14.2 A web interface for following users

Section 14.1 placed some rather heavy demands on our data modeling skills, and it's fine if it takes a while to soak in. In fact, one of the best ways to understand the associations is to use them in the web interface.

In the introduction to this chapter, we saw a preview of the page flow for user following. In this section, we will implement the basic interface and following/unfollowing functionality shown in those mockups. We will also make separate pages to show the user following and followers arrays. In Section 14.3, we'll complete our sample application by adding the user's status feed.

14.2.1 Sample following data

As in previous chapters, we will find it convenient to use **`rails db:seed`** to fill the database with sample relationships. This will allow us to design the look and feel of the web pages first, deferring the back-end functionality until later in this section.

Code to seed the following relationships appears in Listing 14.14. Here we somewhat arbitrarily arrange for the first user to follow users 3 through 51, and then have users 4 through 41 follow that user back. The resulting relationships will be sufficient for developing the application interface.

Listing 14.14: Adding following/follower relationships to the sample data.
db/seeds.rb

```
# Users
User.create!(name:  "Example User",
             email: "example@railstutorial.org",
             password:              "foobar",
             password_confirmation: "foobar",
             admin:     true,
             activated: true,
             activated_at: Time.zone.now)

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,
               activated: true,
               activated_at: Time.zone.now)
end

# Microposts
users = User.order(:created_at).take(6)
50.times do
  content = Faker::Lorem.sentence(5)
  users.each { |user| user.microposts.create!(content: content) }
end

# Create following relationships.
users = User.all
user  = users.first
following = users[2..50]
followers = users[3..40]
following.each { |followed| user.follow(followed) }
followers.each { |follower| follower.follow(user) }
```

To execute the code in Listing 14.14, we reset and reseed the database as usual:

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

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. Using the console, confirm that `User.first.followers.count` matches the value expected from Listing 14.14.

2. Confirm that `User.first.following.count` is correct as well.

Figure 14.10: A mockup of the stats partial.

14.2.2 Stats and a follow form

Now that our sample users have both followed users and followers, we need to update the profile page and Home page to reflect this. We'll start by making a partial to display the following and follower statistics on the profile and home pages. We'll next add a follow/unfollow form, and then make dedicated pages for showing "following" (followed users) and "followers".

As noted in Section 14.1.1, we'll adopt Twitter's convention of using "following" as a label for followed users, as in "50 following". This usage is reflected in the mockup sequence starting in Figure 14.1 and shown in close-up in Figure 14.10.

The stats in Figure 14.10 consist of the number of users the current user is following and the number of followers, each of which should be a link to its respective dedicated display page. In Chapter 5, we stubbed out such links with the dummy text `'#'`, but that was before we had much experience with routes. This time, although we'll defer the actual pages to Section 14.2.3, we'll make the routes now, as seen in Listing 14.15. This code uses the **`:member`** method inside a **`resources`** *block*, which we haven't seen before, but see if you can guess what it does.

Listing 14.15: Adding **`following`** and **`followers`** actions to the Users controller.

config/routes.rb

```
Rails.application.routes.draw do
  root   'static_pages#home'
  get    '/help',    to: 'static_pages#help'
  get    '/about',   to: 'static_pages#about'
  get    '/contact', to: 'static_pages#contact'
  get    '/signup',  to: 'users#new'
```

```
  get    '/login',   to: 'sessions#new'
  post   '/login',   to: 'sessions#create'
  delete '/logout',  to: 'sessions#destroy'
  resources :users do
    member do
      get :following, :followers
    end
  end
  resources :account_activations, only: [:edit]
  resources :password_resets,     only: [:new, :create, :edit, :update]
  resources :microposts,          only: [:create, :destroy]
end
```

You might suspect that the URLs for following and followers will look like /users/1/following and /users/1/followers, and that is exactly what the code in Listing 14.15 arranges. Since both pages will be *showing* data, the proper HTTP verb is a `GET` request, so we use the **`get`** method to arrange for the URLs to respond appropriately. Meanwhile, the **`member`** method arranges for the routes to respond to URLs containing the user id. The other possibility, **`collection`**, works without the id, so that

```
resources :users do
  collection do
    get :tigers
  end
end
```

would respond to the URL /users/tigers (presumably to display all the tigers in our application).[8]

A table of the routes generated by Listing 14.15 appears in Table 14.2. Note the named routes for the followed user and followers pages, which we'll put to use shortly.

With the routes defined, we are now in a position to define the stats partial, which involves a couple of links inside a div, as shown in Listing 14.16.

[8] For more details on such routing options, see the Rails Guides article on "Rails Routing from the Outside In".

HTTP request	URL	Action	Named route
GET	/users/1/following	`following`	`following_user_path(1)`
GET	/users/1/followers	`followers`	`followers_user_path(1)`

Table 14.2: RESTful routes provided by the custom rules in resource in Listing 14.15.

Listing 14.16: A partial for displaying follower stats.
app/views/shared/_stats.html.erb

```
<% @user ||= current_user %>
<div class="stats">
  <a href="<%= following_user_path(@user) %>">
    <strong id="following" class="stat">
      <%= @user.following.count %>
    </strong>
    following
  </a>
  <a href="<%= followers_user_path(@user) %>">
    <strong id="followers" class="stat">
      <%= @user.followers.count %>
    </strong>
    followers
  </a>
</div>
```

Since we will be including the stats on both the user show pages and the home page, the first line of Listing 14.16 picks the right one using

```
<% @user ||= current_user %>
```

As discussed in Box 8.1, this does nothing when **`@user`** is not **`nil`** (as on a profile page), but when it is (as on the Home page) it sets **`@user`** to the current user. Note also that the following/follower counts are calculated through the associations using

```
@user.following.count
```

and

```
@user.followers.count
```

Compare these to the microposts count from Listing 13.24, where we wrote

```
@user.microposts.count
```

to count the microposts. As in that case, Rails calculates the count directly in the database for efficiency.

One final detail worth noting is the presence of CSS ids on some elements, as in

```
<strong id="following" class="stat">
...
</strong>
```

This is for the benefit of the Ajax implementation in Section 14.2.5, which accesses elements on the page using their unique ids.

With the partial in hand, including the stats on the Home page is easy, as shown in Listing 14.17.

Listing 14.17: Adding follower stats to the Home page.
app/views/static_pages/home.html.erb

```
<% if logged_in? %>
  <div class="row">
    <aside class="col-md-4">
      <section class="user_info">
        <%= render 'shared/user_info' %>
      </section>
      <section class="stats">
        <%= render 'shared/stats' %>
```

```
      </section>
      <section class="micropost_form">
        <%= render 'shared/micropost_form' %>
      </section>
    </aside>
    <div class="col-md-8">
      <h3>Micropost Feed</h3>
      <%= render 'shared/feed' %>
    </div>
  </div>
<% else %>
  .
  .
  .
<% end %>
```

To style the stats, we'll add some SCSS, as shown in Listing 14.18 (which contains all the stylesheet code needed in this chapter). The resulting Home page appears in Figure 14.11.

Listing 14.18: SCSS for the Home page sidebar.
app/assets/stylesheets/custom.scss

```
.
.
.
/* sidebar */
.
.
.
.gravatar {
  float: left;
  margin-right: 10px;
}

.gravatar_edit {
  margin-top: 15px;
}

.stats {
  overflow: auto;
  margin-top: 0;
  padding: 0;
  a {
    float: left;
    padding: 0 10px;
    border-left: 1px solid $gray-lighter;
```

```
    color: gray;
    &:first-child {
      padding-left: 0;
      border: 0;
    }
    &:hover {
      text-decoration: none;
      color: blue;
    }
  }
  strong {
    display: block;
  }
}

.user_avatars {
  overflow: auto;
  margin-top: 10px;
  .gravatar {
    margin: 1px 1px;
  }
  a {
    padding: 0;
  }
}

.users.follow {
  padding: 0;
}

/* forms */
.
.
.
```

We'll render the stats partial on the profile page in a moment, but first let's make a partial for the follow/unfollow button, as shown in Listing 14.19.

