Chapter 8 Sign up

Now that we have a working User model, it's time to add an ability few websites can live with out: letting users sign up for the site—thus fulfilling the promise implicit in Section 5.3, "User signup: A first step". We'll use an HTML *form* to submit user signup information to our application in Section 8.1, which will then be used to create a new user and save its attributes to the database in Section 8.3. As usual, we'll write tests as we develop, and in Section 8.4 we'll combine RSpec with the remarkable *Webrat* utility to write succinct and expressive integration tests.

Since we'll be creating a new user in this chapter, you might want to reset the database to clear out any users created at the console (e.g., in Section 7.3.2), so that your results will match those shown in the tutorial. You can do this as follows:

```
$ rake db:reset
```

If you're following along with version control, make a topic branch as usual:

```
$ git checkout master
$ git checkout -b signing-up
```

8.1 Signup form

Recall from Section 5.3.1 that we already have tests for the new users (signup) page, originally seen in Listing 5.25 and reproduced in Listing 8.1. (As promised in Section 7.3.1, we've switched from `get 'new'` to `get :new` because that's what my fingers want to type.) In addition, we saw in Figure 5.11 (shown again in Figure 8.1) that this signup page is currently blank: useless for signing up new users. The goal of this section is to start changing this sad state of affairs by producing the signup form mocked up in Figure 8.2.

Listing 8.1. The tests for the new users page (first seen in Listing 5.25).
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController 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 up/)
    end
  end
  .
  .
  .
end
```

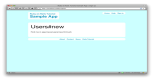
Figure 8.1: The current state of the signup page `/signup`. (full size)

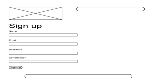
Figure 8.2: A mockup of the user signup page. (full size)

8.1.1 Using `form_for`

The HTML element needed for submitting information to a remote website is a *form*, which suggests a good first step toward registering users is to make a form to accept their signup information. We can accomplish this in Rails with the `form_for` helper method; the result appears in Listing 8.2.

Listing 8.2. A form to sign up new users.
`app/views/users/new.html.erb`

```
<h2>Sign up</h2>

<% form_for(@user) do |f| %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <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="field">
    <%= f.label :password_confirmation, "Confirmation" %><br />
    <%= f.password_field :password_confirmation %>
  </div>
  <div class="actions">
    <%= f.submit "Sign up" %>
  </div>
<% end %>
```

Let’s break this down into pieces. The presence of the `do` keyword indicates that `form_for` takes a block (Section 4.3.2), which has one variable, which we’ve called `f` for “form”. Inside of the `form_for` helper, `f` is an object that represents a form; as is usually the case with Rails helpers, we don't need to know any details about the implementation, but what we *do* need to know is what the `f` object does: when called with a method corresponding to an HTML form element—such as a text field, radio button, or password field—it returns code for that element specifically designed to set an attribute of the `@user` object. In other words,

```
<div class="field">
  <%= f.label :name %><br />
  <%= f.text_field :name %>
</div>
```

creates the HTML needed to make a labeled text field element appropriate for setting the `name` attribute of a User model.

To see this in action, we need to drill down and look at the actual HTML produced by this form, but here we have a problem: the page currently breaks, because we have not set the `@user` variable—like all undefined instance variables (Section 4.2.3), `@user` is currently `nil`. Appropriately, if you run your test suite at this point, you'll see that the signup page tests fail. To get them to pass and get our form to render, we must define an `@user` variable in the controller action corresponding to `new.html.erb`, i.e., the `new` action in the Users controller. The `form_for` helper expects `@user` to be a User object, and since we're creating a *new* user we simply use `User.new`, as seen in Listing 8.3.

Listing 8.3. Adding an `@user` variable to the `new` action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def new
    @user = User.new
    @title = "Sign up"
  end
end
```

With the `@user` variable so defined, the tests should be passing again,[1] and now the form (with the tiny bit of styling from Listing 8.4) appears as in Figure 8.3.

Listing 8.4. A wafer-thin amount of CSS for the signup form.
`public/stylesheets/custom.css`

```
.
.
.
div.field, div.actions {
  margin-bottom: 10px;
}
```

Figure 8.3: The signup form `/signup` for new users. (full size)

8.1.2 The form HTML

As indicated by Figure 8.3, the signup page now renders properly, indicating that the `form_for` code in Listing 8.2 is producing valid HTML. If you look at the HTML for the generated form (using either Firebug or the “view page source” feature of your browser), you should see markup as in Listing 8.5. Although many of the details are irrelevant for our purposes, let’s take a moment to highlight the most important parts of its structure.

