forever" and clearing the session only when the user explicitly signs out. The signin functions themselves will end up crossing the traditional Model-View-Controller lines; in particular, several signin functions will need to be available in both controllers and views. You may recall from Section 4.2.5 that Ruby provides a *module* facility for packaging functions together and including them in multiple places, and that's the plan for the authentication functions. We could make an entirely new module for authentication, but the Sessions controller already comes equipped with a module, namely, `SessionsHelper`. Moreover, such helpers are automatically included in Rails views, so all we need to do to use the Sessions helper functions in controllers is to include the module into the Application controller (Listing 8.14).

Listing 8.14. Including the Sessions helper module into the Application controller.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery
  include SessionsHelper
end
```

By default, all the helpers are available in the views but not in the controllers. We need the methods from the Sessions helper in both places, so we have to include it explicitly.

Because HTTP is a *stateless protocol*, web applications requiring user signin must implement a way to track each user's progress from page to page. One technique for maintaining the user signin status is to use a traditional Rails session (via the special **`session`** function) to store a *remember token* equal to the user's id:

```
session[:remember_token] = user.id
```

This **`session`** object makes the user id available from page to page by storing it in a cookie that expires upon browser close. On each page, the application could simply call

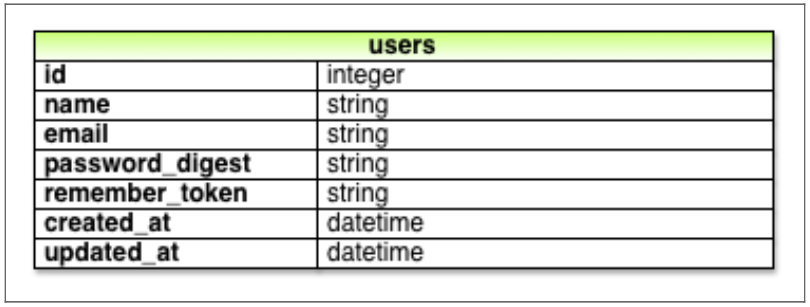
```
User.find(session[:remember_token])
```

to retrieve the user. Because of the way Rails handles sessions, this process is secure; if a malicious user tries to spoof the user id, Rails will detect a mismatch based on a special *session id* generated for each session.

For our application's design choice, which involves *persistent* sessions—that is, signin status that lasts even after browser close—we need to use a *permanent* identifier for the signed-in user. To

accomplish this, we'll generate a unique, secure remember token for each user and store it as a *permanent* cookie rather than one that expires on browser close.

The remember token needs to be associated with a user and stored for future use, so we'll add it as an attribute to the User model, as shown in Figure 8.8.



users	
id	integer
name	string
email	string
password_digest	string
remember_token	string
created_at	datetime
updated_at	datetime

Figure 8.8: The User model with an added `remember_token` attribute.

We'll start with a small addition to the User model specs (Listing 8.15).

Listing 8.15. A first test for the remember token.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:password_confirmation) }
  it { should respond_to(:remember_token) }
  it { should respond_to(:authenticate) }
  .
  .
  .
end
```

We can get this test to pass by generating a remember token at the command line:

```
$ rails generate migration add_remember_token_to_users
```

Next we fill in the resulting migration with the code from Listing 8.16. Note that, because we expect to retrieve users by remember token, we've added an index (Box 6.2) to the `remember_token` column.

Listing 8.16. A migration to add a `remember_token` to the `users` table.
`db/migrate/[timestamp]_add_remember_token_to_users.rb`

```
class AddRememberTokenToUsers < ActiveRecord::Migration
  def change
    add_column :users, :remember_token, :string
    add_index  :users, :remember_token
  end
end
```

Next we update the development and test databases as usual:

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

At this point the User model specs should be passing:

```
$ bundle exec rspec spec/models/user_spec.rb
```

Now we have to decide what to use as a remember token. There are many mostly equivalent possibilities—essentially, any large random string will do just fine. In principle, since the user passwords are securely encrypted, we could use each user's `password_hash` attribute, but it seems like a terrible idea to unnecessarily expose our users' passwords to potential attackers. We'll err on the side of caution and make a custom remember token using the `urlsafe_base64` method from the `SecureRandom` module in the Ruby standard library, which creates a Base64 string safe for use in URIs (and hence safe for use in cookies as well).[3] As of this writing, `SecureRandom.urlsafe_base64` returns a random string of length 16 composed of the characters A–Z, a–z, 0–9, "-", and "_" (for a total of 64 possibilities). This means that the probability of two remember tokens being the same is $1/64^{16} = 2^{-96} \approx 10^{-29}$, which is negligible.

We'll create a remember token using a *callback*, a technique introduced in Section 6.2.5 in the context of email uniqueness. As in that section, we'll use a `before_save` callback, this time to create `remember_token` just before the user is saved.[4] To test for this, we first save the test user and then check that the user's `remember_token` attribute isn't blank. This gives us sufficient flexibility to change the random string if we ever need to. The result appears in Listing 8.17.

