includes a separate presence validation that specifically catches `nil` passwords. (Because `nil` passwords now bypass the main presence validation but are still caught by `has_secure_password`, this also fixes the duplicate error message mentioned in Section 7.3.3.)

With the code in this section, the user edit page should be working (Figure 10.5), as you can double-check by re-running the test suite, which should now be GREEN:

Listing 10.14: 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. Double-check that you can now make edits by making a few changes on the development version of the application.

2. What happens when you change the email address to one without an associated Gravatar?

10.2 Authorization

In the context of web applications, *authentication* allows us to identify users of our site, while *authorization* lets us control what they can do. One nice effect of building the authentication machinery in Chapter 8 is that we are now in a position to implement authorization as well.

Although the edit and update actions from Section 10.1 are functionally complete, they suffer from a ridiculous security flaw: they allow anyone (even non-logged-in users) to access either action and update the information for any

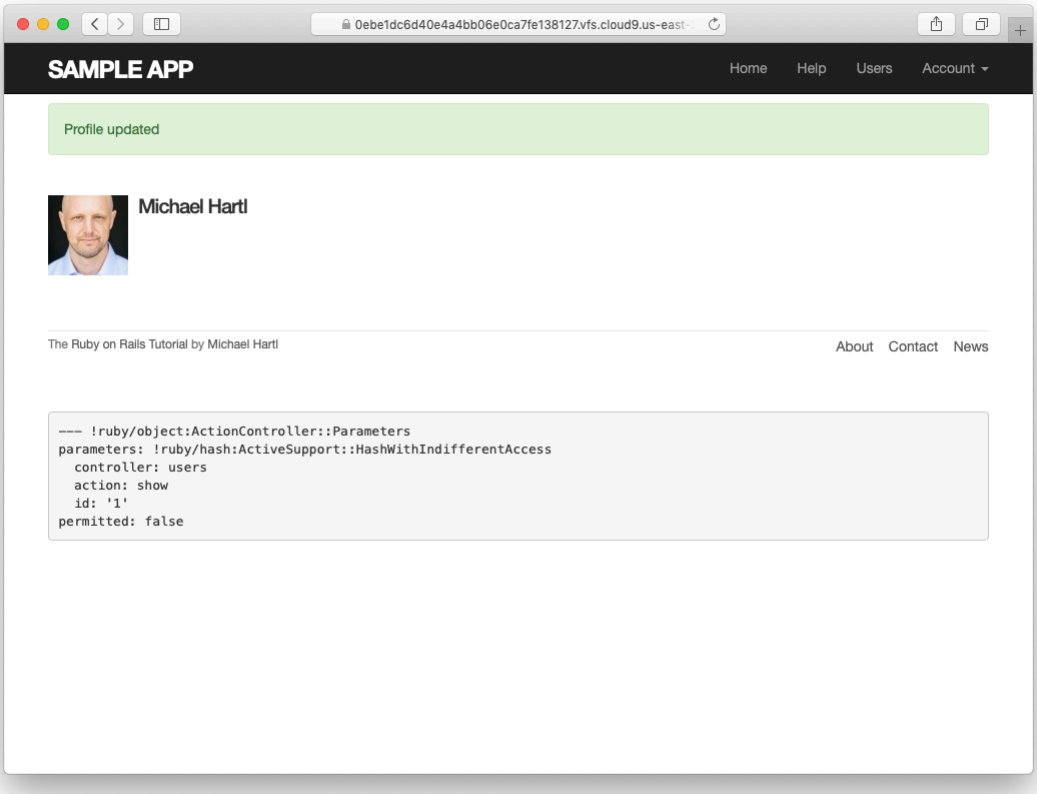

Figure 10.5: The result of a successful edit.

user. In this section, we'll implement a security model that requires users to be logged in and prevents them from updating any information other than their own.