Listing 8.5. The HTML for the form in Figure 8.3.

```
<form action="/users" class="new_user" id="new_user" method="post">
<div style="margin:0;padding:0;display:inline">
<input name="authenticity_token" type="hidden"
       value="rB82sI7Qw5J9J1UMILG/VQL411vH5putR+JwlxLScMQ=" />
</div>

  <div class="field">
    <label for="user_name">Name</label><br />
    <input id="user_name" name="user[name]" size="30" type="text" />
  </div>

  <div class="field">
    <label for="user_email">Email</label><br />
    <input id="user_email" name="user[email]" size="30" type="text" />
  </div>
  <div class="field">
    <label for="user_password">Password</label><br />
    <input id="user_password" name="user[password]" size="30" type="password" />
  </div>

  <div class="field">
    <label for="user_password_confirmation">Confirmation</label><br />
    <input id="user_password_confirmation" name="user[password_confirmation]"
           size="30" type="password" />
  </div>
  <div class="actions">
    <input id="user_submit" name="commit" type="submit" value="Sign up" />
  </div>
</form>
```

We’ll start with the internal structure. Comparing Listing 8.2 with Listing 8.5, we see that the Embedded Ruby

```
<div class="field">
  <%= f.label :name %><br />
  <%= f.text_field :name %>
</div>
```

produces the HTML

```
<div class="field">
  <label for="user_name">Name</label><br />
  <input id="user_name" name="user[name]" size="30" type="text" />
</div>
```

and

```
<div class="field">
  <%= f.label :password %><br />
  <%= f.password_field :password %>
</div>
```

produces the HTML

```
<div class="field">
  <label for="user_password">Password</label><br />
  <input id="user_password" name="user[password]" size="30" type="password" />
</div>
```

As seen in Figure 8.4, text fields (`type="text"`) simply display their contents, whereas password fields (`type="password"`) obscure the input for security purposes, as seen in Figure 8.4.

Figure 8.4: A filled-in form, showing the difference between `text` and `password` fields. (full size)

As we'll see in Section 8.3, the key to creating a user is the special `name` attribute in each `input`:

```
<input id="user_name" name="user[name]" - - - />
.
.
.
<input id="user_password" name="user[password]" - - - />
```

These `name` values allow Rails to construct an initialization hash (via the `params` variable first seen in Section 6.3.2) for creating users using the values entered by the user, as we'll see in Section 8.2.

The second important element is the `form` tag itself. Rails creates the `form` tag using the `@user` object: because every Ruby object knows its own class (Section 4.4.1), Rails figures out that `@user` is of class `User`; moreover, since `@user` is a *new* user, Rails knows to construct a form with the `post` method, which is the proper verb for creating a new object (Box 3.1):

```
<form action="/users" class="new_user" id="new_user" method="post">
```

Here the `class` and `id` attributes are largely irrelevant; what's important is `action="/users"` and `method="post"`. Together, these constitute instructions to issue an HTML `POST` request to the `/users` URL. We'll see in the next two sections what effects this has.

Finally, note the rather obscure code for the "authenticity token":

```
<div style="margin:0;padding:0;display:inline">
<input name="authenticity_token" type="hidden"
       value="rB82sI7Qw5J9J1UMILG/VQL411vH5putR+JwlxLScMQ=" />
</div>
```

Here Rails uses a special unique value to thwart a particular kind of cross-site scripting attack called a *forgery*; see here if you’re interested in the details of how this works and why it’s important. Happily, Rails takes care of the problem for you, and the input tag is `hidden` so you don’t really have to give it a second thought, but it shows up when you view the form source so I wanted at least to address it.

8.2 Signup failure

Though we’ve briefly examined the HTML for the form in Figure 8.3 (shown in Listing 8.5), it’s best understood in the context of *signup failure*. Just getting a signup form that accepts an invalid submission and re-renders the signup page (as mocked up in Figure 8.5) is a significant accomplishment, and it’s the goal of this section.

Figure 8.5: A mockup of the signup failure page. (full size)

8.2.1 Testing failure

Recall from Section 6.3.3 that adding `map.resources :users` to the `routes.rb` file (Listing 6.24) automatically ensures that our Rails application responds to the RESTful URLs from Table 6.2. In particular, it ensures that a `POST` request to `/users` is handled by the `create` action. Our strategy for the `create` action is to use the form submission make a new user object using `User.new`, try (and fail) to save that user, and then render the signup page for possible resubmission. Our task is to write tests for this action, and then add `create` to the Users controller to get it to pass.

Let’s get started by reviewing the code for the signup form:

```
<form action="/users" class="new_user" id="new_user" method="post">
```

As noted in Section 8.1.2, this HTML issues a `POST` request to the `/users` URL. In analogy with the `get` method, which issues a `GET` request inside of tests, we use the `post` method to issue a `POST` request to the `create` action. As we’ll see shortly, `create` takes in a hash corresponding to the object type being created; since this is a test for signup *failure*, we’ll just pass an `@attr` hash with blank entries, as seen in Listing 8.6. This is essentially equivalent to visiting the signup page and clicking on the button without filling in any of the fields.