Listing 14.19: A partial for a follow/unfollow form.
app/views/users/_follow_form.html.erb

```
<% unless current_user?(@user) %>
  <div id="follow_form">
  <% if current_user.following?(@user) %>
    <%= render 'unfollow' %>
  <% else %>
    <%= render 'follow' %>
```

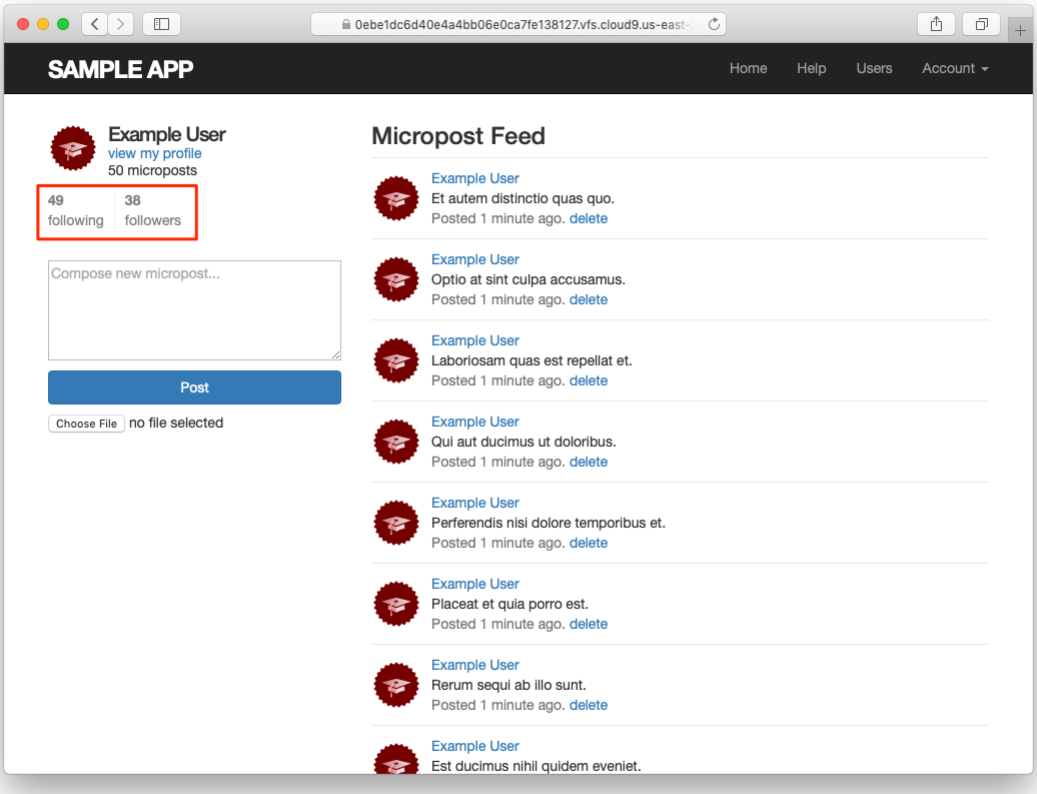

Figure 14.11: The Home page with follow stats.

```
  <% end %>
  </div>
<% end %>
```

This does nothing but defer the real work to **follow** and **unfollow** partials, which need new routes for the Relationships resource, which follows the Microposts resource example (Listing 13.30), as seen in Listing 14.20.

Listing 14.20: Adding the routes for user relationships.
config/routes.rb

```
Rails.application.routes.draw do
  root                'static_pages#home'
  get    'help'    => 'static_pages#help'
  get    'about'   => 'static_pages#about'
  get    'contact' => 'static_pages#contact'
  get    'signup'  => 'users#new'
  get    'login'   => 'sessions#new'
  post   'login'   => 'sessions#create'
  delete 'logout'  => 'sessions#destroy'
  resources :users do
    member do
      get :following, :followers
    end
  end
  resources :account_activations, only: [:edit]
  resources :password_resets,     only: [:new, :create, :edit, :update]
  resources :microposts,          only: [:create, :destroy]
  resources :relationships,       only: [:create, :destroy]
end
```

The follow/unfollow partials themselves are shown in Listing 14.21 and Listing 14.22.

Listing 14.21: A form for following a user.
app/views/users/_follow.html.erb

```
<%= form_with(model: current_user.active_relationships.build, local: true) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

Listing 14.22: A form for unfollowing a user.
app/views/users/_unfollow.html.erb

```
<%= form_with(model: current_user.active_relationships.find_by(followed_id: @user.id),
              html: { method: :delete }, local: true) do |f| %>
  <%= f.submit "Unfollow", class: "btn" %>
<% end %>
```

These two forms both use **form_with** to manipulate a Relationship model object; the main difference between the two is that Listing 14.21 builds a *new* relationship, whereas Listing 14.22 finds the existing relationship. Naturally, the former sends a `POST` request to the Relationships controller to **create** a relationship, while the latter sends a `DELETE` request to **destroy** a relationship. (We'll write these actions in Section 14.2.4.) Finally, you'll note that the follow form doesn't have any content other than the button, but it still needs to send the **followed_id** to the controller. We accomplish this with the **hidden_field_tag** method in Listing 14.21, which produces HTML of the form

```
<input id="followed_id" name="followed_id" type="hidden" value="3" />
```

As we saw in Section 12.3 (Listing 12.14), the hidden **input** tag puts the relevant information on the page without displaying it in the browser.

We can now include the follow form and the following statistics on the user profile page simply by rendering the partials, as shown in Listing 14.23. Profiles with follow and unfollow buttons, respectively, appear in Figure 14.12 and Figure 14.13.

Listing 14.23: Adding the follow form and follower stats to the user profile page.
app/views/users/show.html.erb

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="col-md-4">
    <section>
      <h1>
```

```
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
    <section class="stats">
      <%= render 'shared/stats' %>
    </section>
  </aside>
  <div class="col-md-8">
    <%= render 'follow_form' if logged_in? %>
    <% if @user.microposts.any? %>
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
  </div>
</div>
```

We'll get these buttons working soon enough—in fact, we'll do it two ways, the standard way (Section 14.2.4) and using Ajax (Section 14.2.5)—but first we'll finish the HTML interface by making the following and followers pages.

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. Verify that /users/2 has a follow form and that /users/5 has an unfollow form. Is there a follow form on /users/1?

2. Confirm in the browser that the stats appear correctly on the Home page and on the profile page.

3. Write tests for the stats on the Home page. *Hint*: Add to the test in Listing 13.28. Why don't we also have to test the stats on the profile page?