In Section 10.2.1, we'll handle the case of non-logged-in users who try to access a protected page to which they might normally have access. Because this could easily happen in the normal course of using the application, such users will be forwarded to the login page with a helpful message, as mocked up in Figure 10.6. On the other hand, users who try to access a page for which they would never be authorized (such as a logged-in user trying to access a different user's edit page) will be redirected to the root URL (Section 10.2.2).

10.2.1 Requiring logged-in users

To implement the forwarding behavior shown in Figure 10.6, we'll use a *before filter* in the Users controller. Before filters use the **`before_action`** command to arrange for a particular method to be called before the given actions.[4] To require users to be logged in, we define a **`logged_in_user`** method and invoke it using **`before_action :logged_in_user`**, as shown in Listing 10.15.

Listing 10.15: Adding a **`logged_in_user`** before filter. RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
  private

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

    # Before filters

    # Confirms a logged-in user.
```

[4] The command for before filters used to be called **`before_filter`**, but the Rails core team decided to rename it to emphasize that the filter takes place before particular controller actions.

Figure 10.6: A mockup of the result of visiting a protected page

```
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end
end
```

By default, before filters apply to *every* action in a controller, so here we restrict the filter to act only on the `:edit` and `:update` actions by passing the appropriate `only:` options hash.

We can see the result of the before filter in Listing 10.15 by logging out and attempting to access the user edit page /users/1/edit, as seen in Figure 10.7.

As indicated in the caption of Listing 10.15, our test suite is currently RED:

Listing 10.16: RED

```
$ rails test
```

The reason is that the edit and update actions now require a logged-in user, but no user is logged in inside the corresponding tests.

We'll fix our test suite by logging the user in before hitting the edit or update actions. This is easy using the `log_in_as` helper developed in Section 9.3 (Listing 9.24), as shown in Listing 10.17.

Listing 10.17: Logging in a test user. GREEN

test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
```