Listing 8.6. Tests for failed user signup.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views
  .
  .
  .

  describe "POST 'create'" do
```

```
    describe "failure" do

      before(:each) do
        @attr = { :name => "", :email => "", :password => "",
                  :password_confirmation => "" }
        @user = Factory.build(:user, @attr)
        User.stub!(:new).and_return(@user)
        @user.should_receive(:save).and_return(false)
      end

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

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

By themselves, these two tests are relatively straightforward: we make sure that the title is correct, and then we check that a failed signup attempt just re-renders the new user page. On the other hand, the `before(:each)` block is a little tricky—its purpose is to verify that the `create` action attempts to `save` the user. To understand it, let's break it down into pieces:

```
before(:each) do
  @attr = { :name => "", :email => "", :password => "",
            :password_confirmation => "" }
  @user = Factory.build(:user, @attr)
  User.stub!(:new).and_return(@user)
  @user.should_receive(:save).and_return(false)
end
```

First, note that we use Factory Girl not to simulate an *existing* user (as in, e.g., Listing 7.17), but rather to `build` a *new* user based on a particular attributes hash:[2]

```
@attr = { :name => "", :email => "", :password => "",
          :password_confirmation => "" }
@user = Factory.build(:user, @attr)
```

We then use an RSpec stub (first seen in Listing 7.17) to ensure that a call to `User.new` returns our factory user:

```
User.stub!(:new).and_return(@user)
```

The final line sets up a *message expectation*, indicating that the `@user` object should receive the `save` method in each test:

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

It's essential to understand that, unlike stubs, which merely return values, message expectations are *requirements*. In other words,

```
User.stub!(:new).and_return(@user)
```

ensures that a call to `User.new` returns `@user`, but it doesn't *require* it, whereas

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

leads to an *error* if `@user.save` is never called. This expectation error ensures that the tests initially fail.

It's also important to understand that the expectation

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

arranges for `@user.save` to return `false`. (Recall from Section 6.1.3 that `@user.save` returns `true` if the save succeeds and `false` if it fails.) In particular, it doesn't *require* the method to return `false`; the expectation *enforces* it (Box 8.1).

Box 8.1.An expectation subtlety

Naïvely, it seems that the code

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

requires that `@user.save` gets called *and* requires that it return `false`—but this is not the case. Instead, though it *does* require `@user.save` to be called, it intercepts the call and *arranges* for it to return `false`. The first part is an expectation; the second part is a guarantee.

Why does a message expectation not verify the return value? Doesn't that need to be tested as well? It does—and it has already been tested in the model layer (Chapter 6). Inside a controller test, we *assume* that the model layer does its job correctly. The purpose of a message expectation is to verify that the right methods get called, and then (like stubs) to supply a return value without ever hitting the database.

8.2.2 A working form

We can get the tests from Section 8.2.1 to pass with the code in Listing 8.7. This listing includes a second use of the `render` method, which we first saw in the context of partials (Section 5.1.3); as you can see, `render` works in controller actions as well. Note that we've taken this opportunity to introduce an `if-else` branching structure, which allows us to handle the cases of failure and success separately based on the value of `@user.save`.

Listing 8.7. A `create` action that can handle signup failure (but not success).
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(params[:user])
    if @user.save
      # Handle a successful save.
    else
      @title = "Sign up"
      render 'new'
    end
  end
end
```

The best way understand how the code in [Listing 8.7]() works is to *submit* the form with some invalid signup data; the results appear in [Figure 8.6]().

Figure 8.6: Signup failure with a `params` hash. [(full size)]()

To get a clearer picture of how Rails handles the submission, let's take a closer look at the `params` hash in the debug information at the bottom of [Figure 8.6]():

```
--- !map:HashWithIndifferentAccess
commit: Sign up
authenticity_token: rB82sI7Qw5J9J1UMILG/VQL411vH5puR+Jw1xL5cMQ=
action: create
controller: users
user: !map:HashWithIndifferentAccess
  name: Foo Bar
  password_confirmation: dude
  password: dude
  email: foo@invalid
```

We saw starting in [Section 6.3.2]() that the `params` hash contains information about each request; in the case of a URL like `/users/1`, the value of `params[:id]` is the `id` of the corresponding user (`1` in this example). In the case of posting to the signup form, `params` instead contains a hash of hashes, a construction we first saw in [Section 4.3.3](), which introduced the strategically named `params` variable in a console session. This debug information above shows that submitting the form results in a `user` hash with attributes corresponding to the submitted values, where the keys come from the `name` attributes of the `input` tags seen in [Listing 8.2](); for example, the value of

