Please see the post **Do not buy the print version of the Ruby on Rails Tutorial (yet)**

skip to content | view as single page

# Ruby on Rails Tutorial

## Learn Web Development with Rails

Michael Hartl

## Contents

## Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I've worked my way through many Rails books, this is the one that finally made me "get" it. Everything is done very much "the Rails way"—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it's like to do a real-world project. The tutorial's code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

Derek Sivers (sivers.org)
*Formerly: Founder, CD Baby*
*Currently: Founder, Thoughts Ltd.*

## Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor Aurelius Prochazka. I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

## About the author

Michael Hartl is the author of the *Ruby on Rails Tutorial*, the leading introduction to web development with Ruby on Rails. His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a Ruby Hero Award for his contributions to the Ruby community. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

## Copyright and license

# Chapter 9
# Updating, showing, and deleting users

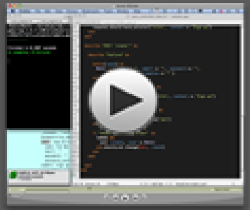
In this chapter, we will complete the REST actions for the Users resource (Table 7.1) by adding **edit**, **update**, **index**, and **destroy** actions. We'll start by giving users the ability to update their profiles, which will also provide a natural opportunity to enforce a security model (made possible by the authorization code in Chapter 8). Then we'll make a listing of all users (also requiring authorization), which will motivate the introduction of sample data and pagination. Finally, we'll add the ability to destroy users, wiping them clear from the database. Since we can't allow just any user to have such dangerous powers, we'll take care to create a privileged class of administrative users (admins) authorized to delete other users.

To get started, let's start work on an **updating-users** topic branch:

```
$ git checkout -b updating-users
```

## 9.1 Updating users

The pattern for editing user information closely parallels that for creating new users (Chapter 7). Instead of a **new** action rendering a view for new users, we have an **edit** action rendering a view to edit users; instead of **create** responding to a POST request, we have an **update** action responding to a PUT request (Box 3.2). The biggest difference is that, while anyone can sign up, only the current user should be able to update his information. This means that we need to enforce access control so that only authorized users can edit and update; the authentication machinery from



Chapter 8 will allow us to use a *before filter* to ensure that this is the case.

## 9.1.1 Edit form

We start with the edit form, whose mockup appears in Figure 9.1.[1] As usual, we'll begin with some tests. First, note the link to change the Gravatar image; if you poke around the Gravatar site, you'll see that the page to add or edit images is located at http://gravatar.com/emails, so we test the **`edit`** page for a link with that URI.[2]

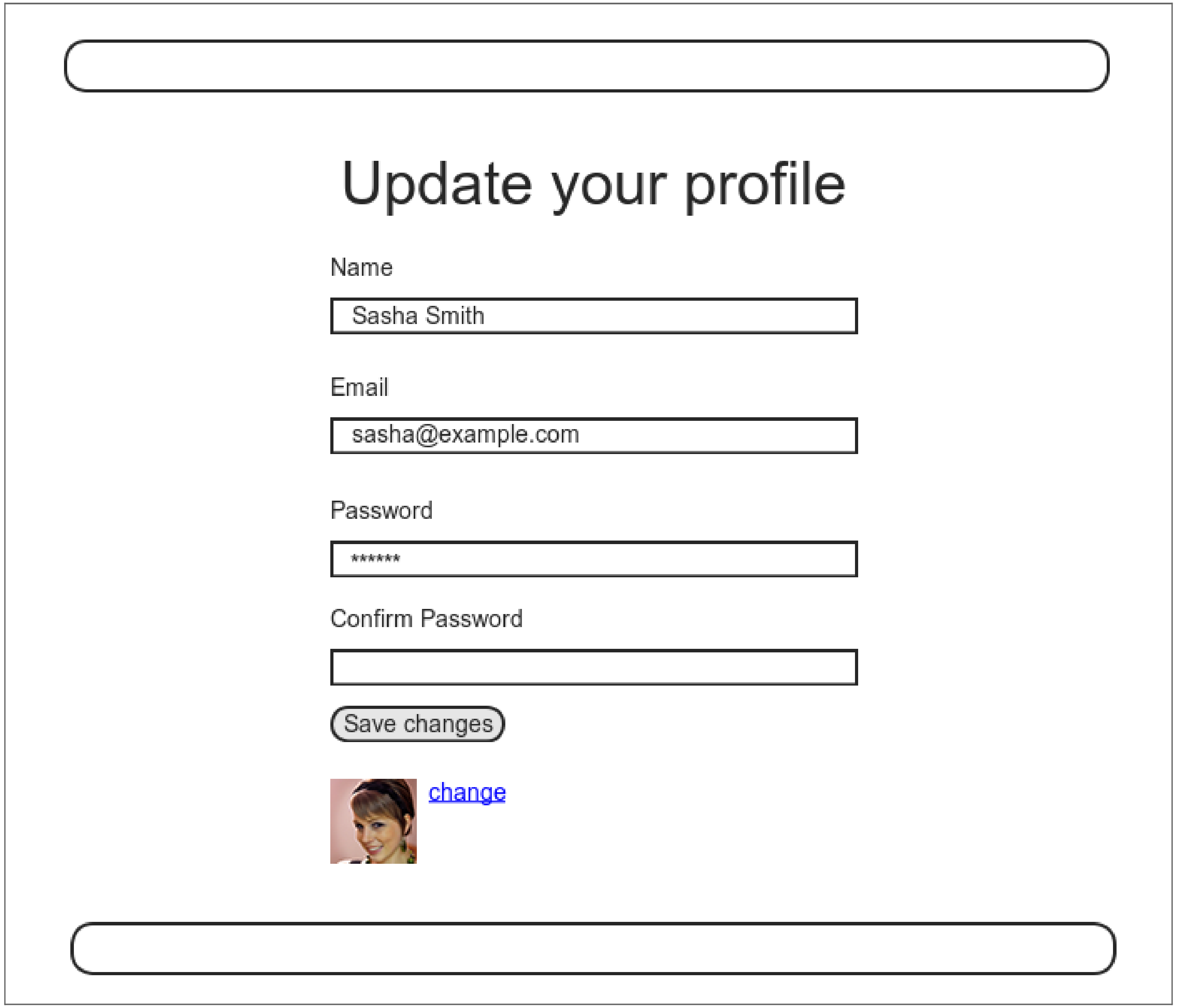


Figure 9.1: A mockup of the user edit page. (full size)

The tests for the edit user form are analogous to the test for the new user form in Listing 7.31 from the Chapter 7 exercises, which added a test for the error message on invalid submission. The result appears in Listing 9.1.

**Listing 9.1.** Tests for the user edit page.
**`spec/requests/user_pages_spec.rb`**

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit edit_user_path(user) }

    describe "page" do
      it { should have_selector('h1',    text: "Update your profile") }
      it { should have_selector('title', text: "Edit user") }
      it { should have_link('change', href: 'http://gravatar.com/emails') }
    end

    describe "with invalid information" do
      before { click_button "Save changes" }

      it { should have_content('error') }
    end
  end
end
```

To write the application code, we need to fill in the `edit` action in the Users controller. Note from Table 7.1 that the proper URI for a user's edit page is /users/1/edit (assuming the user's id is `1`). Recall that the id of the user is available in the `params[:id]` variable, which means that we can find the user with the code in Listing 9.2.

**Listing 9.2.** The user `edit` action.

**app/controllers/users_controller.rb**

```
class UsersController < ApplicationController
  .
  .
  .
  def edit
    @user = User.find(params[:id])
  end
end
```

Getting the tests to pass requires making the actual edit view, shown in Listing 9.3. Note how closely this resembles the new user view from Listing 7.17; the large overlap suggests factoring the repeated code into a partial, which is left as an exercise (Section 9.6).

**Listing 9.3.** The user edit view.

**app/views/users/edit.html.erb**

```
<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name %>

      <%= f.label :email %>
      <%= f.text_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.label :password_confirmation, "Confirm Password" %>
      <%= f.password_field :password_confirmation %>

      <%= f.submit "Save changes", class: "btn btn-large btn-primary" %>
    <% end %>

    <%= gravatar_for @user %>
    <a href="http://gravatar.com/emails">change</a>
  </div>
</div>
```

Here we have reused the shared `error_messages` partial introduced in Section 7.3.2.

With the `@user` instance variable from Listing 9.2, the edit page tests from Listing 9.1 should pass:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "edit page"
```

The corresponding page appears in Figure 9.2, which shows how Rails automatically pre-fills the Name and Email fields using the attributes of the `@user` variable.

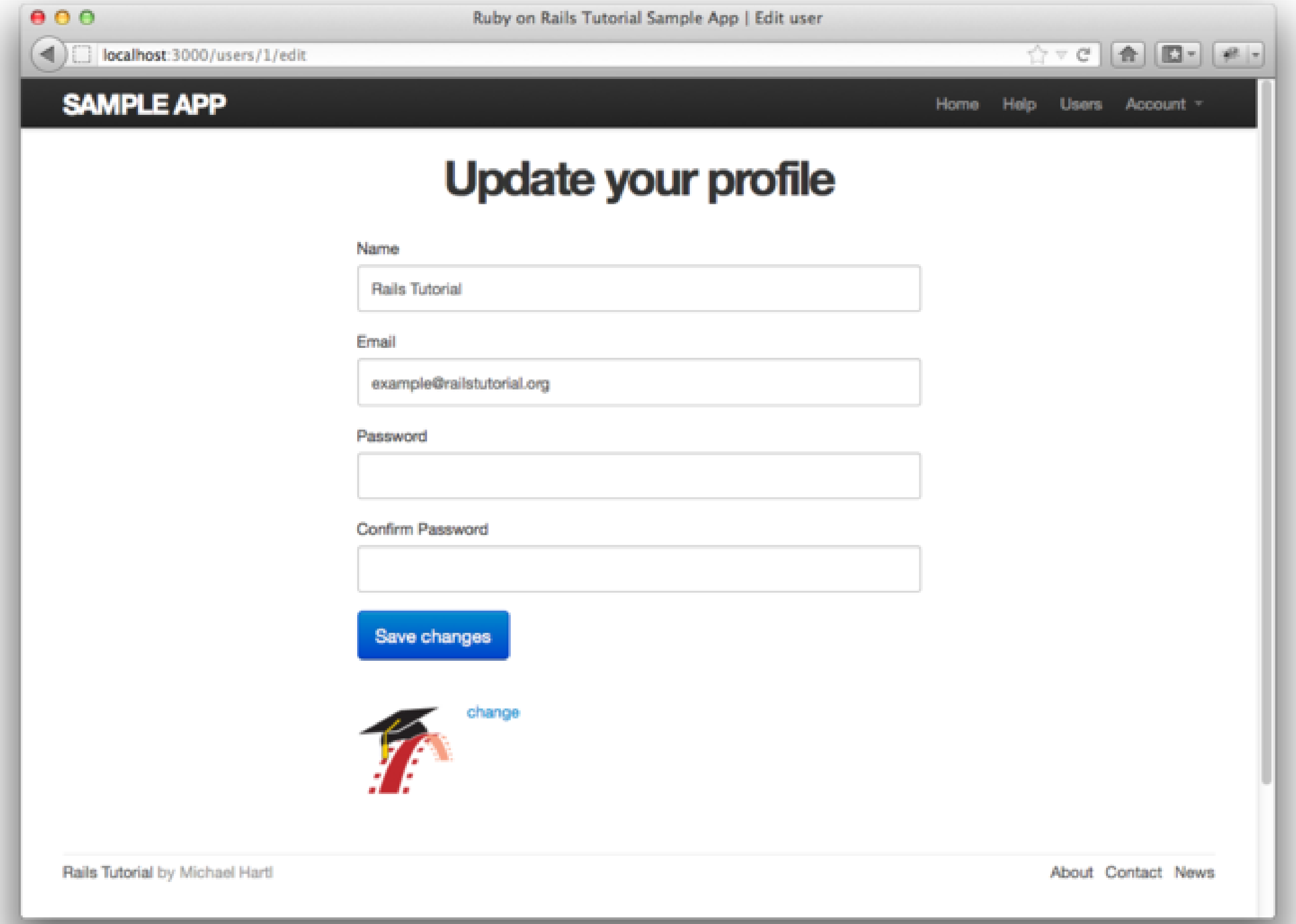


Figure 9.2: The initial user edit page with pre-filled name & email. (full size)

Looking at the HTML source for Figure 9.2, we see a form tag as expected (Listing 9.4).

**Listing 9.4.** HTML for the edit form defined in Listing 9.3 and shown in Figure 9.2.

```
<form action="/users/1" class="edit_user" id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="put" />
  .
  .
  .
</form>
```

Note here the hidden input field

```
<input name="_method" type="hidden" value="put" />
```

Since web browsers can't natively send PUT requests (as required by the REST conventions from Table 7.1), Rails fakes it with a POST request and a hidden **`input`** field.[3]

There's another subtlety to address here: the code **`form_for(@user)`** in Listing 9.3 is *exactly* the same as the code in Listing 7.17—so how does Rails know to use a POST request for new users and a PUT for editing users? The answer is that it is possible to tell whether a user is new or already exists in the database via Active Record's **`new_record?`** boolean method:

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

When constructing a form using **`form_for(@user)`**, Rails uses POST if **`@user.new_record?`** is **`true`** and PUT if it is **`false`**.

