Chapter 10 Updating, showing, and deleting users

In this chapter, we will complete the REST actions for the Users resource (Table 6.2) 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 authentication code in Chapter 9). Then we'll make a listing of all users (also requiring authentication), which will motivate the introduction of sample data and pagination. Finally, we'll add the ability to destroy users, wiping them 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) along the way.

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

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

10.1 Updating users

The pattern for editing user information closely parallels that for creating new users (Chapter 8). 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 3.1). The biggest difference is that, while anyone can sign up, only the current user should be able to update their information. This means that we need to enforce access control so that only authorized users can edit and update; the authentication machinery from Chapter 9 will allow us to use a *before filter* to ensure that this is the case.

10.1.1 Edit form

We start with tests for the edit form, whose mockup appears in Figure 10.1.[1] Two are analogous to tests we saw for the `new` user page (Listing 8.1), checking for the proper response and title; the third test makes sure that there is a link to edit the user's Gravatar image (Section 7.3.2). If you poke around the Gravatar site, you'll see that the page to add or edit images is (somewhat oddly) located at `http://gravatar.com/emails`, so we test the `edit` page for a link with that URL.[2] The result is shown in Listing 10.1.

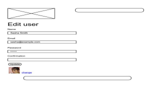


Figure 10.1: A mockup of the user edit page. (full size)

Listing 10.1. Tests for the user `edit` action.

spec/controllers/users_controller_spec.rb

```
require 'spec_helper'

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

    before(:each) do
      @user = Factory(:user)
      test_sign_in(@user)
    end

    it "should be successful" do
      get :edit, :id => @user
      response.should be_success
    end

    it "should have the right title" do
      get :edit, :id => @user
      response.should have_tag("title", /edit user/i)
    end

    it "should have a link to change the Gravatar" do
      get :edit, :id => @user
      gravatar_url = "http://gravatar.com/emails"
      response.should have_tag("a[href=?]", gravatar_url, /change/i)
    end
  end
end
```

Here we've made sure to use `test_sign_in(@user)` to sign in as the user in anticipation of protecting the edit page from unauthorized access (Section 10.2). Otherwise, these tests would break as soon as we implemented our authentication code.

Note from Table 6.2 that the proper URL 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 10.2. This uses `find` to find the relevant user in the database, and then sets the `@title` variable to the proper value.

Listing 10.2. An incomplete user `edit` action.
app/controllers/users_controller.rb

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

Getting the tests to pass requires making the actual edit view, shown in Listing 10.3. Note how closely this resembles the new user view from Listing 8.2; the large overlap suggests factoring the repeated code into a partial, which is left as an exercise (Section 10.6).

Listing 10.3. The user edit view.
`app/views/users/edit.html.erb`

```
<h2>Edit user</h2>

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

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

We’ll also add a link to the site navigation for the user edit page (which we’ll call “Settings”), as mocked up in Figure 10.23 and shown in Listing 10.4.

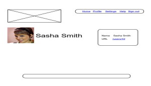


Figure 10.2: A mockup of the user profile page with a “Settings” link. (full size)

Listing 10.4. Adding a Settings link.
`app/views/layouts/_header.html.erb`

```
<div id="header" class="round">
  <%= link_to logo, root_path %>
  <ul class="navigation round">
    <li><%= link_to "Home", root_path %></li>
    <% if signed_in? %>
    <li><%= link_to "Profile", current_user %></li>
    <li><%= link_to "Settings", edit_user_path(current_user) %></li>
    <% end %>
    .
    .
    .
  </ul>
</div>
```

Here we use the named route `edit_user_path` from Table 6.2, together with the handy `current_user` helper method defined in Listing 9.19.

With the `@user` instance variable from Listing 10.2, the tests from Listing 10.1 pass. As seen in Figure 10.3, the `new` page renders, though it doesn't yet work.

Figure 10.3: Editing user settings (`/users/1/edit`). (full size)

Looking at the HTML source for Figure 10.3, we see a form tag as expected (Listing 10.5).

Listing 10.5. HTML for the edit form defined in Listing 10.3 and shown in Figure 10.3.