```
<input id="user_email" name="user[email]" size="30" type="text" />
```

with name `"user[email]"` is precisely the `email` attribute of the `user` hash.

Though the hash keys appears as strings in the debug output, internally Rails uses symbols, so that `params[:user]` is the hash of user attributes—in fact, exactly the attributes needed as an argument to `User.new`, as first seen in [Section 4.4.5]() and appearing in [Listing 8.7](). This means that the line

```
@user = User.new(params[:user])
```

is equivalent to

```
@user = User.new(:name => "Foo Bar", :email => "foo@invalid",
                 :password => "dude", :password_confirmation => "dude")
```

This is exactly the format needed to initialize a User model object with the given attributes.

Of course, instantiating such a variable has implications for successful signup—as we'll see in

Section 8.3, once `@user` is defined properly, calling `@user.save` is all that's needed to complete the registration—but it has consequences even in the failed signup considered here. Note in Figure 8.6 that the fields are *pre-filled* with the data from the failed submission. This is because `form_for` automatically fills in the fields with the attributes of the `@user` object, so that, for example, if `@user.name` is `"Foo"` then

```
<% form_for(@user) do |f| %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  .
  .
  .
```

will produce the HTML

```
<form action="/users" class="new_user" id="new_user" method="post">

  <div class="field">
    <label for="user_name">Name</label><br />
    <input id="user_name" name="user[name]" size="30" type="text" value="Foo"/>
  </div>
  .
  .
  .
```

Here the `value` of the `input` tag is `"Foo"`, so that's what appears in the text field.

8.2.3 Signup error messages

Though not strictly necessary, it's helpful to output error messages on failed signup to indicate the problems that prevented successful user registration. Rails provides just such messages based on the User model validations. For example, consider trying to save a user with invalid email and with a short password:

```
$ script/console
>> user = User.new(:name => "Foo Bar", :email => "foo@invalid",
?>                 :password => "dude", :password_confirmation => "dude")
>> user.save
=> false
>> user.errors.full_messages
=> ["Email is invalid", "Password is too short (minimum is 6 characters)"]
```

Here the `errors.full_messages` object (which we saw briefly in Section 6.2.1) contains an array of error messages.

As in the console session above, the failed save in Listing 8.7 generates a list of error messages associated with the `@user` object. To display the messages in the browser, we use the special `error_messages` object on the form variable `f`, as seen in Listing 8.8.

Listing 8.8. Code to add error message display to the signup form.

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

```
<h2>Sign up</h2>

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

The `error_messages` method inserts the HTML needed for a list of form submission errors, including CSS ids and classes for use in styling the response.3 After adding the CSS in Listing 8.9, on failed submission the error messages appear as in Figure 8.7. Because the messages are generated by the model validations, they will automatically change if you ever change your mind about, say, the format of email addresses, or the minimum length on passwords.

Listing 8.9. CSS for styling error messages.
`public/stylsheets/custom.css`

```
.
.
.
.fieldWithErrors {
  padding: 2px;
  background-color: red;
  display: table;
}

.fieldWithErrors label {
  color: #fff;
}

#errorExplanation {
  width: 400px;
  border: 2px solid red;
  padding: 7px;
  padding-bottom: 12px;
  margin-bottom: 20px;
  background-color: #f0f0f0;
}

#errorExplanation h2 {
  text-align: left;
  font-weight: bold;
  padding: 5px 5px 5px 15px;
  font-size: 12px;
  margin: -7px;
  background-color: #c00;
  color: #fff;
}

#errorExplanation p {
  color: #333;
  margin-bottom: 0;
  padding: 5px;
}

#errorExplanation ul li {
  font-size: 12px;
  list-style: square;
}
```

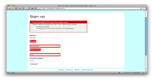
Figure 8.7: Failed signup with error messages. (full size)

8.2.4 Filtering parameter logging

Before moving on to successful signup, there's one loose end to tie off. You might have noticed that, even though we went to great pains to encrypt the password in Chapter 7, both the password and its confirmation appear as cleartext in the debug information. By itself this is no problem—recall from Listing 6.21 that this information only appears for applications running in `development` mode, so actual users would never see it—but it does hint at a real problem: the passwords also appear unencrypted in the *log file* that Rails uses to record information about the running application. Indeed, if you look in the development log file you should see lines like those shown in Listing 8.10. (This information also appears in the server log, which we saw in Figure 1.7.)

Listing 8.10. The development log with visible passwords.
`log/development.log`

```
Parameters: {"commit"=>"Sign up", "action"=>"create",
"authenticity_token"=>"K1HchFF8uYE8ZaQKz5DVG9vF2KGoXJu4JGp/VE3NMjA=",
"controller"=>"users",
  "user"=>{"name"=>"Foo Bar", "password_confirmation"=>"dude",
           "password"=>"dude", "email"=>"foo@invalid"}}
```

It would be a terrible security breach to store unencrypted passwords in the log files—if anyone ever got a hold of the file, they would potentially obtain the passwords for every user on the system. (Of course, here the signup fails, but the problem is exactly the same for successful submissions.) Happily, Rails offers a simple solution: just *filter* the password parameter so that it doesn't appear in the log (Listing 8.11).4 After you add this line, the log appears as in Listing 8.12. (Of course, in production the log file will be `log/production.log`, and the filtering will work the same way.)