As a final touch, we'll add a URI to the user settings link to the site navigation. Since it depends on the signin status of the user, the test for the "Settings" link belongs with the other authentication tests, as shown in Listing 9.5. (It would be nice to have additional tests verifying that such links *don't* appear for users who aren't signed in; writing these tests is left as an exercise (Section 9.6).)

**Listing 9.5.** Adding a test for the "Settings" link.
**`spec/requests/authentication_pages_spec.rb`**

```
require 'spec_helper'

describe "Authentication" do
    .
    .
    .
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before { sign_in user }

      it { should have_selector('title', text: user.name) }
      it { should have_link('Profile',  href: user_path(user)) }
      it { should have_link('Settings', href: edit_user_path(user)) }
      it { should have_link('Sign out', href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
      .
      .
      .
    end
  end
end
```

For convenience, the code in Listing 9.5 uses a helper to sign in a user inside the tests. The method is to visit the signin page and submit valid information, as shown in Listing 9.6.

**Listing 9.6.** A test helper to sign users in.
**`spec/support/utilities.rb`**

```
.
.
.
```

```
def sign_in(user)
  visit signin_path
  fill_in "Email",    with: user.email
  fill_in "Password", with: user.password
  click_button "Sign in"
  # Sign in when not using Capybara as well.
  cookies[:remember_token] = user.remember_token
end
```

As noted in the comment line, filling in the form doesn't work when not using Capybara, so to cover this case we also add the user's remember token to the cookies:

```
# Sign in when not using Capybara as well.
cookies[:remember_token] = user.remember_token
```

This is necessary when using one of the HTTP request methods directly (**`get`**, **`post`**, **`put`**, or **`delete`**), as we'll see in Listing 9.47. (Note that the test **`cookies`** object isn't a perfect simulation of the real cookies object; in particular, the **`cookies.permanent`** method seen in Listing 8.19 doesn't work inside tests.) As you might suspect, the **`sign_in`** method will prove useful in future tests, and in fact it can already be used to eliminate some duplication (Section 9.6).

The application code to add the URI for the "Settings" link is simple: we just use the named route **`edit_user_path`** from Table 7.1, together with the handy **`current_user`** helper method defined in Listing 8.22:

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

The full application code appears in Listing 9.7).

**Listing 9.7.** Adding a "Settings" link.

**app/views/layouts/_header.html.erb**

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", '#' %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
</header>
```

## 9.1.2 Unsuccessful edits

In this section we'll handle unsuccessful edits and get the error messages test in Listing 9.1 to pass. The application code creates an `update` action that uses `update_attributes` (Section 6.1.5) to update the user based on the submitted `params` hash, as shown in Listing 9.8. With invalid information, the update attempt returns `false`, so the `else` branch re-renders the edit page. We've seen this pattern before; the structure closely parallels the first version of the `create` action (Listing 7.21).

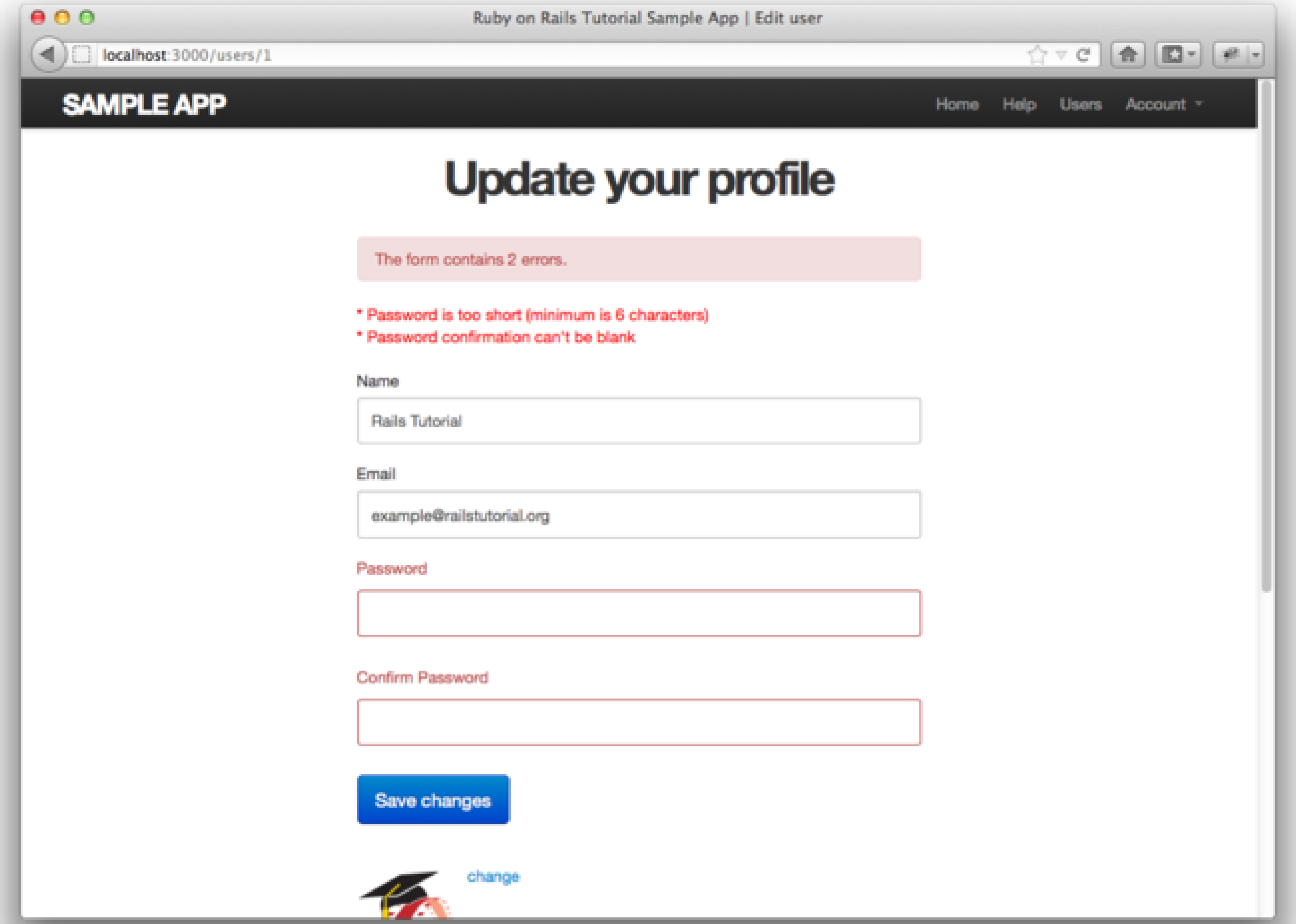
**Listing 9.8.** The initial user `update` action.

`app/controllers/users_controller.rb`

```ruby
class UsersController < ApplicationController
  .
  .
  .
  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      # Handle a successful update.
    else
      render 'edit'
    end
  end
end
```

The resulting error message (Figure 9.3) is the one needed to get the error message test to pass, as you should verify by running the test suite:

```
$ bundle exec rspec spec/
```


Ruby on Rails Tutorial Sample App | Edit user

localhost:3000/users/1

SAMPLE APP

Home Help Users Account

Update your profile

The form contains 2 errors.

- Password is too short (minimum is 6 characters)
- Password confirmation can't be blank

Name

Rails Tutorial

Email

example@railstutorial.org

Password

Confirm Password

Save changes

change

Figure 9.3: Error message from submitting the update form. (full size)

## 9.1.3 Successful edits

Now it’s time to get the edit form to work. Editing the profile images is already functional since we’ve outsourced image upload to Gravatar; we can edit gravatars by clicking on the “change” link from Figure 9.2, as shown in Figure 9.4. Let's get the rest of the user edit functionality working as well.

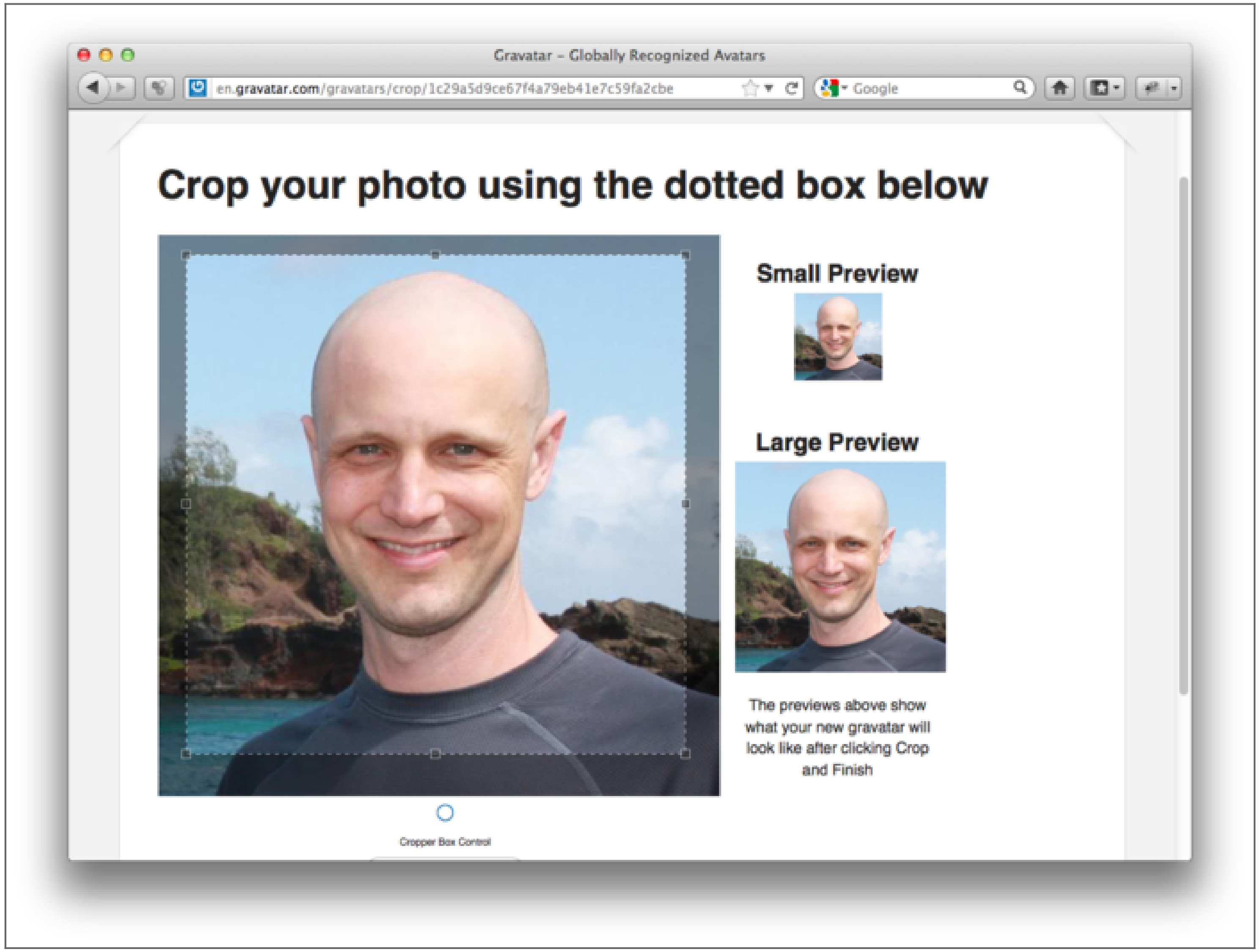

Gravatar - Globally Recognized Avatars
en.gravatar.com/gravatars/crop/1c29a5d9ce67f4a79eb41e7c59fa2cbe
Google
Crop your photo using the dotted box below
Small Preview
Large Preview
The previews above show what your new gravatar will look like after clicking Crop and Finish
Cropper Box Control

Figure 9.4: The Gravatar image-cropping interface, with a picture of some dude.

The tests for the `update` action are similar to those for `create`. Listing 9.9 show how to use Capybara to fill in the form fields with valid information and then test that the resulting behavior is correct. This is a lot of code; see if you can work through it by referring back to the tests in Chapter 7.

**Listing 9.9.** Tests for the user `update` action.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit edit_user_path(user) }
    .
    .
    .
    describe "with valid information" do
      let(:new_name)  { "New Name" }
      let(:new_email) { "new@example.com" }
      before do
        fill_in "Name",             with: new_name
        fill_in "Email",            with: new_email
        fill_in "Password",         with: user.password
        fill_in "Confirm Password", with: user.password
        click_button "Save changes"
      end

      it { should have_selector('title', text: new_name) }
      it { should have_selector('div.alert.alert-success') }
      it { should have_link('Sign out', href: signout_path) }
      specify { user.reload.name.should  == new_name }
      specify { user.reload.email.should == new_email }
    end
  end
end
```

The only real novelty in Listing 9.9 is the `reload` method, which appears in the test for changing the user's attributes:

```
specify { user.reload.name.should  == new_name }
specify { user.reload.email.should == new_email }
```

This reloads the `user` variable from the test database using `user.reload`, and then verifies that the user's new name and email match the new values.