```
<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 6.2), Rails fakes it with a `POST` request and a hidden `input` field.4

There's another subtlety to address here: the code `form_for(@user)` in Listing 10.3 is *exactly* the same as the code in Listing 8.2—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 the `new_record?` boolean method:

```
$ script/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`.

10.1.2 Enabling edits

Although the edit form doesn't yet work, we've outsourced image upload to Gravatar, so it works straightaway by clicking on the "change" link from Figure 10.3, as shown in Figure 10.4.

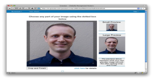


Figure 10.4: The Gravatar image-cropping interface, with a picture of some dude. (full size)

Let's get the rest of the user edit functionality working as well.

The tests for the `update` action are similar to those for `create`. In particular, we test both update failure and update success (Listing 10.6). (This is a lot of code; see if you can work through it by referring back to the tests in Chapter 8.)

Listing 10.6. Tests for the user `update` action.
`spec/controllers/users_controller_spec.rb`

```
describe UsersController do
  integrate_views
  .
  .
  .
  describe "PUT 'update'" do

    before(:each) do
      @user = Factory(:user)
      test_sign_in(@user)
      User.should_receive(:find).with(@user).and_return(@user)
    end

    describe "failure" do

      before(:each) do
        @invalid_attr = { :email => "", :name => "" }
        @user.should_receive(:update_attributes).and_return(false)
      end

      it "should render the 'edit' page" do
        put :update, :id => @user, :user => @invalid_attr
        response.should render_template('edit')
      end

      it "should have the right title" do
        put :update, :id => @user, :user => @invalid_attr
        response.should have_tag("title", /edit user/i)
      end
    end

    describe "success" do

      before(:each) do
        @attr = { :name => "New Name", :email => "user@example.org",
                  :password => "barbaz", :password_confirmation => "barbaz" }
        @user.should_receive(:update_attributes).and_return(true)
      end

      it "should redirect to the user show page" do
        put :update, :id => @user, :user => @attr
        response.should redirect_to(user_path(@user))
      end

      it "should have a flash message" do
        put :update, :id => @user, :user => @attr
        flash[:success].should =~ /updated/
      end
    end
  end
end
```

Here we’ve set up the message expectations to require that `User.find` be called, as in

```
User.should_receive(:find).with(@user).and_return(@user)
```

which also arranges to return the proper user. We also require that `update_attributes` be called, as in

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

and

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

Compare this to the code from Chapter 8, where we required calls to `save`, e.g.,

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

and

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

(The expectations are located in `before` blocks so that they execute before every test, thereby preventing any of the `put` calls from actually hitting the database. In a previous version of this book, the code in Chapter 8 and Chapter 9 didn’t put the expectations in the before block; if you have been reading chapters immediately after their release, you might want to review those chapters before proceeding.)

The `update` action needed to get the tests in Listing 10.6 to pass is similar to the final form of the `create` action (Listing 9.26), as seen in Listing 10.7.

Listing 10.7. 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."
      redirect_to @user
    else
      @title = "Edit user"
      render 'edit'
    end
  end
end
```

With that, the user edit page should be working. As presently constructed, every edit requires the user to reconfirm the password (as implied by the empty confirmation text box in Figure 10.3), which makes updates more secure but is a minor annoyance. (I plan to show how to allow user updates without a password confirmation in the Rails Tutorial screencasts.)

10.2 Protecting pages

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-signed-in users) to access either action, and any signed-in user can update the information for any other user.5 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 10.5.

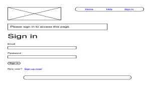


Figure 10.5: A mockup of the result of visiting a protected page (full size)

10.2.1 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 redirected to the signin page, as seen in Listing 10.8.

Listing 10.8. The first tests for authentication.
`spec/controllers/users_controller_spec.rb`

```
describe UsersController do
  integrate_views
  .
  .
  .
  describe "authentication of edit/update pages" do

    before(:each) do
      @user = Factory(:user)
    end

    describe "for non-signed-in users" do
```

```
      it "should deny access to 'edit'" do
        get :edit, :id => @user
        response.should redirect_to(signin_path)
      end

      it "should deny access to 'update'" do
        put :update, :id => @user, :user => {}
        response.should redirect_to(signin_path)
      end
    end
  end
end
```

The application code gets these tests to pass using a *before filter*, which arranges for a particular method to be called before the given actions. In this case, we define an `authenticate` method and invoke it using `before_filter :authenticate`, as shown in Listing 10.9.

Listing 10.9. Adding an `authenticate` before filter.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :authenticate, :only => [:edit, :update]
  .
  .
  .
  private

    def authenticate
      deny_access 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 `:only` options hash.

This code won't work yet, because `deny_access` hasn't been defined. Since access denial is part of authentication, we'll put it in the Sessions helper from Chapter 9. All `deny_access` does is put a message in `flash[:notice]` and then redirect to the signin page (Listing 10.10).