Listing 8.11. Code to filter `password` and `password_confirmation` from the log.
`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  .
  .
  .
  # Scrub sensitive parameters from your log
  filter_parameter_logging :password
end
```

Listing 8.12. The development log with filtered passwords.
`log/development.log`

```
Parameters: {"commit"=>"Sign up", "action"=>"create",
"authenticity_token"=>"K1HchFF8uYE8ZaQKz5DVG9vF2KGoXJu4JGp/VE3NMjA=",
"controller"=>"users",
  "user"=>{"name"=>"Foo Bar", "password_confirmation"=>"[FILTERED]",
           "password"=>"[FILTERED]", "email"=>"foo@invalid"}}
```

8.3 Signup success

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 8.8. If it fails, we simply fall back on the behavior developed in Section 8.2.

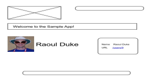

Figure 8.8: A mockup of successful signup. (full size)

8.3.1 Testing success

The tests for a successful signup follow the lead of the failed signup tests from Listing 8.6. Let’s take a look at the result in Listing 8.13.

Listing 8.13. Tests for signup success.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views
  .
  .
  .
  describe "POST 'create'" do
    .
    .
    .
    describe "success" do

      before(:each) do
        @attr = { :name => "New User", :email => "user@example.com",
                  :password => "foobar", :password_confirmation => "foobar" }
        @user = Factory(:user, @attr)
        User.stub!(:new).and_return(@user)
        @user.should_receive(:save).and_return(true)
      end

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

As with the signup failure tests (Listing 8.6), here we use `post :create` to hit the `create` action with an HTTP `POST` request. Unlike in Listing 8.6, which used `Factory.build` to simulate a new user, Listing 8.13 uses `Factory(:user)` to simulate a *saved* user. As before, our expectation is that `@user.save` will be called, but since this is a test for a successful save we arrange for it to return `true`:5

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

After a successful save, we check for a redirect to the user show page using this code:

```
response.should redirect_to(user_path(@user))
```

This is the kind of redirect that happens on nearly every successful form submission on the web, and with RSpec's helpful syntax you don't have to know anything about the underlying HTTP response code.6 The URL itself is generated using the named route `user_path` shown in Table 7.1.

8.3.2 The finished signup form

To get these tests to pass and thereby complete a working signup form, fill in the commented-out section in Listing 8.7 with a redirect, as shown in Listing 8.14.

Listing 8.14. The user `create` action with a save and a redirect.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to @user
    else
      @title = "Sign up"
      render 'new'
    end
  end
end
```

Note that we can omit the `user_path` in the redirect, writing simply `redirect_to @user` to redirect to the user show page, a convention we saw before with `link_to` in Listing 7.24. This syntax is nicely succinct, but unfortunately RSpec doesn't understand it, so we have to use the more verbose `user_path(@user)` in that case.

8.3.3 The flash

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 temporarily and then disappears upon page reload. (If this is unclear now, be patient; a concrete example appears shortly.) The Rails way to accomplish this is to use a special variable called the *flash*, which operates like flash memory in that it stores its data temporarily. The `flash` variable is effectively a hash; you may even recall the console example in Section 4.3.3, where we saw how to iterate through a hash using a strategically named `flash` hash. To recap, try this console session:

```
$ script/console
>> flash = { :success => "It worked!", :error => "It failed. :-(" }
=> {:success=>"It worked!", :error => "It failed. :-("}
>> flash.each do |key, value|
?>   puts "#{key}"
?>   puts "#{value}"
>> end
success
It worked!
error
It failed. :-(
```

We can arrange to display the contents of the flash site-wide by including it in our application layout, as in Listing 8.15.