Listing 8.17. A test for a valid (nonblank) remember token.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }
  .
  .
  .
  describe "remember token" do
```

```
    before { @user.save }
    its(:remember_token) { should_not be_blank }
  end
end
```

Listing 8.17 introduces the **its** method, which is like **it** but applies the subsequent test to the given attribute rather than the subject of the test. In other words,

```
its(:remember_token) { should_not be_blank }
```

is equivalent to

```
it { @user.remember_token.should_not be_blank }
```

The application code introduces several new elements. First, we add a callback method to create the remember token:

```
before_save :create_remember_token
```

This arranges for Rails to look for a method called **create_remember_token** and run it before saving the user. Second, the method itself is only used internally by the User model, so there's no need to expose it to outside users. The Ruby way to accomplish this is to use the **private** keyword:

```
private

  def create_remember_token
```

```
    # Create the token.
  end
```

All methods defined in a class after **private** are automatically hidden, so that

```
$ rails console
>> User.first.create_remember_token
```

will raise a **NoMethodError** exception.

Finally, the **create_remember_token** method needs to *assign* to one of the user attributes, and in this context it is necessary to use the **self** keyword in front of **remember_token**:

```
def create_remember_token
  self.remember_token = SecureRandom.urlsafe_base64
end
```

(*Note*: If you are using Ruby 1.8.7, you should use **SecureRandom.hex** here instead.) Because of the way Active Record synthesizes attributes based on database columns, without **self** the assignment would create a *local* variable called **remember_token**, which isn't what we want at all. Using **self** ensures that assignment sets the user's **remember_token** so that it will be written to the database along with the other attributes when the user is saved.

Putting this all together yields the User model shown in Listing 8.18.

Listing 8.18. A **before_save** callback to create **remember_token**.
app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  has_secure_password

  before_save { |user| user.email = email.downcase }
  before_save :create_remember_token
  .
  .
  .
  private

    def create_remember_token
      self.remember_token = SecureRandom.urlsafe_base64
    end
end
```

By the way, the extra level of indentation on **`create_remember_token`** is there to make it visually apparent which methods are defined after **`private`**.

Since the **`SecureRandom.urlsafe_base64`** string is definitely *not* blank, the tests for the User model should now be passing:

```
$ bundle exec rspec spec/models/user_spec.rb
```

8.2.2 A working `sign_in` method

Now we're ready to write the first signin element, the **`sign_in`** function itself. As noted above, our desired authentication method is to place a remember token as a cookie on the user's browser, and then use the token to find the user record in the database as the user moves from page to page (implemented in Section 8.2.3). The result, Listing 8.19, introduces two new ideas: the **`cookies`** hash and **`current_user`**.

Listing 8.19. The complete (but not-yet-working) `sign_in` function.

`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    cookies.permanent[:remember_token] = user.remember_token
    self.current_user = user
  end
end
```

Listing 8.19 introduces the `cookies` utility supplied by Rails. We can use `cookies` as if it were a hash; each element in the cookie is itself a hash of two elements, a `value` and an optional `expires` date. For example, we could implement user signin by placing a cookie with value equal to the user's remember token that expires 20 years from now:

```
cookies[:remember_token] = { value:   user.remember_token,
                             expires: 20.years.from_now.utc }
```

(This uses one of the convenient Rails time helpers, as discussed in Box 8.1.)

Box 8.1. Cookies expire `20.years.from_now`

You may recall from Section 4.4.2 that Ruby lets you add methods to *any* class, even built-in ones. In that section, we added a `palindrome?` method to the `String` class (and discovered as a result that `"deified"` is a palindrome), and we also saw how Rails adds a `blank?` method to class `Object` (so that `"".blank?`, `" ".blank?`, and `nil.blank?` are all `true`). The cookie code in Listing 8.19 (which internally sets a cookie that expires `20.years.from_now`) gives yet another example of this practice through one of Rails' *time helpers*, which are methods added to `Fixnum` (the base class for numbers):

```
$ rails console
>> 1.year.from_now
=> Sun, 13 Mar 2011 03:38:55 UTC +00:00
>> 10.weeks.ago
=> Sat, 02 Jan 2010 03:39:14 UTC +00:00
```

Rails adds other helpers, too:

```
>> 1.kilobyte
=> 1024
>> 5.megabytes
=> 5242880
```

These are useful for upload validations, making it easy to restrict, say, image uploads to `5.megabytes`.

Though it must be used with caution, the flexibility to add methods to built-in classes allows for extraordinarily natural additions to plain Ruby. Indeed, much of the elegance of Rails ultimately derives from the malleability of the underlying Ruby language.

This pattern of setting a cookie that expires 20 years in the future became so common that Rails added a special **`permanent`** method to implement it, so that we can simply write

```
cookies.permanent[:remember_token] = user.remember_token
```

Under the hood, using **`permanent`** causes Rails to set the expiration to **`20.years.from_now`** automatically.

After the cookie is set, on subsequent page views we can retrieve the user with code like

```
User.find_by_remember_token(cookies[:remember_token])
```

Of course, `cookies` isn't *really* a hash, since assigning to `cookies` actually *saves* a piece of text on the browser, but part of the beauty of Rails is that it lets you forget about that detail and concentrate on writing the application.

You may be aware that storing authentication cookies on a user's browser and transmitting them over the network exposes an application to a *session hijacking* attack, which involves copying the remember token and using it to sign in as the corresponding user. This attack was publicized by the Firesheep application, which showed that many high-profile sites (including Facebook and Twitter) were vulnerable. The solution is to use site-wide SSL as described in Section 7.4.4.

8.2.3 Current user

Having discussed how to store the user's remember token in a cookie for later use, we now need to learn how to retrieve the user on subsequent page views. Let's look again at the `sign_in` function to see where we are:

```
module SessionsHelper

  def sign_in(user)
    cookies.permanent[:remember_token] = user.remember_token
    self.current_user = user
  end
end
```

Our focus now is the second line:

```
self.current_user = user
```

The purpose of this line is to create **current_user**, accessible in both controllers and views, which will allow constructions such as

```
<%= current_user.name %>
```

and

```
redirect_to current_user
```

The use of **self** is necessary in this context for the same essential reason noted in the discussion leading up to Listing 8.18: without **self**, Ruby would simply create a local variable called **current_user**.

To start writing the code for **current_user**, note that the line

```
self.current_user = user
```

is an *assignment*, which we must define. Ruby has a special syntax for defining such an assignment function, shown in Listing 8.20.

Listing 8.20. Defining assignment to **current_user**.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  def sign_in(user)
    .
    .
    .
  end

  def current_user=(user)
    @current_user = user
  end
end
```

This might look confusing—most languages don't let you use the equals sign in a method definition—but it simply defines a method `current_user=` expressly designed to handle assignment to `current_user`. In other words, the code

```
self.current_user = ...
```

is automatically converted to

```
current_user=(...)
```

thereby invoking the `current_user=` method. Its one argument is the right-hand side of the assignment, in this case the user to be signed in. The one-line method body just sets an instance variable `@current_user`, effectively storing the user for later use.

In ordinary Ruby, we could define a second method, `current_user`, designed to return the value of `@current_user`, as shown in Listing 8.21.

Listing 8.21. A tempting but useless definition for `current_user`.

```
module SessionsHelper

  def sign_in(user)
    .
    .
    .
  end

  def current_user=(user)
    @current_user = user
  end

  def current_user
    @current_user     # Useless! Don't use this line.
  end
end
```

If we did this, we would effectively replicate the functionality of `attr_accessor`, which we saw in Section 4.4.5.[5] The problem is that it utterly fails to solve our problem: with the code in Listing 8.21, the user's signin status would be forgotten: as soon as the user went to another page —*poof!*—the session would end and the user would be automatically signed out. To avoid this problem, we can find the user corresponding to the remember token created by the code in Listing 8.19, as shown in Listing 8.22.

Listing 8.22. Finding the current user using the `remember_token`.

`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def current_user=(user)
    @current_user = user
  end
```

```
  def current_user
    @current_user ||= User.find_by_remember_token(cookies[:remember_token])
  end
end
```

Listing 8.22 uses the common but initially obscure `||=` ("or equals") assignment operator (Box 8.2). Its effect is to set the `@current_user` instance variable to the user corresponding to the remember token, but only if `@current_user` is undefined.[6] In other words, the construction

```
@current_user ||= User.find_by_remember_token(cookies[:remember_token])
```

calls the `find_by_remember_token` method the first time `current_user` is called, but on subsequent invocations returns `@current_user` without hitting the database.[7] This is only useful if `current_user` is used more than once for a single user request; in any case, `find_by_remember_token` will be called at least once every time a user visits a page on the site.

Box 8.2. What the *$@! is ||= ?

The `||=` construction is very Rubyish—that is, it is highly characteristic of the Ruby language—and hence important to learn if you plan on doing much Ruby programming. Though at first it may seem mysterious, *or equals* is easy to understand by analogy.

We start by noting a common idiom for changing a currently defined variable. Many computer programs involve incrementing a variable, as in

```
x = x + 1
```

Most languages provide a syntactic shortcut for this operation; in Ruby (and in C, C++, Perl,

Python, Java, etc.), it appears as follows:

```
x += 1
```

Analogous constructs exist for other operators as well:

```
$ rails console
>> x = 1
=> 1
>> x += 1
=> 2
>> x *= 3
=> 6
>> x -= 7
=> -1
```

In each case, the pattern is that `x = x O y` and `x O= y` are equivalent for any operator `O`.

Another common Ruby pattern is assigning to a variable if it's `nil` but otherwise leaving it alone. Recalling the *or* operator `||` seen in Section 4.2.3, we can write this as follows:

```
>> @user
=> nil
>> @user = @user || "the user"
=> "the user"
>> @user = @user || "another user"
=> "the user"
```

Since `nil` is false in a boolean context, the first assignment is `nil || "the user"`, which evaluates to `"the user"`; similarly, the second assignment is `"the user" || "another user"`, which also evaluates to `"the user"`—since strings are `true` in a boolean context, the series of `||` expressions terminates after the first expression is evaluated. (This practice of evaluating `||` expressions from left to right and stopping on the first true value is known as

short-circuit evaluation.)

Comparing the console sessions for the various operators, we see that `@user = @user || value` follows the `x = x O y` pattern with `||` in the place of `O`, which suggests the following equivalent construction:

```
>> @user ||= "the user"
=> "the user"
```

Voilà !

8.2.4 Changing the layout links

We come finally to a practical application of all our signin/out work: we'll change the layout links based on signin status. In particular, as seen in the Figure 8.3 mockup, we'll arrange for the links to change when users sign in or sign out, and we'll also add links for listing all users and user settings (to be completed in Chapter 9) and one for the current user's profile page. In doing so, we'll get the tests in Listing 8.6 to pass, which means our test suite will be green for the first time since the beginning of the chapter.

The way to change the links in the site layout involves using an if-else branching structure inside of Embedded Ruby:

```
<% if signed_in? %>
  # Links for signed-in users
<% else %>
  # Links for non-signed-in-users
<% end %>
```

This kind of code requires the existence of a `signed_in?` boolean, which we'll now define.

A user is signed in if there is a current user in the session, i.e., if `current_user` is non-`nil`. This requires the use of the "not" operator, written using an exclamation point `!` and usually read as "bang". In the present context, a user is signed in if `current_user` is *not* `nil`, as shown in Listing 8.23.

Listing 8.23. The `signed_in?` helper method.

`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    cookies.permanent[:remember_token] = user.remember_token
    self.current_user = user
  end

  def signed_in?
    !current_user.nil?
  end
  .
  .
  .
end
```

With the `signed_in?` method in hand, we're ready to finish the layout links. There are four new links, two of which are stubbed out (to be completed in Chapter 9):

```
<%= link_to "Users", '#' %>
<%= link_to "Settings", '#' %>
```

The signout link, meanwhile, uses the signout path defined in Listing 8.2:

```
<%= link_to "Sign out", signout_path, method: "delete" %>
```

(Notice that the signout link passes a hash argument indicating that it should submit with an HTTP `DELETE` request.[8]) Finally, we'll add a profile link as follows:

```
<%= link_to "Profile", current_user %>
```

Here we could write

```
<%= link_to "Profile", user_path(current_user) %>
```

but Rails allows us to link directly to the user, in this context automatically converting `current_user` into `user_path(current_user)`.

In the process of putting the new links into the layout, we'll take advantage of Bootstrap's ability to make dropdown menus, which you can read more about on the Bootstrap components page. The full result appears in Listing 8.24. Note in particular the CSS ids and classes related to the Bootstrap dropdown menu.

Listing 8.24. Changing the layout links for signed-in users.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
```

```
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", '#' %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

The dropdown menu requires the use of Bootstrap's JavaScript library, which we can include using the Rails asset pipeline by editing the application JavaScript file, as shown in Listing 8.25.

Listing 8.25. Adding the Bootstrap JavaScript library to `application.js`.

`app/assets/javascripts/application.js`

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require_tree .
```

This uses the Sprockets library to include the Bootstrap JavaScript, which in turn is available

thanks to the `bootstrap-sass` gem from Section 5.1.2.

With the code in Listing 8.24, all the tests should be passing:

```
$ bundle exec rspec spec/
```

Unfortunately, if you actually examine the application in the browser, you'll see that it doesn't yet work. This is because the "remember me" functionality requires the user to have a remember token, but the current user doesn't have one: we created the first user back in Section 7.4.3, long before implementing the callback that sets the remember token. To fix this, we need to save each user to invoke the **`before_save`** callback defined in Listing 8.18, which creates a remember token as a side-effect:

```
$ rails console
>> User.first.remember_token
=> nil
>> User.all.each { |user| user.save(validate: false) }
>> User.first.remember_token
=> "Im9P0kWtZvD0RdyiK9UHtg"
```

Here we've iterated over all the users in case you added more than one while playing with the signup form. Note that we've passed an option to the **`save`** method; as currently written, **`save`** by itself wouldn't work because we haven't included the password or its confirmation. Indeed, for a real site we wouldn't even know any of the passwords, but we would still want to be able to save the users. The solution is to pass **`validate: false`** to tell Active Record skip the validations (Rails API save).

With that change, a signed-in user now sees the new links and dropdown menu defined by Listing 8.24, as shown in Figure 8.9.

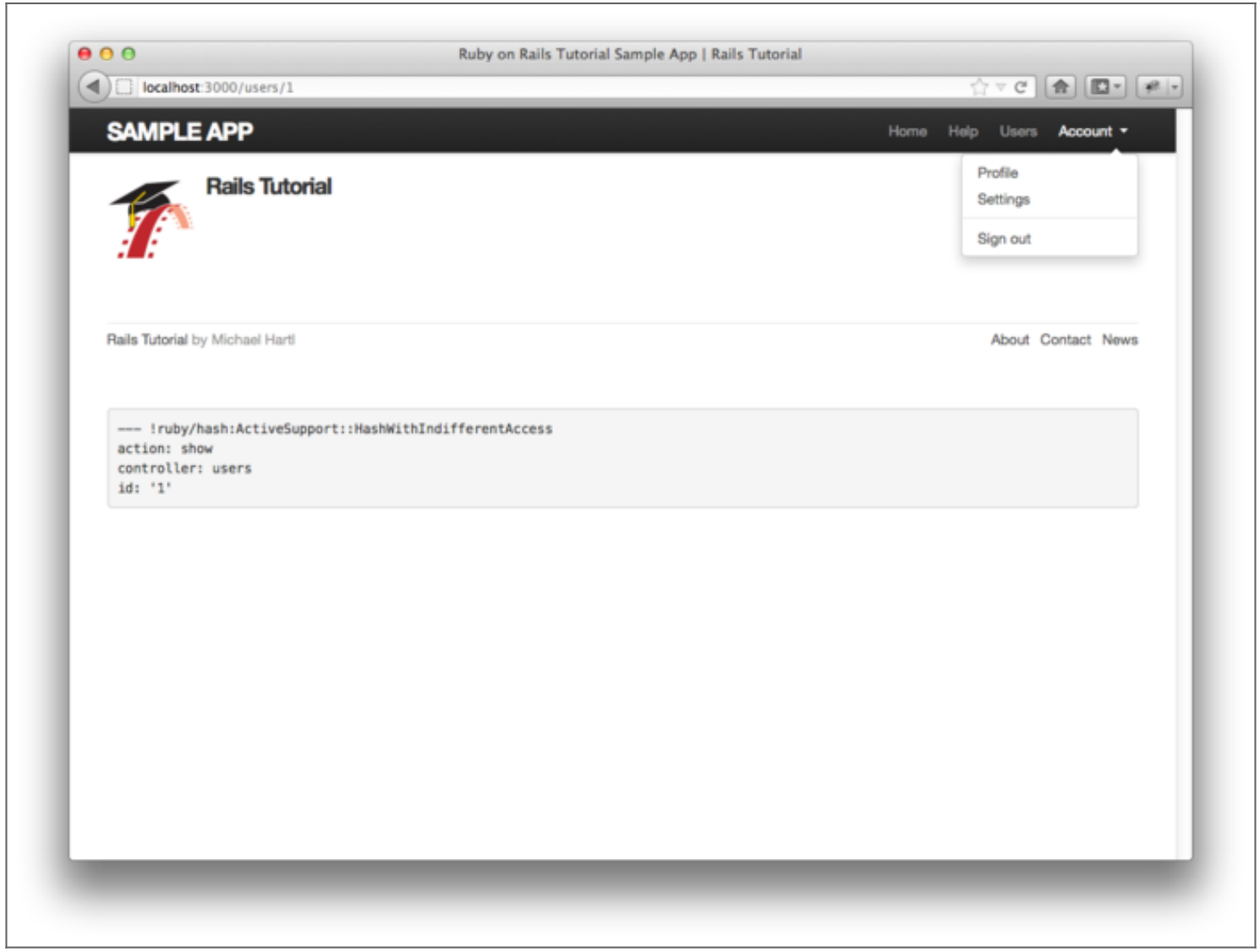


Figure 8.9: A signed-in user with new links and a dropdown menu. (full size)

At this point, you should verify that you can sign in, close the browser, and then still be signed in when you visit the sample application. If you want, you can even inspect the browser cookies to see the result directly (Figure 8.10).

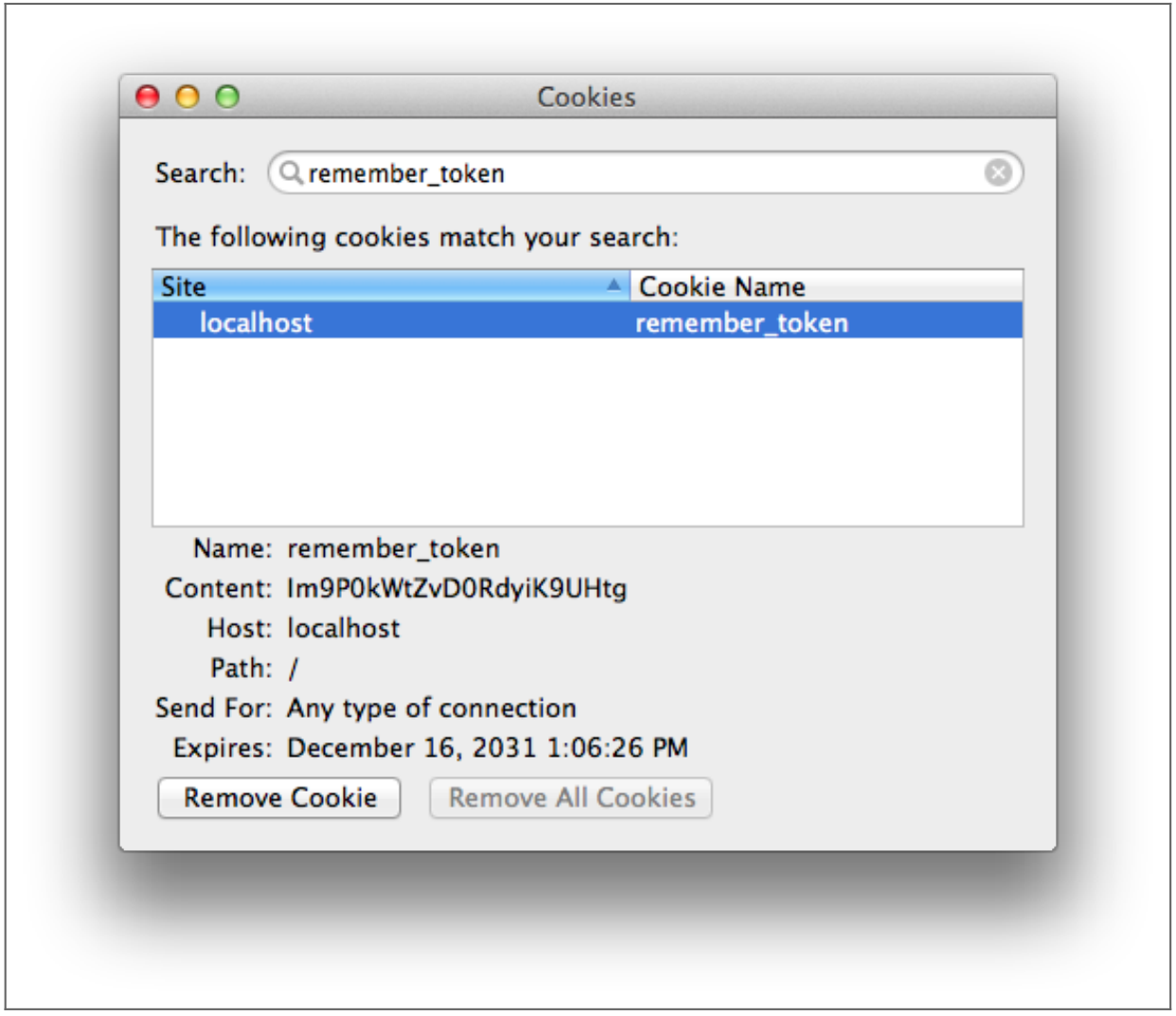


Figure 8.10: The remember token cookie in the local browser. (full size)

8.2.5 Signin upon signup

In principle, although we are now done with authentication, newly registered users might be

confused, as they are not signed in by default. Implementing this is the last bit of polish before letting users sign out. We'll start by adding a line to the authentication tests (Listing 8.26). This includes the "after saving the user" `describe` block from Listing 7.32 (Section 7.6), which you should add to the test now if you didn't already do the corresponding exercise.

Listing 8.26. Testing that newly signed-up users are also signed in.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "after saving the user" do
        .
        .
        .
        it { should have_link('Sign out') }
      end
    end
  end
end
```

