```
      # Log the user in and redirect to the user's show page.
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

We can then verify that both the login integration test and the full test suite are GREEN:

Listing 8.12: GREEN

```
$ rails test test/integration/users_login_test.rb
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Verify in your browser that the sequence from Section 8.1.4 works correctly, i.e., that the flash message disappears when you click on a second page.

8.2 Logging in

Now that our login form can handle invalid submissions, the next step is to handle valid submissions correctly by actually logging a user in. In this section, we'll log the user in with a temporary session cookie that expires automatically upon browser close. In Section 9.1, we'll add sessions that persist even after closing the browser.

Implementing sessions will involve defining a large number of related functions for use across multiple controllers and views. You may recall from Section 4.2.4 that Ruby provides a *module* facility for packaging such functions in one place. Conveniently, a Sessions helper module was generated automatically when generating the Sessions controller (Section 8.1.1). Moreover, such helpers are automatically included in Rails views; by including the module into the base class of all controllers (the Application controller), we arrange to make them available in our controllers as well (Listing 8.13).[2]

Listing 8.13: Including the Sessions helper module into the Application controller.
app/controllers/application_controller.rb

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

With this configuration complete, we're now ready to write the code to log users in.

8.2.1 The `log_in` method

Logging a user in is simple with the help of the `session` method defined by Rails. (This method is separate and distinct from the Sessions controller generated in Section 8.1.1.) We can treat `session` as if it were a hash, and assign to it as follows:

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

This places a temporary cookie on the user's browser containing an encrypted version of the user's id, which allows us to retrieve the id on subsequent pages

[2] I like this technique because it connects to the pure Ruby way of including modules, but Rails 4 introduced a technique called *concerns* that can also be used for this purpose. To learn how to use concerns, run a search for "how to use concerns in Rails".

using `session[:user_id]`. In contrast to the persistent cookie created by the `cookies` method (Section 9.1), the temporary cookie created by the `session` method expires immediately when the browser is closed.

Because we'll want to use the same login technique in a couple of different places, we'll define a method called `log_in` in the Sessions helper, as shown in Listing 8.14.

Listing 8.14: The `log_in` function.
app/helpers/sessions_helper.rb

```ruby
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
end
```

Because temporary cookies created using the `session` method are automatically encrypted, the code in Listing 8.14 is secure, and there is no way for an attacker to use the session information to log in as the user. This applies only to temporary sessions initiated with the `session` method, though, and is *not* the case for persistent sessions created using the `cookies` method. Permanent cookies are vulnerable to a *session hijacking* attack, so in Chapter 9 we'll have to be much more careful about the information we place on the user's browser.

With the `log_in` method defined in Listing 8.14, we're now ready to complete the session `create` action by logging the user in and redirecting to the user's profile page. The result appears in Listing 8.15.[3]

Listing 8.15: Logging in a user.
app/controllers/sessions_controller.rb

```ruby
class SessionsController < ApplicationController

  def new
  end
```

[3]The `log_in` method is available in the Sessions controller because of the module inclusion in Listing 8.13.

```

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Note the compact redirect

```
redirect_to user
```

which we saw before in Section 7.4.1. Rails automatically converts this to the route for the user's profile page:

```
user_url(user)
```

With the **`create`** action defined in Listing 8.15, the login form defined in Listing 8.4 should now be working. It doesn't have any effects on the application display, though, so short of inspecting the browser session directly there's no way to tell that you're logged in. As a first step toward enabling more visible changes, in Section 8.2.2 we'll retrieve the current user from the database using the id in the session. In Section 8.2.3, we'll change the links on the application layout, including a URL to the current user's profile.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Log in with a valid user and inspect your browser's cookies. What is the value of the session content? *Hint*: If you don't know how to view your browser's cookies, Google for it (Box 1.2).

2. What is the value of the **`Expires`** attribute from the previous exercise?

8.2.2 Current user

Having placed the user's id securely in the temporary session, we are now in a position to retrieve it on subsequent pages, which we'll do by defining a **`current_user`** method to find the user in the database corresponding to the session id. The purpose of **`current_user`** is to allow constructions such as

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

and