Listing 8.15. Adding the contents of the `flash` variable to the site layout.
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
      .
      .
      .
      <div id="content" class="round">
        <% flash.each do |key, value| %>
          <div class="flash <%= key %>"><%= value %></div>
        <% end %>
        <%= yield %>
      </div>
      .
      .
      .
</html>
```

This code arranges to insert a `div` tag for each element in the flash, with a CSS class indicating the type of message. For example, if `flash[:success] = "Welcome to the Sample App!"`, then the code

```
<% flash.each do |key, value| %>
  <div class="flash <%= key %>"><%= value %></div>
```

```
<% end %>
```

will produce this HTML:7

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

The reason we iterate through all possible key/value pairs is so that we can include other kinds of flash messages; for example, in Listing 9.8 we'll see `flash[:error]` used to indicate a failed signin attempt.8

Let's test for the right flash message by making sure the right message appears under the key `:success` (Listing 8.16).

Listing 8.16. A test for a flash message on successful user signup.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

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

    describe "success" do
      .
      .
      .

      it "should have a welcome message" do
        post :create, :user => @attr
        flash[:success].should =~ /welcome to the sample app/i
      end
    end
  end
end
```

This uses the case-insensitive regular expression `/.../i`, a construction we last saw in the `EmailRegex` of Listing 6.15. Note that we don't yet test for the appearance of the actual flash message HTML; we'll fix this by testing for the actual `div` tag in Section 8.4.3.

We get the flash test to pass by assigning to `flash[:success]` in the `create` action as in Listing 8.17. The message uses different capitalization from the one in the test, but the test passes anyway because of the `i` at the end of the regular expression. This way we won't break the test if we write, e.g., `sample app` in place of `Sample App`.

Listing 8.17. Adding a flash message to user signup.
`app/controllers/users_controller.rb`

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

8.3.4 The first signup

We can see the result of all this work by signing up our first user (under the name "Rails Tutorial" and email address "`example@railstutorial.org`"), which shows a friendly message upon successful signup, as seen in Figure 8.9. (The nice green styling for the `success` class comes included with the Blueprint CSS framework from Section 4.1.2.) Then, upon reloading the user show page, the flash message disappears as promised (Figure 8.10).

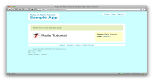
Figure 8.9: The results of a successful user signup, with flash message. (full size)

Figure 8.10: The flash-less profile page after a browser reload. (full size)

We can now check our database just to be double-sure that the new user was actually created:

```
$ script/console
>> user = User.first
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.org",
created_at: "2010-02-17 03:07:53", updated_at: "2010-02-17 03:07:53",
encrypted_password: "48aa8f4444b71f3f713d87d051819b0d44cd89f4a963949f201...",
salt: "f52924ba502d4f92a634d4f9647622ccce26205176cceca2adc...">
```

Success!

8.4 RSpec integration tests

In principle we are done with user signup at this point, but you may have noticed that we haven't tested the structure of the signup form, nor have we tested that submissions actually work. Of course, we *have* checked these things by viewing the pages in our browser, but the whole point of automated testing is to make sure that once things work they stay that way. Making such tests is the goal of this section—and the results are pretty sweet.

One testing method would be to check the HTML structure of the form (using `integrate_views` and the `have_tag` method), and indeed this is a good way to test-drive views. (Section 8.6 has an exercise to this effect.) But I prefer not to test the detailed HTML structure of views—I don't see any reason why we should have to know that Rails implements user email submission using `name="user[email]"`, and indeed any test of that structure would break if a future Rails version changed this convention. Moreover, it would be nice to have a test for the entire signup process: visiting the signup page, filling in the form values, clicking the button, and making sure (if the submission is valid) that a new user gets created in the (test) database.

Though it's not the only way (see Box 8.2), my preferred solution to this problem is to use an RSpec integration test, which we first used in Section 5.2.1 to test the custom routes (such as `/about` for the About page). In that section, we saw only a tiny fraction of the power of integration tests; below, we'll see just how amazing they can be.

Box 8.2.Integration alternatives

As we've seen in this and previous chapters, *Ruby on Rails Tutorial* uses RSpec for all its tests, including integration tests. In my view, there is no match for the simplicity and power of RSpec integration tests, especially when combined with the expressiveness of Webrat. There are a couple of viable alternatives, though. One is the Rails default, integration testing with `Test::Unit`. This is fine if you use `Test::Unit` elsewhere, but we're using RSpec in this tutorial, and I prefer not to mix RSpec and `Test::Unit` in a single project. A second option is Cucumber, which works nicely with RSpec and allows the definition of plain-text stories describing application behavior. Cucumber is quite powerful and has a growing following in the Ruby community; although I find that the plain-text stories can be a bit verbose and (cuc)cumbersome, I suggest taking a look at some Cucumber tutorials to see if it suits you.

8.4.1 Webrat

The first step toward amazing signup integration tests is to install *Webrat*,9 which allows us to write tests using a highly expressive mini-language for interacting with web pages.10 (The Webrat home page has installation instructions, and the Webrat wiki has further documentation.) If you're using Linux, you'll probably have to install a couple libraries first:

```
$ [sudo] apt-get install libxslt1-dev libxml2-dev
```

Otherwise, you just need to install the `webrat` gem (together with `launchy`, a gem needed to spawn new browser windows when Webrat encounters exceptions):

```
$ [sudo] gem install webrat -v 0.7.0
$ [sudo] gem install launchy -v 0.3.5
```

Then add a couple lines in your RSpec helper file (Listing 8.18) and you'll be good to go.

