Chapter 9 Sign in, sign out

Now that new users can sign up for our site (Chapter 8), 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 depending 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 11, we’ll use the identity of a signed-in user to create microposts associated with that user, and in Chapter 12 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 10, 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 in Chapter 10) to delete users from the database.

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
```

9.1 Sessions

A *session* is a semi-permanent connection between two computers, such as a client computer running a web browser and a server running Rails. 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 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”,[2] clearing the session only when the user explicitly signs out.

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. We will therefore need a Sessions controller with `new`, `create`, and `destroy` actions. Unlike the case of the Users controller, which uses a database back-end (via the User model) to persist data, the Sessions controller 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. (Much of this work parallels user signup from Chapter 8.) We’ll then complete user signin with the necessary cookie-manipulation code in Section 9.3.

9.1.1 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 9.2 and Section 9.3), and signing out is handled by sending a `DELETE` request to the `destroy` action (Section 9.4). (Recall the association of HTTP verbs with REST actions from Table 6.2.) Since we know that we'll need a `new` action, we can create it when we generate the Sessions controller (just as with the Users controller in Listing 5.22):3

```
$ script/generate rspec_controller Sessions new
$ rm -rf spec/views
```

Now, as with the signup form in Section 8.1, we create a new file for the Sessions controller specs and add a couple tests for the `new` action and corresponding view (Listing 9.1). (This pattern should start to look familiar by now.)

Listing 9.1. Tests for the `new` session action and view.
`spec/controllers/sessions_controller_spec.rb`

```
require 'spec_helper'

describe SessionsController do
  integrate_views

  describe "GET 'new'" do

    it "should be successful" do
      get :new
      response.should be_success
    end

    it "should have the right title" do
      get :new
      response.should have_tag("title", /sign in/i)
    end
  end
end
```

As in the case of `/users/new` from Section 5.3.1, the first test already passes because `/sessions/new` matches the default `/controller/action/id` pattern (with `nil` id), but it's a good practice to add a RESTful route as well.4 While we're at it, we'll create all the actions needed throughout the chapter as well. We generally follow the example from Listing 6.24, but in this case we define only the particular actions we need, i.e., `new`, `create`, and `destroy`, and also add named routes for signin and signout (Listing 9.2).

Listing 9.2. Adding a resource to get the standard RESTful actions for sessions.
`config/routes.rb`

```
ActionController::Routing::Routes.draw do |map|
  map.resources :users
  map.resources :sessions, :only => [:new, :create, :destroy]
  map.signin  '/signin',  :controller => 'sessions', :action => 'new'
  map.signout '/signout', :controller => 'sessions', :action => 'destroy'
  .
  .
  .
end
```

As you can see, the `map.resources` method can take an options hash, which in this case has key `:only` and value equal to an array of the actions the Sessions controller has to respond to. The resources defined in Listing 9.2 provide URLs and actions similar to those for users (Table 6.2), as shown in Table 9.1.

HTTP request	**URL**	**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 9.1: RESTful routes provided by the sessions rules in Listing 9.2.

We can get the second test in Listing 9.1 to pass by adding the proper title instance variable to the `new` action, as shown in Listing 9.3 (which also defines the `create` and `destroy` actions for future reference).

Listing 9.3. Adding the title for the signin page.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
    @title = "Sign in"
  end

  def create
  end

  def destroy
  end
end
```

With that, the tests in Listing 9.1 should be passing, and we're ready to make the actual signin form.

9.1.2 Signin form

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. A mockup appears in Figure 9.1.

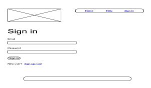


Figure 9.1: A mockup of the signin form. (full size)

Recall from Listing 8.2 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 new session 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 URL `/users`, in the case of sessions we need to indicate both the *name* of the resource and the appropriate URL:

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

Since we're authenticating users with email address and password, we need a field for each one inside the form; the result appears in Listing 9.4.

Listing 9.4. Code for the signin form.
`app/views/sessions/new.html.erb`

```
<h2>Sign in</h2>

<% form_for(:session, :url => sessions_path) do |f| %>
  <div class="field">
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </div>
  <div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </div>
  <div class="actions">
    <%= f.submit "Sign in" %>
  </div>
<% end %>

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

With the code in Listing 9.4, the signin form appears as in Figure 9.2.

Figure 9.2: The signin form (/sessions/new). (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 9.5).

Listing 9.5. HTML for the signin form produced by Listing 9.4.

```
<form action="/sessions" method="post">
  <div class="field">
    <label for="session_email">Email</label><br />
    <input id="session_email" name="session[email]" size="30" type="text" />
  </div>
  <div class="field">
    <label for="session_password">Password</label><br />
    <input id="session_password" name="session[password]" size="30"
           type="password" />
  </div>
  <div class="actions">
    <input id="session_submit" name="commit" type="submit" value="Sign in" />
  </div>
</form>
```

Comparing Listing 9.5 with Listing 8.5, 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. Handling this submission—and, in particular, authenticating users based on the submitted email and password—is the goal of the next two sections.

9.2 Signin failure

As in the case of creating users (signup), the first step in creating sessions (signin) is to handle *invalid* input. 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 9.3.) Finally, we'll lay the foundation for successful signin (Section 9.3) by evaluating each signin submission based on the validity of its email/password combination.

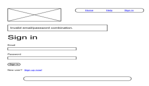


Figure 9.3: A mockup of signin failure. (full size)

9.2.1 Reviewing form submission

Let's start by defining a minimalist `create` action for the Sessions controller (Listing 9.6), which does nothing but render the `new` view. Submitting the `/sessions/new` form with blank fields then yields the result shown in Figure 9.4.

Listing 9.6. A preliminary version of the Sessions `create` action.
`app/controllers/sessions_controller.rb`

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

Figure 9.4: The initial failed signin, with `create` as in Listing 9.6. (full size)

Carefully inspecting the debug information in Figure 9.4 shows that, as hinted at the end of Section 9.1.2, the submission results in a `params` hash containing the email and password under the key `:session`:

