1. Write a test for the error messages implemented in Listing 7.20. How detailed you want to make your tests is up to you; a suggested template appears in Listing 7.25.

Listing 7.25: A template for tests of the error messages.
test/integration/users_signup_test.rb

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

  test "invalid signup information" do
    get signup_path
    assert_no_difference 'User.count' do
      post users_path, params: { user: { name:  "",
                                         email: "user@invalid",
                                         password:              "foo",
                                         password_confirmation: "bar" } }
    end
    assert_template 'users/new'
    assert_select 'div#<CSS id for error explanation>'
    assert_select 'div.<CSS class for field with error>'
  end
  .
  .
  .
end
```

7.4 Successful signups

Having handled invalid form submissions, now it's time to complete the signup form by actually saving a new user (if valid) to the database. First, we try to save the user; if the save succeeds, the user's information gets written to the database automatically, and we then *redirect* the browser to show the user's profile (together with a friendly greeting), as mocked up in Figure 7.20. If it fails, we simply fall back on the behavior developed in Section 7.3.

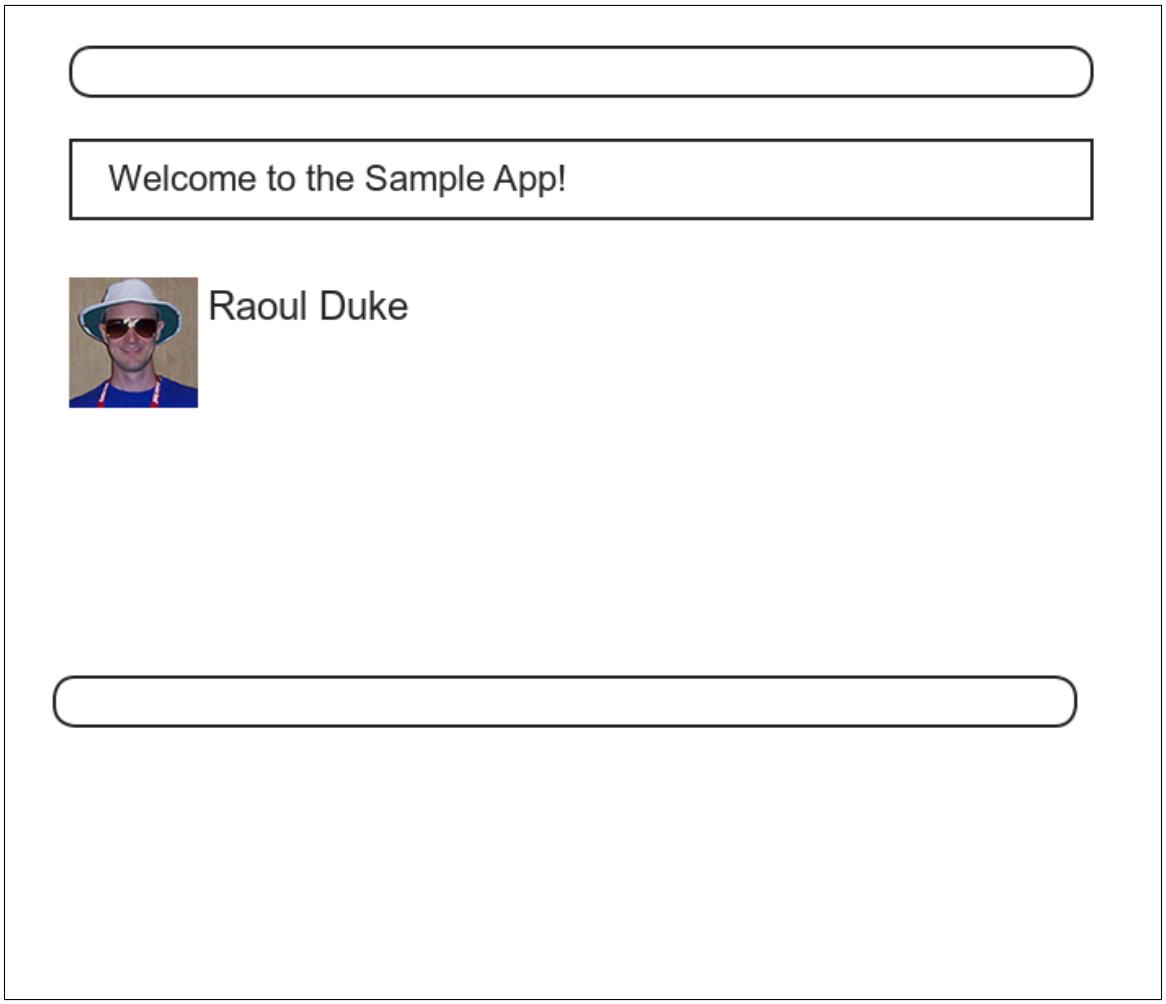

Figure 7.20: A mockup of successful signup.

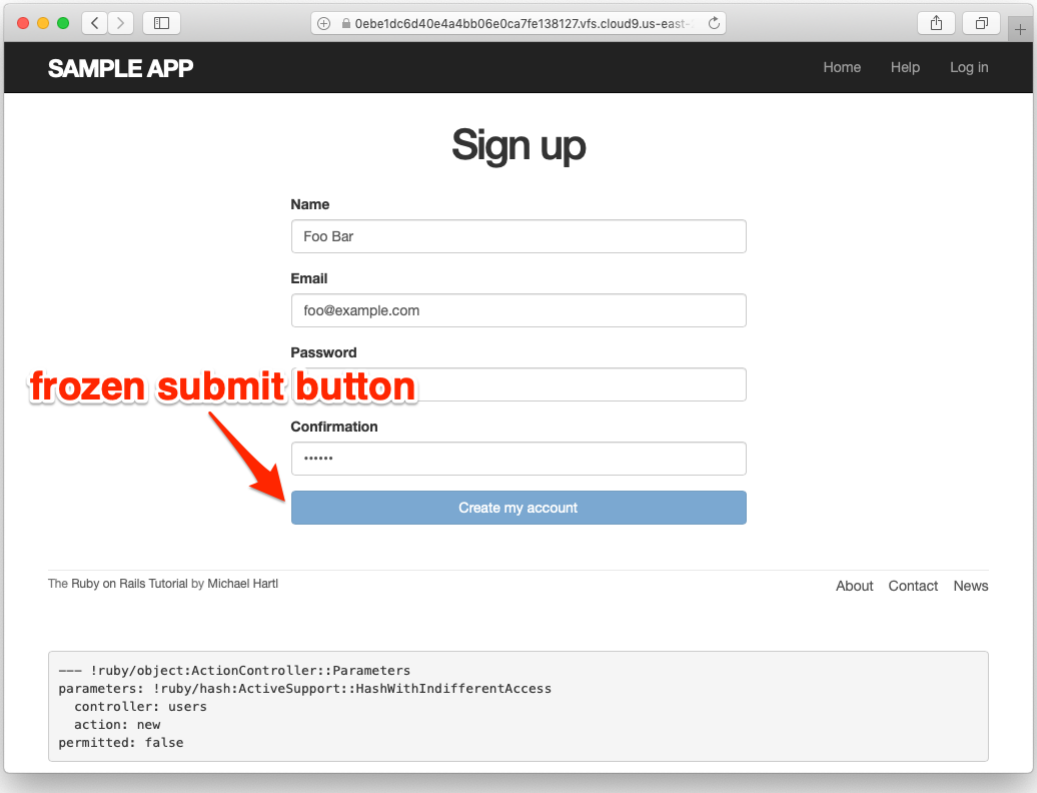

Figure 7.21: The frozen page on valid signup submission.

7.4.1 The finished signup form

To complete a working signup form, we need to fill in the commented-out section in Listing 7.19 with the appropriate behavior. Currently, the form simply freezes on valid submission, as indicated by the subtle color change in the submission button (Figure 7.21), although this behavior may be system-dependent. This is because the default behavior for a Rails action is to render the corresponding view, and there isn't a view template corresponding to the `create` action (Figure 7.22).

Although it's possible to render a template for the `create` action, the usual practice is to *redirect* to a different page instead when the creation is successful.