Listing 10.10. The `deny_access` method for user authentication.
`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def deny_access
    flash[:notice] = "Please sign in to access this page."
    redirect_to signin_path
  end
end
```

Together with `:success` and `:error`, the `:notice` key completes our triumvirate of `flash` styles, all of which are supported natively by Blueprint 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 10.6.

Figure 10.6: The signin form after trying to access a protected page. (full size)

10.2.2 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 10.11). 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 10.11. Authentication tests for signed-in users.
`spec/controllers/users_controller_spec.rb`

```
describe UsersController do
  integrate_views
  .
  .
  .
  describe "authentication of edit/update pages" do
    .
    .
    .
    describe "for signed-in users" do

      before(:each) do
        wrong_user = Factory(:user, :email => "user@example.net")
        test_sign_in(wrong_user)
      end

      it "should require matching users for 'edit'" do
        get :edit, :id => @user
        response.should redirect_to(root_path)
      end

      it "should require matching users for 'update'" do
        put :update, :id => @user, :user => {}
        response.should redirect_to(root_path)
      end
    end
  end
end
```

The application code is simple: we add a second before filter to call the `correct_user` method (which we have to write), as shown in Listing 10.12.

Listing 10.12. A `correct_user` before filter to protect the edit/update pages.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :authenticate, :only => [:edit, :update]
  before_filter :correct_user, :only => [:edit, :update]
  .
  .
  .
  private

    def authenticate
      deny_access unless signed_in?
    end

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

This uses the `current_user?` method, which (as with `deny_access`) we will define in the Sessions helper (Listing 10.13).

Listing 10.13. The `current_user?` method.
`app/helpers/sessions_helper.rb`

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

  def deny_access
    flash[:notice] = "Please sign in to access this page."
    redirect_to signin_path
  end
end
```

Although this implementation now works, your test suite should fail. We'll understand why it fails (and get it to pass) in the next section.

10.2.3 An expectation bonus

As you can verify by filling in the edit form and submitting, the edit/update code currently works, but the test suite fails. This seems like a bug, but it's actually a feature: message expectations like

```
User.should_receive(:find)
```

require that the method be called *exactly once*. The test failure is thus a hint that `find` is being called too many times. Indeed, upon inspecting the current Users controller, we see that `User.find` is called both in the `edit/user` action *and* in the `correct_user` before filter. To remove the superfluous call, we can simply rely on the before filter to find `@user` and remove the other lines, as shown in Listing 10.14; the resulting `@user` instance variable will then be available in all the actions and views as usual.

Listing 10.14. Removing the superfluous `@user` variable assignments.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def edit
    @title = "Edit user"
  end

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

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

At this point, all the tests should pass, and we are nearly done with protecting pages.

10.2.4 Friendly forwarding

Our page protection 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.

The sequence of attempted page visitation, signin, and redirect to destination page is a perfect job for an integration test, so let's make one for friendly forwarding:

```
$ script/generate integration_spec friendly_forwarding
```

The code then appears as in Listing 10.15.

Listing 10.15. An integration test for friendly forwarding.
`spec/integration/friendly_forwardings_spec.rb`

```
require 'spec_helper'

describe "FriendlyForwardings" do

  it "should forward to the requested page after signin" do
    user = Factory(:user)
    visit edit_user_path(user)
    # Webrat automatically follows the redirect to the signin page.
    fill_in :email,    :with => user.email
    fill_in :password, :with => user.password
    click_button
    # Webrat follows the redirect again, this time to users/edit.
    response.should render_template('users/edit')
  end
end
```

(As indicated by the comments, Webrat *follows* redirects, so testing that the response `should redirect_to` some URL won't work. I learned this the hard way.)

Now for the implementation.6 In order to forward users to their intended destination, we need to store the location of the requested page somewhere, and then redirect there instead. 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 9.3.3 that automatically expires upon browser close.7 We also use the `request` object to get the `request_uri`, i.e., the URL of the requested page. The resulting application code appears in Listing 10.16.

Listing 10.16. Code to implement friendly forwarding.
`app/helpers/sessions_helper.rb`

```
module SessionsHelper
  .
  .
  .
  def deny_access
    store_location
    flash[:notice] = "Please sign in to access this page."
    redirect_to signin_path
  end

  def store_location
    session[:return_to] = request.request_uri
  end

  def redirect_back_or(default)
    redirect_to(session[:return_to] || default)
    clear_return_to
  end

  def clear_return_to
    session[:return_to] = nil
  end
```

```
end
```

Here we’ve added a line to the `deny_access` method, first storing the location of the request with `store_location` and then proceeding as before. The `store_location` method puts the requested URL in the `session` variable under the key `:return_to`.

We’ve also defined the `redirect_back_or` method to redirect to the requested URL if it exists, or some default URL otherwise. This method is needed in the Session `create` action to redirect after successful signin (Listing 10.17).