```
--- !map:HashWithIndifferentAccess
commit: Sign in
session: !map:HashWithIndifferentAccess
  password: ""
  email: ""
authenticity_token: LImP18WPIUYQSPbd1nC7ekhZSOFb4ny04ISzFYAALsE=
action: create
controller: sessions
```

As with the case of user signup (Figure 8.6) these parameters form a *nested* hash like the one we saw in Listing 4.5. 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 have already developed exactly the method needed: `User.authenticate` from Section 7.2.4 (Listing 7.12). Recalling that `authenticate` returns `nil` for an invalid authentication, our strategy for user signin can be summarized as follows:

```
def create
  user = User.authenticate(params[:session][:email],
                           params[:session][:password])
  if user.nil?
    # Create an error message and re-render the signin form.
  else
```

```
    # Sign the user in and redirect to the user's show page.
  end
end
```

9.2.2 Failed signin (test and code)

In order to handle a failed signin attempt, first we need to determine that it's a failure. We thus have an expectation that the `User.authenticate` method will be called in the `create` action; we can express this in test code using the same type of message expectation first seen in Listing 8.6, where we indicated the expectation that `@user.save` would be called:

```
@user.should_receive(:save).and_return(false)
```

To describe invalid signin, we'll use a similar construction, in this case expressing the expectation that the `User` model will receive the `authenticate` method, as shown in Listing 9.7. (As with the user signup specs, Listing 9.7 doesn't actually test that the form has right fields for signin or that form submissions actually work. We'll test these aspects of signin with integration tests in Section 9.4.4.)

Listing 9.7. Tests for a failed signin attempt.
`spec/controllers/sessions_controller_spec.rb`

```
require 'spec_helper'

describe SessionsController do
  integrate_views
  .
  .
  .
  describe "POST 'create'" do

    describe "invalid signin" do

      before(:each) do
        @attr = { :email => "email@example.com", :password => "password" }
        User.should_receive(:authenticate).
             with(@attr[:email], @attr[:password]).
             and_return(nil)
      end

      it "should re-render the new page" do
        post :create, :session => @attr
        response.should render_template('new')
      end

      it "should have the right title" do
        post :create, :session => @attr
        response.should have_tag("title", /sign in/i)
      end
    end
  end
end
```

Here the tests follow the example from the analogous tests for user signup (Listing 8.6). The main difference is in the message expectation, which adds the `with` method:5

```
User.should_receive(:authenticate).
     with(@attr[:email], @attr[:password]).
     and_return(nil)
```

In English, this says that the User model should receive the `authenticate` method with the email and password attributes from the form submission. Since this is an invalid signin, we arrange for the return value be `nil`. (Note that we *don't* have to check that the email/password combination is actually invalid, since that logic is covered by the User model tests in Listing 7.11. Recall the message expectation subtlety from Box 8.1: `and_return(nil)` isn't an expectation; it's a guarantee.)

The application code needed to get these tests to pass appears in Listing 9.8. As promised in Section 9.2.1, we extract the submitted email address and password from the `params` hash, and then pass them to the `User.authenticate` method, thereby fulfilling the message expectation from the test. If the user is not authenticated (i.e., if it's `nil`), we set the title and re-render the signin form.6 We'll handle the other branch of the if-else statement in Section 9.3; for now we'll just leave a descriptive comment.

Listing 9.8. Code for a failed signin attempt.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.authenticate(params[:session][:email],
                             params[:session][:password])
    if user.nil?
      flash.now[:error] = "Invalid email/password combination."
      @title = "Sign in"
      render 'new'
    else
      # Sign the user in and redirect to the user's show page.
    end
  end
  .
  .
  .
end
```

Recall from Section 8.4.2 that we displayed signup errors using the User model error messages. Since the session isn't an Active Record model, this strategy won't work here, so instead we've put a message in the flash (or, rather, in `flash.now`;7 see Box 9.1). Thanks to the flash message display in the site layout (Listing 8.15), the `flash[:error]` message automatically gets displayed; thanks to the Blueprint CSS, it automatically gets nice styling (Figure 9.5).

Box 9.1.Flash dot now

There's a subtle difference between `flash` and `flash.now`. The `flash` variable is designed to be used before a *redirect*, and it persists on the resulting page for one request—that is, it appears once, and disappears when you click on another link. Unfortunately, this means that if we *don't* redirect, and instead simply render a page (as in Listing 9.8), the flash message persists for *two* requests: it appears on the rendered page but is still waiting for a "redirect" (i.e., a second request), and thus appears *again* if you click a link.

