```
<%= 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 (inferring the name of the partial from the name of the class). The result is the impressively compact code in Listing 10.52.

As with any refactoring, you should verify that the test suite is still GREEN after changing the application code:

Listing 10.53: GREEN

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Comment out the **`render`** line in Listing 10.52 and confirm that the resulting tests are RED.

10.4 Deleting users

Now that the users 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 10.13, and define the **`destroy`** action necessary to accomplish the

deletion. But first, we'll create the class of administrative users, or *admins*, authorized to do so. In the context of authorization, such a set of special privileges is known as a *role*.

10.4.1 Administrative users

We will identify privileged administrative users with a boolean **admin** attribute in the User model, which will lead automatically to an **admin?** boolean method to test for admin status. The resulting data model appears in Figure 10.14.

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 adds the **admin** column to the **users** table, as shown in Listing 10.54. Note that we've added the argument **default: false** to **add_column** in Listing 10.54, 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.)

Listing 10.54: The migration to add a boolean **admin** attribute to users.
db/migrate/[timestamp]_add_admin_to_users.rb

```
class AddAdminToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

Next, we migrate as usual:

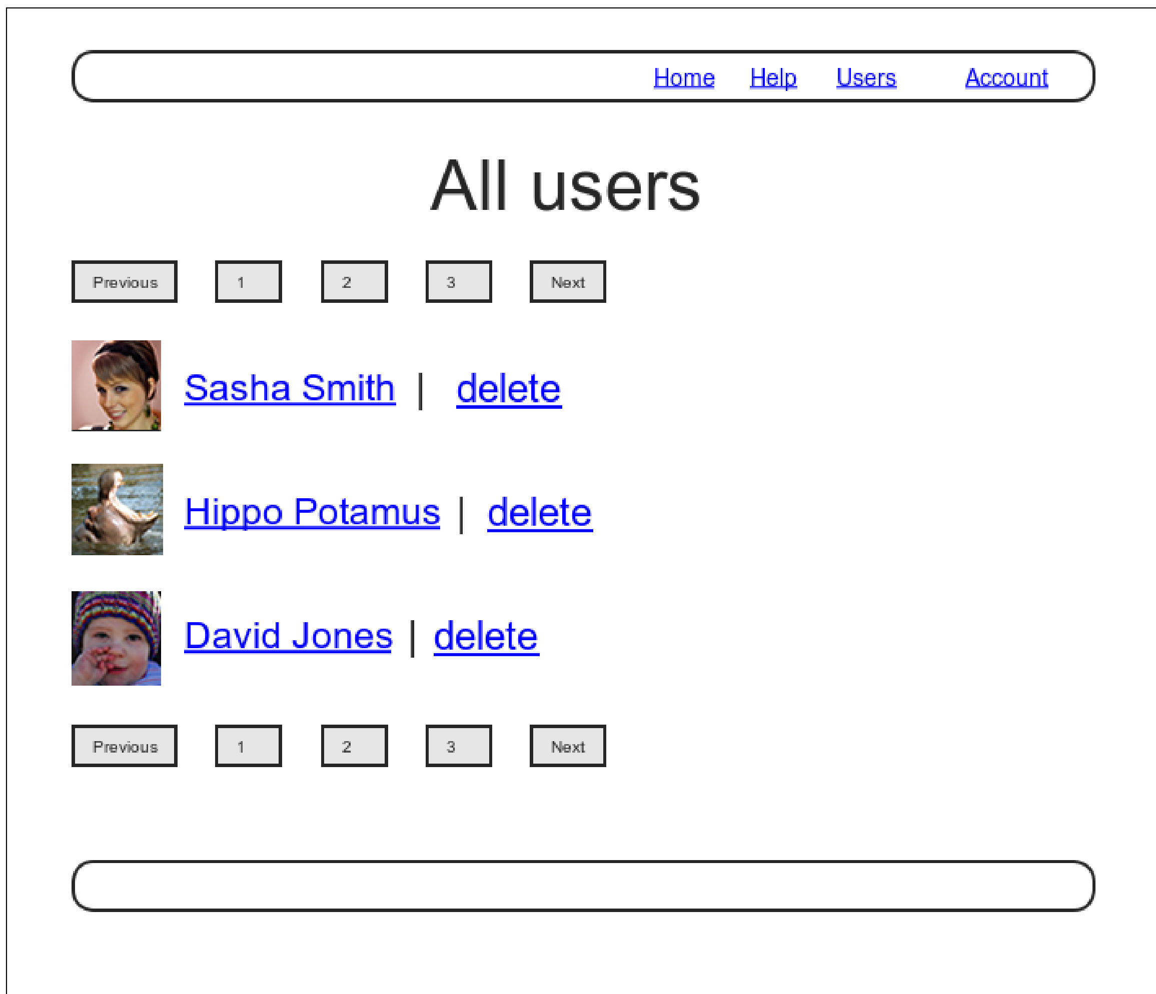

Figure 10.13: A mockup of the users index with delete links.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string
admin	boolean

Figure 10.14: The User model with an added **admin** boolean attribute.

```
$ rails db:migrate
```

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
```

Here we've used the **toggle!** method to flip the **admin** attribute from **false** to **true**.

As a final step, let's update our seed data to make the first user an admin by default (Listing 10.55).

Listing 10.55: The seed data code with an admin user.
db/seeds.rb

```
# Create a main sample user.
User.create!(name:  "Example User",
             email: "example@railstutorial.org",
             password:              "foobar",
             password_confirmation: "foobar",
             admin: true)

# Generate a bunch of additional users.
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
```

Then reset and reseed the database:

```
$ rails db:migrate:reset
$ rails db:seed
```

Revisiting strong parameters