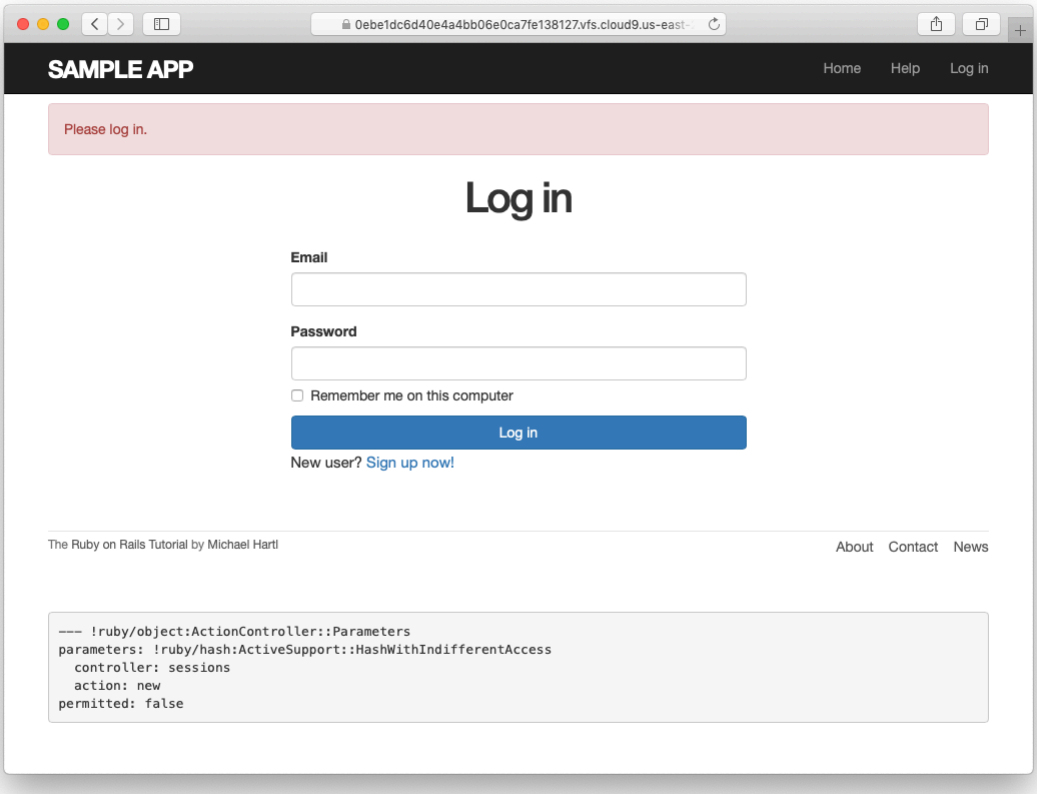

Figure 10.7: The login form after trying to access a protected page.

```
    .
    .
  end

  test "successful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
    .
    .
  end
end
```

(We could eliminate some duplication by putting the test login in the **setup** method of Listing 10.17, but in Section 10.2.3 we'll change one of the tests to visit the edit page *before* logging in, which isn't possible if the login step happens during the test setup.)

At this point, our test suite should be green:

Listing 10.18: GREEN

```
$ rails test
```

Even though our test suite is now passing, we're not finished with the before filter, because the suite is still GREEN even if we remove our security model, as you can verify by commenting it out (Listing 10.19). This is a Bad Thing—of all the regressions we'd like our test suite to catch, a massive security hole is probably #1, so the code in Listing 10.19 should definitely be RED. Let's write tests to arrange that.

Listing 10.19: Commenting out the before filter to test our security model. GREEN

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  # before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

Because the before filter operates on a per-action basis, we'll put the corresponding tests in the Users controller test. The plan is to hit the **edit** and **update** actions with the right kinds of requests and verify that the flash is set and that the user is redirected to the login path. From Table 7.1, we see that the proper requests are GET and PATCH, respectively, which means using the **get** and **patch** methods inside the tests. The results (which include adding a **setup** method to define an **@user** variable) appear in Listing 10.20.

Listing 10.20: Testing that **edit** and **update** are protected. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "should redirect edit when not logged in" do
    get edit_user_path(@user)
    assert_not flash.empty?
    assert_redirected_to login_url
  end

  test "should redirect update when not logged in" do
    patch user_path(@user), params: { user: { name: @user.name,
                                              email: @user.email } }
    assert_not flash.empty?
    assert_redirected_to login_url
  end
end
```

Note that the second test shown in Listing 10.20 involves using the **patch** method to send a PATCH request to **user_path(@user)**. According to Table 7.1, such a request gets routed to the **update** action in the Users controller, as required.

The test suite should now be RED, as required. To get it to GREEN, just uncomment the before filter (Listing 10.21).

Listing 10.21: Uncommenting the before filter. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

With that, our test suite should be GREEN:

Listing 10.22: GREEN

```
$ rails test
```

Any accidental exposure of the edit methods to unauthorized users will now be caught immediately by our test suite.

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. As noted above, by default before filters apply to every action in a controller, which in our cases is an error (requiring, e.g., that users log in to hit the signup page, which is absurd). By commenting out the `only:` hash in Listing 10.15, confirm that the test suite catches this error.

10.2.2 Requiring the right user

Of course, requiring users to log in isn't quite enough; users should only be allowed to edit their *own* information. As we saw in Section 10.2.1, it's easy to have a test suite that misses an essential security flaw, so we'll proceed using test-driven development to be sure our code implements the security model

correctly. To do this, we'll add tests to the Users controller test to complement the ones shown in Listing 10.20.

In order to make sure users can't edit other users' information, we need to be able to log in as a second user. This means adding a second user to our users fixture file, as shown in Listing 10.23.

Listing 10.23: Adding a second user to the fixture file.
test/fixtures/users.yml

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

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>
```

By using the **log_in_as** method defined in Listing 9.24, we can test the **edit** and **update** actions as in Listing 10.24. Note that we expect to redirect users to the root path instead of the login path because a user trying to edit a different user would already be logged in.