The `update` action needed to get the tests in Listing 9.9 to pass is similar to the final form of the `create` action (Listing 8.27), as seen in Listing 9.10. All this does is add

```
flash[:success] = "Profile updated"
sign_in @user
redirect_to @user
```

to the code in Listing 9.8. Note that we sign in the user as part of a successful profile update; this is because the remember token gets reset when the user is saved (Listing 8.18), which invalidates the user's session (Listing 8.22). This is a nice security feature, as it means that any hijacked sessions will automatically expire when the user information is changed.

**Listing 9.10.** The user `update` action.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      flash[:success] = "Profile updated"
      sign_in @user
      redirect_to @user
    else
      render 'edit'
    end
  end
end
```

Note that, as currently constructed, every edit requires the user to reconfirm the password (as implied by the empty confirmation text box in Figure 9.2), which is a minor annoyance but makes updates much more secure.

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

```
$ bundle exec rspec spec/
```

## 9.2 Authorization

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

Although the edit and update actions from Section 9.1 are functionally complete, they suffer from a ridiculous security flaw: they allow anyone (even non-signed-in users) to access either action, and any signed-in user can update the information for any other user. In this section, we'll implement a security model that requires users to be signed in and prevents them from updating any information other than their own. Users who aren't signed in and who try to access protected pages will be forwarded to the signin page with a helpful message, as mocked up in Figure 9.5.

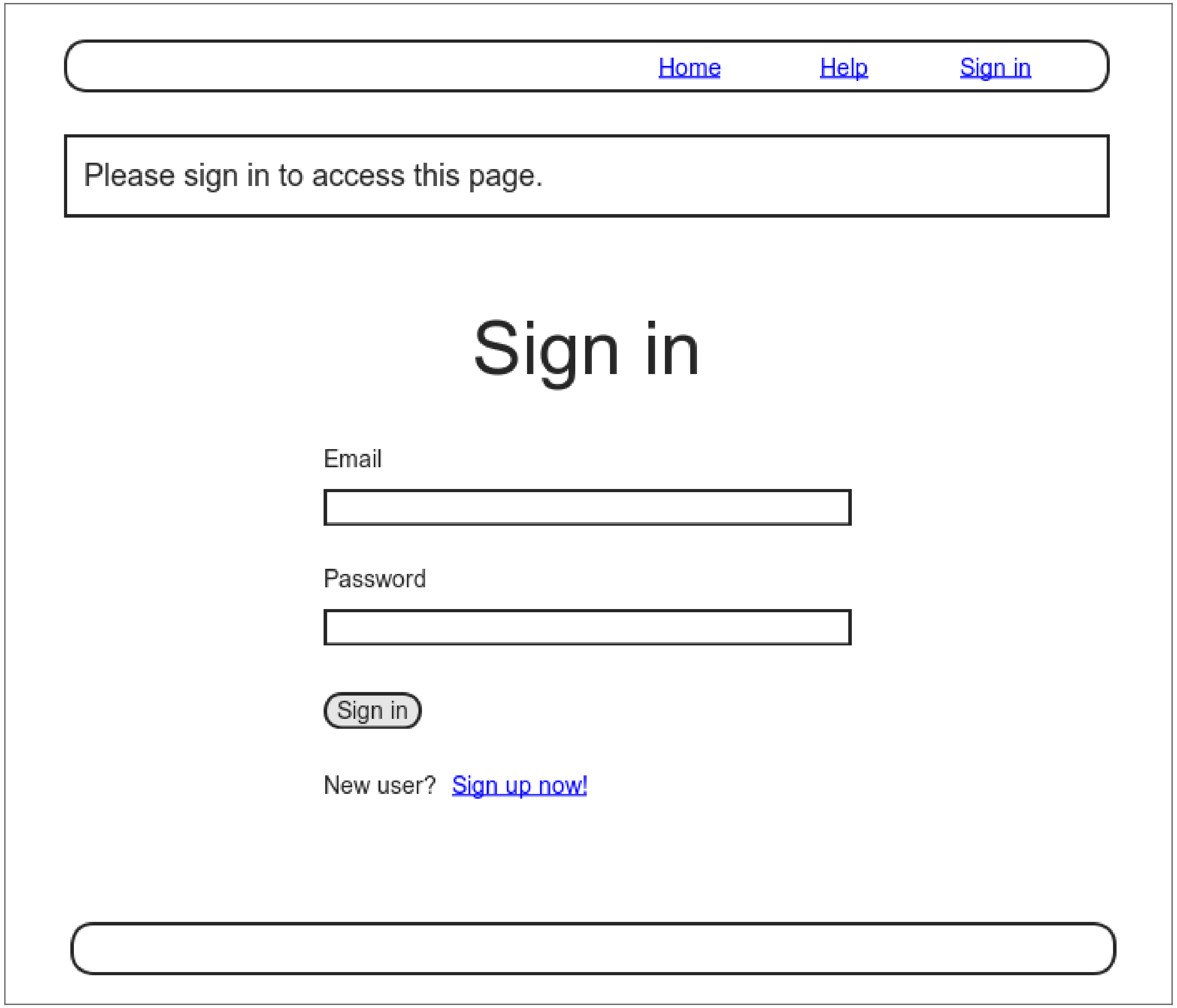


Figure 9.5: A mockup of the result of visiting a protected page (full size)

### 9.2.1 Requiring signed-in users

Since the security restrictions for the **edit** and **update** actions are identical, we'll handle them in a single RSpec **describe** block. Starting with the sign-in requirement, our initial tests verify that non-signed-in users attempting to access either action are simply sent to the signin page, as seen in Listing 9.11.

**Listing 9.11.** Testing that the **edit** and **update** actions are protected.
**spec/requests/authentication_pages_spec.rb**

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "in the Users controller" do

        describe "visiting the edit page" do
          before { visit edit_user_path(user) }
          it { should have_selector('title', text: 'Sign in') }
        end

        describe "submitting to the update action" do
          before { put user_path(user) }
          specify { response.should redirect_to(signin_path) }
        end
      end
    end
  end
end
```

The code in Listing 9.11 introduces a second way, apart from Capybara's `visit` method, to access a controller action: by issuing the appropriate HTTP request directly, in this case using the `put` method to issue a PUT request:

```
describe "submitting to the update action" do
  before { put user_path(user) }
  specify { response.should redirect_to(signin_path) }
end
```

This issues a PUT request directly to `/users/1`, which gets routed to the `update` action of the Users controller (Table 7.1). This is necessary because there is no way for a browser to visit the `update` action directly—it can only get there indirectly by submitting the edit form—so Capybara can't do it either. But visiting the edit page only tests the authorization for the `edit` action, not for `update`. As a result, the only way to test the proper authorization for the `update` action itself is to issue a direct request. (As you might guess, in addition to `put` Rails tests support `get`, `post`, and `delete` as well.)

When using one of the methods to issue HTTP requests directly, we get access to the low-level `response` object. Unlike the Capybara `page` object, `response` lets us test for the server response itself, in this case verifying that the `update` action responds by redirecting to the signin page:

```
specify { response.should redirect_to(signin_path) }
```

The authorization application code uses a *before filter*, which arranges for a particular method to be called before the given actions. To require users to be signed in, we define a `signed_in_user` method and invoke it using `before_filter :signed_in_user`, as shown in Listing 9.12.

**Listing 9.12.** Adding a `signed_in_user` before filter.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:edit, :update]
  .
  .
  .
  private

    def signed_in_user
      redirect_to signin_url, notice: "Please sign in." unless signed_in?
    end
end
```

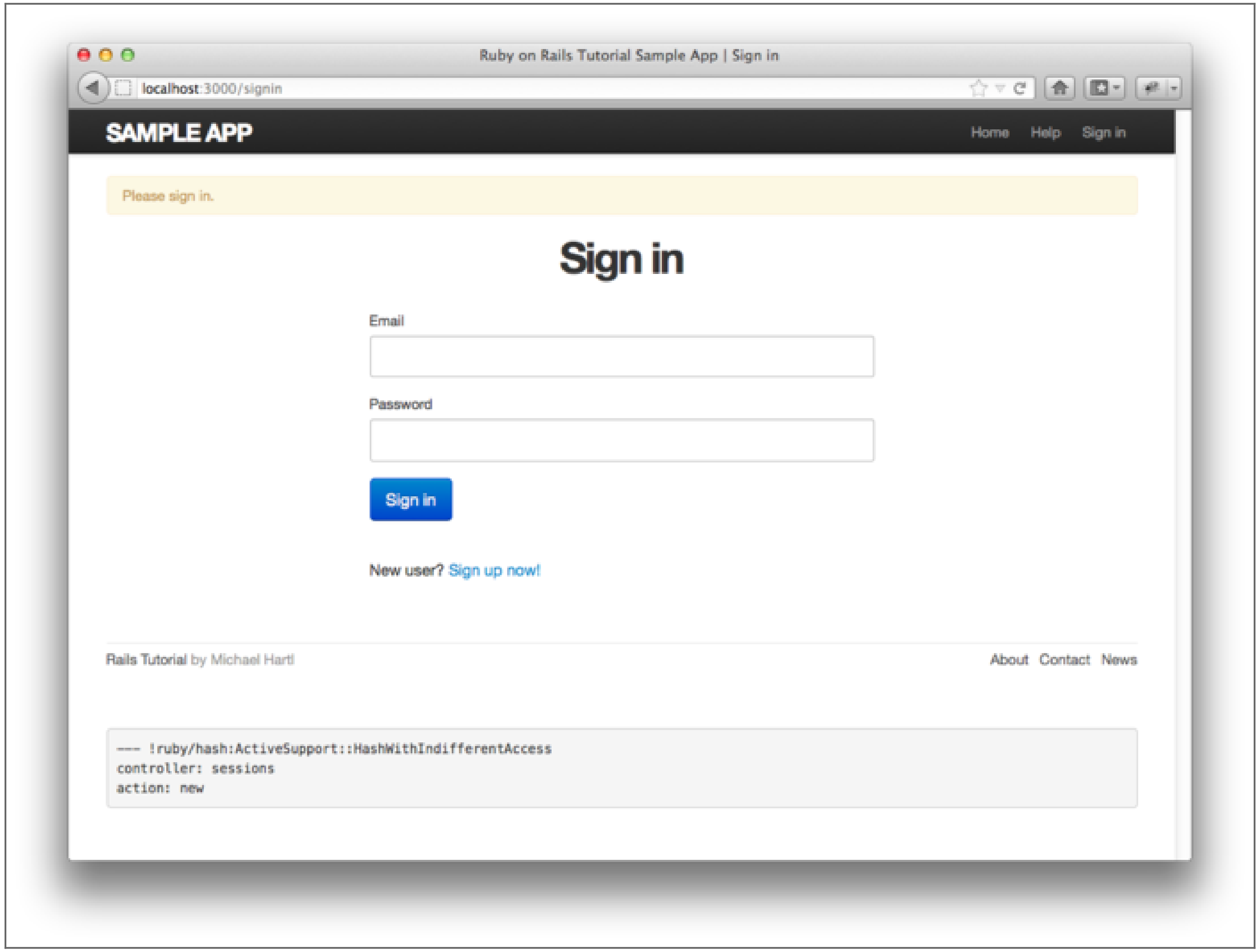
By default, before filters apply to *every* action in a controller, so here we restrict the filter to act only on the `:edit` and `:update` actions by passing the appropriate `:only` options hash.

Note that Listing 9.12 uses a shortcut for setting `flash[:notice]` by passing an options hash to the `redirect_to` function. The code in Listing 9.12 is equivalent to the more verbose

```
flash[:notice] = "Please sign in."
redirect_to signin_url
```

(The same construction works for the `:error` key, but not for `:success`.)

Together with `:success` and `:error`, the `:notice` key completes our triumvirate of `flash` styles, all of which are supported natively by Bootstrap CSS. By signing out and attempting to access the user edit page /users/1/edit, we can see the resulting yellow “notice” box, as seen in Figure 9.6.


Ruby on Rails Tutorial Sample App | Sign in

localhost:3000/signin

SAMPLE APP

Home Help Sign in

Please sign in.

Sign in

Email

Password

Sign in

New user? Sign up now!

Rails Tutorial by Michael Hartl

About Contact News

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: sessions
action: new
```

Figure 9.6: The signin form after trying to access a protected page. (full size)

Unfortunately, in the process of getting the authorization tests from Listing 9.11 to pass, we've broken the tests in Listing 9.1. Code like

```
describe "edit" do
  let(:user) { FactoryGirl.create(:user) }
  before { visit edit_user_path(user) }
  .
  .
  .
```

no longer works because visiting the edit user path requires a signed-in user. The solution is to sign in the user with the `sign_in` utility from Listing 9.6, as shown in Listing 9.13.