Listing 10.17. The Sessions `create` action with friendly forwarding.
`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.authenticate(params[:session][:email],
                             params[:session][:password])
    if user.nil?
      flash.now[:error] = "Invalid email/password combination."
      @title = "Sign in"
      render 'new'
    else
      sign_in user
      redirect_back_or user
    end
  end
  .
  .
  .
end
```

With that, the friendly forwarding integration test in Listing 10.15 should pass, and the basic user authentication and page protection implementation is complete.

10.3 Showing users

In this section, we’ll add the penultimate user action, the `index` action, which is designed to display *all* the users, not 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 10.7.8 In Section 10.4, we’ll add an administrative interface to the user index so that (presumably troublesome) users can be destroyed.

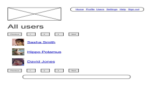


Figure 10.7: A mockup of the user index, with pagination and a “Users” nav link. (full size)

10.3.1 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. Our `index` tests check for this, and also verify that for signed-in users all the site's users are listed (Listing 10.18).

Listing 10.18. Tests for the user index page.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views

  describe "GET 'index'" do

    describe "for non-signed-in users" do
      it "should deny access" do
        get :index
        response.should redirect_to(signin_path)
        flash[:notice].should =~ /sign in/i
      end
    end

    describe "for signed-in users" do

      before(:each) do
        @user = test_sign_in(Factory(:user))
        second = Factory(:user, :email => "another@example.com")
        third  = Factory(:user, :email => "another@example.net")

        @users = [@user, second, third]
        User.should_receive(:all).and_return(@users)
      end

      it "should be successful" do
        get :index
        response.should be_success
      end

      it "should have the right title" do
        get :index
        response.should have_tag("title", /all users/i)
      end

      it "should have an element for each user" do
        get :index
        @users.each do |user|
          response.should have_tag("li", user.name)
        end
      end
    end
  end
  .
  .
  .
end
```

As you can see, the method for checking the index page 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 the users themselves get returned via the message expectation

```
User.should_receive(:all).and_return(@users)
```

which requires the `index` action to call `all` on the `User` class, thereby fetching all the users from the database (as seen in Section 6.1.4).

As expected, the application code uses `User.all` to make an `@users` instance variable in the `index` action of the Users controller (Listing 10.19).

Listing 10.19. The user `index` action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_filter :authenticate, :only => [:index, :edit, :update]
  .
  .
  .
  def index
    @title = "All users"
    @users = User.all
  end
  .
  .
  .
end
```

Note that we have added `:index` to the list of controllers protected by the `authenticate` before filter, thereby getting the first test from Listing 10.18 to pass.

To make the actual 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 10.20). Note that each user's name is both linked to and escaped with `link_to h(user.name), user`.[9]

Listing 10.20. The user index view.
`app/views/users/index.html.erb`

```
<h2>All users</h2>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, :size => 30 %>
      <%= link_to h(user.name), user %>
    </li>
  <% end %>
```

```
</ul>
```

We’ll then add a little CSS for style (Listing 10.21).

Listing 10.21. CSS for the user index.
`public/stylesheets/custom.css`

```
.
.
.
ul.users {
  margin-top: 1em;
}

.users li {
  list-style: none;
}
```

Finally, we’ll add a “Users” link to the site’s navigation header (Listing 10.22). This puts to use the `users_path` named route from Table 6.2.

Listing 10.22. A layout link to the user index.
`app/views/layouts/_header.html.erb`

```
<div id="header" class="round">
  <%= link_to logo, root_path %>
  <ul class="navigation round">
    <li><%= link_to "Home", root_path %></li>
    <% if signed_in? %>
    <li><%= link_to "Users", users_path %></li>
    <li><%= link_to "Profile", current_user %></li>
    <li><%= link_to "Settings", edit_user_path(current_user) %></li>
    <% end %>
    .
    .
    .
  </ul>
</div>
```

With that, the user index is fully functional (with all tests passing), but it is a bit… lonely (Figure 10.8).

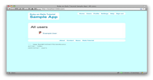


Figure 10.8: The user index page `/users` with only one user. (full size)

10.3.2 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 install the Faker gem, which will allow us to make sample users with semi-realistic names and email addresses:

```
$ [sudo] gem install faker -v 0.3.1
```

Next, we'll add a Rake task to create sample users. Rake tasks live in `lib/tasks`, and are defined using *namespaces* (in this case, `:db`), as seen in Listing 10.23.

Listing 10.23. A Rake task for populating the database with sample users.
`lib/tasks/sample_data.rake`

```
require 'faker'

namespace :db do
  desc "Fill database with sample data"
  task :populate => :environment do
    Rake::Task['db:reset'].invoke
    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 resets the development database using `db:reset` (using slightly weird syntax you shouldn't worry about too much), 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!`).

With the `:db` namespace as in Listing 10.23, we can invoke the Rake task as follows:

```
$ rake db:populate
```

After running the Rake task, our application has 100 sample users, as seen in Figure 10.9. (I've taken the liberty of associating the first few sample addresses with photos so that they're not all the default Gravatar image.)

