7.3 Unsuccessful signups

Although we've briefly examined the HTML for the form in Figure 7.13 (shown in Listing 7.17), we haven't yet covered any details, and the form is best understood in the context of *signup failure*. In this section, we'll create a signup form that accepts an invalid submission and re-renders the signup page with a list of errors, as mocked up in Figure 7.15.

7.3.1 A working form

Recall from Section 7.1.2 that adding `resources :users` to the `routes.rb` file (Listing 7.3) automatically ensures that our Rails application responds to the RESTful URLs from Table 7.1. 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 to make a new user object using `User.new`, try (and fail) to save that user, and then render the signup page for possible resubmission. 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 7.2.2, this HTML issues a `POST` request to the /users URL.

Our first step toward a working signup form is adding the code in Listing 7.18. 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`, which (as we saw in Section 6.1.3) is either `true` or `false` depending on whether or not the save succeeds.

Listing 7.18: A `create` action that can handle signup failure.
app/controllers/users_controller.rb

Figure 7.15: A mockup of the signup failure page.

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])    # Not the final implementation!
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end
end
```

Note the comment: this is not the final implementation. But it's enough to get us started, and we'll finish the implementation in Section 7.3.2.

The best way to understand how the code in Listing 7.18 works is to *submit* the form with some invalid signup data. The result appears in Figure 7.16, and the full debug information appears in Figure 7.17.

To get a better picture of how Rails handles the submission, let's take a closer look at the **user** part of the parameters hash from the debug information (Figure 7.17):

```
"user" => { "name" => "Foo Bar",
            "email" => "foo@invalid",
            "password" => "[FILTERED]",
            "password_confirmation" => "[FILTERED]"
          }
```

This hash gets passed to the Users controller as part of **params**, and we saw starting in Section 7.1.2 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

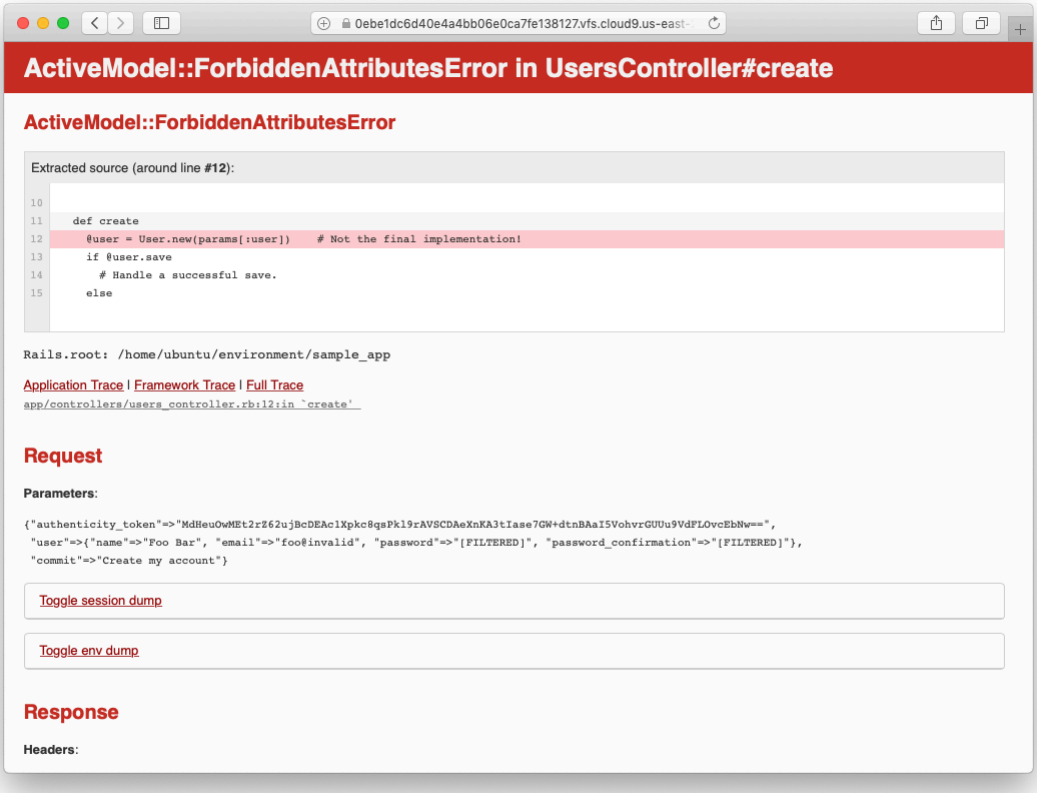

Figure 7.16: Signup failure upon submitting invalid data.

Figure 7.17: Signup failure debug information.

saw in Section 4.3.3, which introduced the strategically named **params** variable in a console session. The 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 7.17. For example, the value of

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

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

Although the hash keys appear as strings in the debug output, we can access them in the Users controller as 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 7.18. This means that the line

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

is mostly equivalent to

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

In previous versions of Rails, using

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

actually worked, but it was insecure by default and required a careful and error-prone procedure to prevent malicious users from potentially modifying the application database. In Rails version later than 4.0, this code raises an error (as seen in Figure 7.16 and Figure 7.17 above), which means it is secure by default.

7.3.2 Strong parameters

We mentioned briefly in Section 4.4.5 the idea of *mass assignment*, which involves initializing a Ruby variable using a hash of values, as in