**Listing 9.13.** Adding a signin step to the edit and update tests.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "edit" do
    let(:user) { FactoryGirl.create(:user) }
    before do
      sign_in user
      visit edit_user_path(user)
    end
    .
    .
    .
  end
end
```

At this point our, test suite should be green:

```
$ bundle exec rspec spec/
```

## 9.2.2 Requiring the right user

Of course, requiring users to sign in isn't quite enough; users should only be allowed to edit their *own* information. We can test for this by first signing in as an incorrect user and then hitting the **`edit`** and **`update`** actions (Listing 9.14). Note that, since users should never even *try* to edit another user's profile, we redirect not to the signin page but to the root URL.

**Listing 9.14.** Testing that the **`edit`** and **`update`** actions require the right user.
**`spec/requests/authentication_pages_spec.rb`**

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as wrong user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:wrong_user) { FactoryGirl.create(:user, email: "wrong@example.com") }
      before { sign_in user }

      describe "visiting Users#edit page" do
        before { visit edit_user_path(wrong_user) }
        it { should_not have_selector('title', text: full_title('Edit user')) }
      end

      describe "submitting a PUT request to the Users#update action" do
        before { put user_path(wrong_user) }
        specify { response.should redirect_to(root_path) }
      end
    end
  end
end
```

Note here that a factory can take an option:

```
FactoryGirl.create(:user, email: "wrong@example.com")
```

This creates a user with a different email address from the default. The tests specify that this wrong user should not have access to the original user's `edit` or `update` actions.

The application code adds a second before filter to call the `correct_user` method, as shown in Listing 9.15.

**Listing 9.15.** A `correct_user` before filter to protect the edit/update pages.
`app/controllers/users_controller.rb`

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

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

```
    def signed_in_user
      redirect_to signin_url, notice: "Please sign in." unless signed_in?
    end

    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_path) unless current_user?(@user)
    end
end
```

The **`correct_user`** filter uses the **`current_user?`** boolean method, which we define in the Sessions helper (Listing 9.16).

**Listing 9.16.** The **`current_user?`** method.
**`app/helpers/sessions_helper.rb`**

```
module SessionsHelper
  .
  .
  .
  def current_user
    @current_user ||= User.find_by_remember_token(cookies[:remember_token])
  end

  def current_user?(user)
    user == current_user
  end
  .
  .
  .
end
```

Listing 9.15 also shows the updated **`edit`** and **`update`** actions. Before, in Listing 9.2, we had

```
def edit
  @user = User.find(params[:id])
```

```
end
```

and similarly for **`update`**. Now that the **`correct_user`** before filter defines **`@user`**, we can omit it from both actions.

Before moving on, you should verify that the test suite passes:

```
$ bundle exec rspec spec/
```

## 9.2.3 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 signing 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.

To test for such "friendly forwarding", we first visit the user edit page, which redirects to the signin page. We then enter valid signin information and click the "Sign in" button. The resulting page, which by default is the user's profile, should in this case be the "Edit user" page. The test for this sequence appears in Listing 9.17.

**Listing 9.17.** A test for friendly forwarding.
**`spec/requests/authentication_pages_spec.rb`**

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }

      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email",    with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do

          it "should render the desired protected page" do
            page.should have_selector('title', text: 'Edit user')
          end
        end
      end
    end
    .
    .
    .
  end
end
```

Now for the implementation.[4] 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. We accomplish this with a pair of methods, `store_location` and `redirect_back_or`, both defined in the Sessions helper (Listing 9.18).

**Listing 9.18.** Code to implement friendly forwarding.
`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def redirect_back_or(default)
    redirect_to(session[:return_to] || default)
    session.delete(:return_to)
  end

  def store_location
    session[:return_to] = request.url
  end
end
```

The storage mechanism is the `session` facility provided by Rails, which you can think of as being like an instance of the `cookies` variable from Section 8.2.1 that automatically expires upon browser close. We also use the `request` object to get the `url`, i.e., the URI/URL of the requested page. The `store_location` method puts the requested URI in the `session` variable under the key `:return_to`.

To make use of `store_location`, we need to add it to the `signed_in_user` before filter, as shown in Listing 9.19.

**Listing 9.19.** Adding `store_location` to the signed-in user before filter.
`app/controllers/users_controller.rb`

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

```
    def signed_in_user
      unless signed_in?
        store_location
        redirect_to signin_url, notice: "Please sign in."
      end
    end

    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_path) unless current_user?(@user)
    end
end
```

To implement the forwarding itself, we use the `redirect_back_or` method to redirect to the requested URI if it exists, or some default URI otherwise, which we add to the Sessions controller `create` action to redirect after successful signin (Listing 9.20). The `redirect_back_or` method uses the or operator `||` through

```
session[:return_to] || default
```

This evaluates to `session[:return_to]` unless it's `nil`, in which case it evaluates to the given default URI. Note that Listing 9.18 is careful to remove the forwarding URI; otherwise, subsequent signin attempts would forward to the protected page until the user closed his browser. (Testing for this is left as an exercise (Section 9.6.)

**Listing 9.20.** The Sessions `create` action with friendly forwarding.
`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])
        sign_in user
        redirect_back_or user
      else
        flash.now[:error] = 'Invalid email/password combination'
        render 'new'
      end
    end
    .
    .
    .
  end
```

With that, the friendly forwarding integration test in Listing 9.17 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:

```
$ bundle exec rspec spec/
```

## 9.3 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 about populating the database with sample users and *paginating* 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 9.7.[5] In Section 9.4, we'll add an administrative interface to the user index so that (presumably troublesome) users can be destroyed.

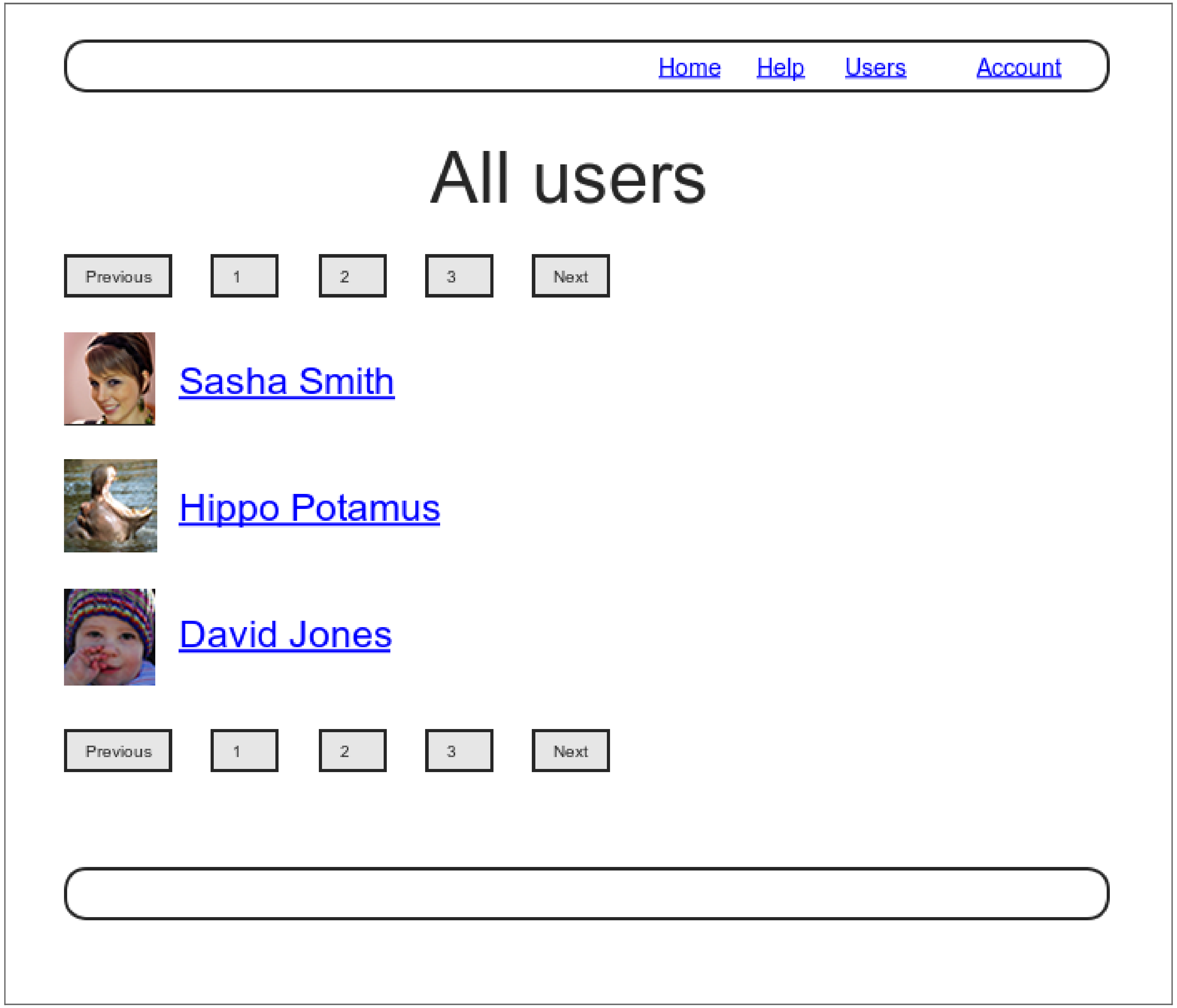


Figure 9.7: A mockup of the user index, with pagination and a “Users” nav link. (full size)

### 9.3.1 User index

Although we'll keep individual user `show` pages visible to all site visitors, the user `index` will be restricted to signed-in users so that there's a limit to how much unregistered users can see by default. We'll start by testing that the `index` action is protected by visiting the `users_path` (Table 7.1) and verifying that we are redirected to the signin page. As with other authorization tests, we'll put this example in the authentication integration test, as shown in Listing 9.21.

**Listing 9.21.** Testing that the `index` action is protected.
`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      .
      .
      .
      describe "in the Users controller" do
        .
        .
        .
        describe "visiting the user index" do
          before { visit users_path }
          it { should have_selector('title', text: 'Sign in') }
        end
      end
      .
      .
      .
    end
  end
end
```

The corresponding application code simply involves adding `index` to the list of actions protected by the `signed_in_user` before filter, as shown in Listing 9.22.

**Listing 9.22.** Requiring a signed-in user for the `index` action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
  end
  .
  .
  .
end
```

The next set of tests makes sure that, for signed-in users, the index page has the right title/heading and lists all of the site's users. The method is to make three factory users (signing in as the first one) and then verify that the index page has a list element (`li`) tag for the name of each one. Note that we've taken care to give the users different names so that each element in the list of users has a unique entry, as shown in Listing 9.23.

**Listing 9.23.** Tests for the user index page.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do
```

```
    before do
      sign_in FactoryGirl.create(:user)
      FactoryGirl.create(:user, name: "Bob", email: "bob@example.com")
      FactoryGirl.create(:user, name: "Ben", email: "ben@example.com")
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1',    text: 'All users') }

    it "should list each user" do
      User.all.each do |user|
        page.should have_selector('li', text: user.name)
      end
    end
  end
  .
  .
  .
end
```

As you may recall from the corresponding action in the demo app (Listing 2.4), the application code uses **`User.all`** to pull all the users out of the database, assigning them to an **`@users`** instance variable for use in the view, as seen in Listing 9.24. (If displaying all the users at once seems like a bad idea, you're right, and we'll remove this blemish in Section 9.3.3.)

**Listing 9.24.** The user **`index`** action.
**`app/controllers/users_controller.rb`**

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

To make the actual index page, we need to make a view that iterates through the users and wraps each one in an `li` tag. We do this with the `each` method, displaying each user’s Gravatar and name, while wrapping the whole thing in an unordered list (`ul`) tag (Listing 9.25). The code in Listing 9.25 uses the result of Listing 7.29 from Section 7.6, which allows us to pass an option to the Gravatar helper specifying a size other than the default. If you didn’t do that exercise, update your Users helper file with the contents of Listing 7.29 before proceeding.