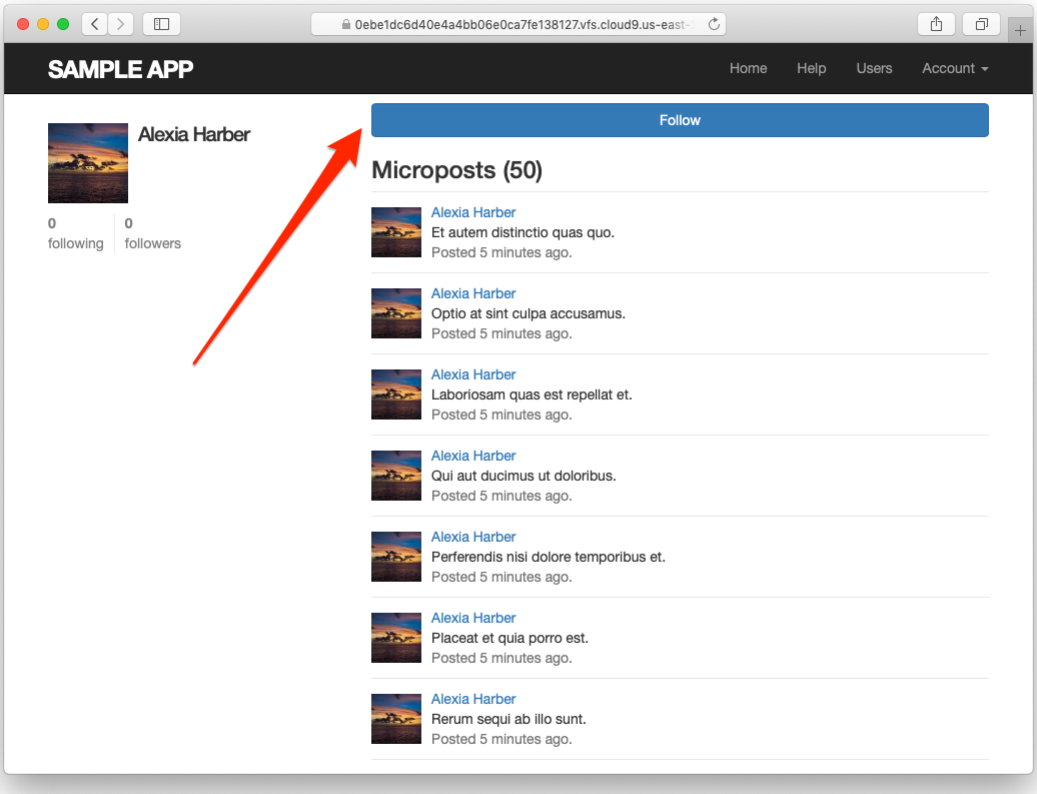

Figure 14.12: A user profile with a follow button (/users/2).

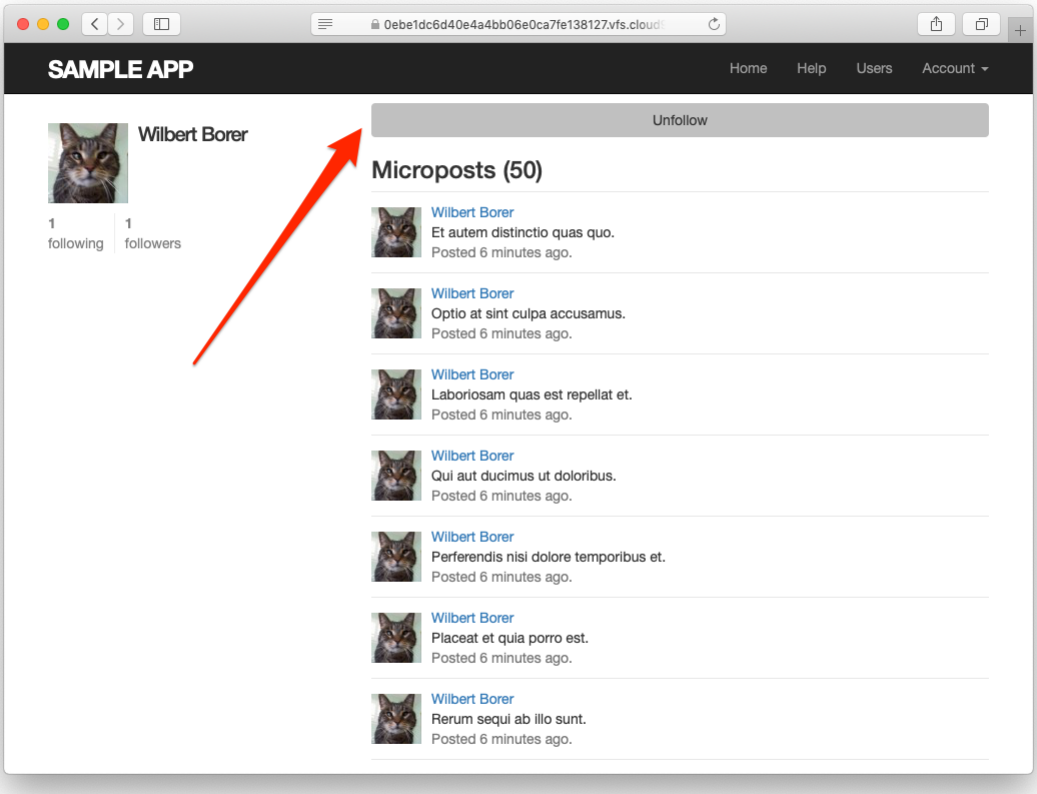

Figure 14.13: A user profile with an unfollow button (/users/5).

14.2.3 Following and followers pages

Pages to display followed users and followers will resemble a hybrid of the user profile page and the user index page (Section 10.3.1), with a sidebar of user information (including the following stats) and a list of users. In addition, we'll include a raster of smaller user profile image links in the sidebar. Mockups matching these requirements appear in Figure 14.14 (following) and Figure 14.15 (followers).

Our first step is to get the following and followers links to work. We'll follow Twitter's lead and have both pages require user login. As with most previous examples of access control, we'll write the tests first, as shown in Listing 14.24. Note that Listing 14.24 uses the named routes from Table 14.2.

Listing 14.24: Tests for the authorization of the following and followers pages. RED

test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
    @other_user = users(:archer)
  end
  .
  .
  .
  test "should redirect following when not logged in" do
    get following_user_path(@user)
    assert_redirected_to login_url
  end