To avoid this weird behavior, when `render`ing rather than `redirect`ing we use `flash.now` instead of `flash`. The `flash.now` object is specifically designed for displaying flash messages on rendered pages. If you ever find yourself wondering why a flash message is showing up where you don't expect it, the chances are good that you need to replace `flash` with `flash.now`.

Figure 9.5: A failed signin (with a flash message). (full size)

9.3 Signin success

Having handled a failed signin, we now need to actually sign a user in. A hint of where we're going—the user profile page, with modified navigation links—is mocked up in Figure 9.6.8 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.

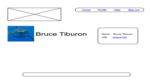


Figure 9.6: A mockup of the user profile after a successful signin (with updated nav links). (full size)

9.3.1 The completed `create` action

Filling in the area now occupied by the signin comment (Listing 9.8) is simple: upon successful signin, we sign the user in using the `sign_in` function, and then redirect to the profile page (Listing 9.9). 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 9.9. The completed Sessions controller `create` action (not yet working).
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.authenticate(params[:session][:email],
                             params[:session][:password])
    if user.nil?
      flash.now[:error] = "Invalid email/password combination."
      @title = "Sign in"
      render 'new'
    else
      sign_in user
      redirect_to user
    end
  end
  .
  .
  .
end
```

Even though we lack the `sign_in` function, we can still write the tests (Listing 9.10). (We'll fill in the body of the first test in Section 9.3.4.)

Listing 9.10. Pending tests for user signin (to be completed in Section 9.3.4).
`spec/controllers/sessions_controller_spec.rb`

```
describe SessionsController do
  .
  .
  .
  describe "POST 'create'" do
    .
    .
    .
    describe "with valid email and password" do

      before(:each) do
        @user = Factory(:user)
        @attr = { :email => @user.email, :password => @user.password }
        User.should_receive(:authenticate).
             with(@user.email, @user.password).
             and_return(@user)
      end

      it "should sign the user in" do
        post :create, :session => @attr
        # Fill in with tests for a signed-in user.
      end

      it "should redirect to the user show page" do
        post :create, :session => @attr
        response.should redirect_to(user_path(@user))
      end
    end
  end
end
```

In analogy with the message expectation in Listing 9.7, Listing 9.10 sets up the expectation that `User.authenticate` gets called with the right email/password combination, in this case (for a valid combination) returning the factory user:

```
@user = Factory(:user)
@attr = { :email => @user.email, :password => @user.password }
User.should_receive(:authenticate).
     with(@user.email, @user.password).
```

```
      and_return(@user)
```

9.3.2 Remember me

There are many moving parts in the signin machinery. Our strategy is to push a few elements onto the stack, and then pop them off as the section unfolds. Have patience if you get bogged down; you'll probably have to get all the way through before you see how it all fits together.

The signin functions 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, 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 9.11).

Listing 9.11. Including the Sessions helper module into the Application controller.
`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time
  include SessionsHelper
  .
  .
  .
end
```

Here the line `helper :all` includes the helpers in all 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.

Box 9.2.Sessions and cookies

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 can simply call

```
  User.find_by_id(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—storing the user id is a security hole. As soon as we break the tie between the special session id and the stored user id, a malicious user could sign in as that user with a `remember_token` equal to the user's id. To fix this flaw, we generate a unique, secure remember token for each user based on the user's salt and id. Moreover, a *permanent* remember token would also represent a security hole—by inspecting the browser cookies, a malicious user could find the token and then use it to sign in from any other computer, any time. We solve this by adding a *timestamp* to the token, and reset the token every time the user signs into the application. This results in a persistent session essentially impervious to attack.

Now we're ready for the first signin element, the `sign_in` function itself. Our authentication method is to place a *remember token* as a cookie on the user's browser (Box 9.2), and then use the token to find the user record in the database as the user moves from page to page (implemented in Section 9.3.4). The result, Listing 9.12, pushes three things onto the stack: the `remember_me!` method, the `cookies` hash, and `current_user`. Let's start popping them off.

Listing 9.12. The complete (but not-yet-working) `sign_in` function.
`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  def sign_in(user)
    user.remember_me!
    cookies[:remember_token] = { :value   => user.remember_token,
                                 :expires => 20.years.from_now.utc }
    self.current_user = user
  end
end
```

We begin with the `remember_me!` method. This method's job is to save the user's remember token to the database. We can outline the expected behavior using some tests, which, because `remember_me!` acts on users, are tests on the User model (Listing 9.13).9

Listing 9.13. Tests for the `remember_me!` method.
`spec/models/user_spec.rb`

```
describe User do
  .
  .
  .
  describe "remember me" do

    before(:each) do
      @user = User.create!(@attr)
    end

    it "should have a remember token" do
      @user.should respond_to(:remember_token)
    end

    it "should have a remember_me! method" do
      @user.should respond_to(:remember_me!)
    end
    it "should set the remember token" do
      @user.remember_me!
      @user.remember_token.should_not be_nil
    end
  end
end
```

The `remember_me!` method itself then appears as in Listing 9.14 (with the `has_password?` method shown to give you some context).