**Listing 9.25.** The user index view.
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>
```

Let’s also add a little CSS (or, rather, SCSS) for style (Listing 9.26).

**Listing 9.26.** CSS for the user index.
`app/assets/stylesheets/custom.css.scss`

```
.
.
.
```

```
/* users index */

.users {
  list-style: none;
  margin: 0;
  li {
    overflow: auto;
    padding: 10px 0;
    border-top: 1px solid $grayLighter;
    &:last-child {
      border-bottom: 1px solid $grayLighter;
    }
  }
}
```

Finally, we'll add the URI to the users link in the site's navigation header using **`users_path`**, thereby using the last of the unused named routes in Table 7.1. The test (Listing 9.27) and application code (Listing 9.28) are both straightforward.

**Listing 9.27.** A test for the "Users" link URI.
**`spec/requests/authentication_pages_spec.rb`**

```
require 'spec_helper'

describe "Authentication" do
    .
    .
    .
    describe "with valid information" do
      let(:user) { FactoryGirl.create(:user) }
      before { sign_in user }

      it { should have_selector('title', text: user.name) }

      it { should have_link('Users',    href: users_path) }
      it { should have_link('Profile',  href: user_path(user)) }
      it { should have_link('Settings', href: edit_user_path(user)) }
      it { should have_link('Sign out', href: signout_path) }
      it { should_not have_link('Sign in', href: signin_path) }
      .
      .
      .
    end
  end
end
```

**Listing 9.28.** Adding the URI to the users link.

`app/views/layouts/_header.html.erb`

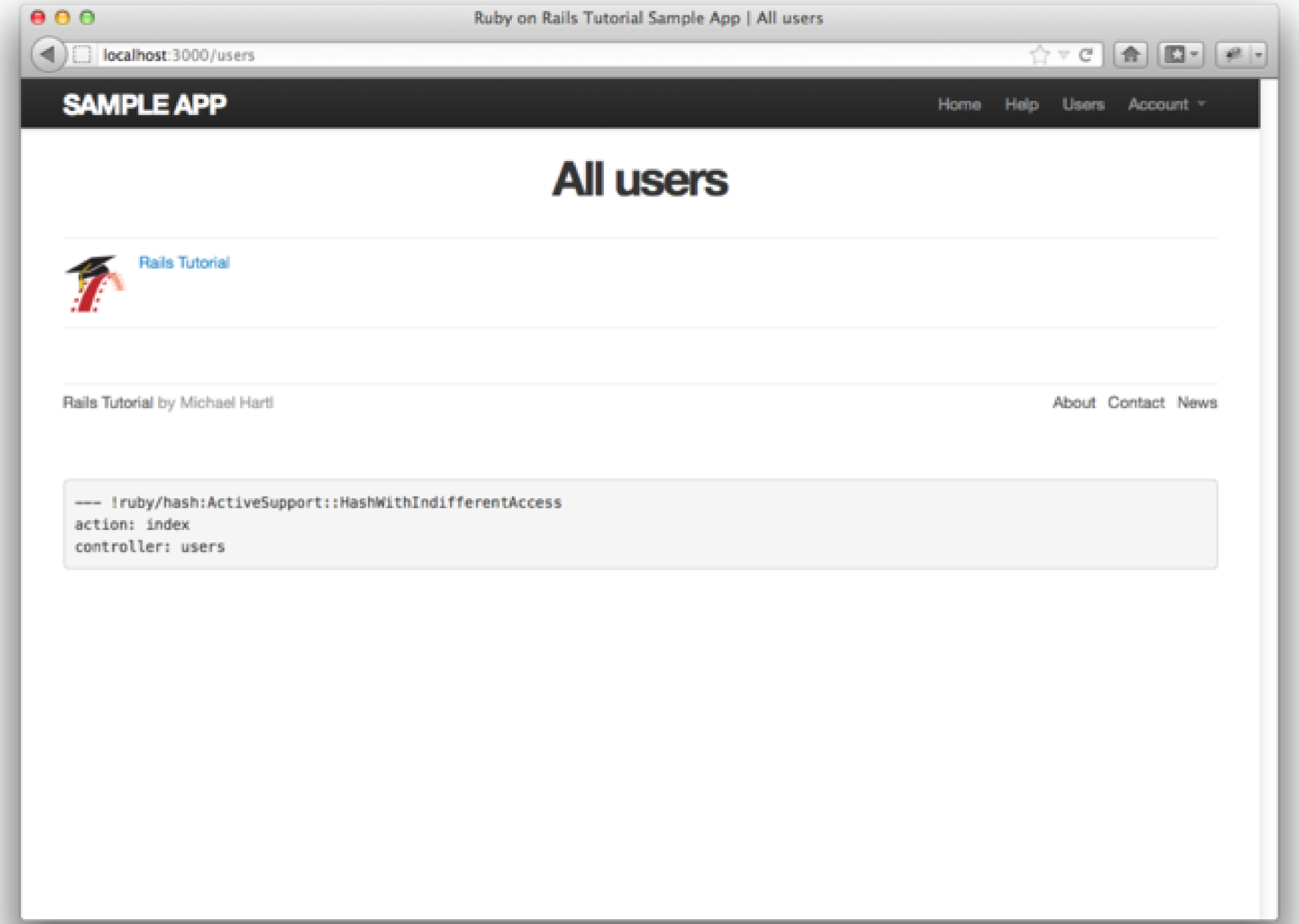
```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", root_path, id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home", root_path %></li>
          <li><%= link_to "Help", help_path %></li>
          <% if signed_in? %>
            <li><%= link_to "Users", users_path %></li>
            <li id="fat-menu" class="dropdown">
              <a href="#" class="dropdown-toggle" data-toggle="dropdown">
                Account <b class="caret"></b>
              </a>
              <ul class="dropdown-menu">
                <li><%= link_to "Profile", current_user %></li>
                <li><%= link_to "Settings", edit_user_path(current_user) %></li>
                <li class="divider"></li>
                <li>
                  <%= link_to "Sign out", signout_path, method: "delete" %>
                </li>
              </ul>
            </li>
          <% else %>
            <li><%= link_to "Sign in", signin_path %></li>
          <% end %>
        </ul>
      </nav>
    </div>
  </div>
```

```
</header>
```

With that, the user index is fully functional, with all tests passing:

```
$ bundle exec rspec spec/
```

On the other hand, as seen in Figure 9.8, it is a bit… lonely. Let’s remedy this sad situation.



```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: index
controller: users
```

Figure 9.8: The user index page /users with only one user. (full size)

### 9.3.2 Sample users

In this section, we'll give our lonely sample user some company. Of course, to create enough users to make a decent user index, we *could* use our web browser to visit the signup page and make the new users one by one, but far a better solution is to use Ruby (and Rake) to make the users for us.

First, we'll add the *Faker* gem to the **`Gemfile`**, which will allow us to make sample users with semi-realistic names and email addresses (Listing 9.29).

**Listing 9.29.** Adding the Faker gem to the **`Gemfile`**.

```
source 'https://rubygems.org'

gem 'rails', '3.2.8'
gem 'bootstrap-sass', '2.0.4'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.0.1'
.
.
.
```

Then install as usual:

```
$ bundle install
```

Next, we'll add a Rake task to create sample users. Rake tasks live in the **`lib/tasks`** directory, and are defined using *namespaces* (in this case, **`:db`**), as seen in Listing 9.30. (This is a bit advanced, so don't worry too much about the details.)

**Listing 9.30.** A Rake task for populating the database with sample users.
**`lib/tasks/sample_data.rake`**

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    User.create!(name: "Example User",
                 email: "example@railstutorial.org",
                 password: "foobar",
                 password_confirmation: "foobar")
    99.times do |n|
      name  = Faker::Name.name
      email = "example-#{n+1}@railstutorial.org"
      password  = "password"
      User.create!(name: name,
                   email: email,
                   password: password,
                   password_confirmation: password)
    end
  end
end
```

This defines a task **db:populate** that creates an example user with name and email address replicating our previous one, and then makes 99 more. The line

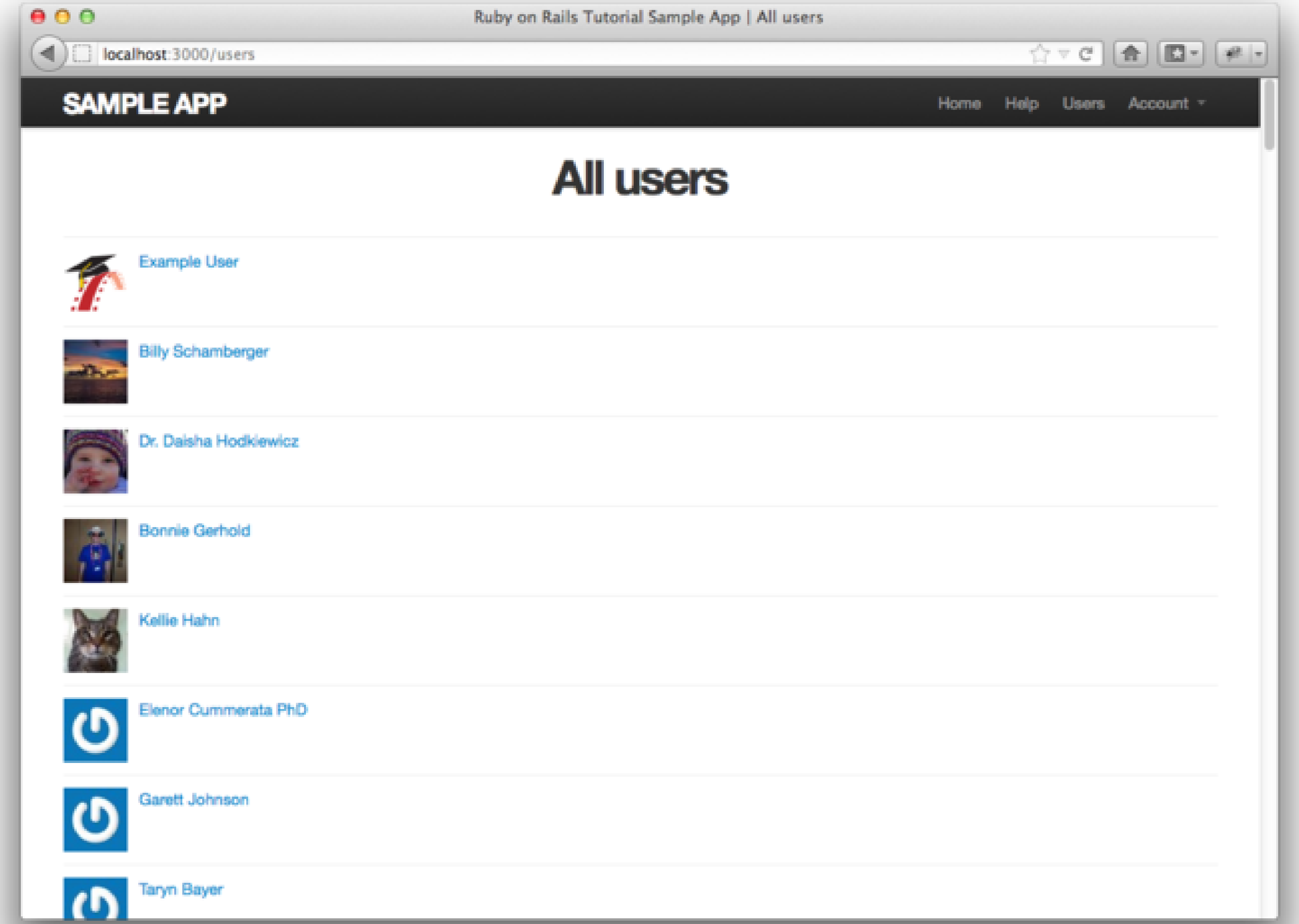
```
task populate: :environment do
```

ensures that the Rake task has access to the local Rails environment, including the User model (and hence **User.create!**). Here **create!** is just like the **create** method, except it raises an exception (Section 6.1.4) for an invalid user rather than returning **false**. This noisier construction makes debugging easier by avoiding silent errors.

With the **:db** namespace as in Listing 9.30, we can invoke the Rake task as follows:

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake db:test:prepare
```

After running the Rake task, our application has 100 sample users, as seen in Figure 9.9. (I've taken the liberty of associating the first few sample addresses with photos so that they're not all the default Gravatar image.)



All users

Example User
Billy Schamberger
Dr. Daisha Hodkiewicz
Bonnie Gerhold
Kellie Hahn
Elenor Cummerata PhD
Garett Johnson
Taryn Bayer

Figure 9.9: The user index page /users with 100 sample users. (full size)

### 9.3.3 Pagination

Our original user doesn't suffer from loneliness any more, but now we have the opposite problem: our user has *too many* companions, and they all appear on the same page. Right now there are a hundred, which is already a reasonably large number, and on a real site it could be thousands. The solution is to *paginate* the users, so that (for example) only 30 show up on a page at any one time.

There are several pagination methods in Rails; we'll use one of the simplest and most robust, called will_paginate. To use it, we need to include both the `will_paginate` gem and `bootstrap-will_paginate`, which configures will_paginate to use Bootstrap's pagination styles. The updated **`Gemfile`** appears in Listing 9.31.

**Listing 9.31.** Including `will_paginate` in the **`Gemfile`**.