  test "should redirect followers when not logged in" do
    get followers_user_path(@user)
    assert_redirected_to login_url
  end
end
```

The only tricky part of the implementation is realizing that we need to add two new actions to the Users controller. Based on the routes defined in Listing 14.15, we need to call them **`following`** and **`followers`**. Each action

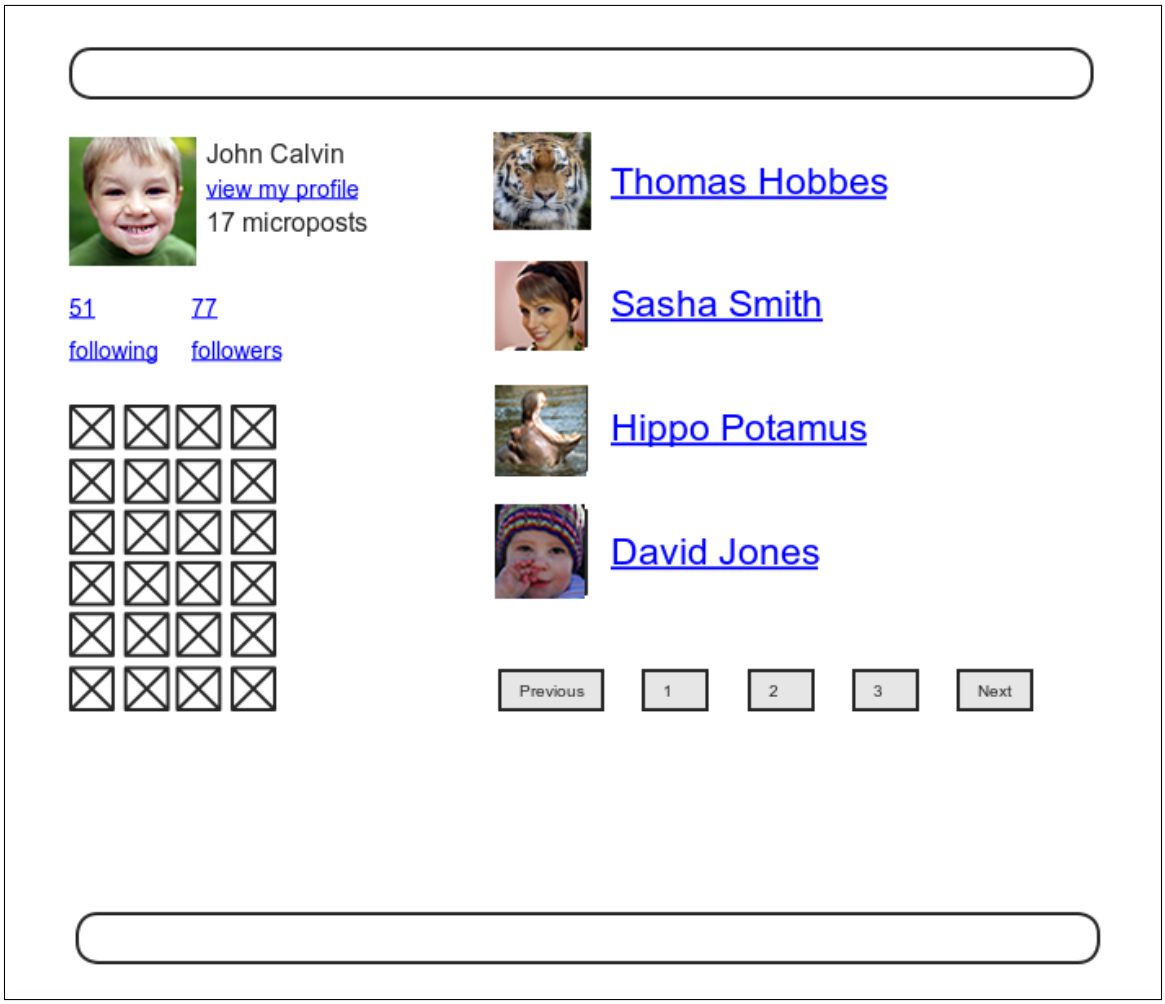

Figure 14.14: A mockup of the user following page.

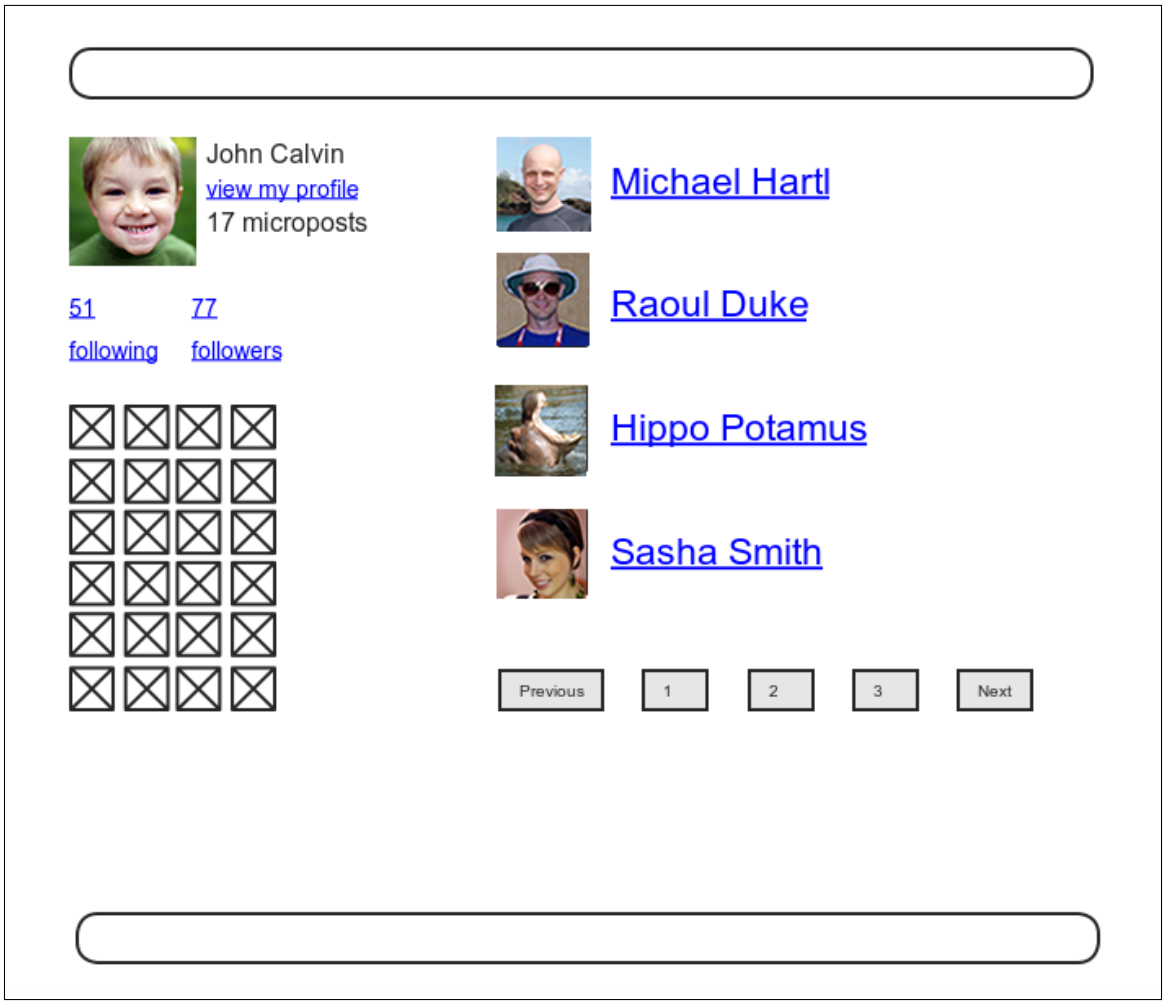

Figure 14.15: A mockup of the user followers page.

needs to set a title, find the user, retrieve either **`@user.following`** or **`@user.followers`** (in paginated form), and then render the page. The result appears in Listing 14.25.

Listing 14.25: The **`following`** and **`followers`** actions. RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update, :destroy,
                                        :following, :followers]
  .
  .
  .
  def following
    @title = "Following"
    @user  = User.find(params[:id])
    @users = @user.following.paginate(page: params[:page])
    render 'show_follow'
  end

  def followers
    @title = "Followers"
    @user  = User.find(params[:id])
    @users = @user.followers.paginate(page: params[:page])
    render 'show_follow'
  end

  private
  .
  .
  .
end
```

As we've seen throughout this tutorial, the usual Rails convention is to implicitly render the template corresponding to an action, such as rendering **`show.html.erb`** at the end of the **`show`** action. In contrast, both actions in Listing 14.25 make an *explicit* call to **`render`**, in this case rendering a view called **`show_follow`**, which we must create. The reason for the common view is that the ERb is nearly identical for the two cases, and Listing 14.26 covers them both.