Listing 10.24: Tests for trying to edit as the wrong user. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user       = users(:michael)
    @other_user = users(:archer)
  end
  .
  .
  .
  test "should redirect edit when logged in as wrong user" do
    log_in_as(@other_user)
    get edit_user_path(@user)
    assert flash.empty?
```

```
    assert_redirected_to root_url
  end

  test "should redirect update when logged in as wrong user" do
    log_in_as(@other_user)
    patch user_path(@user), params: { user: { name: @user.name,
                                              email: @user.email } }
    assert flash.empty?
    assert_redirected_to root_url
  end
end
```

To redirect users trying to edit another user's profile, we'll add a second method called **`correct_user`**, together with a before filter to call it (Listing 10.25). Note that the **`correct_user`** before filter defines the **`@user`** variable, so Listing 10.25 also shows that we can eliminate the **`@user`** assignments in the **`edit`** and **`update`** actions.

Listing 10.25: A before filter to protect the edit/update pages. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
  private

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

```
    end

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless @user == current_user
    end
end
```

At this point, our test suite should be GREEN:

Listing 10.26: GREEN

```
$ rails test
```

As a final refactoring, we'll adopt a common convention and define a **`current_user?`** boolean method for use in the **`correct_user`** before filter. We'll use this method to replace code like

```
unless @user == current_user
```

with the more expressive

```
unless current_user?(@user)
```

The result appears in Listing 10.27. Note that by writing **`user && user == current_user`**, we also catch the edge case where **`user`** is **`nil`**.[5]

[5] Thanks to reader Andrew Moor for pointing this out. Andrew also noted that we can use the safe navigation operator introduced in Section 8.2.4 to write this as **`user&. == current_user`**.

Listing 10.27: The `current_user?` method.
app/helpers/sessions_helper.rb

```
module SessionsHelper

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

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end

  # Returns true if the given user is the current user.
  def current_user?(user)
    user && user == current_user
  end
  .
  .
  .
end
```

Replacing the direct comparison with the boolean method gives the code shown in Listing 10.28.

Listing 10.28: The final `correct_user` before filter. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
```

```
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
  private

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

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

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. Why is it important to protect both the `edit` and `update` actions?
2. Which action could you more easily test in a browser?

10.2.3 Friendly forwarding

Our site authorization is complete as written, but there is one minor blemish: when users try to access a protected page, they are currently redirected to their profile pages regardless of where they were trying to go. In other words, if a non-logged-in user tries to visit the edit page, after logging in the user will be redirected to /users/1 instead of /users/1/edit. It would be much friendlier to redirect them to their intended destination instead.

The application code will turn out to be relatively complicated, but we can write a ridiculously simple test for friendly forwarding just by reversing the order of logging in and visiting the edit page in Listing 10.17. As seen in Listing 10.29, the resulting test tries to visit the edit page, then logs in, and then checks that the user is redirected to the *edit* page instead of the default profile page. (Listing 10.29 also removes the test for rendering the edit template since that's no longer the expected behavior.)

Listing 10.29: A test for friendly forwarding. RED
test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "successful edit with friendly forwarding" do
    get edit_user_path(@user)
    log_in_as(@user)
    assert_redirected_to edit_user_url(@user)
    name  = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), params: { user: { name:  name,
```

```
                                                email: email,
                                                password:              "",
                                                password_confirmation: "" } }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name,  @user.name
    assert_equal email, @user.email
  end
end
```

Now that we have a failing test, we're ready to implement friendly forwarding.[6] In order to forward users to their intended destination, we need to store the location of the requested page somewhere, and then redirect to that location instead of to the default. We accomplish this with a pair of methods, **`store_location`** and **`redirect_back_or`**, both defined in the Sessions helper (Listing 10.30).