```
source 'https://rubygems.org'

gem 'rails', '3.2.8'
gem 'bootstrap-sass', '2.0.4'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.0.1'
gem 'will_paginate', '3.0.3'
gem 'bootstrap-will_paginate', '0.0.6'
.
.
.
```

Then run **`bundle install`**:

```
$ bundle install
```

You should also restart the web server to insure that the new gems are loaded properly.

Because the `will_paginate` gem is in wide use, there's no need to test it thoroughly, so we'll take a lightweight approach. First, we'll test for a `div` with CSS class "pagination", which is what gets output by `will_paginate`. Then we'll verify that the correct users appear on the first page of results. This requires the use of the `paginate` method, which we'll cover shortly.

As before, we'll use Factory Girl to simulate users, but immediately we have a problem: user email addresses must be unique, which would appear to require creating more than 30 users by hand—a terribly cumbersome job. In addition, when testing for user listings it would be convenient for them all to have different names. Fortunately, Factory Girl anticipates this issue, and provides *sequences* to solve it. Our original factory (Listing 7.8) hard-coded the name and email address:

```
FactoryGirl.define do
  factory :user do
    name     "Michael Hartl"
    email    "michael@example.com"
    password "foobar"
    password_confirmation "foobar"
  end
end
```

Instead, we can arrange for a sequence of names and email addresses using the `sequence` method:

```
factory :user do
  sequence(:name)  { |n| "Person #{n}" }
  sequence(:email) { |n| "person_#{n}@example.com"}
  .
  .
  .
```

Here `sequence` takes a symbol corresponding to the desired attribute (such as `:name`) and a block with one variable, which we have called `n`. Upon successive invocations of the `FactoryGirl` method,

```
FactoryGirl.create(:user)
```

The block variable **`n`** is automatically incremented, so that the first user has name “Person 1” and email address “person_1@example.com”, the second user has name “Person 2” and email address “person_2@example.com”, and so on. The full code appears in Listing 9.32.

**Listing 9.32.** Defining a Factory Girl sequence.
**`spec/factories.rb`**

```
FactoryGirl.define do
  factory :user do
    sequence(:name)  { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com"}
    password "foobar"
    password_confirmation "foobar"
  end
end
```

Applying the idea of factory sequences, we can make 30 users in our test, which (as we will see) will be sufficient to invoke pagination:

```
before(:all) { 30.times { FactoryGirl.create(:user) } }
after(:all)  { User.delete_all }
```

Note here the use of **`before(:all)`**, which ensures that the sample users are created *once*, before all the tests in the block. This is an optimization for speed, as creating 30 users can be slow on some systems. We use the complementary method **`after(:all)`** to delete the users once we’re done.

The tests for the appearance of the pagination `div` and the right users appears in Listing 9.33. Note the replacement of the `User.all` array from Listing 9.23 with `User.paginate(page: 1)`, which (as we'll see momentarily) is how to pull out the first page of users from the database. Note also that Listing 9.33 uses `before(:each)` to emphasize the contrast with `before(:all)`.

**Listing 9.33.** Tests for pagination.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do

    let(:user) { FactoryGirl.create(:user) }

    before(:each) do
      sign_in user
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1',    text: 'All users') }

    describe "pagination" do

      before(:all) { 30.times { FactoryGirl.create(:user) } }
      after(:all)  { User.delete_all }

      it { should have_selector('div.pagination') }

      it "should list each user" do
        User.paginate(page: 1).each do |user|
          page.should have_selector('li', text: user.name)
        end
      end
    end
    end
    .
    .
    .
  end
```

To get pagination working, we need to add some code to the index view telling Rails to paginate the users, and we need to replace `User.all` in the `index` action with an object that knows about pagination. We'll start by adding the special `will_paginate` method in the view (Listing 9.34); we'll see in a moment why the code appears both above and below the user list.

**Listing 9.34.** The user index with pagination.
`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 52 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>

<%= will_paginate %>
```

The `will_paginate` method is a little magical; inside a `users` view, it automatically looks for an `@users` object, and then displays pagination links to access other pages. The view in Listing 9.34 doesn't work yet, though, because currently `@users` contains the results of `User.all` (Listing 9.24), which is of class `Array`, whereas `will_paginate` expects an object of class `ActiveRecord::Relation`. Happily, this is just the kind of object returned by the `paginate` method added by the `will_paginate` gem to all Active Record objects:

```
$ rails console
>> User.all.class
=> Array
>> User.paginate(page: 1).class
=> ActiveRecord::Relation
```

Note that **`paginate`** takes a hash argument with key **`:page`** and value equal to the page requested. **`User.paginate`** pulls the users out of the database one chunk at a time (30 by default), based on the **`:page`** parameter. So, for example, page 1 is users 1–30, page 2 is users 31–60, etc. If the page is **`nil`**, **`paginate`** simply returns the first page.

Using the **`paginate`** method, we can paginate the users in the sample application by using **`paginate`** in place of **`all`** in the **`index`** action (Listing 9.35). Here the **`:page`** parameter comes from **`params[:page]`**, which is generated automatically by **`will_paginate`**.

**Listing 9.35.** Paginating the users in the **`index`** action.
**`app/controllers/users_controller.rb`**

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.paginate(page: params[:page])
  end
  .
  .
  .
end
```

The user index page should now be working, appearing as in Figure 9.10. (On some systems, you may have to restart the Rails server at this point.) Because we included `will_paginate` both above and below the user list, the pagination links appear in both places.

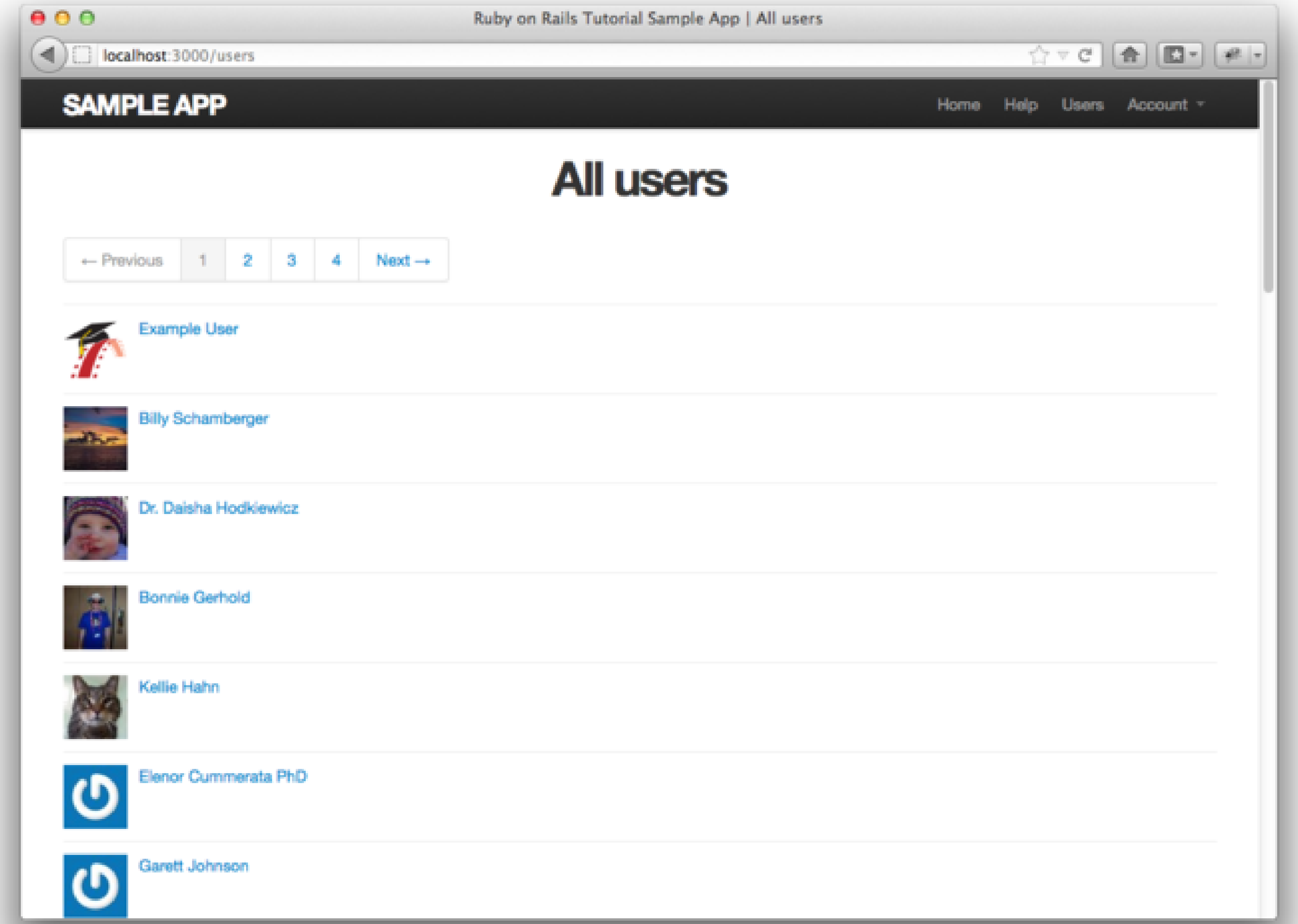
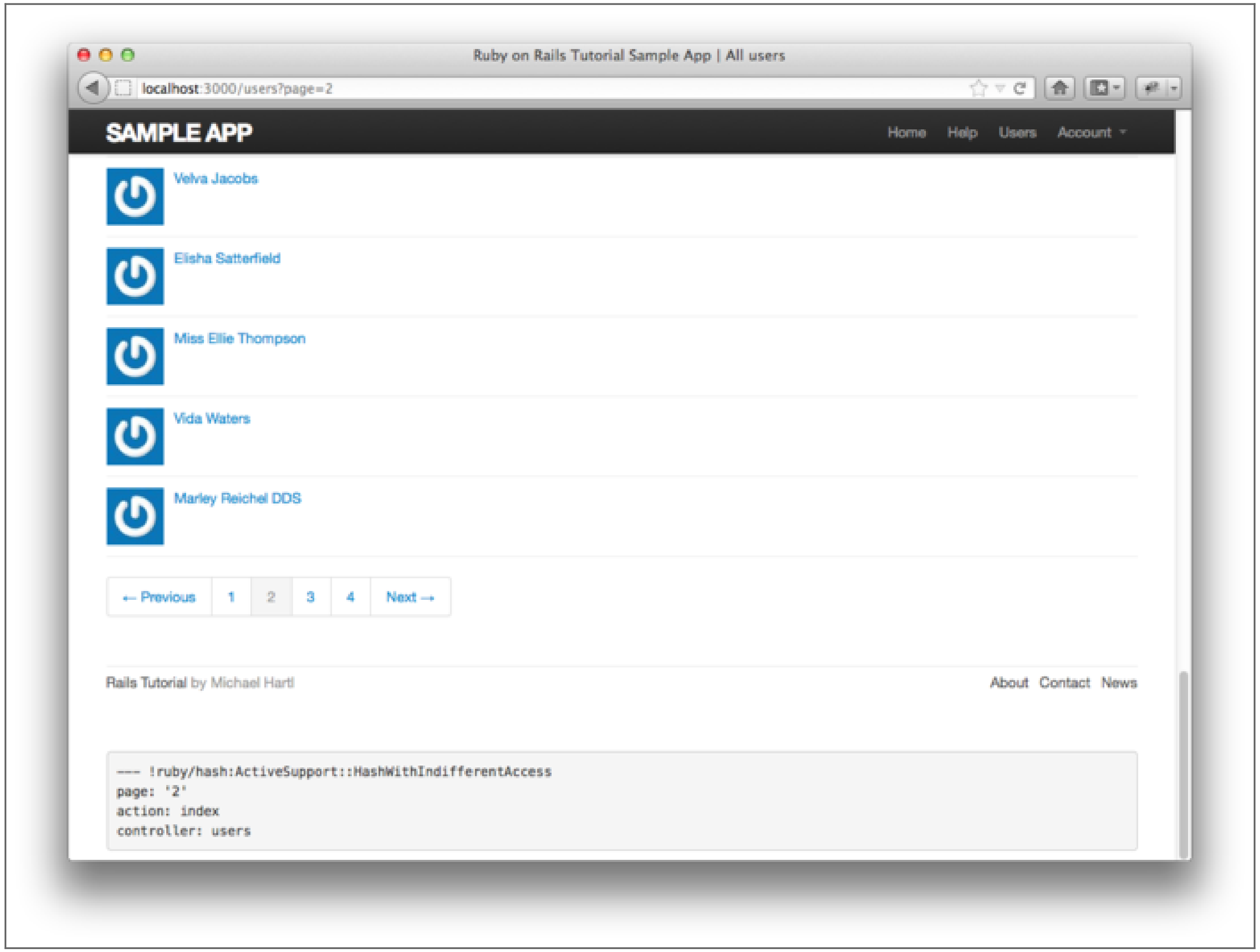


Figure 9.10: The user index page /users with pagination. (full size)

If you now click on either the 2 link or Next link, you'll get the second page of results, as shown in Figure 9.11.



```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
page: '2'
action: index
controller: users
```

Figure 9.11: Page 2 of the user index (/users?page=2). (full size)

You should also verify that the tests are passing:

```
$ bundle exec rspec spec/
```

### 9.3.4 Partial refactoring

The paginated user index is now complete, but there's one improvement I can't resist including: Rails has some incredibly slick tools for making compact views, and in this section we'll refactor the index page to use them. Because our code is well-tested, we can refactor with confidence, assured that we are unlikely to break our site's functionality.

The first step in our refactoring is to replace the user `li` from Listing 9.34 with a `render` call (Listing 9.36).