Here we've tested the appearance of the signout link to verify that the user was successfully signed in after signing up.

With the `sign_in` method from Section 8.2, getting this test to pass by actually signing in the user is easy: just add `sign_in @user` right after saving the user to the database (Listing 8.27).

Listing 8.27. Signing in the user upon signup.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(params[:user])
    if @user.save
      sign_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end
end
```

8.2.6 Signing out

As discussed in Section 8.1, our authentication model is to keep users signed in until they sign out explicitly. In this section, we'll add this necessary signout capability.

So far, the Sessions controller actions have followed the RESTful convention of using `new` for a signin page and `create` to complete the signin. We'll continue this theme by using a `destroy` action to delete sessions, i.e., to sign out. To test this, we'll click on the "Sign out" link and then look for the reappearance of the signin link (Listing 8.28).

Listing 8.28. A test for signing out a user.

spec/requests/authentication_pages_spec.rb

```
require 'spec_helper'

describe "Authentication" do
  .
```

```
  .
  .
  describe "signin" do
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "followed by signout" do
        before { click_link "Sign out" }
        it { should have_link('Sign in') }
      end
    end
  end
end
```

As with user signin, which relied on the `sign_in` function, user signout just defers to a `sign_out` function (Listing 8.29).

Listing 8.29. Destroying a session (user signout).
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def destroy
    sign_out
    redirect_to root_url
  end
end
```

As with the other authentication elements, we'll put `sign_out` in the Sessions helper module. The implementation is simple: we set the current user to `nil` and use the `delete` method on cookies to remove the remember token from the session (Listing 8.30). (Setting the current user to `nil` isn't

currently necessary because of the immediate redirect in the `destroy` action, but it's a good idea in case we ever want to use `sign_out` without a redirect.)

Listing 8.30. The `sign_out` method in the Sessions helper module.

`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    cookies.permanent[:remember_token] = user.remember_token
    self.current_user = user
  end
  .
  .
  .
  def sign_out
    self.current_user = nil
    cookies.delete(:remember_token)
  end
end
```

This completes the signup/signin/signout triumvirate, and the test suite should pass:

```
$ bundle exec rspec spec/
```

It's worth noting that our test suite covers most of the authentication machinery, but not all of it. For instance, we don't test how long the "remember me" cookie lasts or whether it gets set at all. It is possible to do so, but experience shows that direct tests of cookie values are brittle and have a tendency to rely on implementation details that sometimes change from one Rails release to the next. The result is breaking tests for application code that still works fine. By focusing on high-level functionality—verifying that users can sign in, stay signed in from page to page, and can sign out—we test the core application code without focusing on less important details.

8.3 Introduction to Cucumber (optional)

Having finished the foundation of the sample application's authentication system, we're going to take this opportunity to show how to write signin tests using Cucumber, a popular tool for behavior-driven development that enjoys significant popularity in the Ruby community. This section is optional and can be skipped without loss of continuity.

Cucumber allows the definition of plain-text *stories* describing application behavior. Many Rails programmers find Cucumber especially convenient when doing client work; since they can be read even by non-technical users, Cucumber tests can be shared with (and can sometimes even be written by) the client. Of course, using a testing framework that isn't pure Ruby has a downside, and I find that the plain-text stories can be a bit verbose. Nevertheless, Cucumber does have a place in the Ruby testing toolkit, and I especially like its emphasis on high-level behavior over low-level implementation.

Since the emphasis in this book is on RSpec and Capybara, the presentation that follows is necessarily superficial and incomplete, and will be a bit light on explanation. Its purpose is just to give you a taste of Cucumber (crisp and juicy, no doubt)—if it strikes your fancy, there are entire books on the subject waiting to satisfy your appetite. (I particularly recommend *The RSpec Book* by David Chelimsky, *Rails 3 in Action* by Ryan Bigg and Yehuda Katz, and *The Cucumber Book* by Matt Wynne and Aslak Hellesøy.)

8.3.1 Installation and setup

To install Cucumber, first add the `cucumber-rails` gem and a utility gem called `database_cleaner` to the `:test` group in the `Gemfile` (Listing 8.31).

Listing 8.31. Adding the `cucumber-rails` gem to the `Gemfile`.