You might have noticed that Listing 10.55 makes the user an admin by including **`admin: true`** in the initialization hash. This underscores the danger of exposing our objects to the wild Web—if we simply passed an initialization hash in from an arbitrary web request, a malicious user could send a `PATCH` request as follows:[14]

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

This request would make user 17 an admin, which would be a potentially serious security breach.

[14]Command-line tools such as `curl` can issue `PATCH` requests of this form.

Because of this danger, it is essential that we only update attributes that are safe to edit through the web. As noted in Section 7.3.2, this is accomplished using *strong parameters* by calling **require** and **permit** on the **params** hash:

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

Note in particular that **admin** is *not* in the list of permitted attributes. This is what prevents arbitrary users from granting themselves administrative access to our application. Because of its importance, it's a good idea to write a test for any attribute that isn't editable, and writing such a test for the **admin** attribute is left as an exercise (Section 10.4.1).

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. By issuing a PATCH request directly to the user path as shown in Listing 10.56, verify that the **admin** attribute isn't editable through the web. To be sure your test is covering the right thing, your first step should be to *add* **admin** to the list of permitted parameters in **user_params** so that the initial test is RED. For the final line, make sure to load the updated user information from the database (Section 6.1.5).

Listing 10.56: Testing that the **admin** attribute is forbidden.
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user       = users(:michael)
```

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

  test "should not allow the admin attribute to be edited via the web" do
    log_in_as(@other_user)
    assert_not @other_user.admin?
    patch user_path(@other_user), params: {
                                    user: { password:              "password",
                                            password_confirmation: "password",
                                            admin: FILL_IN } }
    assert_not @other_user.FILL_IN.admin?
  end
  .
  .
  .
end
```

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 users index page, restricting access to administrative users. The resulting **`"delete"`** links will be displayed only if the current user is an admin (Listing 10.57).

Listing 10.57: User delete links (viewable only by admins).
app/views/users/_user.html.erb

