Chapter 13 User microposts

In the course of developing the core sample application, we've now encountered four resources—users, sessions, account activations, and password resets—but only the first of these is backed by an Active Record model with a table in the database. The time has finally come to add a second such resource: user *microposts*, which are short messages associated with a particular user.[1] We first saw microposts in larval form in Chapter 2, and in this chapter we will make a full-strength version of the sketch from Section 2.3 by constructing the Micropost data model, associating it with the User model using the `has_many` and `belongs_to` methods, and then making the forms and partials needed to manipulate and display the results (including, in Section 13.4, uploaded images). In Chapter 14, we'll complete our tiny Twitter clone by adding the notion of *following* users in order to receive a *feed* of their microposts.

13.1 A Micropost model

We begin the Microposts resource by creating a Micropost model, which captures the essential characteristics of microposts. What follows builds on the work from Section 2.3; as with the model in that section, our new Micropost model will include data validations and an association with the User model. Unlike that model, the present Micropost model will be fully tested, and will also have a default *ordering* and automatic *destruction* if its parent user is destroyed.

If you're using Git for version control, I suggest making a topic branch at this time:

```
$ git checkout -b user-microposts
```

13.1.1 The basic model

The Micropost model needs only two attributes: a `content` attribute to hold the micropost's content and a `user_id` to associate a micropost with a particular user. The result is a Micropost model with the structure shown in Figure 13.1.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime

Figure 13.1: The Micropost data model.

It's worth noting that the model in Figure 13.1 uses the `text` data type for micropost content (instead of `string`), which is capable of storing an arbitrary amount of text. Even though the content will be restricted to fewer than 140 characters (Section 13.1.2) and hence would fit inside the 255-character `string` type, using `text` better expresses the nature of microposts, which are more naturally thought of as blocks of text. Indeed, in Section 13.3.2 we'll use a text *area* instead of a text field for submitting microposts. In addition, using `text` gives us greater flexibility should we wish to increase the length limit at a future date (as part of internationalization, for example). Finally, using the `text` type results in no performance difference in production,[2] so it costs us nothing to use it here.

As with the case of the User model (Listing 6.1), we generate the Micropost model using `generate model` (Listing 13.1).

Listing 13.1: Generating the Micropost model.

```
$ rails generate model Micropost content:text user:references
```

This migration leads to the creation of the Micropost model shown in Listing 13.2. In addition to inheriting from `ApplicationRecord` as usual (Section 6.1.2), the generated model includes a line indicating that a micropost `belongs_to` a user, which is included as a result of the `user:references` argument in Listing 13.1. We'll explore the implications of this line in Section 13.1.3.

Listing 13.2: The generated Micropost model. `app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  belongs_to :user
end
```

The `generate` command in Listing 13.1 also produces a migration to create a `microposts` table in the database (Listing 13.3); compare it to the analogous migration for the `users` table from Listing 6.2. The biggest difference is the use of `references`, which automatically adds a `user_id` column (along with an index and a foreign key reference)[3] for use in the user/micropost association. As with the User model, the Micropost model migration automatically includes the `t.timestamps` line, which (as mentioned in Section 6.1.1) adds the magic `created_at` and `updated_at` columns shown in Figure 13.1. (We'll put the `created_at` column to work starting in Section 13.1.4.)

Listing 13.3: The Micropost migration with added index. `db/migrate/[timestamp]_create_microposts.rb`

```
class CreateMicroposts < ActiveRecord::Migration[5.0]
  def change
    create_table :microposts do |t|
      t.text :content
      t.references :user, foreign_key: true

      t.timestamps
    end
    add_index :microposts, [:user_id, :created_at]
  end
end
```

Because we expect to retrieve all the microposts associated with a given user id in reverse order of creation, Listing 13.3 adds an index (Box 6.2) on the `user_id` and `created_at` columns:

```
add_index :microposts, [:user_id, :created_at]
```

By including both the `user_id` and `created_at` columns as an array, we arrange for Rails to create a *multiple key index*, which means that Active Record uses *both* keys at the same time.

With the migration in Listing 13.3, we can update the database as usual:

```
$ rails db:migrate
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Using `Micropost.new` in the console, instantiate a new Micropost object called `micropost` with content "Lorem ipsum" and user id equal to the id of the first user in the database. What are the values of the magic columns `created_at` and `updated_at`?
2. What is `micropost.user` for the micropost in the previous exercise? What about `micropost.user.name`?
3. Save the micropost to the database. What are the values of the magic columns now?