```
.
.
.
group :test do
  .
  .
  .
  gem 'cucumber-rails', '1.2.1', require: false
  gem 'database_cleaner', '0.7.0'
end
.
.
.
```

Then install as usual:

```
$ bundle install
```

To set up the application to use Cucumber, we next generate some necessary support files and directories:

```
$ rails generate cucumber:install
```

This creates a **`features/`** directory where the files associated with Cucumber will live.

8.3.2 Features and steps

Cucumber features are descriptions of expected behavior using a plain-text language called Gherkin. Gherkin tests read much like well-written RSpec examples, but because they are plain-text they are more accessible to those more comfortable reading English than Ruby code.

Our Cucumber features will implement a subset of the signin examples in Listing 8.5 and Listing 8.6. To get started, we'll create a file in the `features/` directory called `signing_in.feature`.

Cucumber features start with a short description of the feature, as follows:

```
Feature: Signing in
```

Then they add individual *scenarios*. For example, to test unsuccessful signin, we could write the following scenario:

```
  Scenario: Unsuccessful signin
    Given a user visits the signin page
    When he submits invalid signin information
    Then he should see an error message
```

Similarly, to test successful signin, we could add this:

```
  Scenario: Successful signin
    Given a user visits the signin page
      And the user has an account
      And the user submits valid signin information
    Then he should see his profile page
      And he should see a signout link
```

Collecting these together yields the Cucumber feature file shown in Listing 8.32.

Listing 8.32. Cucumber features to test user signin.
`features/signing_in.feature`

```
Feature: Signing in

  Scenario: Unsuccessful signin
    Given a user visits the signin page
    When he submits invalid signin information
    Then he should see an error message

  Scenario: Successful signin
    Given a user visits the signin page
      And the user has an account
      And the user submits valid signin information
    Then he should see his profile page
      And he should see a signout link
```

To run the features, we use the **`cucumber`** executable:

```
$ bundle exec cucumber features/
```

Compare this to

```
$ bundle exec rspec spec/
```

In this context, it's worth noting that, like RSpec, Cucumber can be invoked using a Rake task:

```
$ bundle exec rake cucumber
```

(For reasons that escape me, this is sometimes written as **`rake cucumber:ok`**.)

All we've done so far is write some plain text, so it shouldn't be surprising that the Cucumber

scenarios aren't yet passing. To get the test suite to green, we need to add a *step* file that maps the plain-text lines to Ruby code. The step file goes in the **`features/step_definitions`** directory; we'll call it **`authentication_steps.rb`**.

The **`Feature`** and **`Scenario`** lines are mainly for documentation, but each of the other lines needs some corresponding Ruby. For example, the line

```
Given a user visits the signin page
```

in the feature file gets handled by the step definition

```
Given /^a user visits the signin page$/ do
  visit signin_path
end
```

In the feature, **`Given`** is just a string, but in the step file **`Given`** is a *method* that takes a regular expression and a block. The regex matches the text of the line in the scenario, and the contents of the block are the Ruby code needed to implement the step. In this case, "a user visits the signin page" is implemented by

```
visit signin_path
```

If this looks familiar, it should: it's just Capybara, which is included by default in Cucumber step files. The next two lines should also look familiar; the scenario steps

```
When he submits invalid signin information
Then he should see an error message
```

in the feature file are handled by these steps:

```
When /^he submits invalid signin information$/ do
  click_button "Sign in"
end

Then /^he should see an error message$/ do
  page.should have_selector('div.alert.alert-error')
end
```

The first step also uses Capybara, while the second uses Capybara's `page` object with RSpec. Evidently, all the testing work we've done so far with RSpec and Capybara is also useful with Cucumber.

The rest of the steps proceed similarly. The final step definition file appears in Listing 8.33. Try adding one step at a time, running

```
$ bundle exec cucumber features/
```

each time until the tests pass.

Listing 8.33. The complete steps needed to get the signin features to pass.
features/step_definitions/authentication_steps.rb

```
Given /^a user visits the signin page$/ do
  visit signin_path
end

When /^he submits invalid signin information$/ do
  click_button "Sign in"
```

```
end

Then /^he should see an error message$/ do
  page.should have_selector('div.alert.alert-error')
end

Given /^the user has an account$/ do
  @user = User.create(name: "Example User", email: "user@example.com",
                      password: "foobar", password_confirmation: "foobar")
end

When /^the user submits valid signin information$/ do
  fill_in "Email",    with: @user.email
  fill_in "Password", with: @user.password
  click_button "Sign in"
end

Then /^he should see his profile page$/ do
  page.should have_selector('title', text: @user.name)
end

Then /^he should see a signout link$/ do
  page.should have_link('Sign out', href: signout_path)
end
```

With the code in Listing 8.33, the Cucumber tests should pass:

```
$ bundle exec cucumber features/
```

8.3.3 Counterpoint: RSpec custom matchers

Having written some simple Cucumber scenarios, it's worth comparing the result to the equivalent RSpec examples. First, take a look at the Cucumber feature in Listing 8.32 and the corresponding step definitions in Listing 8.33. Then take a look at the RSpec request specs (integration tests):