Listing 9.14. The `remember_me!` method.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  before_save :encrypt_password
  .
  .
  .
  def has_password?(submitted_password)
    encrypted_password == encrypt(submitted_password)
  end

  def remember_me!
    self.remember_token = encrypt("#{salt}--#{id}--#{Time.now.utc}")
    save_without_validation
  end
  .
  .
  .
  private

    def encrypt_password
      unless password.nil?
        self.salt = make_salt
        self.encrypted_password = encrypt(password)
      end
    end
  .
  .
  .
end
```

Here we set the remember token for each user using the `encrypt` function from Section 7.2.3. Since this token will be placed on the user's browser, it needs to be both *unique* and *secure*. We accomplish this by encrypting the `salt` with the `id`, thereby guaranteeing that the token is different for each user and is impervious to a rainbow attack (Section 7.2.2). As promised in Box 9.2, we also add a timestamp (`Time.now.utc`) as a security precaution, thereby ensuring that the token changes every time the user signs in.

In order to establish the connection between the remember token and the user, we'll store the remember token in the database and at some point (Listing 9.19) use the `find_by_remember_token` method (created automatically by Active Record) to retrieve the user. This means that we must *save* the user after setting the remember token, but recall that users have a virtual `password` attribute which is required to be present (Section 7.1.1). When calling the `remember_me!` method, there is no password, so the validations will fail if we simply use `save`. We bypass this in Listing 9.14 using the special `save_without_validation` method, which saves the user without running the validations.

Finally, note that we've updated the `encrypt_password` method from Chapter 6. Even though we skip the validation step using `save_without_validation`, the `before_save` callback still fires; with the original code from Listing 7.10,

```
def encrypt_password
  self.salt = make_salt
  self.encrypted_password = encrypt(password)
end
```

when `password` is `nil` the user's encrypted password would be reset to `encrypt(nil)`, thus preventing the user from ever signing in again. (Section 9.6 has an exercise to write an integration test that catches this problem.) To prevent this issue, we skip the encryption step when `password` is `nil`:10

```
def encrypt_password
  unless password.nil?
    self.salt = make_salt
    self.encrypted_password = encrypt(password)
  end
end
```

With the code in Listing 9.14, we've pushed the `remember_token` onto the stack. Since it needs to be stored in the database, it should be an attribute on the User model; we can add it with a migration:

```
$ script/generate migration add_remember_token_to_users remember_token:string
```

This generates a nearly complete migration; the only extra step (shown in Listing 9.15) is to add an index on the `remember_token` column. (Recall from Box 6.2 that an index is important to avoid a full-table scan when finding by a particular attribute.)

Listing 9.15. The migration for the `remember_token` attribute.
`db/migrate/<timestamp>_add_remember_token_to_users.rb`

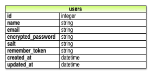
```
class AddRememberTokenToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :remember_token, :string
    add_index  :users, :remember_token
  end

  def self.down
    remove_column :users, :remember_token
  end
end
```

Then run the migration as usual:

```
$ rake db:migrate
```

The resulting data model is shown in Figure 9.7.



users	
id	integer
name	string
email	string
encrypted_password	string
salt	string
remember_token	string
created_at	datetime
updated_at	datetime

Figure 9.7: The User model with an added remember token.

Since we’ve changed the data model by adding a new column, we also need to prepare the test database:

```
$ rake db:test:prepare
```

At this point, although the tests in Listing 9.10 don’t yet pass, all the tests in Listing 9.13 should be passing, as you can see by running the User model specs directly:

```
$ spec spec/models/user_spec.rb
```

9.3.3 Cookies

Let’s look again at the `sign_in` function to see where we are:

```
module SessionsHelper

  def sign_in(user)
    user.remember_me!
    cookies[:remember_token] = { :value   => user.remember_token,
                                 :expires => 20.years.from_now.utc }
    self.current_user = user
  end
end
```

The whole point of the previous section was to get the first line to work by adding the `remember_me!` method to the User model. Now that we’ve done that, the cookie line will work as well, since after the call to `remember_me!` the user has a valid remember token.

As you can see, the `cookies` utility can be used like a hash; in this case, we define a hash-of-hashes—the `cookies` assignment makes a hash with key `:remember_token` and value equal to another hash, consisting of the user’s remember token set to expire twenty years from now (effectively lasting “forever”). (See Box 9.3 for more on the `20.years.from_now` syntax.) Of course, `cookies` isn’t *really* a hash, since assigning to `cookies` actually *saves* a piece of text on the browser (as seen in Figure 9.8), but part of the beauty of Rails is that it lets you forget about that detail and concentrate on writing the application.

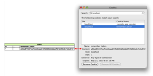


Figure 9.8: The correspondence between the remember token and the browser cookie. (full size)

Box 9.3.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 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 9.12 shows yet another example of this practice, through one of Rails' *time helpers*, which are methods added to Fixnum (the base class for numbers):

```
  $ script/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.

Now that we know how to manipulate cookies, there's only one piece left: it's time to pop current_user off the stack.