Figure 10.9: The user index page `/users` with 100 sample users. (full size)

10.3.3 Pagination

Having solved the problem of too few sample users, we now encounter the opposite problem: having too many users on a 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`. It comes as a gem, which means we could install it using `gem install`; on the other hand, unlike previous gems such as RSpec and Factory Girl, `will_paginate` is needed in the application itself, not just the tests. There is a convenient way to indicate this requirement, and include the gem at the same time, using `config.gem` in the `environment.rb` file (Listing 10.24).10

Listing 10.24. Code to include the `will_paginate` gem.
`config/environment.rb`

```
.
.
.
Rails::Initializer.run do |config|
  .
  .
  .
  config.gem 'will_paginate', :version => '2.3.12'
end
```

With the configuration line in Listing 10.24, we can install the required gem dependencies as follows:

```
$ [sudo] rake gems:install
```

If you're also deploying the sample application to Heroku, you'll need to include `will_paginate` in the *gems manifest*, which involves creating a file called `.gems` in the Rails root.11 Navigate to the Rails root (if you're not already there) and make a new `.gems` file using your favorite text editor:

```
$ cd ~/rails_projects/sample_app
$ mate .gems
```

Then fill the `.gems` with contents of Listing 10.25.

Listing 10.25. The Heroku gems manifest.
`.gems`

```
will_paginate --version 2.3.12
faker
```

(Here we've included the `faker` gem from Section 10.3.2 since otherwise Heroku complains when running `heroku rake db:migrate`. The other gems in our application, such as Factory Girl and Webrat, are only needed for tests and so are not needed on the Heroku server.)

With `will_paginate` installed, we are now ready to paginate the results of finding users. We'll start by adding the special `will_paginate` method in the view (Listing 10.26); we'll see in a moment why the code appears both above and below the user list.

Listing 10.26. The user index with pagination.
`app/views/users/index.html.erb`

```
<h2>All users</h2>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, :size => 30 %>
      <%= link_to h(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 10.26 doesn't work yet, though, because currently `@users` contains the results of `User.all` (Listing 10.19), which is of class Array, whereas `will_paginate` expects an object of class `WillPaginate::Collection`. Happily, this is just the kind of object returned by the `paginate` method supplied by the `will_paginate` gem:

```
$ script/console
>> User.all.class
=> Array
>> User.paginate(:page => 1).class
=> WillPaginate::Collection
```

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.

We can paginate the users in the sample application by using `paginate` in place of `all` in the `index` action (Listing 10.27). Here the `:page` parameter comes from `params[:page]`, which is generated automatically by `will_paginate`.

Listing 10.27. Paginating the users in the `index` action.
`app/controllers/users_controller.rb`

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

The user index page should now be working, appearing as in Figure 10.10; because we included `will_paginate` both above and below the user list, the pagination links appear in both places. The tests, on the other hand, have now broken: by calling `User.paginate`, we have violated the expectation from Listing 10.18 that `User.all` should be called in the `index` action. We'll fix this problem next.

Figure 10.10: 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 10.11.

Figure 10.11: Page 2 of the user index (`/users?page=2`). (full size)

Testing pagination

Testing pagination requires detailed knowledge of how `will_paginate` works, so we did the implementation first, but it's still a good idea to test it. To do this, we need to invoke pagination in a test, which means making more than 30 (factory) users.

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. Fortunately, Factory Girl anticipates this issue, and provides *sequences* to solve it, as shown in Listing 10.28.

Listing 10.28. Defining a Factory Girl sequence.
`spec/factories.rb`

```
Factory.define :user do |user|
  user.name                  "Michael Hartl"
  user.email                 "mhartl@example.com"
  user.password              "foobar"
  user.password_confirmation "foobar"
end

Factory.sequence :email do |n|
  "person-#{n}@example.com"
end
```

This arranges to return email addresses like `person-1@example.com`, `person-2@example.com`, etc., which we invoke using the `next` method:

```
Factory(:user, :email => Factory.next(:email))
```

Applying the idea of factory sequences, we can make 31 users (the original `@user` plus 30 more) inside a test, and then verify that the response has the HTML expected from `will_paginate` (which you should be able to determine using Firebug or by viewing the page source). The result appears in Listing 10.29.

Listing 10.29. A test for pagination.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe "UsersController" do
  integrate_views

  describe "GET 'index'" do
    .
    .
    .
    describe "for signed-in users" do

      before(:each) do
        .
        .
        .
        @users = [@user, second, third]
        30.times do
          @users << Factory(:user, :email => Factory.next(:email))
        end
        User.should_receive(:paginate).and_return(@users.paginate)
      end
      .
      .
      .
      it "should have an element for each user" do
        get :index
        @users[0..2].each do |user|
          response.should have_tag("li", user.name)
        end
      end

      it "should paginate users" do
        get :index
        response.should have_tag("div.pagination")
        response.should have_tag("span", "&laquo; Previous")
        response.should have_tag("span", "1")
        response.should have_tag("a[href=?]", "/users?page=2", "2")
        response.should have_tag("a[href=?]", "/users?page=2", "Next &raquo;")
      end
    end
  end
  .
  .
  .
end
```

This code ensures that the tests invoke pagination by adding 30[12] users to the `@users` variable using the `Array` push notation `<<`, which appends an element to an existing array:

```
$ script/console
>> a = [1, 2, 5]
=> [1, 2, 5]
>> a << 17
=> [1, 2, 5, 17]
>> a << 42 << 1337
=> [1, 2, 5, 17, 42, 1337]
```

We see from the last example that occurrences of `<<` can be chained.