13.1.2 Micropost validations

Now that we've created the basic model, we'll add some validations to enforce the desired design constraints. One of the necessary aspects of the Micropost model is the presence of a user id to indicate which user made the micropost. The idiomatically correct way to do this is to use Active Record *associations*, which we'll implement in Section 13.1.3, but for now we'll work with the `Micropost` model directly.

The initial micropost tests parallel those for the User model (Listing 6.7). In the `setup` step, we create a new micropost while associating it with a valid user from the fixtures, and then check that the result is valid. Because every micropost should have a user id, we'll add a test for a `user_id` presence validation. Putting these elements together yields the test in Listing 13.4.

Listing 13.4: Tests for the validity of a new micropost. **green** `test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

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

```
# This code is not idiomatically correct.
@micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
end

test "should be valid" do
  assert @micropost.valid?
end

test "user id should be present" do
  @micropost.user_id = nil
assert_not @micropost.valid?
end
end
```

As indicated by the comment in the `setup` method, the code to create the micropost is not idiomatically correct, which we'll fix in Section 13.1.3.

As with the original User model test (Listing 6.5), the first test in Listing 13.4 is just a sanity check, but the second is a test of the presence of the user id, for which we'll add the presence validation shown in Listing 13.5.

Listing 13.5: A validation for the micropost's `user_id`. **green** `app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
validates :user_id, presence: true
end
```

By the way, as of Rails 5 the tests in Listing 13.4 actually pass without the validation in Listing 13.5, but only when using the idiomatically incorrect line highlighted in Listing 13.4. The user id presence validation is necessary after switching to the idiomatically correct code in Listing 13.12, so we include it here for convenience.

With the code in Listing 13.5 the tests should (still) be **green**:

Listing 13.6: **green**

```
$ rails test:models
```

Next, we'll add validations for the micropost's `content` attribute (following the example from Section 2.3.2). As with the `user_id`, the `content` attribute must be present, and it is further constrained to be no longer than 140 characters (which is what puts the *micro* in micropost).

As with the User model validations (Section 6.2), we'll add the micropost content validations using test-driven development. The resulting tests generally follow the examples from the User model validation tests, as shown in Listing 13.7.

Listing 13.7: Tests for the Micropost model validations. **red** `test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

def setup
  @user = users(:michael)
@micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
end

test "should be valid" do
  assert @micropost.valid?
end

test "user id should be present" do
  @micropost.user_id = nil
assert_not @micropost.valid?
end

test "content should be present" do
  @micropost.content = "   "
assert_not @micropost.valid?
end

test "content should be at most 140 characters" do
```

```
@micropost.content = "a" * 141
assert_not @micropost.valid?
end
end
```

As in Section 6.2, the code in Listing 13.7 uses string multiplication to test the micropost length validation:

```
$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

The corresponding application code is virtually identical to the `name` validation for users (Listing 6.16), as shown in Listing 13.8.

Listing 13.8: The Micropost model validations. **green** `app/models/micropost.rb`