9.3.4 Current user

Let's look at the sign_in function one last time:

```
module SessionsHelper

  def sign_in(user)
    user.remember_me!
    cookies[:remember_token] = { :value   => user.remember_token,
                                 :expires => 20.years.from_now.utc }
    self.current_user = user
  end
end
```

Our focus now is the final 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
```

In the context of the Sessions helper, the `self` in `self.current_user` is the Sessions *controller*, since the module is being included into the Application controller, which is the base class for all the other controllers (including Sessions). But there is no such variable as `current_user` right now; we need to define it.

To describe the behavior of the remaining signin machinery, we'll first fill in the test for signing a user in (Listing 9.16).

Listing 9.16. Filling in the test for signing the user in.
`spec/controllers/sessions_controller_spec.rb`

```
describe SessionsController do
  .
  .
  .
  describe "POST 'create'" do
    .
    .
    .
    describe "with valid email and password" do

      before(:each) do
        @user = Factory(:user)
        @attr = { :email => @user.email, :password => @user.password }
        User.should_receive(:authenticate).
             with(@user.email, @user.password).
             and_return(@user)
      end

      it "should sign the user in" do
        post :create, :session => @attr
        controller.current_user.should == @user
        controller.should be_signed_in
      end

      it "should redirect to the user show page" do
        post :create, :session => @attr
        response.should redirect_to(user_path(@user))
      end
    end
  end
end
```

The new test uses the `controller` variable (which is available inside Rails tests) to check that the `current_user` variable is set to the signed-in user, and that the user is signed in:

```
it "should sign the user in" do
  post :create, :session => @attr
  controller.current_user.should == @user
  controller.should be_signed_in
end
```

The second line may be a little confusing at this point, but you can guess based on the RSpec convention for boolean methods that

```
controller.should be_signed_in
```

is equivalent to

```
controller.signed_in?.should be_true
```

This is a hint that we will be defining a `signed_in?` method that returns `true` if a user is signed in and `false` otherwise. Moreover, the `signed_in?` method will be attached to the *controller*, not to a user, which is why we write `controller.signed_in?` instead of `current_user.signed_in?`. (If no user is signed in, how could we call `signed_in?` on it?)

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

```
self.current_user = user
```

is an *assignment*. Ruby has a special syntax for defining such an assignment function, shown in Listing 9.17.

Listing 9.17. 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, but it simply defines a method `current_user=` expressly designed to handle assignment to `current_user`. 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` (Listing 9.18).

Listing 9.18. 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 code.
  end
end
```

If we did this, we would effectively replicate the functionality of `attr_accessor`, first seen in Section 4.4.5 and used to make the virtual `password` attribute in Section 7.1.1.11 The problem is that it utterly fails to solve our problem: with the code in Listing 9.18, 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.

Luckily, thanks to our work with the remember token in Section 9.3.3, we can find the user in the database using the `cookies` hash and the `find_by_remember_token` (Listing 9.19).12

Listing 9.19. Finding the current user by `remember_token`.
`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def current_user
    @current_user ||= user_from_remember_token
  end

  def user_from_remember_token
    remember_token = cookies[:remember_token]
    User.find_by_remember_token(remember_token) unless remember_token.nil?
  end
end
```

This code uses the common but initially obscure `||=` ("or equals") assignment operator (Box 9.4). 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. In other words, the construction

```
@current_user ||= user_from_remember_token
```

calls the `user_from_remember_token` method the first time `current_user` is called, but on subsequent invocations returns `@current_user` without calling `user_from_remember_token`.13

Box 9.4.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:

```
$ script/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" || "the 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!

At this point, the signin test is almost passing; the only thing remaining is to define the required `signed_in?` boolean method. Happily, it's easy with the use of the "not" operator `!`: a user is signed in if `current_user` is not `nil` (Listing 9.20).

Listing 9.20. The `signed_in?` helper method.
`app/helpers/sessions_helper.rb`

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

Though it’s already useful for the test, we’ll put the `signed_in?` method to even better use in Section 9.4.3 and again in Chapter 10.

With that, all the tests should pass.

9.4 Signing out

As discussed in Section 9.1, our authentication model is to keep users signed in until they sign out explicitly. In this section we’ll add this necessary signout capability. Once we’re done, we’ll add some integration tests to put our authentication machinery through its paces.

9.4.1 Destroying sessions

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.

In order to test the signout action, we first need a way to sign in within a test. The easiest way to do this is to use the `controller` object we saw in Section 9.3.4 and set its `current_user` to the given user. In order to use the resulting `test_sign_in` function in all our tests, we need to put it in the spec helper file, as shown in Listing 9.21.

Listing 9.21. A `test_sign_in` function to simulate user signin inside tests.
`spec/spec_helper.rb`

```
.
.
.
Spec::Runner.configure do |config|
  .
  .
  .
  def test_sign_in(user)
    controller.current_user = user
  end
end
```

After running `test_sign_in`, the `current_user` will not be `nil`, so `signed_in?` will be `true`.

With this spec helper in hand, the test for signout is straightforward: sign in as a (factory) user, check that it is signed in, and then hit the `destroy` action and verify that the user gets signed out (Listing 9.22).

Listing 9.22. A test for destroying a session (user signout).

spec/controllers/sessions_controller_spec.rb

```
describe SessionsController do
  .
  .
  .
  describe "DELETE 'destroy'" do

    it "should sign a user out" do
      test_sign_in(Factory(:user))
      controller.should be_signed_in
      delete :destroy
      controller.should_not be_signed_in
      response.should redirect_to(root_path)
    end
  end
end
```

The only novel element here is the `delete` method, which issues an HTTP `DELETE` request (in analogy with the `get` and `post` methods seen in previous tests), as required by the REST conventions (Table 9.1).