```
  ↳ app/controllers/users_controller.rb:13:in `create'
No template found for UsersController#create, rendering head :no_content
Completed 204 No Content in 332ms (ActiveRecord: 18.1ms | Allocations: 6720)
```

Figure 7.22: The **`create`** template error in the server log.

In particular, we'll follow the common convention of redirecting to the newly created user's profile, although the root path would also work. The application code, which introduces the **`redirect_to`** method, appears in Listing 7.26.

Listing 7.26: The user **`create`** action with a save and a redirect.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to @user
    else
      render 'new'
    end
  end

  private

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

Note that we've written

```
redirect_to @user
```

where we could have used the equivalent

```
redirect_to user_url(@user)
```

This is because Rails automatically infers from **`redirect_to @user`** that we want to redirect to **`user_url(@user)`**.

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 Rails console, verify that a user is in fact created when submitting valid information.

2. Confirm by updating Listing 7.26 and submitting a valid user that **`redirect_to user_url(@user)`** has the same effect as **`redirect_to @user`**.

7.4.2 The flash

With the code in Listing 7.26, our signup form is actually working, but before submitting a valid registration in a browser we're going to add a bit of polish common in web applications: a message that appears on the subsequent page (in this case, welcoming our new user to the application) and then disappears upon visiting a second page or on page reload.

The Rails way to display a temporary message is to use a special method called the *flash*, which we can treat like a hash. Rails adopts the convention of a **`:success`** key for a message indicating a successful result (Listing 7.27).

Listing 7.27: Adding a flash message to user signup.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      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
```

By assigning a message to the **flash**, we are now in a position to display the message on the first page after the redirect. Our method is to iterate through the **flash** and insert all relevant messages into the site layout. You may recall the console example in Section 4.3.3, where we saw how to iterate through a hash using the strategically named **flash** variable (Listing 7.28).

Listing 7.28: Iterating through a **flash** hash in the console.

```
$ rails console
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", danger: "It failed."}
>> flash.each do |key, value|
?>   puts "#{key}"
?>   puts "#{value}"
>> end
success
It worked!
danger
It failed.
```

By following this pattern, we can arrange to display the contents of the flash site-wide using code like this:

```
<% flash.each do |message_type, message| %>
  <div class="alert alert-<%= message_type %>"><%= message %></div>
<% end %>
```

(This code is a particularly ugly and difficult-to-read combination of HTML and ERb; making it prettier is left as an exercise (Section 7.4.4).) Here the embedded Ruby

```
alert-<%= message_type %>
```

makes a CSS class corresponding to the type of message, so that for a **`:success`** message the class is

```
alert-success
```

(The key **`:success`** is a symbol, but embedded Ruby automatically converts it to the string **`"success"`** before inserting it into the template.) Using a different class for each key allows us to apply different styles to different kinds of messages. For example, in Section 8.1.4 we'll use **`flash[:danger]`** to indicate a failed login attempt.[12] (In fact, we've already used **`alert-danger`** once, to style the error message div in Listing 7.21.) Bootstrap CSS supports styling for four such flash classes for increasingly urgent message types (**`success`**, **`info`**, **`warning`**, and **`danger`**), and we'll find occasion to use all of them in the course of developing the sample application (**`info`** in Section 11.2, **`warning`** in Section 11.3, and **`danger`** for the first time in Section 8.1.4).

Because the message is also inserted into the template, the full HTML result for

```
flash[:success] = "Welcome to the Sample App!"
```

appears as follows:

[12] Actually, we'll use the closely related **`flash.now`**, but we'll defer that subtlety until we need it.

```
<div class="alert alert-success">Welcome to the Sample App!</div>
```

Putting the embedded Ruby discussed above into the site layout leads to the code in Listing 7.29.

Listing 7.29: Adding the contents of the `flash` variable to the site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  .
  .
  .
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <% flash.each do |message_type, message| %>
        <div class="alert alert-<%= message_type %>"><%= message %></div>
      <% end %>
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
    .
    .
    .
  </body>
</html>
```

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. In the console, confirm that you can use interpolation (Section 4.2.1) to interpolate a raw symbol. For example, what is the return value of `"#{:success}"`?

2. How does the previous exercise relate to the flash iteration shown in Listing 7.28?

7.4.3 The first signup

We can see the result of all this work by signing up the first user for the sample app. Even though previous submissions didn't work properly (as shown in Figure 7.21), the `user.save` line in the Users controller still works, so users might still have been created. To clear them out, we'll reset the database as follows:

```
$ rails db:migrate:reset
```

On some systems you might have to restart the webserver (using Ctrl-C) for the changes to take effect (Box 1.2).

We'll create the first user with the name "Rails Tutorial" and email address "example@railstutorial.org", as shown in Figure 7.23). The resulting page (Figure 7.24) shows a friendly flash message upon successful signup, including nice green styling for the `success` class, which comes included with the Bootstrap CSS framework from Section 5.1.2. Then, upon reloading the user show page, the flash message disappears as promised (Figure 7.25).

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 Rails console, find by the email address to double-check that the new user was actually created. The result should look something like Listing 7.30.

2. Create a new user with your primary email address. Verify that the Gravatar correctly appears.

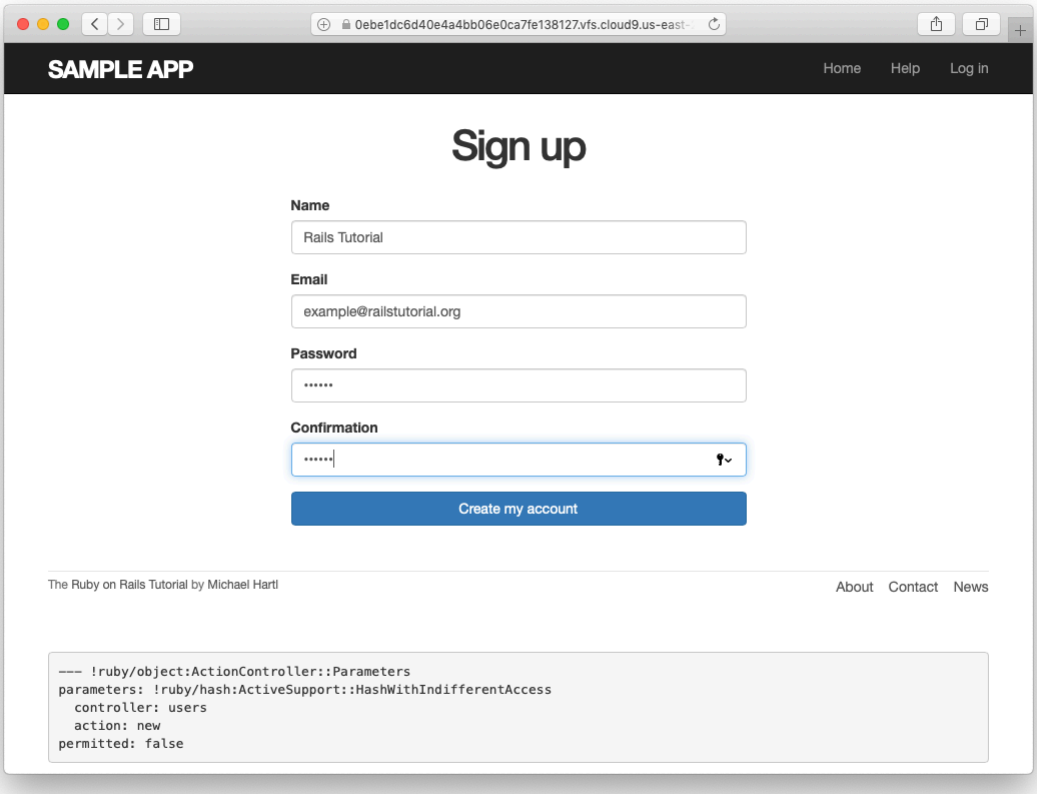

Figure 7.23: Filling in the information for the first signup.

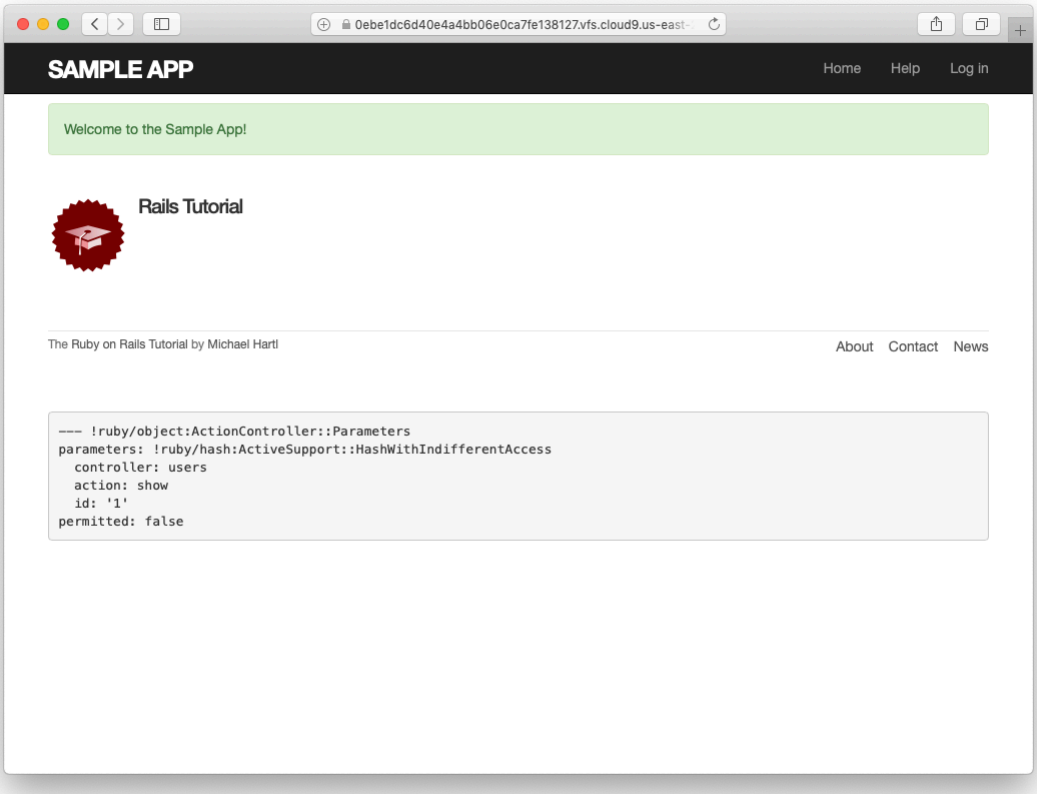

Figure 7.24: The results of a successful user signup, with flash message.

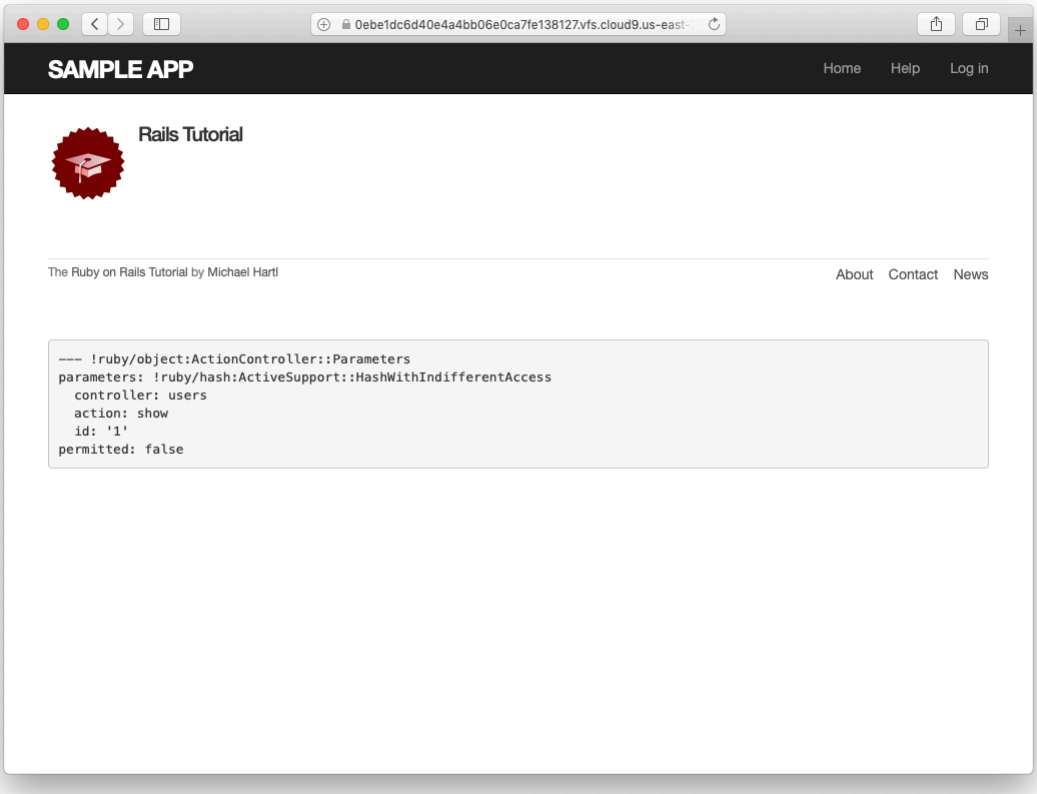

Figure 7.25: The `flash`-less profile page after a browser reload.

Listing 7.30: Finding the newly created user in the database.