```
@user = User.new(params[:user])    # Not the final implementation!
```

The comment included in Listing 7.18 and reproduced above indicates that this is not the final implementation. The reason is that initializing the entire **params** hash is *extremely* dangerous—it arranges to pass to **User.new** *all* data submitted by a user. In particular, suppose that, in addition to the current attributes, the User model included an **admin** attribute used to identify administrative users of the site. (We will implement just such an attribute in Section 10.4.1.) The way to set such an attribute to **true** is to pass the value **admin='1'** as part of **params[:user]**, a task that is easy to accomplish using a command-line HTTP client such as `curl`. The result would be that, by passing in the entire **params** hash to **User.new**, we would allow any user of the site to gain administrative access by including **admin='1'** in the web request.

Previous versions of Rails used a method called **attr_accessible** in the *model* layer to solve this problem, and you may still see that method in legacy Rails applications, but as of Rails 4.0 the preferred technique is to use so-called *strong parameters* in the controller layer. This allows us to specify which parameters are *required* and which ones are *permitted*. In addition, passing in a raw **params** hash as above will cause an error to be raised, so that Rails applications are now immune to mass assignment vulnerabilities by default.

In the present instance, we want to require the **params** hash to have a **:user** attribute, and we want to permit the name, email, password, and password confirmation attributes (but no others). We can accomplish this as follows:

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

This code returns a version of the **params** hash with only the permitted attributes (while raising an error if the **:user** attribute is missing).

To facilitate the use of these parameters, it's conventional to introduce an auxiliary method called **user_params** (which returns an appropriate initialization hash) and use it in place of **params[:user]**:

```
@user = User.new(user_params)
```

Since **user_params** will only be used internally by the Users controller and need not be exposed to external users via the web, we'll make it *private* using Ruby's **private** keyword, as shown in Listing 7.19. (We'll discuss **private** in more detail in Section 9.1.)

Listing 7.19: Using strong parameters in the **create** action.
app/controllers/users_controller.rb

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

  private

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

By the way, the extra level of indentation on the **user_params** method is designed to make it visually apparent which methods are defined after **private**. (Experience shows that this is a wise practice; in classes with a large number of methods, it is easy to define a private method accidentally, which leads to considerable confusion when it isn't available to call on the corresponding object.)

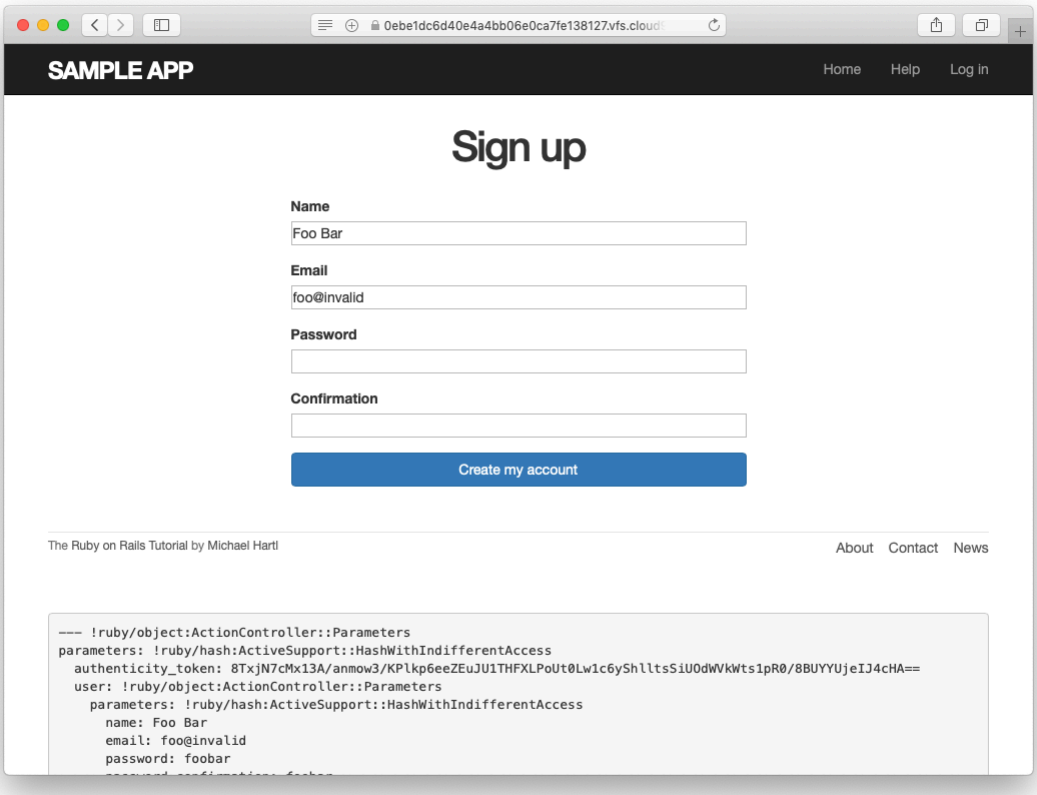

Figure 7.18: The signup form submitted with invalid information.

At this point, the signup form is working, at least in the sense that it no longer produces an error upon submission. On the other hand, as seen in Figure 7.18, it doesn't display any feedback on invalid submissions (apart from the development-only debug area), which is potentially confusing. It also doesn't actually create a new user. We'll fix the first issue in Section 7.3.3 and the second in Section 7.4.

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. By hitting the URL /signup?admin=1, confirm that the **admin** attribute appears in the **params** debug information.

7.3.3 Signup error messages

As a final step in handling failed user creation, we'll add helpful error messages to indicate the problems that prevented successful signup. Conveniently, Rails automatically provides such messages based on the User model validations. For example, consider trying to save a user with an invalid email address and with a password that's too short:

```
$ rails 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 before in Section 6.2.2) contains an array of error messages.

As in the console session above, the failed save in Listing 7.18 generates a list of error messages associated with the **@user** object. To display the messages in the browser, we'll render an error-messages partial on the user **new** page while adding the CSS class **form-control** (which has special meaning to Bootstrap) to each entry field, as shown in Listing 7.20. It's worth noting that this error-messages partial is only a first attempt; the final version appears in Section 13.3.2.

Listing 7.20: Code to display error messages on the signup form.
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
```

```
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, local: true) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%= f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Notice here that we **render** a partial called **'shared/error_messages'**; this reflects the common Rails convention of using a dedicated **shared/** directory for partials expected to be used in views across multiple controllers. (We'll see this expectation fulfilled in Section 10.1.1.)

This means that we have to create a new **app/views/shared** directory using **mkdir** and an error messages partial using (Table 1.1):

```
$ mkdir app/views/shared
```

We then need to create the **_error_messages.html.erb** partial file using **touch** or the text editor as usual. The contents of the partial appear in Listing 7.21.

Listing 7.21: A partial for displaying form submission error messages.
app/views/shared/_error_messages.html.erb

```
<% if @user.errors.any? %>
  <div id="error_explanation">
```

```
    <div class="alert alert-danger">
      The form contains <%= pluralize(@user.errors.count, "error") %>.
    </div>
    <ul>
    <% @user.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

This partial introduces several new Rails and Ruby constructs, including two methods for Rails error objects. The first method is **`count`**, which simply returns the number of errors:

```
>> user.errors.count
=> 2
```

The other new method is **`any?`**, which (together with **`empty?`**) is one of a pair of complementary methods:

```
>> user.errors.empty?
=> false
>> user.errors.any?
=> true
```

We see here that the **`empty?`** method, which we first saw in Section 4.2.2 in the context of strings, also works on Rails error objects, returning **`true`** for an empty object and **`false`** otherwise. The **`any?`** method is just the opposite of **`empty?`**, returning **`true`** if there are any elements present and **`false`** otherwise. (By the way, all of these methods—**`count`**, **`empty?`**, and **`any?`**—work on Ruby arrays as well. We'll put this fact to good use starting in Section 13.2.)

The other new idea is the **`pluralize`** text helper, which is available in the console via the **`helper`** object:

```
>> helper.pluralize(1, "error")
=> "1 error"
>> helper.pluralize(5, "error")
=> "5 errors"
```

We see here that **pluralize** takes an integer argument and then returns the number with a properly pluralized version of its second argument. Underlying this method is a powerful *inflector* that knows how to pluralize a large number of words, including many with irregular plurals:

```
>> helper.pluralize(2, "woman")
=> "2 women"
>> helper.pluralize(3, "erratum")
=> "3 errata"
```

As a result of its use of **pluralize**, the code

```
<%= pluralize(@user.errors.count, "error") %>
```

returns **"0 errors"**, **"1 error"**, **"2 errors"**, and so on, depending on how many errors there are, thereby avoiding ungrammatical phrases such as **"1 errors"** (a distressingly common mistake in both web and desktop applications).

Note that Listing 7.21 includes the CSS id **error_explanation** for use in styling the error messages. (Recall from Section 5.1.2 that CSS uses the pound sign **#** to style ids.) In addition, after an invalid submission Rails automatically wraps the fields with errors in **div**s with the CSS class **field_with_errors**. These labels then allow us to style the error messages with the SCSS shown in Listing 7.22, which makes use of Sass's **@extend** function to include the functionality of the Bootstrap class **has-error**.

Listing 7.22: CSS for styling error messages.
app/assets/stylesheets/custom.scss

```
.
.
.
/* forms */
.
.
.
#error_explanation {
  color: red;
  ul {
    color: red;
    margin: 0 0 30px 0;
  }
}

.field_with_errors {
  @extend .has-error;
  .form-control {
    color: $state-danger-text;
  }
}
```

With the code in Listing 7.20 and Listing 7.21 and the SCSS from Listing 7.22, helpful error messages now appear when submitting invalid signup information, as seen in Figure 7.19. 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 of passwords. (*Note*: Because both the presence validation and the **`has_secure_password`** validation catch the case of *empty* (**`nil`**) passwords, the signup form currently produces duplicate error messages when the user submits empty passwords. We could manipulate the error messages directly to eliminate duplicates, but luckily this issue will be fixed automatically by the addition of **`allow_nil: true`** in Section 10.1.4.)

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 changing the minimum length of passwords to 5 that the error message updates automatically as well.

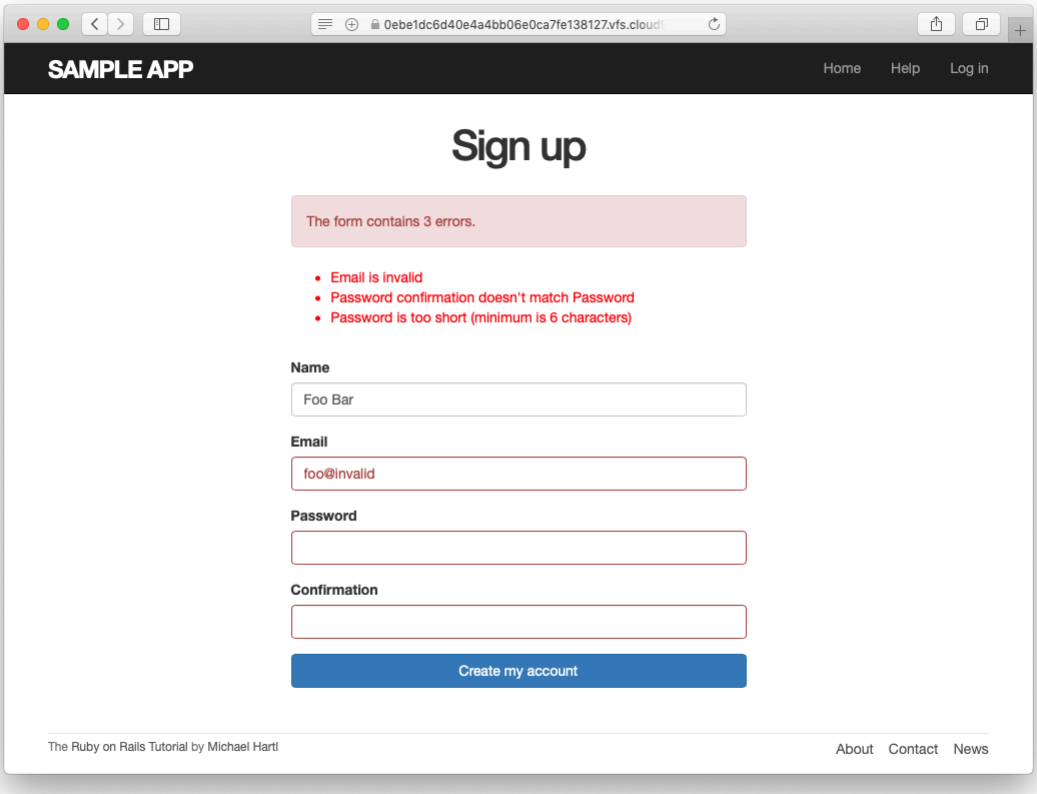

Figure 7.19: Failed signup with error messages.

2. How does the URL on the unsubmitted signup form (Figure 7.13) compare to the URL for a submitted signup form (Figure 7.19)? Why don't they match?

7.3.4 A test for invalid submission

In the days before powerful web frameworks with automated testing capabilities, developers had to test forms by hand. For example, to test a signup page manually, we would have to visit the page in a browser and then submit alternately invalid and valid data, verifying in each case that the application's behavior was correct. Moreover, we would have to remember to repeat the process any time the application changed. This process was painful and error-prone.

Happily, with Rails we can write tests to automate the testing of forms. In this section, we'll write one such test to verify the correct behavior upon invalid form submission; in Section 7.4.4, we'll write a corresponding test for valid submission.

To get started, we first generate an integration test file for signing up users, which we'll call **`users_signup`** (adopting the controller convention of a plural resource name):