As with user signin, which relied on the `sign_in` function, user signout just defers the hard work to a `sign_out` function (Listing 9.23).

Listing 9.23. Destroying a session (user signout).
app/controllers/sessions_controller.rb

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

As with the other authentication elements, we'll put `sign_out` in the Sessions helper module (Listing 9.24).

Listing 9.24. The `sign_out` method in the Sessions helper module.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  def sign_in(user)
    user.remember_me!
    cookies[:remember_token] = { :value   => user.remember_token,
                                 :expires => 20.years.from_now.utc }
    self.current_user = user
  end
  .
  .
  .
  def sign_out
    cookies.delete(:remember_token)
    self.current_user = nil
  end
end
```

As you can see, the `sign_out` method effectively undoes the `sign_in` method by deleting the remember token and by setting the current user to `nil`.14 With this, the test in Listing 9.22 should pass.

9.4.2 Signin upon signup

In principle, we are now done with authentication, but as currently constructed there are no links to the signin or signout actions. Moreover, newly registered users might be confused, as they are not signed in by default.

We'll fix the second problem first, starting with testing that a new user is automatically signed in (Listing 9.25).

Listing 9.25. Testing that newly signed-up users are also signed in.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views
  .
  .
  .
  describe "POST 'create'" do
    .
    .
    .
    describe "success" do
      .
      .
      .
      it "should sign the user in" do
        post :create, :user => @attr
        controller.should be_signed_in
      end
      .
      .
      .
    end
  end
end
```

With the `sign_in` method from Section 9.3, 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 9.26).

Listing 9.26. 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
      @title = "Sign up"
      render 'new'
    end
  end
```

9.4.3 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 9.6 mockup, we'll arrange for the links change when users sign in or sign out, and we'll also add a profile link to the user show page for signed-in users.

We start with two integration tests: one to check that a `"Sign in"` link appears for non-signed-in users, and one to check that a `"Sign out"` link appears for signed-in users; both cases verify that the link goes to the proper URL. We'll put these tests in the layout links test we created in Section 5.2.1; the result appears in Listing 9.27.

Listing 9.27. Tests for the signin/signout links on the site layout.
`spec/integration/layout_links_spec.rb`

```
describe "Layout links" do
  .
  .
  .
  describe "when not signed in" do
    it "should have a signin link" do
      visit root_path
      response.should have_tag("a[href=?]", signin_path, "Sign in")
    end
  end

  describe "when signed in" do

    before(:each) do
      @user = Factory(:user)
      visit signin_path
      fill_in :email,    :with => @user.email
      fill_in :password, :with => @user.password
      click_button
    end

    it "should have a signout link" do
      visit root_path
      response.should have_tag("a[href=?]", signout_path, "Sign out")
    end

    it "should have a profile link"
  end
end
```

Here the `before(:each)` block signs in by visiting the signin page and submitting a valid email/password pair.15 We do this instead of using the `test_sign_in` function from Listing 9.21 because `test_sign_in` doesn't work inside integration tests for some reason. (See Section 9.6 for an exercise to make an `integration_sign_in` function for use in integration tests.)

The application code uses an if-then branching structure inside of Embedded Ruby, using the `signed_in?` method defined in Listing 9.20:

```
<% if signed_in? %>
<li><%= link_to "Sign out", signout_path, :method => :delete %></li>
<% else %>
<li><%= link_to "Sign in", signin_path %></li>
<% end %>
```

Notice that the signout link passes a hash argument indicating that it should submit with an HTTP `DELETE` request.16 With this snippet added, the full header partial appears as in Listing 9.28.

Listing 9.28. Changing the layout links for signed-in users.
`app/views/layouts/_header.html.erb`

```
<div id="header" class="round">
  <%= link_to logo, root_path %>
  <ul class="navigation round">
    <li><%= link_to "Home", root_path %></li>
    <li><%= link_to "Help", help_path %></li>
    <% if signed_in? %>
    <li><%= link_to "Sign out", signout_path, :method => :delete %></li>
    <% else %>
    <li><%= link_to "Sign in", signin_path %></li>
    <% end %>
  </ul>
</div>
```

In Listing 9.28 we've used the `logo` helper from the Chapter 5 exercises (Section 5.5); in case you didn't work that exercise, the answer appears in Listing 9.29.

Listing 9.29. A helper for the site logo.
`app/helpers/application_helper.rb`