```
class Micropost < ApplicationRecord
belongs_to :user
validates :user_id, presence: true
validates :content, presence: true, length: { maximum: 140 }
end
```

At this point, the full test suite should be **green**:

Listing 13.9: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. At the console, instantiate a micropost with no user id and blank content. Is it valid? What are the full error messages?
2. At the console, instantiate a second micropost with no user id and content that's too long. Is it valid? What are the full error messages?

13.1.3 User/Micropost associations

When constructing data models for web applications, it is essential to be able to make *associations* between individual models. In the present case, each micropost is associated with one user, and each user is associated with (potentially) many microposts—a relationship seen briefly in Section 2.3.3 and shown schematically in Figure 13.2 and Figure 13.3. As part of implementing these associations, we'll write tests for the Micropost model and add a couple of tests to the User model.

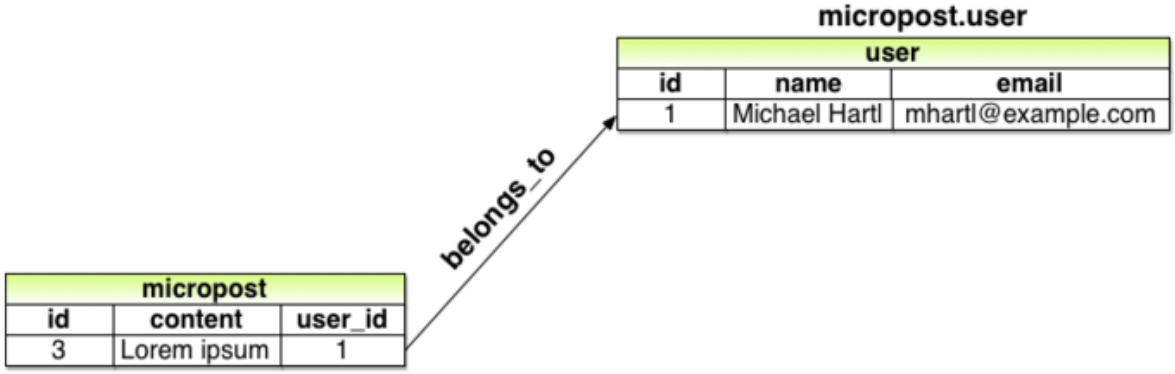

Figure 13.2: The `belongs_to` relationship between a micropost and its associated user.

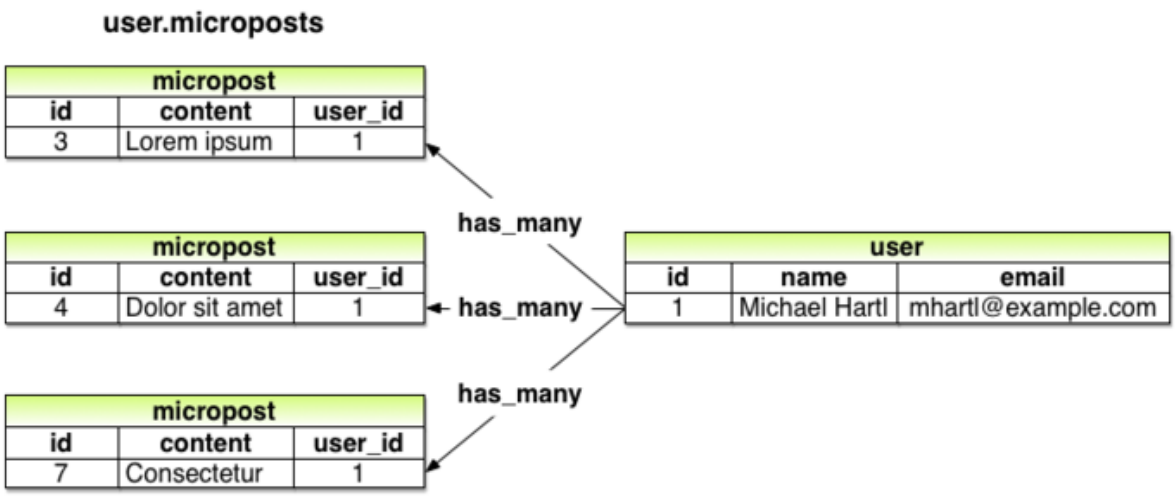

Figure 13.3: The `has_many` relationship between a user and its microposts.

Using the `belongs_to`/`has_many` association defined in this section, Rails constructs the methods shown in Table 13.1. Note from Table 13.1 that instead of

```
Micropost.create
Micropost.create!
Micropost.new
```

we have

```
user.microposts.create
user.microposts.create!
user.microposts.build
```

These latter methods constitute the idiomatically correct way to make a micropost, namely, *through* its association with a user. When a new micropost is made in this way, its `user_id` is automatically set to the right value. In particular, we can replace the code

```
@user = users(:michael)
# This code is not idiomatically correct.
@micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
```

from Listing 13.4 with this:

```
@user = users(:michael)
@micropost = @user.microposts.build(content: "Lorem ipsum")
```

(As with `new`, `build` returns an object in memory but doesn't modify the database.) Once we define the proper associations, the resulting `@micropost` variable will automatically have a `user_id` attribute equal to its associated user's id.

Method	Purpose
`micropost.user`	Returns the User object associated with the micropost
`user.microposts`	Returns a collection of the user's microposts
`user.microposts.create(arg)`	Creates a micropost associated with `user`
`user.microposts.create!(arg)`	Creates a micropost associated with `user` (exception on failure)
`user.microposts.build(arg)`	Returns a new Micropost object associated with `user`
`user.microposts.find_by(id: 1)`	Finds the micropost with id `1` and `user_id` equal to `user.id`

Table 13.1: A summary of user/micropost association methods.

To get code like `@user.microposts.build` to work, we need to update the User and Micropost models with code to associate them. The first of these was included automatically by the migration in Listing 13.3 via `belongs_to :user`, as shown in Listing 13.10. The second half of the association, `has_many :microposts`, needs to be added by hand, as shown in (Listing 13.11).

Listing 13.10: A micropost `belongs_to` a user. **green** `app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  belongs_to :user
```

```
validates :user_id, presence: true
validates :content, presence: true, length: { maximum: 140 }
end
```

Listing 13.11: A user `has_many` microposts. **green** `app/models/user.rb`

```
class User < ApplicationRecord
  has_many :microposts
  .
  .
  .
end
```

With the association thus made, we can update the `setup` method in Listing 13.4 with the idiomatically correct way to build a new micropost, as shown in Listing 13.12.