To get the message expectation to pass, Listing 10.29 changes

```
User.should_receive(:all).and_return(@users)
```

with

```
User.should_receive(:pagination).and_return(@users.paginate)
```

This shows that we can convert an `Array` object into a `WillPaginate::Collection` object using `@users.paginate`.

Finally, in the test itself, note the compact notation `have_tag("div.pagination")`, which borrows the class convention from CSS (first seen in Listing 5.3) to check for a `div` tag with class `pagination`. Also note that, since there are now 33 users, we've updated the user element test to use only the first three elements (`[0..2]`) of the `@users` array, which is what we had before in Listing 10.18:

```
@users[0..2].each do |user|
  response.should have_tag("li", :content => user.name)
end
```

With that, our pagination code is well-tested, and there's only one minor detail left.

10.3.4 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 10.26 with a `render` call (Listing 10.30).

Listing 10.30. The first refactoring attempt at the index view.
`app/views/users/index.html.erb`

```
<h2>All users</h2>

<%= 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`;13 in this context, Rails automatically looks for a partial called `_user.html.erb`, which we must create (Listing 10.31).

Listing 10.31. A partial to render a single user.
`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, :size => 30 %>
  <%= link_to h(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 10.32).

Listing 10.32. The fully refactored user index.
`app/views/users/index.html.erb`

```
<h2>All users</h2>

<%= will_paginate %>

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

<%= will_paginate %>
```

Here Rails infers that `@users` is an array 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 10.32.

10.4 Destroying 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 10.12, and define the `destroy` action necessary to accomplish the deletion. But first, we'll create the class of administrative users authorized to do so.

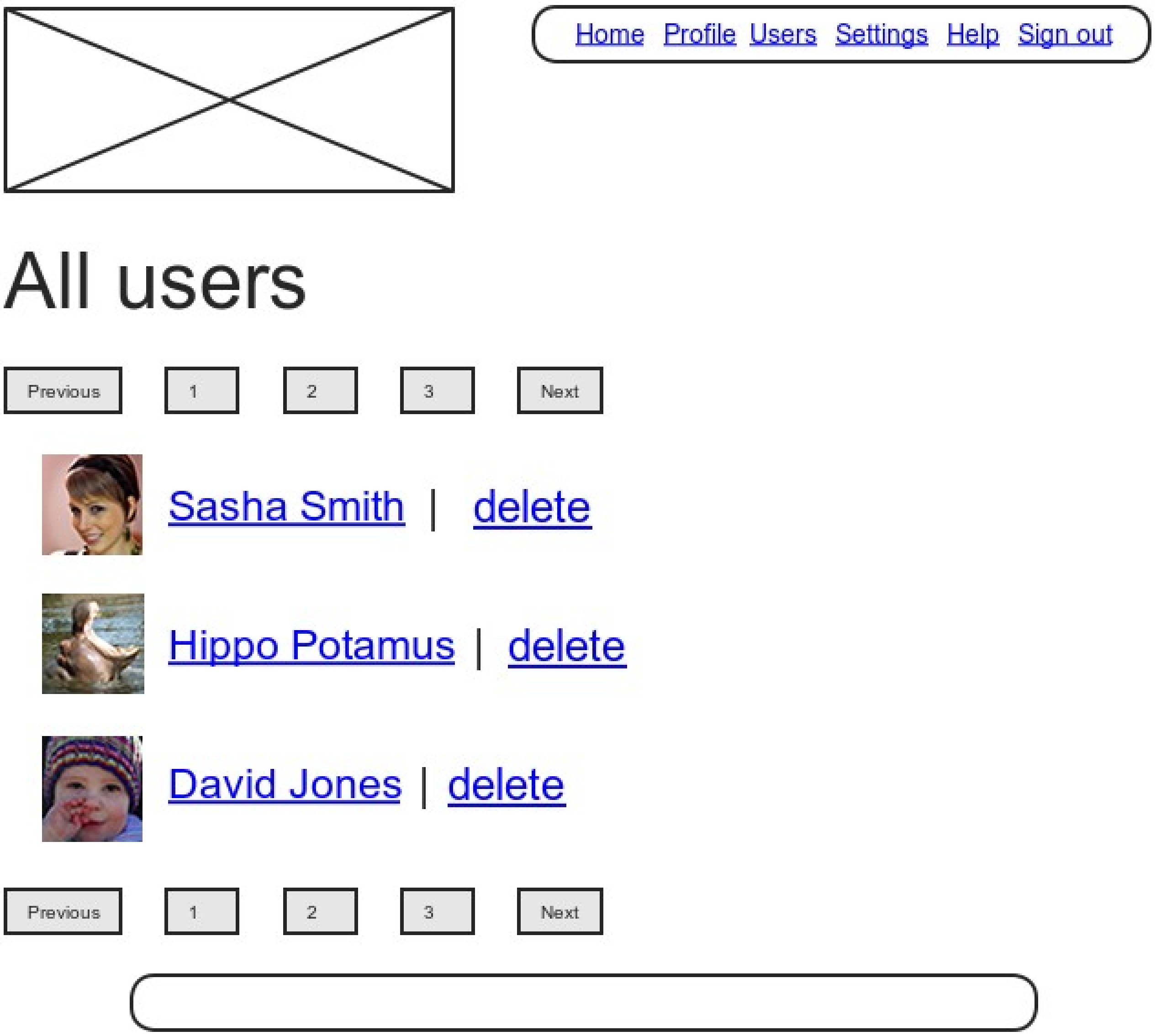


Figure 10.12: A mockup of the user index with delete links. (full size)

10.4.1 Administrative users

We will identify privileged administrative users with a boolean `admin` attribute in the User model, which will lead to an `admin?` method to test for admin status. Following the model of the tests for “remember me” in Listing 9.13, we can write tests for this attribute as in Listing 10.33.

Listing 10.33. Tests for an `admin` attribute.
`spec/models/user_spec.rb`