Listing 14.26: The `show_follow` view used to render following and followers. GREEN

app/views/users/show_follow.html.erb

```
<% provide(:title, @title) %>
<div class="row">
  <aside class="col-md-4">
    <section class="user_info">
      <%= gravatar_for @user %>
      <h1><%= @user.name %></h1>
      <span><%= link_to "view my profile", @user %></span>
      <span><b>Microposts:</b> <%= @user.microposts.count %></span>
    </section>
    <section class="stats">
      <%= render 'shared/stats' %>
      <% if @users.any? %>
        <div class="user_avatars">
          <% @users.each do |user| %>
            <%= link_to gravatar_for(user, size: 30), user %>
          <% end %>
        </div>
      <% end %>
    </section>
  </aside>
  <div class="col-md-8">
    <h3><%= @title %></h3>
    <% if @users.any? %>
      <ul class="users follow">
        <%= render @users %>
      </ul>
      <%= will_paginate %>
    <% end %>
  </div>
</div>
```

The actions in Listing 14.25 render the view from Listing 14.26 in two contexts, "following" and "followers", with the results shown in Figure 14.16 and Figure 14.17. Note that nothing in the above code uses the current user, so the same links work for other users, as shown in Figure 14.18.

At this point, the tests in Listing 14.24 should be GREEN due to the before filter in Listing 14.25:

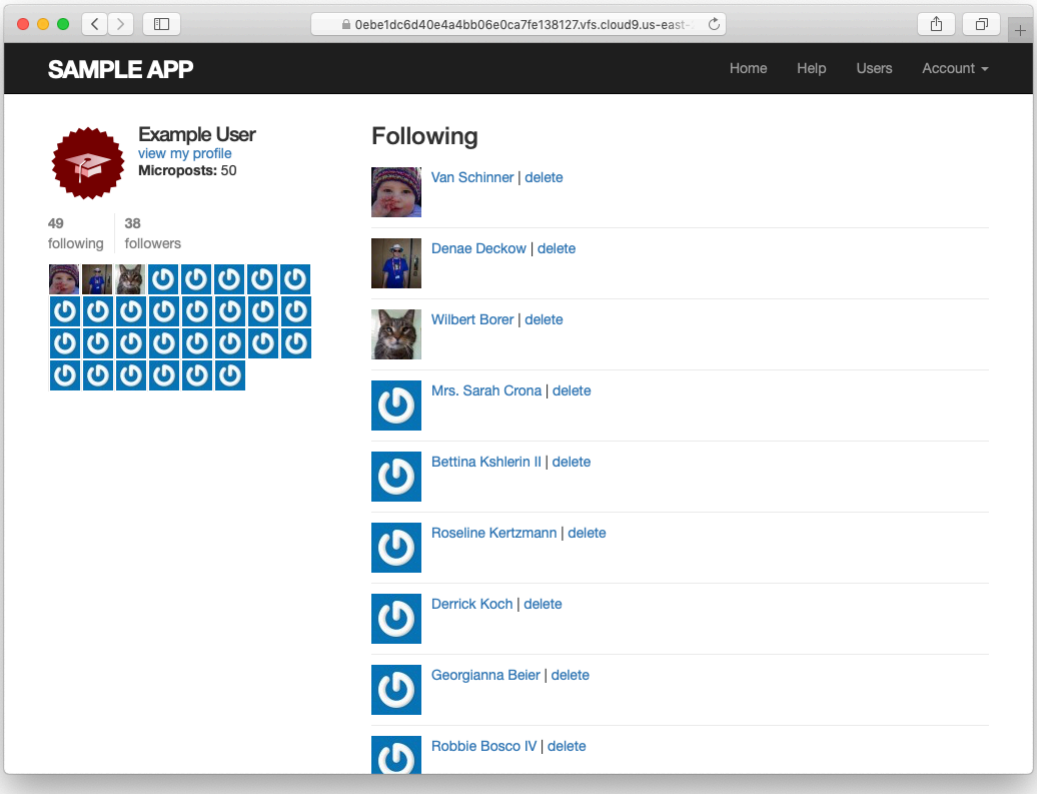

Figure 14.16: Showing the users the given user is following.

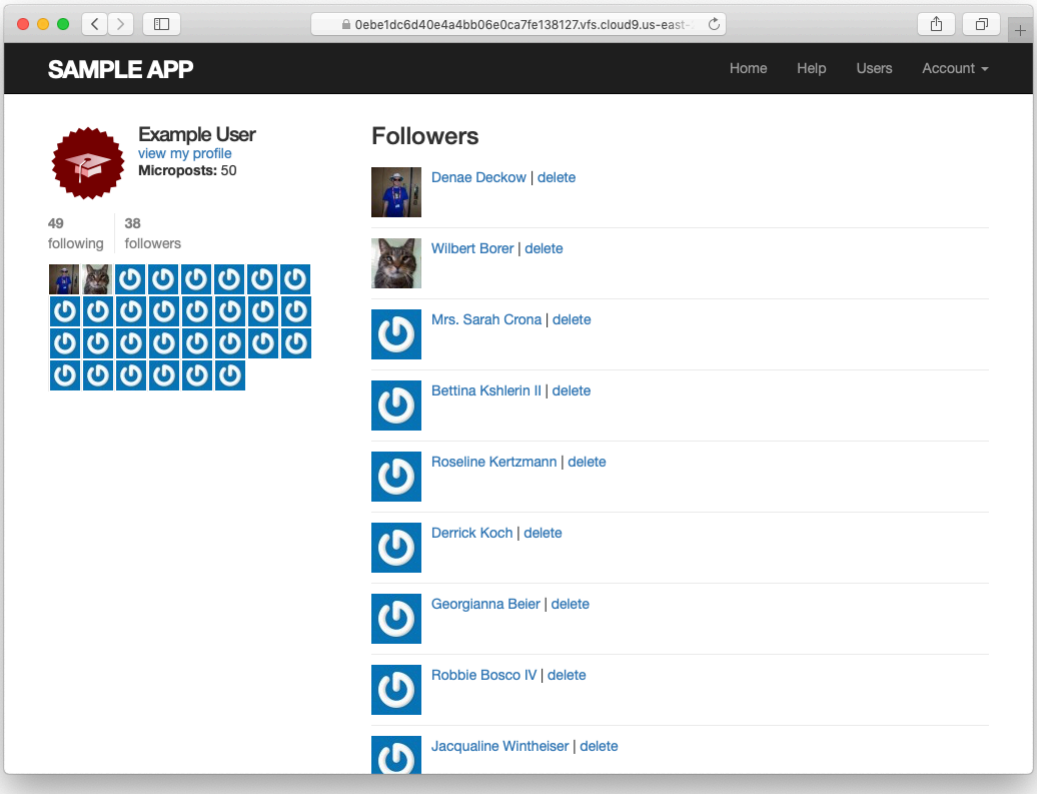

Figure 14.17: Showing the given user's followers.