```
<li>
  <%= gravatar_for user, size: 50 %>
  <%= link_to user.name, user %>
  <% if current_user.admin? && !current_user?(user) %>
    | <%= link_to "delete", user, method: :delete,
                                  data: { confirm: "You sure?" } %>
  <% end %>
</li>
```

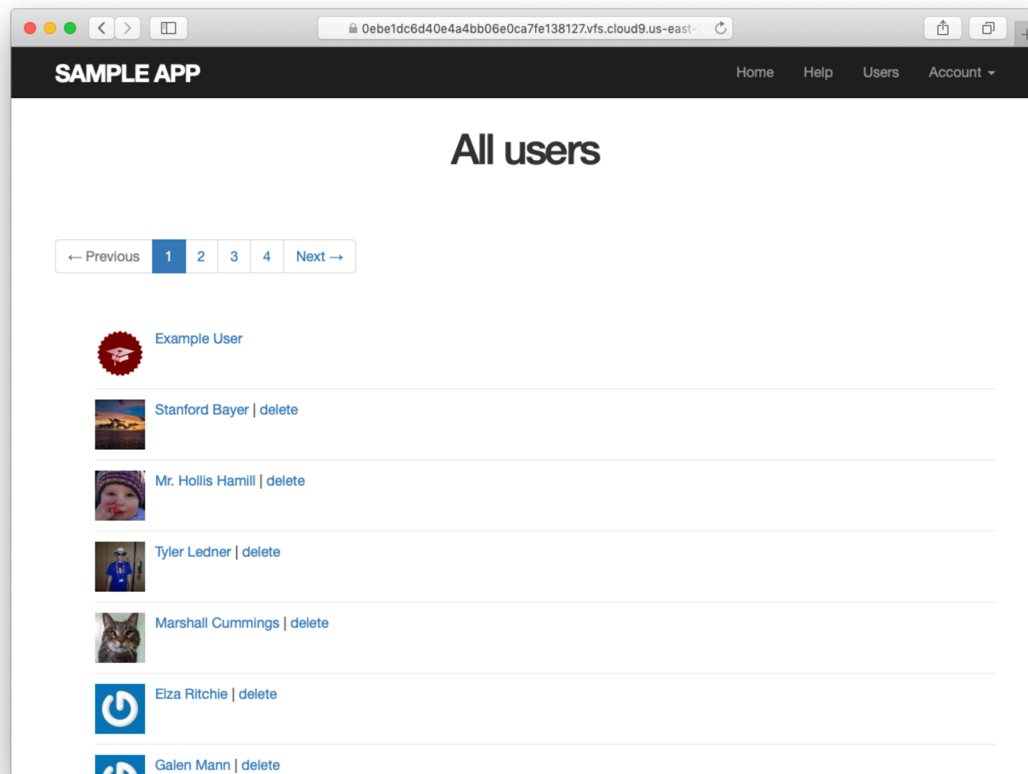

Figure 10.15: The users index with delete links.

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 10.15.

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. 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.[15]

[15] See the RailsCast on "Destroy Without JavaScript" for details.

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 users index, as seen in Listing 10.58. Because users have to be logged in to delete users, Listing 10.58 also adds **:destroy** to the **logged_in_user** before filter.

Listing 10.58: Adding a working **destroy** action.
app/controllers/users_controller.rb

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

  private
  .
  .
  .
end
```

As constructed, only admins can destroy users through the web since only they can see the delete links, but there's still a terrible security hole: any sufficiently sophisticated attacker could simply issue a DELETE request 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 that *only* admins can delete users.

As in Section 10.2.1 and Section 10.2.2, we'll enforce access control using a before filter, this time to restrict access to the **destroy** action to admins. The resulting **admin_user** before filter appears in Listing 10.59.

Listing 10.59: A before filter restricting the **destroy** action to admins.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update, :destroy]
  before_action :correct_user,   only: [:edit, :update]
  before_action :admin_user,     only: :destroy
  .
  .
  .
  private
    .
    .
    .
    # Confirms an admin user.
    def admin_user
      redirect_to(root_url) unless current_user.admin?
    end
end
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. As the admin user, destroy a few sample users through the web interface. What are the corresponding entries in the server log?

10.4.3 User destroy tests

With something as dangerous as destroying users, it's important to have good tests for the expected behavior. We start by arranging for one of our fixture users to be an admin, as shown in Listing 10.60.