```
redirect_to current_user
```

To find the current user, one possibility is to use the **`find`** method, as on the user profile page (Listing 7.5):

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

But recall from Section 6.1.4 that **`find`** raises an exception if the user id doesn't exist. This behavior is appropriate on the user profile page because it will only happen if the id is invalid, but in the present case **`session[:user_id]`** will often be **`nil`** (i.e., for non-logged-in users). To handle this possibility, we'll use the same **`find_by`** method used to find by email address in the **`create`** method, with **`id`** in place of **`email`**:

```
User.find_by(id: session[:user_id])
```

Rather than raising an exception, this method returns **`nil`** (indicating no such user) if the id is invalid.

We could now define the **`current_user`** method as follows:

```
def current_user
  if session[:user_id]
    User.find_by(id: session[:user_id])
  end
end
```

(If the session user id doesn't exist, the function just falls off the end and returns **`nil`** automatically, which is exactly what we want.) This would work fine, but it would hit the database multiple times if, e.g., **`current_user`** appeared multiple times on a page. Instead, we'll follow a common Ruby convention by storing the result of **`User.find_by`** in an instance variable, which hits the database the first time but returns the instance variable immediately on subsequent invocations:[4]

```
if @current_user.nil?
  @current_user = User.find_by(id: session[:user_id])
else
  @current_user
end
```

Recalling the *or* operator `||` seen in Section 4.2.2, we can rewrite this as follows:

```
@current_user = @current_user || User.find_by(id: session[:user_id])
```

[4]This practice of remembering variable assignments from one method invocation to the next is known as *memoization*. (Note that this is a technical term; in particular, it's *not* a misspelling of "memorization"—a subtlety lost on the hapless copyeditor of a previous edition of this book.)

Because a User object is true in a boolean context, the call to `find_by` only gets executed if `@current_user` hasn't yet been assigned.

Although the preceding code would work, it's not idiomatically correct Ruby; instead, the proper way to write the assignment to `@current_user` is like this:

```
@current_user ||= User.find_by(id: session[:user_id])
```

This uses the potentially confusing but frequently used `||=` ("or equals") operator (Box 8.1).

Box 8.1. What the *$@! is ||= ?

The `||=` ("or equals") assignment operator is a common Ruby idiom and is thus important for aspiring Rails developers to recognize. Although at first it may seem mysterious, *or equals* is easy to understand by analogy.

We start by noting the common pattern of incrementing a variable:

```
x = x + 1
```

Many languages provide a syntactic shortcut for this operation; in Ruby (and in C, C++, Perl, Python, Java, etc.), it can also appear as follows:

```
x += 1
```

Analogous constructs exist for other operators as well:

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

```
=> 6
>> x -= 8
=> -2
>> x /= 2
=> -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 4.2.2, we can write this as follows:

```
>> @foo
=> nil
>> @foo = @foo || "bar"
=> "bar"
>> @foo = @foo || "baz"
=> "bar"
```

Since `nil` is false in a boolean context, the first assignment to `@foo` is `nil || "bar"`, which evaluates to `"bar"`. Similarly, the second assignment is `@foo || "baz"`, i.e., `"bar" || "baz"`, which also evaluates to `"bar"`. This is because anything other than `nil` or `false` is `true` in a boolean context, and the series of `||` expressions terminates after the first true 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*. The same principle applies to `&&` statements, except in this case evaluation stops on the first *false* value.)

Comparing the console sessions for the various operators, we see that `@foo = @foo || "bar"` follows the `x = x O y` pattern with `||` in the place of `O`:

```
x    =    x    +    1        ->       x       +=    1
x    =    x    *    3        ->       x       *=    3
x    =    x    -    8        ->       x       -=    8
```

```
x     =  x   /   2       ->      x      /=   2
@foo = @foo || "bar"     ->      @foo ||= "bar"
```

Thus we see that `@foo = @foo || "bar"` and `@foo ||= "bar"` are equivalent. In the context of the current user, this suggests the following construction:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Voilà !

(Technically, Ruby evaluates the expression `@foo || @foo = "bar"`, which avoids an unnecessary assignment when `@foo` is not `nil` or `false`. But this expression doesn't explain the `||=` notation as well, so the above discussion uses the nearly equivalent `@foo = @foo || "bar"`.)

Applying the results of the above discussion yields the succinct **`current_user`** method shown in Listing 8.16. (There's a slight amount of repetition in the use of **`session[:user_id]`**, which we'll eliminate in Section 9.1.2.)