```
$ rails console
>> User.find_by(email: "example@railstutorial.org")
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.
org", created_at: "2016-05-31 17:17:33", updated_at: "2016-05-31 17:17:33",
password_digest: "$2a$10$8MaeHdnOhZvMk3GmFdmpPOeG6a7u7/k2Z9TMjOanC9G...">
```

7.4.4 A test for valid submission

Before moving on, we'll write a test for valid submission to verify our application's behavior and catch regressions. As with the test for invalid submission in Section 7.3.4, our main purpose is to verify the contents of the database. In this case, we want to submit valid information and then confirm that a user *was* created. In analogy with Listing 7.23, which used

```
assert_no_difference 'User.count' do
  post users_path, ...
end
```

here we'll use the corresponding **`assert_difference`** method:

```
assert_difference 'User.count', 1 do
  post users_path, ...
end
```

As with **`assert_no_difference`**, the first argument is the string **`'User.count'`**, which arranges for a comparison between **`User.count`** before and after the contents of the **`assert_difference`** block. The second (optional) argument specifies the size of the difference (in this case, 1).

Incorporating **`assert_difference`** into the file from Listing 7.23 yields the test shown in Listing 7.31. Note that we've used the **`follow_redirect!`** method after posting to the users path. This simply arranges to follow the redirect after submission, resulting in a rendering of the **`'users/show'`** template. (It's probably a good idea to write a test for the flash as well, which is left as an exercise (Section 7.4.4).)

Listing 7.31: A test for a valid signup. GREEN
test/integration/users_signup_test.rb

```
require 'test_helper'

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

Note that Listing 7.31 also verifies that the user show template renders following successful signup. For this test to work, it's necessary for the Users routes (Listing 7.3), the Users **show** action (Listing 7.5), and the **show.html.erb** view (Listing 7.8) to work correctly. As a result, the one line

```
assert_template 'users/show'
```

is a sensitive test for almost everything related to a user's profile page. This sort of end-to-end coverage of important application features illustrates one reason why integration tests are so useful.

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. Write a test for the flash implemented in Section 7.4.2. How detailed you want to make your tests is up to you; a suggested ultra-minimalist template appears in Listing 7.32, which you should complete by replacing `FILL_IN` with the appropriate code. (Even testing for the right key, much less the text, is likely to be brittle, so I prefer to test only that the flash isn't empty.)

2. As noted above, the flash HTML in Listing 7.29 is ugly. Verify by running the test suite that the cleaner code in Listing 7.33, which uses the Rails `content_tag` helper, also works.

3. Verify that the test fails if you comment out the redirect line in Listing 7.26.

4. Suppose we changed `@user.save` to `false` in Listing 7.26. How does this change verify that the `assert_difference` block is testing the right thing?

Listing 7.32: A template for a test of the flash.
test/integration/users_signup_test.rb

```
require 'test_helper'
  .
  .
  .
  test "valid signup information" do
    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_not flash.FILL_IN
  end
end
```

Listing 7.33: The `flash` ERb in the site layout using `content_tag`.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
      .
      .
      .
      <% flash.each do |message_type, message| %>
        <%= content_tag(:div, message, class: "alert alert-#{message_type}") %>
      <% end %>
      .
      .
      .
</html>
```

7.5 Professional-grade deployment

Now that we have a working signup page, it's time to deploy our application and get it working in production. Although we started deploying our application in Chapter 3, this is the first time it will actually *do* something, so we'll take this opportunity to make the deployment professional-grade. In particular, we'll add an important feature to the production application to make signup secure, we'll replace the default webserver with one suitable for real-world use, and we'll add some configuration for our production database.

As preparation for the deployment, you should merge your changes into the `master` branch at this point:

```
$ git add -A
$ git commit -m "Finish user signup"
$ git checkout master
$ git merge sign-up
```

7.5.1 SSL in production

When submitting the signup form developed in this chapter, the name, email address, and password get sent over the network, and hence are vulnerable to