```
.
.
.
  describe "admin attribute" do
```

```
    before(:each) do
      @user = User.create!(@attr)
    end

    it "should respond to admin" do
      @user.should respond_to(:admin)
    end

    it "should not be an admin by default" do
      @user.should_not be_admin
    end

    it "should be convertible to an admin" do
      @user.toggle!(:admin)
      @user.should be_admin
    end
  end
end
```

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

```
@user.should be_admin
```

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

We add the `admin` attribute with a migration as usual, indicating the `boolean` type on the command line:

```
$ script/generate migration add_admin_to_users admin:boolean
```

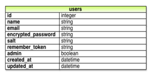
The migration simply adds the `admin` column to the `users` table (Listing 10.34), yielding the data model in Figure 10.13.

Listing 10.34. The migration to add a boolean `admin` attribute to users.
`db/migrate/<timestamp>_add_admin_to_users.rb`

```
class AddAdminToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :admin, :boolean, :default => false
  end

  def self.down
    remove_column :users, :admin
  end
end
```

Note that we've added the argument `:default => false` to `add_column` in Listing 10.34, 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
encrypted_password	string
salt	string
remember_token	string
admin	boolean
created_at	datetime
updated_at	datetime

Figure 10.13: The User model with an added `admin` boolean attribute.

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

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

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

```
$ script/console
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

As a final step, let's update our sample data populator to make the first user an admin (Listing 10.35).

Listing 10.35. The sample data populator code with an admin user.
`lib/tasks/sample_data.rake`

```
require 'faker'

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

Finally, re-run the populator to reset the database and then rebuild it from scratch:

```
$ rake db:populate
```

Revisiting `attr_accessible`

You might have noticed that Listing 10.35 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, and the `admin` attribute isn't accessible. Listing 10.36 reproduces the most recent list of `attr_accessible` attributes—note that `:admin` is *not* on the list.

Listing 10.36. A review of the `attr_accessible` attributes in the User model.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessor :password
  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:14

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

This request would make user 17 an admin, which could 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.

10.4.2 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 (Listing 10.37).

Listing 10.37. User delete links (viewable only by admins).
`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, :size => 30 %>
  <%= link_to h(user.name), user %>
  <% if current_user.admin? %>
  | <%= link_to "delete", user, :method => :delete, :confirm => "You sure?" %>
  <% end %>
</li>
```

Note the `:method => :delete` argument, which arranges for the link to issue the necessary `DELETE` request;15 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 10.14.

Figure 10.14: The user index `/users` with delete links. (full size)

Even though only admins can see the delete links, there's still a terrible security breach: any sufficiently sophisticated attacker could simply issue `DELETE` requests from the command line and delete any user on the site. To secure the site properly, we also need access control, so our tests should check not only that admins *can* delete users, but also that other users *can't*. The results appear in Listing 10.38. Note that, in analogy with the `get`, `post`, and `put` methods, we use `delete` to issue `DELETE` requests inside of tests.