Listing 8.16: Finding the current user in the session.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end
end
```

With the working **current_user** method in Listing 8.16, we're now in a position to make changes to our application based on user login status.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm at the console that **User.find_by(id: ...)** returns **nil** when the corresponding user doesn't exist.

2. In a Rails console, create a **session** hash with key **:user_id**. By following the steps in Listing 8.17, confirm that the **||=** operator works as required.

Listing 8.17: Simulating **session** in the console.

```
>> session = {}
>> session[:user_id] = nil
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
>> session[:user_id]= User.first.id
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
```

8.2.3 Changing the layout links

The first practical application of logging in involves changing the layout links based on login status. In particular, as seen in the Figure 8.8 mockup,[5] we'll add links for logging out, for user settings, for listing all users, and for the current user's profile page. Note in Figure 8.8 that the logout and profile links appear in a dropdown "Account" menu; we'll see in Listing 8.19 how to make such a menu with Bootstrap.

[5]Image retrieved from https://www.flickr.com/photos/elevy/14730820387 on 2016-06-03. Copyright © 2014 by Elias Levy and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

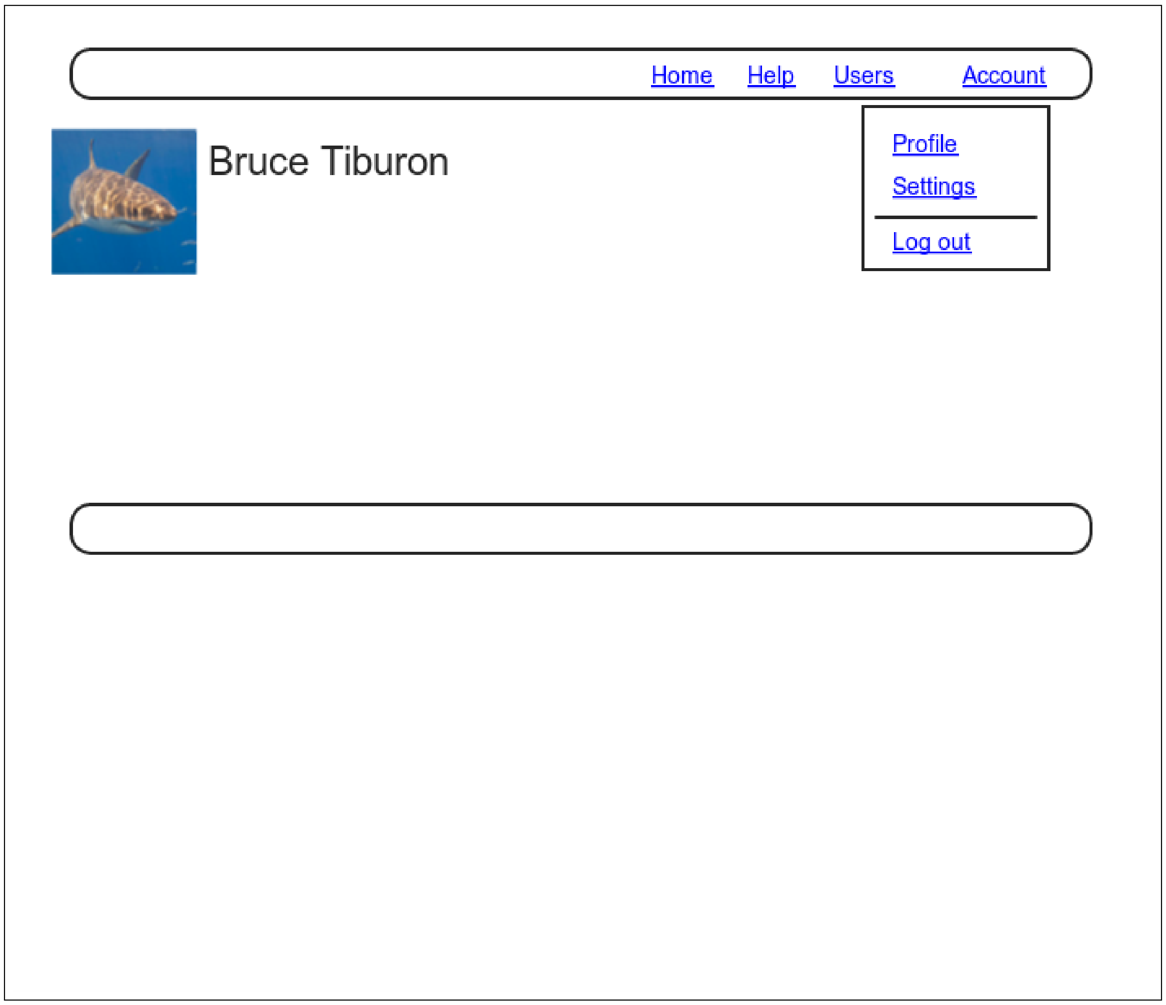


Figure 8.8: A mockup of the user profile after a successful login.

At this point, in real life I would consider writing an integration test to capture the behavior described above. As noted in Box 3.3, as you become more familiar with the testing tools in Rails you may find yourself more inclined to write tests first. In this case, though, such a test involves several new ideas, so for now it's best deferred to its own section (Section 8.2.4).

The way to change the links in the site layout involves using an if-else statement inside embedded Ruby to show one set of links if the user is logged in and another set of links otherwise:

```
<% if logged_in? %>
  # Links for logged-in users
<% else %>
  # Links for non-logged-in-users
<% end %>
```

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

A user is logged in if there is a current user in the session, i.e., if **`current_user`** is not **`nil`**. Checking for this requires the use of the "not" operator (Section 4.2.2), written using an exclamation point **`!`** and usually read as "bang". The resulting **`logged_in?`** method appears in Listing 8.18.

Listing 8.18: The **`logged_in?`** helper method.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end

  # Returns true if the user is logged in, false otherwise.
  def logged_in?
    !current_user.nil?
  end
end
```

With the addition in Listing 8.18, we're now ready to change the layout links if a user is logged in. There are four new links, two of which are stubbed out (to be completed in Chapter 10):

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

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

```
<%= link_to "Log out", logout_path, method: :delete %>
```

Notice that the logout link passes a hash argument indicating that it should submit with an HTTP `DELETE` request.[6] We'll also add a profile link as follows:

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

Here we could write

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

but as usual Rails allows us to link directly to the user by automatically converting **`current_user`** into **`user_path(current_user)`** in this context. Finally, when users *aren't* logged in, we'll use the login path defined in Listing 8.2 to make a link to the login form:

[6]Web browsers can't actually issue `DELETE` requests; Rails fakes it with JavaScript.

```
<%= link_to "Log in", login_path %>
```

Putting everything together gives the updated header partial shown in Listing 8.19.

Listing 8.19: Changing the layout links for logged-in users.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
          <li 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 "Log out", logout_path, method: :delete %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

As part of including the new links into the layout, Listing 8.19 takes advantage of Bootstrap's ability to make dropdown menus.[7] Note in particular the inclusion of the special Bootstrap CSS classes such as **`dropdown`**, **`dropdown-menu`**, etc. To activate the dropdown menu, we need to include Bootstrap's

[7]See the Bootstrap components page for more information.

custom JavaScript library into our application (which is *not* included automatically as part of the `bootstrap-sass` gem in Listing 5.5), as well as the jQuery library.

Section 5.2 mentioned briefly that the Rails asset pipeline works in parallel with Webpack and Yarn, and we need to put both to work in order to include the above JavaScript. The first step is to install both jQuery and Bootstrap's JavaScript library in our application, which coincidentally needs the same version number for each:

```
$ yarn add jquery@3.4.1 bootstrap@3.4.1
```

In order to make jQuery available in our application, we need to edit Webpack's environment file and add the content shown in Listing 8.20.

Listing 8.20: Adding jQuery configuration to Webpack.
`config/webpack/environment.js`

```
const { environment } = require('@rails/webpacker')

const webpack = require('webpack')
environment.plugins.prepend('Provide',
  new webpack.ProvidePlugin({
    $: 'jquery/src/jquery',
    jQuery: 'jquery/src/jquery'
  })
)

module.exports = environment
```

Finally, we need to require jQuery and import Bootstrap in our **`application.js`** file, as shown in Listing 8.21.[8]

Listing 8.21: Requiring and importing the necessary JavaScript libraries.
`app/javascript/packs/application.js`

[8]For what it's worth, I don't know offhand why one uses **`require`** and the other used **`import`**.

```
require("@rails/ujs").start()
require("turbolinks").start()
require("@rails/activestorage").start()
require("channels")
require("jquery")
import "bootstrap"
```

At this point, you should visit the login path and log in as a valid user (username **`example@railstutorial.org`**, password **`foobar`**), which effectively tests the code in the previous three sections.[9] With the code in Listing 8.19 and Listing 8.21, you should see the dropdown menu and links for logged-in users, as shown in Figure 8.9.