**Listing 9.36.** The first refactoring attempt at the index view.
**`app/views/users/index.html.erb`**

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <%= render user %>
  <% end %>
</ul>

<%= will_paginate %>
```

Here we call `render` not on a string with the name of a partial, but rather on a `user` variable of class `User`;[6] in this context, Rails automatically looks for a partial called `_user.html.erb`, which we must create (Listing 9.37).

**Listing 9.37.** A partial to render a single user.
**`app/views/users/_user.html.erb`**

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
</li>
```

This is a definite improvement, but we can do even better: we can call `render` *directly* on the `@users` variable (Listing 9.38).

**Listing 9.38.** The fully refactored user index.
**`app/views/users/index.html.erb`**

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <%= render @users %>
</ul>

<%= will_paginate %>
```

Here Rails infers that `@users` is a list of `User` objects; moreover, when called with a collection of users, Rails automatically iterates through them and renders each one with the `_user.html.erb` partial. The result is the impressively compact code in Listing 9.38. As with any refactoring, you should verify that the test suite is still green after changing the application code:

```
$ bundle exec rspec spec/
```

## 9.4 Deleting users

Now that the user index is complete, there’s only one canonical REST action left: **`destroy`**. In this section, we’ll add links to delete users, as mocked up in Figure 9.12, and define the **`destroy`** action necessary to accomplish the deletion. But first, we’ll create the class of administrative users authorized to do so.

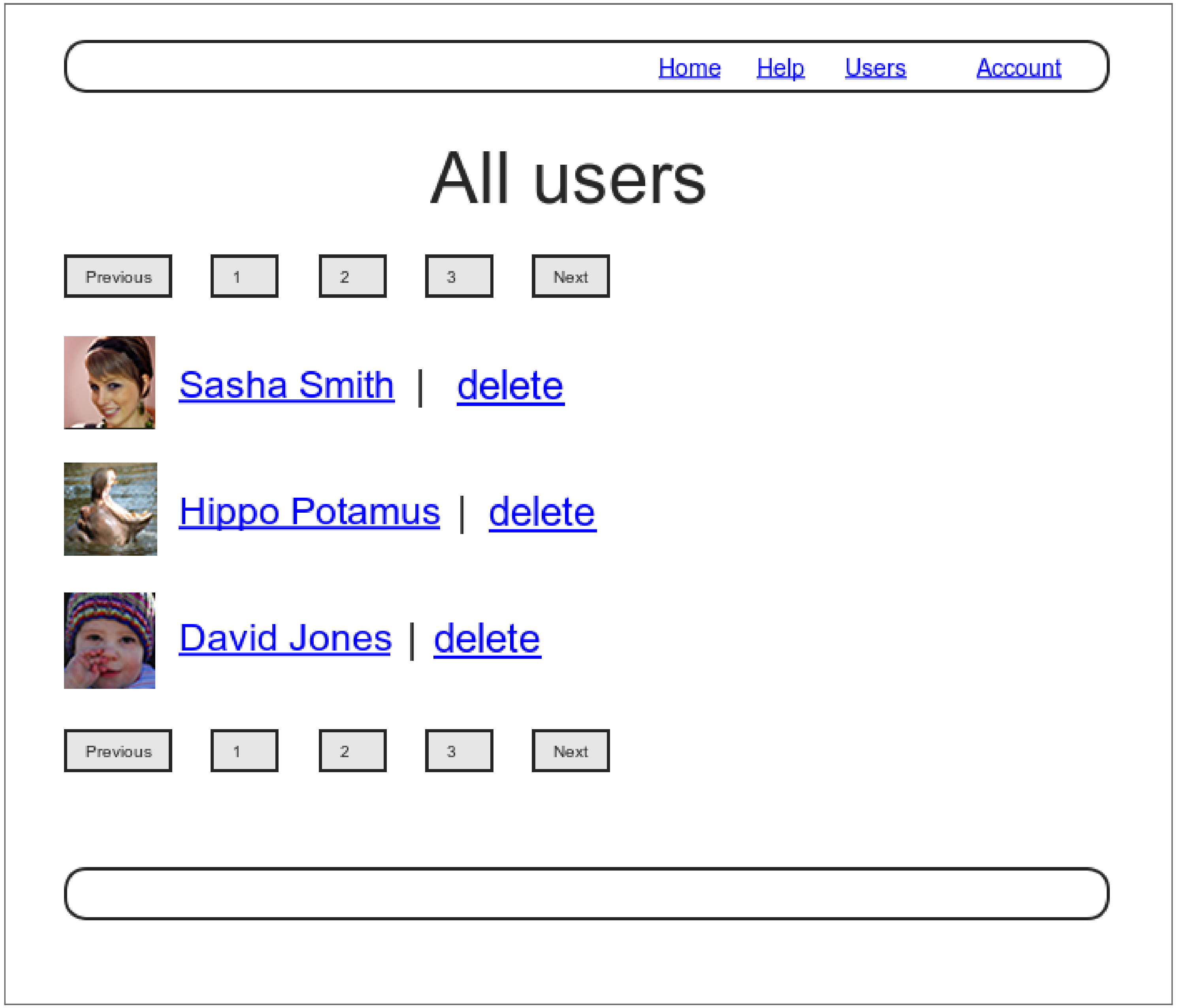


Figure 9.12: A mockup of the user index with delete links. (full size)

### 9.4.1 Administrative users

We will identify privileged administrative users with a boolean `admin` attribute in the User model, which, as we'll see, will automatically lead to an `admin?` boolean method to test for admin status. We can write tests for this attribute as in Listing 9.39.

**Listing 9.39.** Tests for an `admin` attribute.
`spec/models/user_spec.rb`

```ruby
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:admin) }
  it { should respond_to(:authenticate) }

  it { should be_valid }
  it { should_not be_admin }

  describe "with admin attribute set to 'true'" do
    before do
      @user.save!
      @user.toggle!(:admin)
    end

    it { should be_admin }
  end
  .
  .
  .
end
```

Here we've used the `toggle!` method to flip the `admin` attribute from `false` to `true`. Also note that the line

```
it { should be_admin }
```

implies (via the RSpec boolean convention) that the user should have an **`admin?`** boolean method.

As usual, we add the **`admin`** attribute with a migration, indicating the **`boolean`** type on the command line:

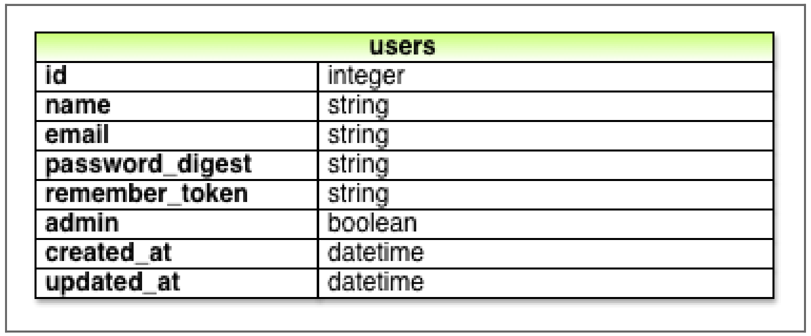
```
$ rails generate migration add_admin_to_users admin:boolean
```

The migration simply adds the **`admin`** column to the **`users`** table (Listing 9.40), yielding the data model in Figure 9.13.

**Listing 9.40.** The migration to add a boolean **`admin`** attribute to users.
**`db/migrate/[timestamp]_add_admin_to_users.rb`**

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

Note that we've added the argument **`default: false`** to **`add_column`** in Listing 9.40, which means that users will *not* be administrators by default. (Without the **`default: false`** argument, **`admin`** will be **`nil`** by default, which is still **`false`**, so this step is not strictly necessary. It is more explicit, though, and communicates our intentions more clearly both to Rails and to readers of our code.)



| users | |
|---|---|
| id | integer |
| name | string |
| email | string |
| password_digest | string |
| remember_token | string |
| admin | boolean |
| created_at | datetime |
| updated_at | datetime |

Figure 9.13: The User model with an added **`admin`** boolean attribute.

Finally, we migrate the development database and prepare the test database:

```
$ bundle exec rake db:migrate
$ bundle exec rake db:test:prepare
```

As expected, Rails figures out the boolean nature of the **`admin`** attribute and automatically adds the question-mark method **`admin?`**:

```
$ rails console --sandbox
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

As a result, the admin tests should pass:

```
$ bundle exec rspec spec/models/user_spec.rb
```

As a final step, let’s update our sample data populator to make the first user an admin by default (Listing 9.41).

**Listing 9.41.** The sample data populator code with an admin user.
**`lib/tasks/sample_data.rake`**

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    admin = User.create!(name: "Example User",
                         email: "example@railstutorial.org",
                         password: "foobar",
                         password_confirmation: "foobar")
    admin.toggle!(:admin)
    .
    .
    .
  end
end
```

Then reset the database and re-populate the sample data:

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
$ bundle exec rake db:test:prepare
```

## Revisiting `attr_accessible`

You might have noticed that Listing 9.41 makes the user an admin with **`toggle!(:admin)`**, but why not just add **`admin: true`** to the initialization hash? The answer is, it won’t work, and this is by design: only `attr_accessible` attributes can be assigned through mass assignment (that is, using an initialization hash, as in `User.new(name: "Foo", ...)`), and the `admin` attribute isn't accessible. Listing 9.42 reproduces the most recent list of `attr_accessible` attributes—note that `:admin` is *not* on the list.

**Listing 9.42.** The `attr_accessible` attributes for the User model *without* an `:admin` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  .
  .
  .
end
```

Explicitly defining accessible attributes is crucial for good site security. If we omitted the `attr_accessible` list in the User model (or foolishly added `:admin` to the list), a malicious user could send a PUT request as follows:[7]

```
put /users/17?admin=1
```

This request would make user 17 an admin, which would be a potentially serious security breach, to say the least. Because of this danger, it is a good practice to define `attr_accessible` for every model. In fact, it's a good idea to write a test for any attribute that *isn't* accessible; writing such a test for the `admin` attribute is left as an exercise (Section 9.6).

### 9.4.2 The `destroy` action

The final step needed to complete the Users resource is to add delete links and a `destroy` action. We'll start by adding a delete link for each user on the user index page, restricting access to administrative users.

To write tests for the delete functionality, it's helpful to be able to have a factory to create admins. We can accomplish this by adding an `:admin` block to our factories, as shown in Listing 9.43.

**Listing 9.43.** Adding a factory for administrative users.
`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    sequence(:name)  { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com"}
    password "foobar"
    password_confirmation "foobar"

    factory :admin do
      admin true
    end
  end
end
```

With the code in Listing 9.43, we can now use `FactoryGirl.create(:admin)` to create an administrative user in our tests.

Our security model requires that ordinary users not see delete links:

```
it { should_not have_link('delete') }
```

But administrative users should see such links, and by clicking on a delete link we expect an admin to delete the user, i.e., to change the `User` count by `-1`:

```
it { should have_link('delete', href: user_path(User.first)) }
it "should be able to delete another user" do
  expect { click_link('delete') }.to change(User, :count).by(-1)
end
it { should_not have_link('delete', href: user_path(admin)) }
```

Note that we have added a test to verify that the admin does not see a link to delete himself. The full set of delete link tests appears in Listing 9.44.