Listing 10.60: Making one of the fixture users an admin.
test/fixtures/users.yml

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

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

lana:
  name: Lana Kane
  email: hands@example.gov
  password_digest: <%= User.digest('password') %>

malory:
  name: Malory Archer
  email: boss@example.gov
  password_digest: <%= User.digest('password') %>

<% 30.times do |n| %>
user_<%= n %>:
  name:  <%= "User #{n}" %>
  email: <%= "user-#{n}@example.com" %>
  password_digest: <%= User.digest('password') %>
<% end %>
```

Following the practice from Section 10.2.1, we'll put action-level tests of access control in the Users controller test file. As with the logout test in Listing 8.35, we'll use **`delete`** to issue a `DELETE` request directly to the **`destroy`** action. We need to check two cases: first, users who aren't logged in should be redirected to the login page; second, users who are logged in but who aren't admins should be redirected to the Home page. The result appears in Listing 10.61.

Listing 10.61: Action-level tests for admin access control. GREEN
`test/controllers/users_controller_test.rb`

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

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

```
  test "should redirect destroy when not logged in" do
    assert_no_difference 'User.count' do
      delete user_path(@user)
    end
    assert_redirected_to login_url
  end

  test "should redirect destroy when logged in as a non-admin" do
    log_in_as(@other_user)
    assert_no_difference 'User.count' do
      delete user_path(@user)
    end
    assert_redirected_to root_url
  end
end
```

Note that Listing 10.61 also makes sure that the user count doesn't change using the **assert_no_difference** method (seen before in Listing 7.23).

The tests in Listing 10.61 verify the behavior in the case of an unauthorized (non-admin) user, but we also want to check that an admin can use a delete link to successfully destroy a user. Since the delete links appear on the users index, we'll add these tests to the users index test from Listing 10.48. The only really tricky part is verifying that a user gets deleted when an admin clicks on a delete link, which we'll accomplish as follows:

```
assert_difference 'User.count', -1 do
  delete user_path(@other_user)
end
```

This uses the **assert_difference** method first seen in Listing 7.31 when creating a user, this time verifying that a user is *destroyed* by checking that **User.count** changes by -1 when issuing a **delete** request to the corresponding user path.

Putting everything together gives the pagination and delete test in Listing 10.62, which includes tests for both admins and non-admins.

Listing 10.62: An integration test for delete links and destroying users. GREEN
test/integration/users_index_test.rb

```
require 'test_helper'

class UsersIndexTest < ActionDispatch::IntegrationTest

  def setup
    @admin     = users(:michael)
    @non_admin = users(:archer)
  end

  test "index as admin including pagination and delete links" do
    log_in_as(@admin)
    get users_path
    assert_template 'users/index'
    assert_select 'div.pagination'
    first_page_of_users = User.paginate(page: 1)
    first_page_of_users.each do |user|
      assert_select 'a[href=?]', user_path(user), text: user.name
      unless user == @admin
        assert_select 'a[href=?]', user_path(user), text: 'delete'
      end
    end
    assert_difference 'User.count', -1 do
      delete user_path(@non_admin)
    end
  end

  test "index as non-admin" do
    log_in_as(@non_admin)
    get users_path
    assert_select 'a', text: 'delete', count: 0
  end
end
```

Note that Listing 10.62 checks for the right delete links, including skipping the test if the user happens to be the admin (which lacks a delete link due to Listing 10.57).

At this point, our deletion code is well-tested, and the test suite should be GREEN:

Listing 10.63: GREEN

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. By commenting out the admin user before filter in Listing 10.59, confirm that the tests go RED.

10.5 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, log in, log out, view their profiles, edit their settings, and see an index of all users—and some can even destroy other users.

As it presently stands, the sample application forms a solid foundation for any website requiring users with authentication and authorization. In Chapter 11 and Chapter 12, we'll add two additional refinements: an account activation link for newly registered users (verifying a valid email address in the process) and password resets to help users who forget their passwords.

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

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

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):

```
$ rails test
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rails db:migrate
$ heroku run rails db:seed
```