Listing 8.18. Code needed to use Webrat with RSpec.
`spec/spec_helper.rb`

```
.
.
```

```
.
require 'spec/rails'
require 'webrat'

Webrat.configure do |config|
  config.mode = :rails
end
.
.
.
```

8.4.2 Users signup failure should not make a new user

Now we're ready to make an integration test for signing up users. As we saw in Section 5.2.1, RSpec comes with a generator to make such integration specs; in the present case, our integration tests will contain various actions taken by users, so we'll name the test `users` accordingly:

```
$ script/generate integration_spec users
      create  spec/integration/users_spec.rb
```

As in Section 5.2.1, the generator automatically appends a spec identifier, yielding `users_spec.rb`.11

We start with signup failure. A simple way to arrange a failing signup is to visit the signup URL and just click the button, resulting in a page as in Figure 8.11. Upon failed submission, the response should render the `users/new` template. If you inspect the resulting HTML, you should see something like the markup in Listing 8.19. This means that we can test for the presence of error messages by looking for a `div` tag with the CSS id `"errorExplanation"`. A test for these steps appears in Listing 8.20.

Figure 8.11: The result of visiting `/signup` and just clicking "Sign up". (full size)

Listing 8.19. The error explanation `div` from the page in Figure 8.11.

```
<div class="errorExplanation" id="errorExplanation">
  <h2>5 errors prohibited this user from being saved</h2>
  <p>There were problems with the following fields:</p>
  <ul>
    <li>Name can't be blank</li>
    <li>Email can't be blank</li>
    <li>Email is invalid</li>
    <li>Password can't be blank</li>
    <li>Password is too short (minimum is 6 characters)</li>
  </ul>
</div>
```

Listing 8.20. Testing signup failure.
`spec/integration/users_spec.rb`

```
require 'spec_helper'

describe "Users" do

  describe "signup" do

    describe "failure" do

      it "should not make a new user" do
        visit signup_path
        click_button
        response.should render_template('users/new')
        response.should have_tag("div#errorExplanation")
      end
    end
  end
end
```

Here `"div#errorExplanation"` is CSS-inspired shorthand for

```
<div id="errorExplanation">...</div>
```

(Recall that CSS uses the pound sign `#` to style ids.)

Notice how natural the language is in Listing 8.20. The only problem is that it doesn't *quite* test what we want: we're not actually testing that a failed submission fails to create a new user. To do so, we need two new elements. First, Active Record models all have a `count` method, which simply returns how many records of that type are in the database. If you cleared the database at the beginning of the chapter and then created a new user in Section 8.3, this number should currently be `1`:

```
$ script/console
>> User.count
=> 1
```

Second, we need a way to wrap the test steps in a single package, and then check that it doesn't change the `User` count. The way to do this is to use a `lambda`,[12] as shown in Listing 8.21.

Listing 8.21. Testing signup failure with a `lambda`.
`spec/integration/users_spec.rb`

```
require 'spec_helper'

describe "Users" do

  describe "signup" do

    describe "failure" do

      it "should not make a new user" do
        lambda do
          visit signup_path
          click_button
          response.should render_template('new')
          response.should have_tag("div#errorExplanation")
        end.should_not change(User, :count)
      end
    end
  end
end
```

This wraps the signup failure steps in a `lambda` package and verifies that `User.count` doesn't get changed by the operation. The `change` method is part of RSpec, and

```
should_not change(User, :count)
```

verifies that the code inside the `lambda` block doesn't change the value of `User.count`.

This test should pass, and it provides an end-to-end verification that our signup machinery is working, at least for failed submissions.

8.4.3 Users signup success should make a new user

We come now to the integration test for successful signup. In this case, we need to fill in the signup fields with valid user data. When we do, it should change the User count by 1. Listing 8.22 shows how to do it.

Listing 8.22. Testing signup success.
`spec/integration/users_spec.rb`

```
require 'spec_helper'

describe "Users" do

  describe "signup" do
    .
    .
    .
    describe "success" do

      it "should make a new user" do
        lambda do
          visit signup_path
          fill_in "Name",         :with => "Example User"
          fill_in "Email",        :with => "user@example.com"
          fill_in "Password",     :with => "foobar"
          fill_in "Confirmation", :with => "foobar"
          click_button
          response.should render_template('users/show')
        end.should change(User, :count).by(1)
      end
    end
  end
end
```

By the way, although it's not obvious from the Webrat documentation, you can use the CSS id of the text box instead of the label, so `fill_in :user_name` also works.13 (This is especially nice for forms that don't use labels.)

I hope you agree that the combination of Webrat and RSpec syntax is incredibly natural and succinct.