Listing 10.30: Code to implement friendly forwarding. RED
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Redirects to stored location (or to the default).
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
  end

  # Stores the URL trying to be accessed.
  def store_location
    session[:forwarding_url] = request.original_url if request.get?
  end
end
```

Here the storage mechanism for the forwarding URL is the same **`session`** facility we used in Section 8.2.1 to log the user in. Listing 10.30 also uses the **`request`** object (via **`request.original_url`**) to get the URL of the requested page.

[6] The code in this section is adapted from the Clearance gem by thoughtbot.

The **store_location** method in Listing 10.30 puts the requested URL in the **session** variable under the key **:forwarding_url**, but only for a **GET** request. This prevents storing the forwarding URL if a user, say, submits a form when not logged in (which is an edge case but could happen if, e.g., a user deleted the session cookies by hand before submitting the form). In such a case, the resulting redirect would issue a **GET** request to a URL expecting **POST**, **PATCH**, or **DELETE**, thereby causing an error. Including **if request.get?** prevents this from happening.[7]

To make use of **store_location**, we need to add it to the **logged_in_user** before filter, as shown in Listing 10.31.

Listing 10.31: Adding **store_location** to the logged-in user before filter.
RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end
  .
  .
  .
  private

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

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
```

[7] Thanks to reader Yoel Adler for pointing out this subtle issue, and for discovering the solution.

```
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

To implement the forwarding itself, we use the **redirect_back_or** method to redirect to the requested URL if it exists, or some default URL otherwise, which we add to the Sessions controller **create** action to redirect after successful login (Listing 10.32). The **redirect_back_or** method uses the or operator **||** through

```
session[:forwarding_url] || default
```

This evaluates to **session[:forwarding_url]** unless it's **nil**, in which case it evaluates to the given default URL. Note that Listing 10.30 is careful to remove the forwarding URL (via **session.delete(:forwarding_url)**); otherwise, subsequent login attempts would forward to the protected page until the user closed their browser. (Testing for this is left as an exercise (Section 10.2.3).) Also note that the session deletion occurs even though the line with the redirect appears first; redirects don't happen until an explicit **return** or the end of the method, so any code appearing after the redirect is still executed.

Listing 10.32: The Sessions **create** action with friendly forwarding. GREEN
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
```

```
      params[:session][:remember_me] == '1' ? remember(user) : forget(user)
      redirect_back_or user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end
  .
  .
  .
end
```

With that, the friendly forwarding integration test in Listing 10.29 should pass, and the basic user authentication and page protection implementation is complete. As usual, it's a good idea to verify that the test suite is GREEN before proceeding:

Listing 10.33: 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 to make sure that friendly forwarding only forwards to the given URL the first time. On subsequent login attempts, the forwarding URL should revert to the default (i.e., the profile page). *Hint*: Add to the test in Listing 10.29 by checking for the right value of **`session[:forwarding_url]`**.

2. Put a **`debugger`** (Section 7.1.3) in the Sessions controller's **`new`** action, then log out and try to visit /users/1/edit. Confirm in the debugger that the value of **`session[:forwarding_url]`** is correct. What is the value of **`request.get?`** for the **`new`** action? (Sometimes the terminal can freeze

up or act strangely when you're using the debugger; use your technical sophistication (Box 1.2) to resolve any issues.)

10.3 Showing all users

In this section, we'll add the penultimate user action, the **index** action, which is designed to display *all* the users instead of just one. Along the way, we'll learn how to seed the database with sample users and how to *paginate* the user output so that the index page can scale up to display a potentially large number of users. A mockup of the result—users, pagination links, and a "Users" navigation link—appears in Figure 10.8.[8] In Section 10.4, we'll add an administrative interface to the users index so that users can also be destroyed.

10.3.1 Users index

To get started with the users index, we'll first implement a security model. Although we'll keep individual user **show** pages visible to all site visitors, the user **index** will be restricted to logged-in users so that there's a limit to how much unregistered users can see by default.[9]

To protect the **index** page from unauthorized access, we'll first add a short test to verify that the **index** action is redirected properly (Listing 10.34).

Listing 10.34: Testing the **index** action redirect. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user       = users(:michael)
    @other_user = users(:archer)
  end
```

[8] Image retrieved from https://www.flickr.com/photos/glasgows/338937124/ on 2014-08-25. Copyright © 2008 by M&R Glasgow and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

[9] This is the same authorization model used by Twitter.