Listing 10.38. Tests for destroying users.
`spec/controllers/users_controller_spec.rb`

```
describe UsersController do
  integrate_views
  .
  .
  .
  describe "DELETE 'destroy'" do

    before(:each) do
      @user = Factory(:user)
    end

    describe "as a non-signed-in user" do
      it "should deny access" do
        delete :destroy, :id => @user
        response.should redirect_to(signin_path)
      end
    end

    describe "as a non-admin user" do
      it "should protect the page" do
        test_sign_in(@user)
        delete :destroy, :id => @user
        response.should redirect_to(root_path)
      end
    end

    describe "as an admin user" do

      before(:each) do
        admin = Factory(:user, :email => "admin@example.com", :admin => true)
        test_sign_in(admin)
        User.should_receive(:find).with(@user).and_return(@user)
        @user.should_receive(:destroy).and_return(@user)
      end

      it "should destroy the user" do
        delete :destroy, :id => @user
        response.should redirect_to(users_path)
      end
    end
  end
end
```

(You might notice that we've set an admin user using `:admin => true`; user factories are not bound by the rules of `attr_accessible` parameters.)

As you might suspect by now, the implementation uses a before filter, this time to restrict access to the `destroy` action to admins. The `destroy` action itself finds the user, destroys it, and then redirects to user index (Listing 10.39).

Listing 10.39. A before filter restricting the `destroy` action to admins.
`app/controllers/users_controller.rb`

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

  private
    .
    .
    .
    def admin_user
      redirect_to(root_path) unless current_user.admin?
    end
end
```

Note that the `destroy` action uses *method chaining* (seen briefly in Section 4.2.3) in the line

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

which saves a line of code.

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

10.5 Conclusion

We've come a long way since introducing the Users controller way back in Section 5.3. 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 authentication system) to make a site with Twitter-like microposts (Chapter 11) and user following (Chapter 12). 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 -am "Done with user edit/update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
```

10.6 Exercises

1. Arrange for the Gravatar “change” link in Listing 10.3 to open in a new window (or tab). *Hint:* Search the web; you should find one particularly robust method involving something called `_blank`. This solution is not valid HTML (so if you installed the HTML Validator in Chapter 1 it will indicate an error), but it works in every browser I’ve ever seen.
2. Remove the duplicated form code by refactoring the `new.html.erb` and `edit.html.erb` views to use the partial in Listing 10.40. Note that you will have to pass the form variable `f` explicitly as a local variable, as shown in Listing 10.41.
3. 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.
4. Add tests to check that the delete links in Listing 10.37 appear for admins but not for normal users.
5. Modify the `destroy` action to prevent admin users from destroying themselves. (Write a test first.)

Listing 10.40. A partial for the new and edit form fields.
`app/views/users/_fields.html.erb`

```
<%= f.error_messages %>
<div class="field">
  <%= f.label :name %><br />
  <%= f.text_field :name %>
</div>
<div class="field">
  <%= f.label :email %><br />
  <%= f.text_field :email %>
</div>
<div class="field">
  <%= f.label :password %><br />
  <%= f.password_field :password %>
</div>
<div class="field">
  <%= f.label :password_confirmation, "Confirmation" %><br />
  <%= f.password_field :password_confirmation %>
</div>
```

Listing 10.41. The new user view with partial.
`app/views/users/new.html.erb`

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

<% form_for(@user) do |f| %>
  <%= render :partial => 'fields', :locals => { :f => f } %>
  <div class="actions">
    <%= f.submit "Sign up" %>
  </div>
<% end %>
```

« Chapter 9 Sign in, sign out Chapter 11 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. Image from http://www.flickr.com/photos/sashawolff/4598355045/. ↑
4. Don't be worried about how this works; the details are of interest to developers of the Rails framework itself, but by design are not important for Rails application developers. ↑
5. To be fair, they would need the user's password, but if we ever made the password unnecessary (as planned for the screencasts) it would open up a *huge* security hole. ↑
6. The code in this section is adapted from the Clearance gem by thoughtbot. ↑
7. Indeed, as noted in Section 9.6, `session` is implemented in just this way. ↑
8. Baby photo from http://www.flickr.com/photos/glasgows/338937124/. ↑
9. Note that Ruby allows the omission of *one* pair of parentheses, but not two, so instead of `link_to h user.name` we write `link_to h(user.name)`. ↑
10. In Rails 3, the `config.gem` method for including gems will be replaced by Bundler. ↑
11. The Heroku gems manifest will also be replaced (or, rather, supplemented) by Bundler in Rails 3. ↑
12. Technically, we only need to create 28 additional factory users since we already have three, but I find the meaning clearer if we create 30 instead. ↑
13. 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. ↑
14. Command-line tools such as `curl` (seen in Box 3.1) can issue `PUT` requests of this form. ↑
15. 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. For the purposes of this tutorial, we can just require our site admins to enable JavaScript, but 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. ↑