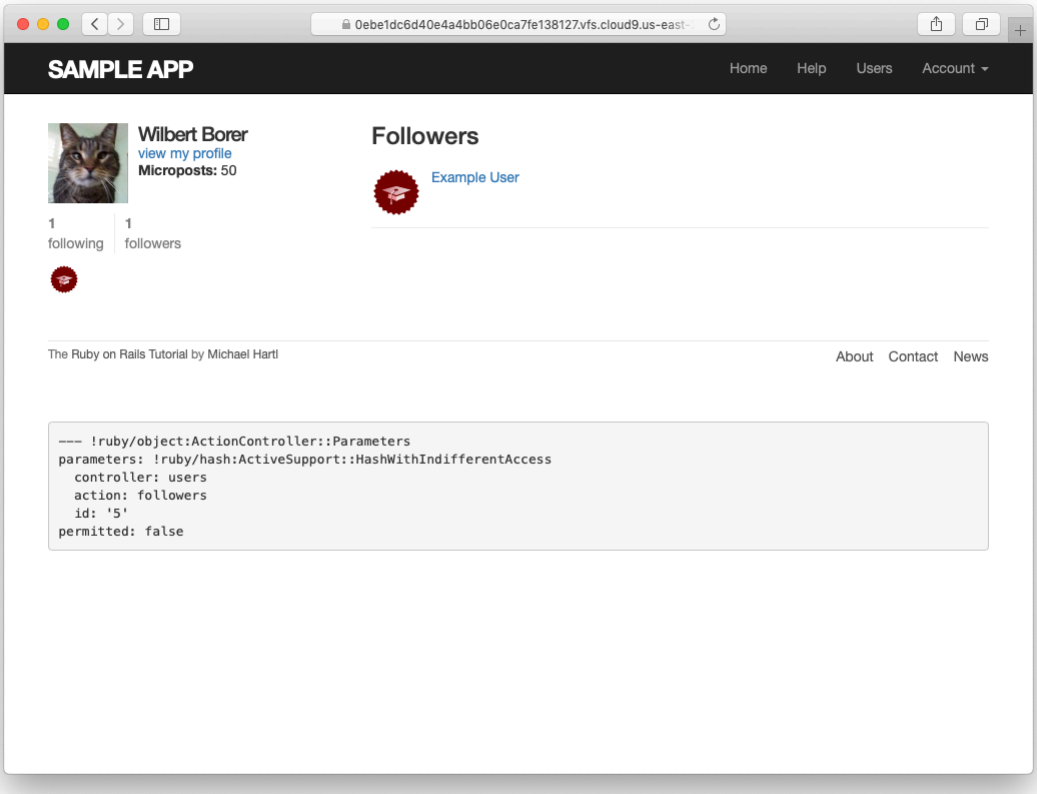

Figure 14.18: Showing a different user's followers.

Listing 14.27: GREEN

```
$ rails test
```

To test the **show_follow** rendering, we'll write a couple of short integration tests that verify the presence of working following and followers pages. They are designed to be a reality check, not to be comprehensive; indeed, as noted in Section 5.3.4, comprehensive tests of things like HTML structure are likely to be brittle and thus counter-productive. Our plan in the case of following/followers pages is to check the number is correctly displayed and that links with the right URLs appear on the page.

To get started, we'll generate an integration test as usual:

```
$ rails generate integration_test following
      invoke  test_unit
      create    test/integration/following_test.rb
```

Next, we need to assemble some test data, which we can do by adding some relationships fixtures to create following/follower relationships. Recall from Section 13.2.3 that we can use code like

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

to associate a micropost with a given user. In particular, we can write

```
user: michael
```

instead of

```
user_id: 1
```

Applying this idea to the relationships fixtures gives the associations in Listing 14.28.

Listing 14.28: Relationships fixtures for use in following/follower tests.
test/fixtures/relationships.yml

```
one:
  follower: michael
  followed: lana

two:
  follower: michael
  followed: malory

three:
  follower: lana
  followed: michael

four:
  follower: archer
  followed: michael
```

The fixtures in Listing 14.28 first arrange for Michael to follow Lana and Malory, and then arrange for Michael to be followed by Lana and Archer. To test for the right count, we can use the same **assert_match** method we used in Listing 13.28 to test for the display of the number of microposts on the user profile page. Adding in assertions for the right links yields the tests shown in Listing 14.29.

Listing 14.29: Tests for following/follower pages. GREEN
test/integration/following_test.rb

```
require 'test_helper'

class FollowingTest < ActionDispatch::IntegrationTest

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

```
    log_in_as(@user)
  end

  test "following page" do
    get following_user_path(@user)
    assert_not @user.following.empty?
    assert_match @user.following.count.to_s, response.body
    @user.following.each do |user|
      assert_select "a[href=?]", user_path(user)
    end
  end

  test "followers page" do
    get followers_user_path(@user)
    assert_not @user.followers.empty?
    assert_match @user.followers.count.to_s, response.body
    @user.followers.each do |user|
      assert_select "a[href=?]", user_path(user)
    end
  end
end
```

In Listing 14.29, note that we include the assertion

```
assert_not @user.following.empty?
```

which is included to make sure that

```
@user.following.each do |user|
  assert_select "a[href=?]", user_path(user)
end
```

isn't vacuously true (and similarly for **`followers`**). In other words, if **`@user.following.empty?`** were true, not a single **`assert_select`** would execute in the loop, leading the tests to pass and thereby give us a false sense of security.

The test suite should now be GREEN:

Listing 14.30: 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. Verify in a browser that /users/1/followers and /users/1/following work. Do the image links in the sidebar work as well?

2. Comment out the application code needed to turn the `assert_select` tests in Listing 14.29 RED to confirm they're testing the right thing.

14.2.4 A working follow button the standard way

Now that our views are in order, it's time to get the follow/unfollow buttons working. Because following and unfollowing involve creating and destroying relationships, we need a Relationships controller, which we generate as usual

```
$ rails generate controller Relationships
```

As we'll see in Listing 14.32, enforcing access control on the Relationships controller actions won't much matter, but we'll still follow our previous practice of enforcing the security model as early as possible. In particular, we'll check that attempts to access actions in the Relationships controller require a logged-in user (and thus get redirected to the login page), while also not changing the Relationship count, as shown in Listing 14.31.

Listing 14.31: Basic access control tests for relationships. RED
test/controllers/relationships_controller_test.rb

```
require 'test_helper'

class RelationshipsControllerTest < ActionDispatch::IntegrationTest

  test "create should require logged-in user" do
    assert_no_difference 'Relationship.count' do
      post relationships_path
    end
    assert_redirected_to login_url
  end

  test "destroy should require logged-in user" do
    assert_no_difference 'Relationship.count' do
      delete relationship_path(relationships(:one))
    end
    assert_redirected_to login_url
  end
end
```

We can get the tests in Listing 14.31 to pass by adding the **logged_in_user** before filter (Listing 14.32).

Listing 14.32: Access control for relationships. GREEN
app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
  end

  def destroy
  end
end
```

To get the follow and unfollow buttons to work, all we need to do is find the user associated with the **followed_id** in the corresponding form (i.e., Listing 14.21 or Listing 14.22), and then use the appropriate **follow** or **unfollow** method from Listing 14.10. The full implementation appears in Listing 14.33.

Listing 14.33: The Relationships controller. GREEN
app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
    user = User.find(params[:followed_id])
    current_user.follow(user)
    redirect_to user
  end