If you quit your browser completely, you should also be able to verify that the application forgets your login status, requiring you to log in again to see the changes described above.[10]

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

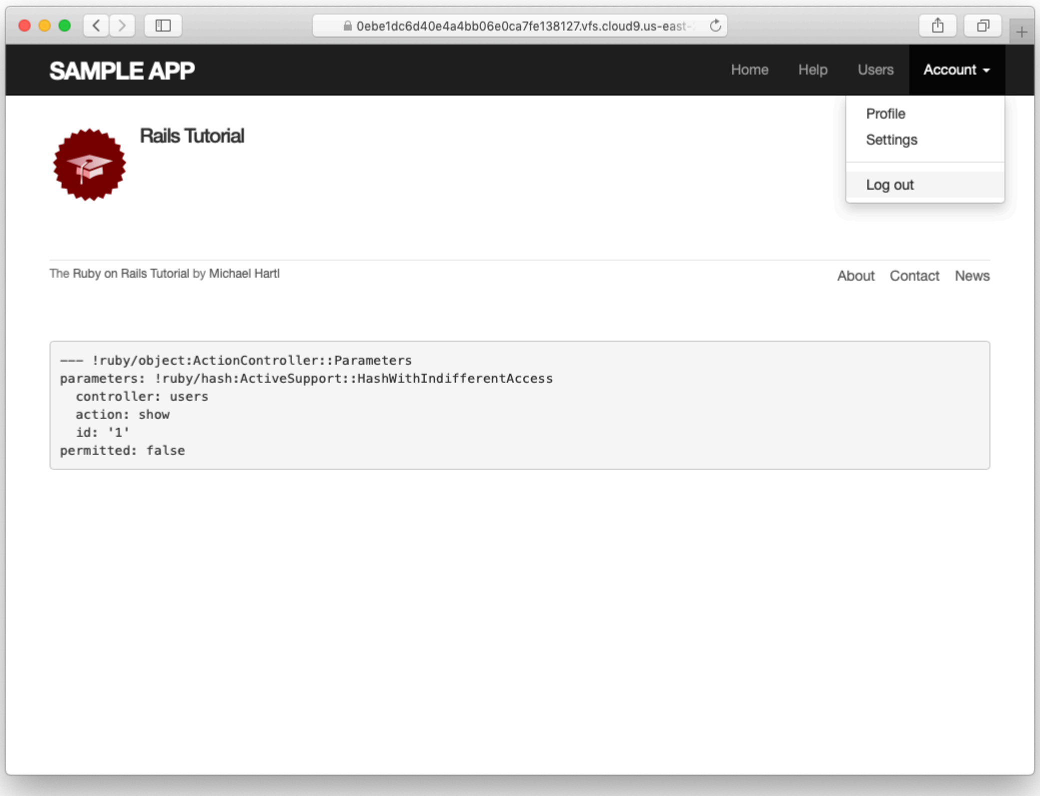
To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Using the cookie inspector in your browser (Section 8.2.1), remove the session cookie and confirm that the layout links revert to the non-logged-in state.

2. Log in again, confirming that the layout links change correctly. Then quit your browser and start it again to confirm that the layout links revert to the non-logged-in state. (If your browser has a "remember where I left off" feature that automatically restores the session, be sure to disable it in this step (Box 1.2).)

[9]You may have to restart the webserver to get this to work (Box 1.2).

[10]If you're using the cloud IDE, I recommend using a different browser to test the login behavior so that you don't have to close down the browser running the IDE.