For example, to fill in a field with a value, we just use code like this:

```
fill_in "Name",         :with => "Example User"
fill_in "Email",        :with => "user@example.com"
fill_in "Password",     :with => "foobar"
fill_in "Confirmation", :with => "foobar"
```

Here the first arguments to `fill_in` are the label values, i.e., exactly the text the user sees in the browser; there's no need to know anything about the underlying HTML structure generated by the Rails `form_for` helper.

Finally, we come to the coup de grâce—testing that successful signup actually creates a user in the database:

```
it "should make a new user" do
  lambda do
    .
    .
    .
  end.should change(User, :count).by(1)
```

As in Listing 8.21, we've wrapped the code for a successful signup in a `lambda` block. In this case, instead of making sure that the User count *doesn't* change, we verify that it increases by 1 due to a User record being created in the test database.

With that, our signup integration tests are complete, and we can be confident that, if users don't join our site, it's not because the signup form is broken.

8.5 Conclusion

Being able to sign up users is a major milestone for our application. Though the sample app has yet to accomplish anything useful, we have laid an essential foundation for all future development. In the next two chapters, we will complete two more major milestones: first, in Chapter 9 we will complete our authentication machinery by allowing users to sign in and out of the application; second, in Chapter 10 we will allow all users to update their account information and will allow site administrators to delete users, while also adding page protection to enforce a site security model, thereby completing the full suite of the Users resource REST actions from Table 6.2.

As usual, if you're using Git, you should merge your changes into the `master` branch at this point:

```
$ git add .
$ git commit -am "User signup complete"
$ git checkout master
$ git merge signing-up
```

8.6 Exercises

1. **(advanced)** Dig into the latest Rails API to discover how the `error_messages` method from Listing 8.8 is implemented.
2. Using the model in Listing 8.23, write tests to check for the presence of each field on the signup form. (Don't forget the `integrate_views` line, which is essential for this to work.)
3. Oftentimes signup forms will clear the password field for failed submissions, as shown in

Figure 8.12. Modify the Users controller `create` action to replicate this behavior. *Hint:* Reset `@user.password`.

4. The flash HTML in Listing 8.15 is a particularly ugly combination of HTML and ERb. Verify by running the test suite that the cleaner code in Listing 8.24, which uses the Rails `content_tag` helper, also works.

Listing 8.23. A template for testing for each field on the signup form.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views
  .
  .
  .
  describe "GET 'new'" do
    .
    .
    .

    it "should have a name field" do
      get :new
      response.should have_tag("input[name=?][type=?]", "user[name]", "text")
    end

    it "should have an email field"

    it "should have a password field"

    it "should have a password confirmation field"
  end
  .
  .
  .
end
```

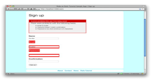
Figure 8.12: A failed signup form submission with the password field cleared. (full size)

Listing 8.24. The `flash` ERb in the site layout using `content_tag`.
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
      .
      .
      .
      <div id="content" class="round">
        <% flash.each do |key, value| %>
          <%= content_tag(:div, value, :class => "flash #{key}") %>
        <% end %>
        <%= yield %>
      </div>
      .
      .
      .
</html>
```

« Chapter 7 Modeling and viewing users, part II Chapter 9 Sign in, sign out »

1. If you get an error like `views/users/new.html.erb_spec.rb fails`, remove those accursed view specs with `$ rm -rf spec/views`. ↑
2. If this code fails for you, try using `Factory(:user, @attr)` instead. ↑
3. Exactly how this happens is a bit obscure; the form variable `f` has access to the `@user` variable inside the block, and must at some point iterate through the user's error messages in order to display them as HTML. Discovering exactly how `error_messages` works is left as an exercise (Section 8.6). ↑
4. Password parameter filtering is enabled by default in Rails 3. ↑
5. Recall from Box 8.1 that the `and_return` part of the statement is not an expectation, but rather a guarantee. ↑
6. In case you're curious, the response code is `302`, in contrast to the "permanent" `301` redirect discussed briefly in Box 3.1. ↑
7. Note that the key `:success` is a symbol, but Embedded Ruby automatically converts it to the string `"success"` before inserting it into the template. ↑
8. Actually, we'll use the closely related `flash.now`, but we'll defer that subtlety until we need it. ↑
9. Webrat is not compatible with Rails 3, but its replacement, called Capybara, has a nearly identical syntax. ↑
10. If you installed Webrat as part of the exercises in Section 5.5, you can skip this section. ↑
11. Note the plural; this is *not* the User spec `user_spec.rb`, which is a model test, not an integration test. ↑
12. The name comes from the lambda calculus, a mathematical system for representing functions and their operations. ↑
13. You can use Firebug or your browser's "view source" if you need to figure out the id. Or you can note that Rails uses the name of the resource and the name of the attribute separated with an underscore, yielding `user_name`, `user_email`, etc. ↑