  def destroy
    user = Relationship.find(params[:id]).followed
    current_user.unfollow(user)
    redirect_to user
  end
end
```

We can see from Listing 14.33 why the security issue mentioned above is minor: if an unlogged-in user were to hit either action directly (e.g., using a command-line tool like **`curl`**), **`current_user`** would be **`nil`**, and in both cases the action's second line would raise an exception, resulting in an error but no harm to the application or its data. It's best not to rely on that, though, so we've taken the extra step and added an additional layer of security.

With that, the core follow/unfollow functionality is complete, and any user can follow or unfollow any other user, as you can verify by clicking the corresponding buttons in your browser. (We'll write integration tests to verify this behavior in Section 14.2.6.) The result of following user #2 is shown in Figure 14.19 and Figure 14.20.

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. Follow and unfollow /users/2 through the web. Did it work?
2. According to the server log, which templates are rendered in each case?

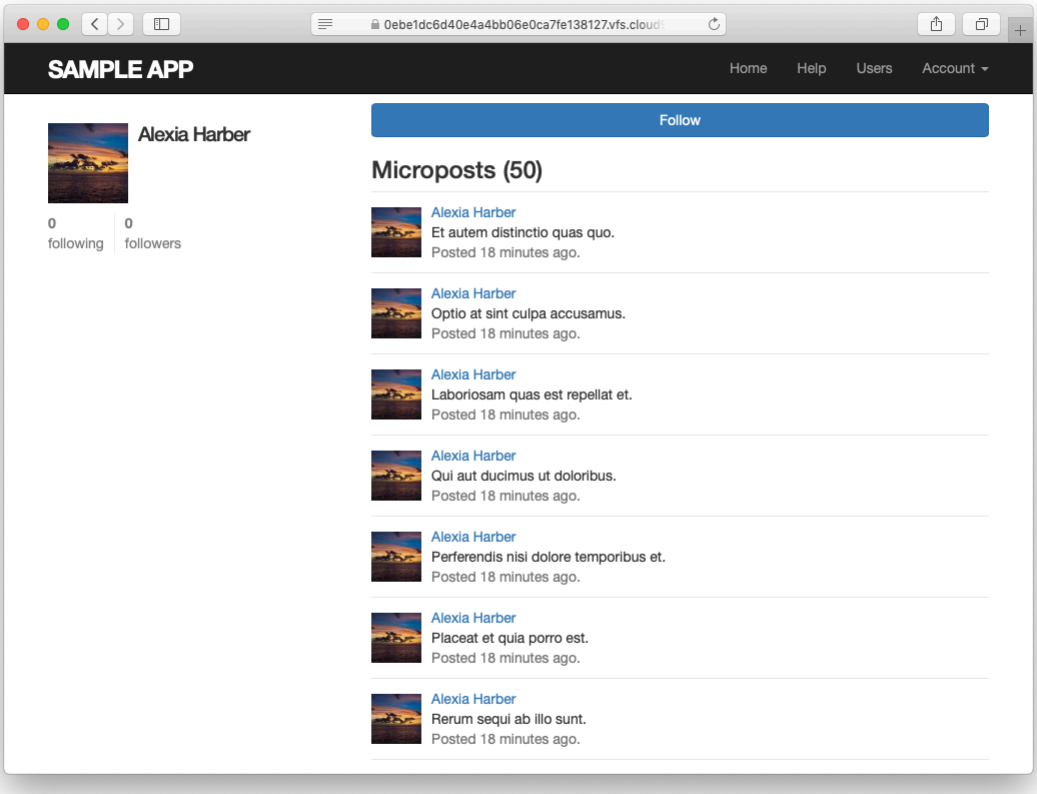

Figure 14.19: An unfollowed user.

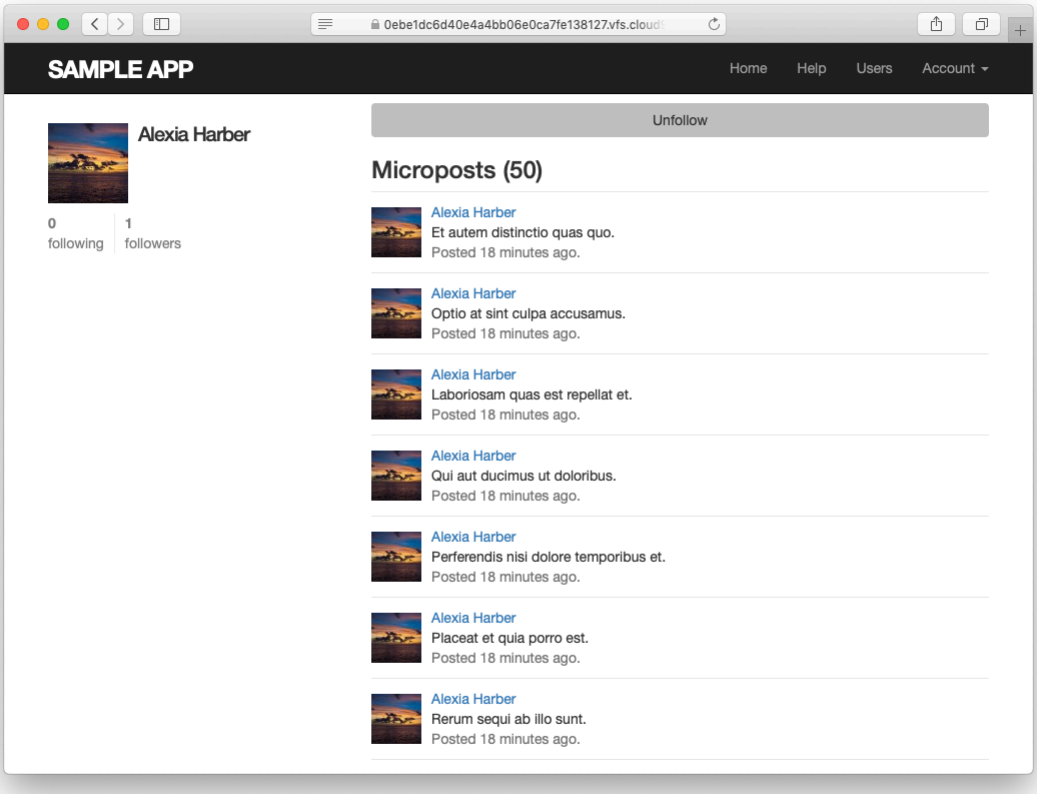

Figure 14.20: The result of following an unfollowed user.

14.2.5 A working follow button with Ajax

Although our user following implementation is complete as it stands, we have one bit of polish left to add before starting work on the status feed. You may have noticed in Section 14.2.4 that both the **create** and **destroy** actions in the Relationships controller simply redirect *back* to the original profile. In other words, a user starts on another user's profile page, follows the other user, and is immediately redirected back to the original page. It is reasonable to ask why the user needs to leave that page at all.

This is exactly the problem solved by *Ajax*, which allows web pages to send requests asynchronously to the server without leaving the page.[9] Because adding Ajax to web forms is a common practice, Rails makes Ajax easy to implement. Indeed, updating the follow/unfollow form partials is trivial: just change

```
form_with(model: ..., local: true)
```

to

```
form_with(model: ..., remote: true)
```

and Rails automagically uses Ajax.[10] The updated partials appear in Listing 14.34 and Listing 14.35.

Listing 14.34: A form for following a user using Ajax.
app/views/users/_follow.html.erb

```
<%= form_with(model: current_user.active_relationships.build, remote: true) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

[9] Because it is nominally an acronym for *asynchronous JavaScript and XML*, Ajax is sometimes misspelled "AJAX", even though the original Ajax article spells it as "Ajax" throughout.

[10] In fact, the default behavior for **form_with** is to make remote submissions, but at least when starting out I prefer to be explicit.

Listing 14.35: A form for unfollowing a user using Ajax.
app/views/users/_unfollow.html.erb

```
<%= form_with(model: current_user.active_relationships.find_by(followed_id: @user.id),
              html: { method: :delete }, remote: true) do |f| %>
  <%= f.submit "Unfollow", class: "btn" %>
<% end %>
```

The actual HTML generated by this ERb isn't particularly relevant, but you might be curious, so here's a peek at a schematic view (details may differ):

```
<form action="/relationships/117" class="edit_relationship" data-remote="true"
      id="edit_relationship_117" method="post">
  .
  .
  .
</form>
```

This sets the variable **`data-remote="true"`** inside the form tag, which tells Rails to allow the form to be handled by JavaScript. By using a simple HTML property instead of inserting the full JavaScript code (as in previous versions of Rails), Rails follows the philosophy of *unobtrusive JavaScript*.