```
module ApplicationHelper
  .
  .
  .
  def logo
    image_tag("logo.png", :alt => "Sample App", :class => "round")
  end
end
```

Finally, let's add a profile link. The test (Listing 9.30) and application code (Listing 9.31) are both straightforward. Notice that the profile link's URL is simply `current_user`,17 which is our first use of that helpful method. (It won't be our last.)

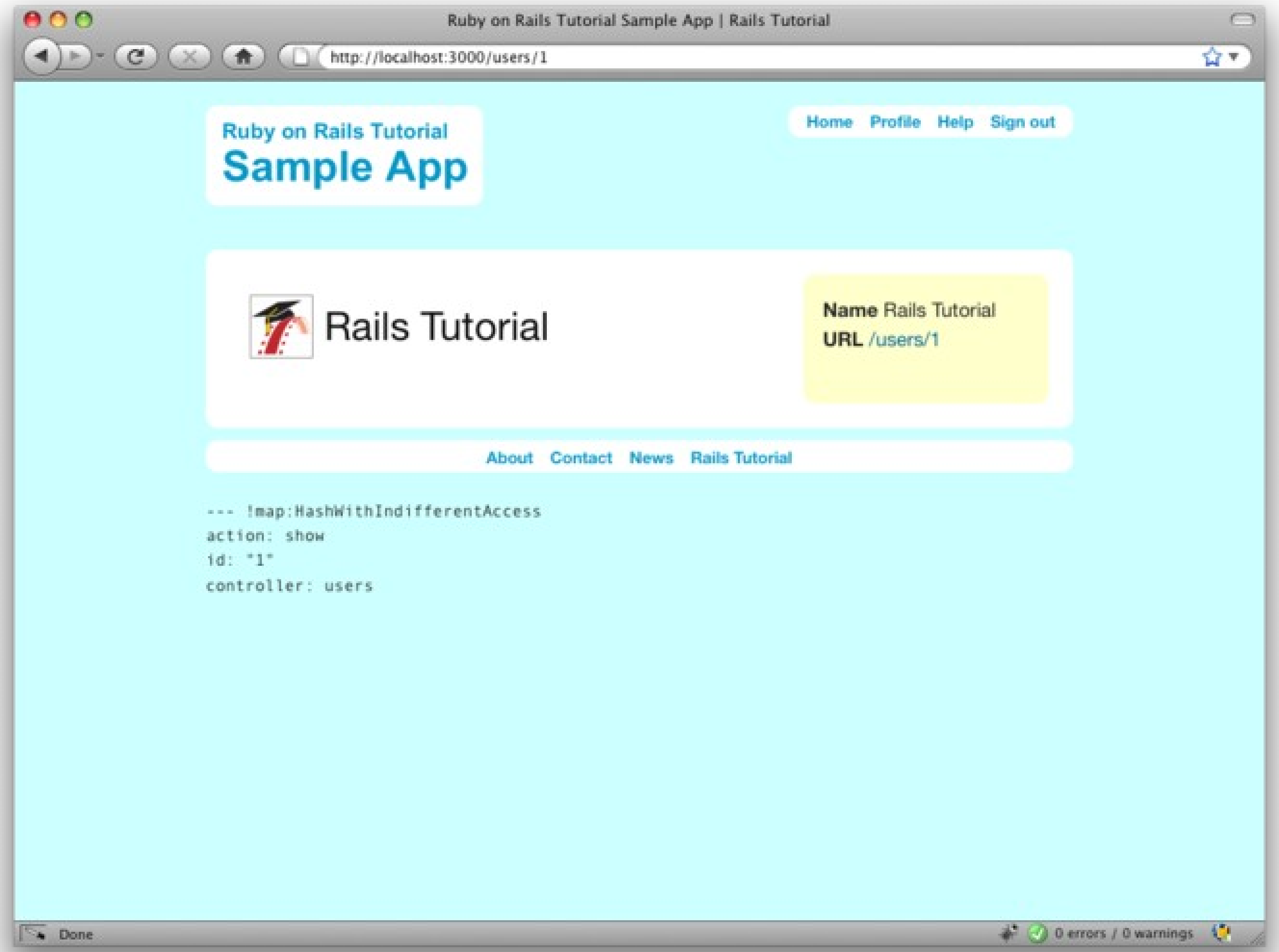
Listing 9.30. A test for a profile link.
`spec/integration/layout_links_spec.rb`

```
describe "Layout links" do
  .
  .
  .
  describe "when signed in" do
    .
    .
    .
    it "should have a profile link" do
      visit root_path
      response.should have_tag("a[href=?]", user_path(@user), "Profile")
    end
  end
end
```

Listing 9.31. Adding a profile link.
`app/views/layouts/_header.html.erb`

```
<div id="header" class="round">
  <%= link_to logo, root_path %>
  <ul class="navigation round">
    <li><%= link_to "Home", root_path %></li>
    <% if signed_in? %>
    <li><%= link_to "Profile", current_user %></li>
    <% end %>
    <li><%= link_to "Help", help_path %></li>
    <% if signed_in? %>
    <li><%= link_to "Sign out", signout_path, :method => :delete %></li>
    <% else %>
    <li><%= link_to "Sign in", signin_path %></li>
    <% end %>
  </ul>
</div>
```

With the code in this section, a signed-in user now sees both signout and profile links, as expected (Figure 9.9).