```
describe "Authentication" do

  subject { page }

  describe "signin" do
    before { visit signin_path }

    describe "with invalid information" do
      before { click_button "Sign in" }

      it { should have_selector('title', text: 'Sign in') }
      it { should have_selector('div.alert.alert-error', text: 'Invalid') }
    end

    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        fill_in "Email",    with: user.email
        fill_in "Password", with: user.password
        click_button "Sign in"
      end

      it { should have_selector('title', text: user.name) }
      it { should have_selector('a', 'Sign out', href: signout_path) }
    end
  end
end
```

You can see how a case could be made for either Cucumber or integration tests. Cucumber features are easily readable, but they are entirely separate from the code that implements them—a property that cuts both ways. I find that Cucumber is easy to read and awkward to write, while integration tests are (for a programmer) a little harder to read and *much* easier to write.

One nice effect of Cucumber's separation of concerns is that it operates at a higher level of abstraction. For example, we write

```
Then he should see an error message
```

to express the expectation of seeing an error message, and

```
Then /^he should see an error message$/ do
  page.should have_selector('div.alert.alert-error', text: 'Invalid')
end
```

to implement the test. What's especially convenient about this is that only the second element (the step) is dependent on the implementation, so that if we change, e.g., the CSS class used for error messages, the feature file would stay the same.

In this vein, it might make you unhappy to write

```
should have_selector('div.alert.alert-error', text: 'Invalid')
```

in a bunch of places, when what you really want is to indicate that the page should have an error message. This practice couples the test tightly to the implementation, and we would have to change it everywhere if the implementation changed. In the context of pure RSpec, there is a solution, which is to use a *custom matcher*, allowing us to write the following instead:

```
should have_error_message('Invalid')
```

We can define such a matcher in the same utilities file where we put the **`full_title`** test helper in Section 5.3.4. The code itself looks like this:

```
RSpec::Matchers.define :have_error_message do |message|
  match do |page|
```

```
    page.should have_selector('div.alert.alert-error', text: message)
  end
end
```

We can also define helper functions for common operations:

```
def valid_signin(user)
  fill_in "Email",    with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
end
```

The resulting support code is shown in Listing 8.34 (which incorporates the results of Listing 5.37 and Listing 5.38 from Section 5.6). I find this approach to be more flexible than Cucumber step definitions, particularly when the matchers or should helpers naturally take an argument, such as `valid_signin(user)`. Step definitions can replicate this functionality with regex matchers, but I generally find this approach to be more (cu)cumbersome.

Listing 8.34. Adding a helper method and a custom RSpec matcher.
spec/support/utilities.rb

```
include ApplicationHelper

def valid_signin(user)
  fill_in "Email",    with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
end

RSpec::Matchers.define :have_error_message do |message|
  match do |page|
    page.should have_selector('div.alert.alert-error', text: message)
  end
end
```

With the code in Listing 8.34, we can write

```
it { should have_error_message('Invalid') }
```

and

```
describe "with valid information" do
  let(:user) { FactoryGirl.create(:user) }
  before { valid_signin(user) }
  .
  .
  .
```

There are many other examples of coupling between our tests and the site's implementation. Sweeping through the current test suite and decoupling the tests from the implementation details by making custom matchers and methods is left as an exercise (Section 8.5).

8.4 Conclusion

We've covered a lot of ground in this chapter, transforming our promising but unformed application into a site capable of the full suite of registration and login behaviors. All that is needed to complete the authentication functionality is to restrict access to pages based on signin status and user identity. We'll accomplish this task en route to giving users the ability to edit their information and giving administrators the ability to remove users from the system, which are the main goals of Chapter 9.

Before moving on, merge your changes back into the master branch:

```
$ git add .
$ git commit -m "Finish sign in"
$ git checkout master
$ git merge sign-in-out
```

Then push up the remote GitHub repository and the Heroku production server:

```
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

If you've created any users on the production server, I recommend following the steps in Section 8.2.4 to give each user a valid remember token. The only difference is using the Heroku console instead of the local one:

```
$ heroku run console
>> User.all.each { |user| user.save(validate: false) }
```

8.5 Exercises

1. Refactor the signin form to use `form_tag` in place of `form_for`. Make sure the test suite still passes. *Hint*: See the RailsCast on authentication in Rails 3.1, and note in particular the change in the structure of the `params` hash.

2. Following the example in Section 8.3.3, go through the user and authentication request specs (i.e., the files currently in the `spec/requests` directory) and define utility functions in `spec/support/utilities.rb` to decouple the tests from the implementation. *Extra credit:* Organize the support code into separate files and modules, and get everything to work by including the modules properly in the spec helper file.

1. Another common model is to expire the session after a certain amount of time. This is especially appropriate on sites containing sensitive information, such as banking and financial trading accounts. ↑
2. Image from http://www.flickr.com/photos/hermanusbackpackers/3343254977/. ↑
3. This choice is based on the RailsCast on remember me. ↑
4. For more details on the kind of callbacks supported by Active Record, see the discussion of callbacks at the Rails Guides. ↑
5. In fact, the two are exactly equivalent; `attr_accessor` is merely a convenient way to create just such getter/setter methods automatically. ↑
6. Typically, this means assigning to variables that are initially `nil`, but note that `false` values will also be overwritten by the `||=` operator. ↑
7. This is an example of *memoization*, which we discussed before in Box 6.3. ↑
8. Web browsers can't actually issue `DELETE` requests; Rails fakes it with JavaScript. ↑