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-
SAMPLE APP
Home
Help
Users
Account
Profile
Settings
Log out
Rails Tutorial
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  controller: users
  action: show
  id: '1'
permitted: false
```

Figure 8.9: A logged-in user with new links and a dropdown menu.

8.2.4 Testing layout changes

Having verified by hand that the application is behaving properly upon successful login, before moving on we'll write an integration test to capture that behavior and catch regressions. We'll build on the test from Listing 8.9 and write a series of steps to verify the following sequence of actions:

1. Visit the login path.
2. Post valid information to the sessions path.
3. Verify that the login link disappears.
4. Verify that a logout link appears
5. Verify that a profile link appears.

In order to see these changes, our test needs to log in as a previously registered user, which means that such a user must already exist in the database. The default Rails way to do this is to use *fixtures*, which are a way of organizing data to be loaded into the test database. We discovered in Section 6.2.5 that we needed to delete the default fixtures so that our email uniqueness tests would pass (Listing 6.31). Now we're ready to start filling in that empty file with custom fixtures of our own.

In the present case, we need only one user, whose information should consist of a valid name and email address. Because we'll need to log the user in, we also have to include a valid password to compare with the password submitted to the Sessions controller's **`create`** action. Referring to the data model in Figure 6.9, we see that this means creating a **`password_digest`** attribute for the user fixture, which we'll accomplish by defining a **`digest`** method of our own.

As discussed in Section 6.3.1, the password digest is created using bcrypt (via **`has_secure_password`**), so we'll need to create the fixture password using the same method. By inspecting the secure password source code, we find that this method is

```
BCrypt::Password.create(string, cost: cost)
```

where **`string`** is the string to be hashed and **`cost`** is the *cost parameter* that determines the computational cost to calculate the hash. Using a high cost makes it computationally intractable to use the hash to determine the original password, which is an important security precaution in a production environment, but in tests we want the **`digest`** method to be as fast as possible. The secure password source code has a line for this as well:

```
cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                              BCrypt::Engine.cost
```

This rather obscure code, which you don't need to understand in detail, arranges for precisely the behavior described above: it uses the minimum cost parameter in tests and a normal (high) cost parameter in production. (We'll learn more about the strange **`?-:`** notation in Section 9.2.)

There are several places we could put the resulting **`digest`** method, but we'll have an opportunity in Section 9.1.1 to reuse **`digest`** in the User model. This suggests placing the method in **`user.rb`**. Because we won't necessarily have access to a user object when calculating the digest (as will be the case in the fixtures file), we'll attach the **`digest`** method to the User class itself, which (as we saw briefly in Section 4.4.1) makes it a *class method*. The result appears in Listing 8.22.

Listing 8.22: Adding a digest method for use in fixtures.
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }
  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end
end
```

With the **`digest`** method from Listing 8.22, we are now ready to create a user fixture for a valid user, as shown in Listing 8.23.[11]

Listing 8.23: A fixture for testing user login.
`test/fixtures/users.yml`

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>
```

Note in particular that fixtures support embedded Ruby, which allows us to use

```
<%= User.digest('password') %>
```

to create the valid password digest for the test user.

Although we've defined the **`password_digest`** attribute required by **`has_secure_password`**, sometimes it's convenient to refer to the plain (virtual) password as well. Unfortunately, this is impossible to arrange with fixtures, and adding a **`password`** attribute to Listing 8.23 causes Rails to complain that there is no such column in the database (which is true). We'll make do by adopting the convention that all fixture users have the same password (**`'password'`**).

Having created a fixture with a valid user, we can retrieve it inside a test as follows:

[11]It's worth noting that indentation in fixture files must take the form of spaces, not tabs, so take care when copying code like that shown in Listing 8.23.

```
user = users(:michael)
```

Here **users** corresponds to the fixture filename **users.yml**, while the symbol **:michael** references user with the key shown in Listing 8.23.

With the fixture user as above, we can now write a test for the layout links by converting the sequence enumerated at the beginning of this section into code, as shown in Listing 8.24.

Listing 8.24: A test for user logging in with valid information. GREEN
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid information" do
    get login_path
    post login_path, params: { session: { email:    @user.email,
                                          password: 'password' } }
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
  end
end
```

Here we've used

```
assert_redirected_to @user
```

to check the right redirect target and

```
follow_redirect!
```

to actually visit the target page. Listing 8.24 also verifies that the login link disappears by verifying that there are *zero* login path links on the page:

```
assert_select "a[href=?]", login_path, count: 0
```

By including the extra **`count: 0`** option, we tell **`assert_select`** that we expect there to be zero links matching the given pattern. (Compare this to **`count: 2`** in Listing 5.32, which checks for exactly two matching links.)