Having updated the form, we now need to arrange for the Relationships controller to respond to Ajax requests. We can do this using the **`respond_to`** method, responding appropriately depending on the type of request. The general pattern looks like this:

```
respond_to do |format|
  format.html { redirect_to user }
  format.js
end
```

The syntax is potentially confusing, and it's important to understand that in the code above only *one* of the lines gets executed. (In this sense, **`respond_to`** is more like an if-then-else statement than a series of sequential lines.) Adapting the Relationships controller to respond to Ajax involves adding **`respond_to`** as above to the **`create`** and **`destroy`** actions from Listing 14.33. The result

appears as in Listing 14.36. Note the change from the local variable **user** to the instance variable **@user**; in Listing 14.33 there was no need for an instance variable, but now such a variable is necessary for use in Listing 14.34 and Listing 14.35.

Listing 14.36: Responding to Ajax requests in the Relationships controller.
app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
    @user = User.find(params[:followed_id])
    current_user.follow(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end

  def destroy
    @user = Relationship.find(params[:id]).followed
    current_user.unfollow(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end
end
```

The actions in Listing 14.36 degrade gracefully, which means that they work fine in browsers that have JavaScript disabled (although a small amount of configuration is necessary, as shown in Listing 14.37).

Listing 14.37: Configuration needed for graceful degradation of form submission.
config/application.rb

```
require_relative 'boot'
.
.
.
module SampleApp
```

```
  class Application < Rails::Application
    .
    .
    .
    # Include the authenticity token in remote forms.
    config.action_view.embed_authenticity_token_in_remote_forms = true
  end
end
```

On the other hand, we have yet to respond properly when JavaScript is enabled. In the case of an Ajax request, Rails automatically calls a *JavaScript embedded Ruby* (**.js.erb**) file with the same name as the action, i.e., **create.js.erb** or **destroy.js.erb**. As you might guess, such files allow us to mix JavaScript and embedded Ruby to perform actions on the current page. It is these files that we need to create and edit in order to update the user profile page upon being followed or unfollowed.

Inside a JS-ERb file, Rails automatically provides the jQuery JavaScript helpers to manipulate the page using the Document Object Model (DOM). The jQuery library (which we saw briefly in Section 13.4.2) provides a large number of methods for manipulating the DOM, but here we will need only two. First, we will need to know about the dollar-sign syntax to access a DOM element based on its unique CSS id. For example, to manipulate the **follow_form** element, we will use the syntax

```
$("#follow_form")
```

(Recall from Listing 14.19 that this is a **div** that wraps the form, not the form itself.) This syntax, inspired by CSS, uses the **#** symbol to indicate a CSS id. As you might guess, jQuery, like CSS, uses a dot **.** to manipulate CSS classes.

The second method we'll need is **html**, which updates the HTML inside the relevant element with the contents of its argument. For example, to replace the entire follow form with the string **"foobar"**, we would write

```
$("#follow_form").html("foobar")
```

Unlike plain JavaScript files, JS-ERb files also allow the use of embedded Ruby, which we apply in the **`create.js.erb`** file to update the follow form with the **`unfollow`** partial (which is what should show after a successful following) and update the follower count. The result is shown in Listing 14.38. This uses the **`escape_javascript`** method, which is needed to escape out the result when inserting HTML in a JavaScript file.

Listing 14.38: The JavaScript embedded Ruby to create a following relationship.
app/views/relationships/create.js.erb

```
$("#follow_form").html("<%= escape_javascript(render('users/unfollow')) %>");
$("#followers").html('<%= @user.followers.count %>');
```

Note the presence of line-ending semicolons, which are characteristic of languages with syntax descended from ALGOL.

The **`destroy.js.erb`** file is analogous (Listing 14.39).

Listing 14.39: The Ruby JavaScript (RJS) to destroy a following relationship.
app/views/relationships/destroy.js.erb

```
$("#follow_form").html("<%= escape_javascript(render('users/follow')) %>");
$("#followers").html('<%= @user.followers.count %>');
```

With that, you should navigate to a user profile page and verify that you can follow and unfollow without a page refresh.

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. Unfollow and refollow /users/2 through the web. Did it work?
2. According to the server log, which templates are rendered in each case?

14.2.6 Following tests

Now that the follow buttons are working, we'll write some simple tests to prevent regressions. To follow a user, we post to the relationships path and verify that the number of followed users increases by 1:

```
assert_difference '@user.following.count', 1 do
  post relationships_path, params: { followed_id: @other.id }
end
```

This tests the standard implementation, but testing the Ajax version is almost exactly the same; the only difference is the addition of the option **`xhr: true`**:

```
assert_difference '@user.following.count', 1 do
  post relationships_path, params: { followed_id: @other.id }, xhr: true
end
```

Here **`xhr`** stands for XmlHttpRequest; setting the **`xhr`** option to **`true`** issues an Ajax request in the test, which causes the **`respond_to`** block in Listing 14.36 to execute the proper JavaScript method.

The same parallel structure applies to deleting users, with **`delete`** instead of **`post`**. Here we check that the followed user count goes down by 1 and include the relationship and followed user's id:

```
assert_difference '@user.following.count', -1 do
  delete relationship_path(relationship)
end
```

and

```
assert_difference '@user.following.count', -1 do
  delete relationship_path(relationship), xhr: true
end
```

Putting the two cases together gives the tests in Listing 14.40.

Listing 14.40: Tests for the follow and unfollow buttons. GREEN
test/integration/following_test.rb

```
require 'test_helper'

class FollowingTest < ActionDispatch::IntegrationTest

  def setup
    @user  = users(:michael)
    @other = users(:archer)
    log_in_as(@user)
  end
  .
  .
  .
  test "should follow a user the standard way" do
    assert_difference '@user.following.count', 1 do
      post relationships_path, params: { followed_id: @other.id }
    end
  end

  test "should follow a user with Ajax" do
    assert_difference '@user.following.count', 1 do
      post relationships_path, xhr: true, params: { followed_id: @other.id }
    end
  end

  test "should unfollow a user the standard way" do
    @user.follow(@other)
    relationship = @user.active_relationships.find_by(followed_id: @other.id)
    assert_difference '@user.following.count', -1 do
      delete relationship_path(relationship)
    end
  end

  test "should unfollow a user with Ajax" do
    @user.follow(@other)
    relationship = @user.active_relationships.find_by(followed_id: @other.id)
    assert_difference '@user.following.count', -1 do
      delete relationship_path(relationship), xhr: true
    end
  end
end
```

At this point, the tests should be GREEN:

Listing 14.41: 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 and uncommenting each of the lines in the `respond_to` blocks (Listing 14.36), verify that the tests are testing the right things. Which test fails in each case?

2. What happens if you delete one of the occurrences of `xhr: true` in Listing 14.40? Explain why this is a problem, and why the procedure in the previous exercise would catch it.

14.3 The status feed

We come now to the pinnacle of our sample application: the status feed of microposts. Appropriately, this section contains some of the most advanced material in the entire tutorial. The full status feed builds on the proto-feed from Section 13.3.3 by assembling an array of the microposts from the users being followed by the current user, along with the current user's own microposts. Throughout this section, we'll proceed through a series of feed implementations of increasing sophistication. To accomplish this, we will need some fairly advanced Rails, Ruby, and even SQL programming techniques.

Because of the heavy lifting ahead, it's especially important to review where we're going. A recap of the final status feed, shown in Figure 14.5, appears again in Figure 14.21.