```
$ rails generate integration_test users_signup
      invoke  test_unit
      create    test/integration/users_signup_test.rb
```

(We'll use this same file in Section 7.4.4 to test a valid signup.)

The main purpose of our test is to verify that clicking the signup button results in *not* creating a new user when the submitted information is invalid. (Writing a test for the error messages is left as an exercise (Section 7.3.4).) The way to do this is to check the *count* of users, and under the hood our tests will use the **`count`** method available on every Active Record class, including **`User`**:

```
$ rails console
>> User.count
=> 1
```

(Here `User.count` is `1` because of the user created in Section 6.3.4, though it may differ if you've added or deleted any users in the interim.) As in Section 5.3.4, we'll use `assert_select` to test HTML elements of the relevant pages, taking care to check only elements unlikely to change in the future.

We'll start by visiting the signup path using `get`:

```
get signup_path
```

In order to test the form submission, we need to issue a `POST` request to the `users_path` (Table 7.1), which we can do with the `post` function:

```
assert_no_difference 'User.count' do
  post users_path, params: { user: { name:  "",
                                     email: "user@invalid",
                                     password:              "foo",
                                     password_confirmation: "bar" } }
end
```

Here we've included the `params[:user]` hash expected by `User.new` in the `create` action (Listing 7.27). (In versions of Rails before 5, `params` was implicit, and only the `user` hash would be passed. This practice was deprecated in Rails 5.0, and now the recommended method is to include the full `params` hash explicitly.)

By wrapping the `post` in the `assert_no_difference` method with the string argument `'User.count'`, we arrange for a comparison between `User.count` before and after the contents inside the `assert_no_difference` block. This is equivalent to recording the user count, posting the data, and verifying that the count is the same:

```
before_count = User.count
post users_path, ...
after_count  = User.count
assert_equal before_count, after_count
```

Although the two are equivalent, using `assert_no_difference` is cleaner and is more idiomatically correct Ruby.

It's worth noting that the **get** and **post** steps above are technically unrelated, and it's actually not necessary to get the signup path before posting to the users path. I prefer to include both steps, though, both for conceptual clarity and to double-check that the signup form renders without error.

Putting the above ideas together leads to the test in Listing 7.23. We've also included a call to **assert_template** to check that a failed submission re-renders the **new** action. Adding lines to check for the appearance of error messages is left as an exercise (Section 7.3.4).

Listing 7.23: A test for an invalid signup. GREEN
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'
  end
end
```

Because we wrote the application code before the integration test, the test suite should be GREEN:

Listing 7.24: 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. 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.