Because the application code was already working, this test should be GREEN:

Listing 8.25: GREEN

```
$ rails test test/integration/users_login_test.rb
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm by commenting out everything after **`if user`** in Line 8 of Listing 8.15 that the tests still pass even if we don't authenticate the user by email and password, as shown in Listing 8.26. This is because Listing 8.9 doesn't test the case of a correct user email but incorrect password. Fix this serious omission in our test suite by adding a valid email to the Users login test by (Listing 8.27). Verify that the tests are RED, then remove the Line 8 comment to get back to GREEN. (Because it's so important, we'll add this test to the main code in Section 8.3.)

2. Use the "safe navigation" operator `&.` to simplify the boolean test in Line 8 of Listing 8.15, as shown in Line 8 of Listing 8.28.[12] This Ruby feature allows us to condense the common pattern of `obj && obj.method` into `obj&.method`. Confirm that the tests in Listing 8.27 still pass after the change.

Listing 8.26: Commenting out the authentication code, but tests still GREEN.
app/controllers/sessions_controller.rb

```ruby
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user # && user.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Listing 8.27: Testing the case of valid user email, invalid password.
test/integration/users_login_test.rb

```ruby
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid email/invalid password" do
    get login_path
```

[12]Thanks to reader Aviv Levinsky for suggesting this addition.

```
    assert_template 'sessions/new'
    post login_path, params: { session: { email:    FILL_IN,
                                          password: "invalid" } }
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end
  .
  .
  .
end
```

Listing 8.28: Using the "safe navigation" operator `&.` to simplify the login code.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user&.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out
    redirect_to root_url
  end
end
```

8.2.5 Login upon signup

Although our authentication system is now working, newly registered users might be confused, as they are not logged in by default. Because it would be strange to force users to log in immediately after signing up, we'll log in new users automatically as part of the signup process. To arrange this behavior, all we need to do is add a call to **`log_in`** in the Users controller **`create`** action, as shown in Listing 8.29.[13]

Listing 8.29: Logging in the user upon signup.
`app/controllers/users_controller.rb`

```ruby
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

To test the behavior from Listing 8.29, we can add a line to the test from Listing 7.31 to check that the user is logged in. It's helpful in this context to define an **`is_logged_in?`** helper method to parallel the **`logged_in?`** helper defined in Listing 8.18, which returns **`true`** if there's a user id in the (test) session and false otherwise (Listing 8.30). (Because helper methods aren't available in tests, we can't use the **`current_user`** as in Listing 8.18, but the **`session`** method

[13]As with the Sessions controller, the **`log_in`** method is available in the Users controller because of the module inclusion in Listing 8.13.

is available, so we use that instead.) Here we use **`is_logged_in?`** instead of **`logged_in?`** so that the test helper and Sessions helper methods have different names, which prevents them from being mistaken for each other.[14] (In this case we could actually just include the Sessions helper and use **`logged_in?`** directly, but this technique would fail in Chapter 9 due to details of how cookies are handled in tests, so instead we define a test-specific method that will work in all cases.)

Listing 8.30: A boolean method for login status inside tests.
`test/test_helper.rb`

```ruby
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end
end
```

With the code in Listing 8.30, we can assert that the user is logged in after signup using the line shown in Listing 8.31.

Listing 8.31: A test of login after signup. GREEN
`test/integration/users_signup_test.rb`

```ruby
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest
  .
  .
  .
  test "valid signup information" do
```

[14]For example, I once had a test suite that was GREEN even after accidentally deleting the main **`log_in`** method in the Sessions helper. The reason is that the tests were happily using a test helper with the same name, thereby passing even though the application was completely broken. As with **`is_logged_in?`**, we'll avoid this issue by defining the test helper **`log_in_as`** in Listing 9.24.

```
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, params: { user: { name:  "Example User",
                                         email: "user@example.com",
                                         password:              "password",
                                         password_confirmation: "password" } }
    end
    follow_redirect!
    assert_template 'users/show'
    assert is_logged_in?
  end
end
```

At this point, the test suite should still be GREEN:

Listing 8.32: GREEN

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Is the test suite RED or GREEN if you comment out the **`log_in`** line in Listing 8.29?

2. By using your text editor's ability to comment out code, toggle back and forth between commenting out code in Listing 8.29 and confirm that the test suite toggles between RED and GREEN. (You will need to save the file between toggles.)

8.3 Logging out

As discussed in Section 8.1, our authentication model is to keep users logged in until they log out explicitly. In this section, we'll add this necessary logout