```
--- !map:HashWithIndifferentAccess
action: show
id: "1"
controller: users
```

Figure 9.9: A signed-in user with signout and profile links. (full size)

9.4.4 Signin/out integration tests

As a capstone to our hard work on authentication, we'll finish with integration tests for signin and signout (placed in the `users_spec.rb` file for convenience). The combination of RSpec and Webrat is expressive enough that Listing 9.32 should need little explanation; I especially like the use of `click_link "Sign out"`, which not only simulates a browser clicking the signout link, but also raises an error if no such link exists—thereby testing the URL, the named route, the link text, and the changing of the layout links, all in one line. If that's not an *integration* test, I don't know what is.

Listing 9.32. An integration test for signing in and out.
`spec/integration/users_spec.rb`

```
require 'spec_helper'

describe "Users" do

  describe "signup" do
    .
    .
    .
  end

  describe "sign in/out" do

    describe "failure" do
      it "should not sign a user in" do
        visit signin_path
        fill_in :email,    :with => ""
        fill_in :password, :with => ""
        click_button
        response.should render_template('sessions/new')
        response.should have_tag("div.flash.error", /invalid/i)
      end
    end

    describe "success" do
      it "should sign a user in and out" do
        user = Factory(:user)
        visit signin_path
        fill_in :email,    :with => user.email
        fill_in :password, :with => user.password
        click_button
        controller.should be_signed_in
        click_link "Sign out"
        controller.should_not be_signed_in
      end
    end
  end
end
```

9.5 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.

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

```
$ git add .
$ git commit -am "Done with sign in"
$ git checkout master
$ git merge sign-in-out
```

9.6 Exercises

The second and third exercises are more difficult than usual. Solving them will require some outside research (e.g., Rails API reading and Google searches), and they can be skipped without loss of continuity.

1. Several of the integration specs use the same code to sign a user in. Replace that code with the `integration_sign_in` function in Listing 9.33 and verify that the tests still pass.
2. Write an integration test for signing in, signing out, and then signing back in, which will catch the password re-encryption issue mentioned after Listing 9.14.
3. Use `session` instead of `cookies` so that users are automatically signed out when they close their browsers.18 *Hint:* Do a Google search on “Rails session”.
4. **(advanced)** Some sites use secure HTTP (HTTPS) for their signin pages. Search online to learn how to use HTTPS in Rails, and then secure the Sessions controller `new` and `create` actions. *Hint:* Take a look at the `ssl_requirement` plugin. *Extra challenge:* Write tests for the HTTPS functionality. (*Note:* I suggest doing this exercise only in development, which does not require obtaining an SSL certificate or setting up the SSL encryption machinery. Actually deploying an SSL-enabled site is *much* more difficult.)

Listing 9.33. A function to sign users in inside of integration tests.
`spec/spec_helper.rb`

```
.
.
.
Spec::Runner.configure do |config|
  .
  .
  .
  def test_sign_in(user)
    controller.current_user = user
  end

  def integration_sign_in(user)
    visit signin_path
    fill_in :email,    :with => user.email
    fill_in :password, :with => user.password
    click_button
  end
end
```

« Chapter 8 Sign up Chapter 10 Updating, showing, and deleting users »

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. We’ll see in Section 9.3.2 just how long “forever” is. ↑
3. If given the `create` and `destroy` actions as well, the generate script would make *views* for those actions, which we don’t need. Of course, we could delete the views, but I’ve elected to omit them from `generate` and instead define the actions by hand. ↑
4. In Rails 3, the `/controller/action/id` pattern is disabled by default, so the RESTful route will be necessary in this case. ↑
5. See the RSpec documentation on message expectations for more about `with`. ↑
6. If case you’re wondering why we use `user` instead of `@user` in Listing 9.8, it’s because this user variable is never needed in any view, so there is no reason to use an instance variable here. (Using `@user` still works, though.) ↑
7. We don’t have a controller test for `flash.now` because Rails 2.3 broke `flash.now` support

in tests (it worked in Rails 2.2). It's not a big deal; we'll cover it as part of the integration tests in Section 9.4.4. ↑

8. Image from http://www.flickr.com/photos/hermanusbackpackers/3343254977/. ↑
9. You might reasonably wonder why we have a separate test that a user responds to `remember_me!`; after all, we use it in the very next test, which would fail without a `remember_me!` method. The answer is that in TDD we typically write as little code as possible at each step, so here we would actually write an *empty* `remember_me!` method to get the `respond_to(:remember_me!)` test to pass, and only then move on to the full implementation. It's difficult to show in a book just how this works; the Ruby on Rails Tutorial screencasts will make the process clearer. ↑
10. There's a second way to do this, by passing an `:if` parameter to the `before_save` filter. The syntax is a smell, though, and I expect it to change in future versions of Active Record. ↑
11. In fact, the two are exactly equivalent; `attr_accessor` is merely a convenient way to create just such getter/setter methods automatically. ↑
12. Recall from Section 6.1.4 that Active Record synthesizes find methods for all the user attributes, including `remember_token`. ↑
13. This optimization technique to avoid repeated function calls is known as memoization. ↑
14. You can learn about things like `cookies.delete` by reading the cookies entry in the Rails API. ↑
15. Note that we can use symbols in place of strings for the labels, e.g., `fill_in :email` instead of `fill_in "Email"`. We used the latter in Listing 8.22, but by now it shouldn't surprise you that Rails allows us to use symbols instead. ↑
16. Web browsers can't actually issue `DELETE` requests; Rails fakes it with JavaScript. ↑
17. Recall from Section 7.3.3 that we can link directly to a user object and allow Rails to figure out the appropriate URL. ↑
18. Somewhat confusingly, we've used `cookies` to implement sessions, and `session` is implemented with cookies! ↑