**Listing 9.44.** Tests for delete links.

**spec/requests/user_pages_spec.rb**

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "index" do

    let(:user) { FactoryGirl.create(:user) }

    before do
      sign_in user
      visit users_path
    end

    it { should have_selector('title', text: 'All users') }
    it { should have_selector('h1',    text: 'All users') }

    describe "pagination" do
      .
      .
      .
    end

    describe "delete links" do
```

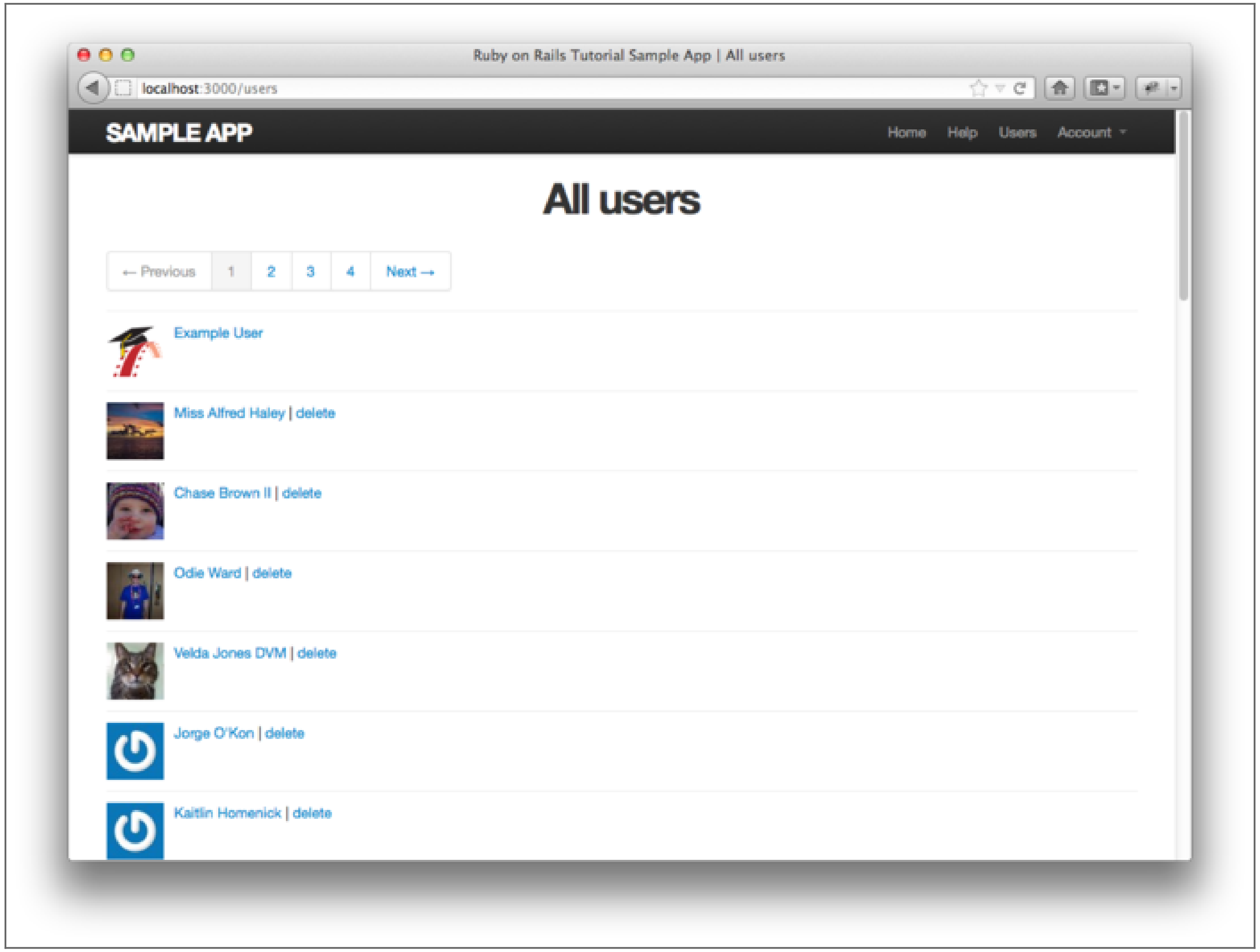
```
      it { should_not have_link('delete') }

      describe "as an admin user" do
        let(:admin) { FactoryGirl.create(:admin) }
        before do
          sign_in admin
          visit users_path
        end

        it { should have_link('delete', href: user_path(User.first)) }
        it "should be able to delete another user" do
          expect { click_link('delete') }.to change(User, :count).by(-1)
        end
        it { should_not have_link('delete', href: user_path(admin)) }
      end
    end
  end
  .
  .
  .
end
```

The application code links to `"delete"` if the current user is an admin (Listing 9.45). Note the `method: :delete` argument, which arranges for the link to issue the necessary `DELETE` request. We've also wrapped each link inside an `if` statement so that only admins can see them. The result for our admin user appears in Figure 9.14.

**Listing 9.45.** User delete links (viewable only by admins).
`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, size: 52 %>
  <%= link_to user.name, user %>
  <% if current_user.admin? && !current_user?(user) %>
    | <%= link_to "delete", user, method: :delete,
                                  data: { confirm: "You sure?" } %>
  <% end %>
</li>
```

Web browsers can't send DELETE requests natively, so Rails fakes them with JavaScript. This means that the delete links won't work if the user has JavaScript disabled. If you must support non-JavaScript-enabled browsers you can fake a DELETE request using a form and a POST request, which works even without JavaScript; see the RailsCast on "Destroy Without JavaScript" for details.



All users

← Previous 1 2 3 4 Next →

Example User

Miss Alfred Haley | delete

Chase Brown II | delete

Odie Ward | delete

Velda Jones DVM | delete

Jorge O'Kon | delete

Kaitlin Homenick | delete

Figure 9.14: The user index /users with delete links. (full size)

To get the delete links to work, we need to add a **destroy** action (Table 7.1), which finds the corresponding user and destroys it with the Active Record **destroy** method, finally redirecting to the user index, as seen in Listing 9.46. Note that we also add **:destroy** to the **signed_in_user** before filter.

**Listing 9.46.** Adding a working **destroy** action.
**app/controllers/users_controller.rb**

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_filter :correct_user,   only: [:edit, :update]
  .
  .
  .
  def destroy
    User.find(params[:id]).destroy
    flash[:success] = "User destroyed."
    redirect_to users_url
  end
  .
  .
  .
end
```

Note that the **destroy** action uses method chaining to combine the **find** and **destroy** into one line:

```
User.find(params[:id]).destroy
```

As constructed, only admins can destroy users through the web, because only admins can see the delete links. Unfortunately, there's still a terrible security hole: any sufficiently sophisticated attacker could simply issue `DELETE` requests directly from the command line to delete any user on the site. To secure the site properly, we also need access control on the **`destroy`** action, so our tests should check not only that admins *can* delete users, but also that other users *can't*. The results appear in Listing 9.47. Note that, in analogy with the **`put`** method from Listing 9.11, we use **`delete`** to issue a `DELETE` request directly to the specified URI (in this case, the user path, as required by Table 7.1).

**Listing 9.47.** A test for protecting the **`destroy`** action.
**`spec/requests/authentication_pages_spec.rb`**

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do
    .
    .
    .
    describe "as non-admin user" do
      let(:user) { FactoryGirl.create(:user) }
      let(:non_admin) { FactoryGirl.create(:user) }

      before { sign_in non_admin }

      describe "submitting a DELETE request to the Users#destroy action" do
        before { delete user_path(user) }
        specify { response.should redirect_to(root_path) }
      end
    end
  end
end
```

In principle, there's still a minor security hole, which is that an admin could delete himself by issuing a `DELETE` request directly. One might argue that such an admin is only getting what he deserves, but it would be nice to prevent such an occurrence, and doing so is left as an exercise (Section 9.6).

As you might suspect by now, the application code uses a before filter, this time to restrict access to the **`destroy`** action to admins. The resulting **`admin_user`** before filter appears in Listing 9.48.

**Listing 9.48.** A before filter restricting the **`destroy`** action to admins.
**`app/controllers/users_controller.rb`**

```
class UsersController < ApplicationController
  before_filter :signed_in_user, only: [:index, :edit, :update, :destroy]
  before_filter :correct_user,   only: [:edit, :update]
  before_filter :admin_user,     only: :destroy
  .
  .
  .
  private
    .
    .
    .
    def admin_user
      redirect_to(root_path) unless current_user.admin?
    end
end
```

At this point, all the tests should be passing, and the Users resource—with its controller, model, and views—is functionally complete.

```
$ bundle exec rspec spec/
```

## 9.5 Conclusion

We've come a long way since introducing the Users controller way back in Section 5.4. Those users couldn't even sign up; now users can sign up, sign in, sign out, view their profiles, edit their settings, and see an index of all users—and some can even destroy other users.

The rest of this book builds on the foundation of the Users resource (and associated authorization system) to make a site with Twitter-like microposts (Chapter 10) and a status feed of posts from followed users (Chapter 11). These chapters will introduce some of the most powerful features of Rails, including data modeling with **`has_many`** and **`has_many through`**.

Before moving on, be sure to merge all the changes into the master branch:

```
$ git add .
$ git commit -m "Finish user edit, update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
```

You can also deploy the application and even populate the production database with sample users (using the **`pg:reset`** task to reset the production database):

```
$ git push heroku
$ heroku pg:reset <DATABASE>
$ heroku run rake db:migrate
$ heroku run rake db:populate
```

Here you'll have to replace **`<DATABASE>`** with the name of your database, which you can determine with the command

```
$ heroku config | grep POSTGRESQL
```

You may also have to redeploy to Heroku to force an app restart. Here's a hack that will force Heroku to restart the application:

```
$ touch foo
$ git add foo
$ git commit -m "foo"
$ git push heroku
```

It's also worth noting that this chapter saw the last of the necessary gem installations. For reference, the final `Gemfile` is shown in Listing 9.49.

**Listing 9.49.** The final `Gemfile` for the sample application.

```
source 'https://rubygems.org'

gem 'rails', '3.2.8'
gem 'bootstrap-sass', '2.0.4'
gem 'bcrypt-ruby', '3.0.1'
gem 'faker', '1.0.1'
gem 'will_paginate', '3.0.3'
gem 'bootstrap-will_paginate', '0.0.6'

group :development, :test do
  gem 'sqlite3', '1.3.5'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '3.2.5'
  gem 'coffee-rails', '3.2.2'
  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

group :test, :development do
```

```
  gem 'rspec-rails', '2.11.0'
  gem 'guard-rspec', '0.5.5'
  gem 'guard-spork', '0.3.2'
  gem 'spork', '0.9.0'
end

group :test do
  gem 'capybara', '1.1.2'
  gem 'factory_girl_rails', '1.4.0'
  gem 'cucumber-rails', '1.2.1', require: false
  gem 'database_cleaner', '0.7.0'
end

group :production do
  gem 'pg', '0.12.2'
end
```

## 9.6 Exercises

1. Following the model in Listing 10.8, add a test to verify that the User **`admin`** attribute isn't accessible. Be sure to get first to Red, and then to Green. (*Hint*: Your first step should be to *add* **`admin`** to the accessible list.)

2. Arrange for the Gravatar "change" link in Listing 9.3 to open in a new window (or tab). *Hint:* Search the web; you should find one particularly robust method involving something called **`_blank`**.

3. The current authentication tests check that navigation links such as "Profile" and "Settings" appear when a user is signed in. Add tests to make sure that these links *don't* appear when a user isn't signed in.

4. Use the **`sign_in`** test helper from Listing 9.6 in as many places as you can find.

5. Remove the duplicated form code by refactoring the **`new.html.erb`** and **`edit.html.erb`** views to use the partial in Listing 9.50. Note that you will have to pass the form variable **`f`**

explicitly as a local variable, as shown in Listing 9.51. You will also have to update the tests, as the forms aren't currently *exactly* the same; identify the slight difference and update the tests accordingly.

6. Signed-in users have no reason to access the **new** and **create** actions in the Users controller. Arrange for such users to be redirected to the root URL if they do try to hit those pages.
7. Learn about the **request** object by inserting some of the methods listed in the Rails API[8] into the site layout. (Refer to Listing 7.1 if you get stuck.)
8. Write a test to make sure that the friendly forwarding only forwards to the given URI the first time. On subsequent signin attempts, the forwarding URI should revert to the default (i.e., the profile page). See Listing 9.52 for a hint (and, by a hint, I mean the solution).
9. Modify the **destroy** action to prevent admin users from destroying themselves. (Write a test first.)

**Listing 9.50.** A partial for the new and edit form fields.
**app/views/users/_fields.html.erb**

```
<%= render 'shared/error_messages' %>

<%= f.label :name %>
<%= f.text_field :name %>

<%= f.label :email %>
<%= f.text_field :email %>

<%= f.label :password %>
<%= f.password_field :password %>

<%= f.label :password_confirmation, "Confirm Password" %>
<%= f.password_field :password_confirmation %>
```

**Listing 9.51.** The new user view with partial.

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

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

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'fields', f: f %>
      <%= f.submit "Create my account", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

**Listing 9.52.** A test for forwarding to the default page after friendly forwarding.

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      .
      .
      .
      describe "when attempting to visit a protected page" do
        before do
          visit edit_user_path(user)
          fill_in "Email",    with: user.email
          fill_in "Password", with: user.password
          click_button "Sign in"
        end

        describe "after signing in" do
```

```
          it "should render the desired protected page" do
            page.should have_selector('title', text: 'Edit user')
          end

          describe "when signing in again" do
            before do
              visit signin_path
              fill_in "Email",    with: user.email
              fill_in "Password", with: user.password
              click_button "Sign in"
            end

            it "should render the default (profile) page" do
              page.should have_selector('title', text: user.name)
            end
          end
        end
      end
    end
    .
    .
    .
  end
end
```

« Chapter 8 Sign in, sign out                                        Chapter 10 User microposts »

---

1. Image from http://www.flickr.com/photos/sashawolff/4598355045/. ↑
2. The Gravatar site actually redirects this to http://en.gravatar.com/emails, which is for English language users, but I've omitted the en part to account for the use of other languages. ↑
3. Don't worry about how this works; the details are of interest to developers of the Rails framework itself, and by design are not important for Rails application developers. ↑
4. The code in this section is adapted from the Clearance gem by thoughtbot. ↑

5. Baby photo from http://www.flickr.com/photos/glasgows/338937124/. ↑

6. The name `user` is immaterial—we could have written `@users.each do |foobar|` and then used `render foobar`. The key is the *class* of the object—in this case, `User`. ↑

7. Command-line tools such as `curl` can issue PUT requests of this form. ↑

8. http://api.rubyonrails.org/v3.2.0/classes/ActionDispatch/Request.html ↑