Listing 13.12: Using idiomatically correct code to build a micropost. **green** `test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
@micropost = @user.microposts.build(content: "Lorem ipsum")
  end

  test "should be valid" do
    assert @micropost.valid?
  end

test "user id should be present" do
    @micropost.user_id = nil
  assert_not @micropost.valid?
  end
  .
  .
  .
end
```

Of course, after this minor refactoring the test suite should still be **green**:

Listing 13.13: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Set `user` to the first user in the database. What happens when you execute the command `micropost = user.microposts.create(content: "Lorem ipsum")`?
2. The previous exercise should have created a micropost in the database. Confirm this by running `user.microposts.find(micropost.id)`. What if you write `micropost` instead of `micropost.id`?
3. What is the value of `user == micropost.user`? How about `user.microposts.first == micropost`?

13.1.4 Micropost refinements

In this section, we'll add a couple of refinements to the user/micropost association. In particular, we'll arrange for a user's microposts to be retrieved in a specific *order*, and we'll also make microposts *dependent* on users so that they will be automatically destroyed if their associated user is destroyed.

Default scope

By default, the `user.microposts` method makes no guarantees about the order of the posts, but (following the convention of blogs and Twitter) we want the microposts to come out in reverse order of when they were created so that the most recent post is first.[4] We'll arrange for this to happen using a *default scope*.

This is exactly the sort of feature that could easily lead to a spurious passing test (i.e., a test that would pass even if the application code were wrong), so we'll proceed using test-driven development to be sure we're testing the right thing. In particular, let's write a test to verify that the first micropost in the database is the same as a fixture micropost we'll call `most_recent`, as shown in Listing 13.14.

Listing 13.14: Testing the micropost order. **red** `test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase
.
.
.
test "order should be most recent first" do
assert_equal microposts(:most_recent), Micropost.first
end
end
```

Listing 13.14 relies on having some micropost fixtures, which we define as shown in Listing 13.15.

Listing 13.15: Micropost fixtures. `test/fixtures/microposts.yml`

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

tau_manifesto:
content: "Check out the @tauday site by @mhartl: http://tauday.com"
created_at: <%= 3.years.ago %>

cat_video:
content: "Sad cats are sad: http://youtu.be/PKffm2uI4dk"
created_at: <%= 2.hours.ago %>

most_recent:
content: "Writing a short test"
created_at: <%= Time.zone.now %>
```

Note that we have explicitly set the `created_at` column using embedded Ruby. Because it's a "magic" column automatically updated by Rails, setting it by hand isn't ordinarily possible, but it is possible in fixtures. In practice this might not be necessary, and in fact on many systems the fixtures are created in order. In this case, the final fixture in the file is created last (and hence is most recent), but it would be foolish to rely on this behavior, which is brittle and probably system-dependent.

With the code in Listing 13.14 and Listing 13.15, the test suite should be **red**:

Listing 13.16: **red**

```
$ rails test test/models/micropost_test.rb
```

We'll get the test to pass using a Rails method called `default_scope`, which among other things can be used to set the default order in which elements are retrieved from the database. To enforce a particular order, we'll include the `order` argument in `default_scope`, which lets us order by the `created_at` column as follows:

```
order(:created_at)
```

Unfortunately, this orders the results in *ascending* order from smallest to biggest, which means that the oldest microposts come out first. To pull them out in reverse order, we can push down one level deeper and include a string with some raw SQL:

```
order('created_at DESC')
```

Here `DESC` is SQL for "descending", i.e., in descending order from newest to oldest.[5] In older versions of Rails, using this raw SQL used to be the only option to get the desired behavior, but as of Rails 4.0 we can use a more natural pure-Ruby syntax as well:

```
order(created_at: :desc)
```

Adding this in a default scope for the Micropost model gives Listing 13.17.

Listing 13.17: Ordering the microposts with `default_scope`. **green** `app/models/micropost.rb`

```
class Micropost < ApplicationRecord
belongs_to :user
default_scope -> { order(created_at: :desc) }
validates :user_id, presence: true
validates :content, presence: true, length: { maximum: 140 }
end
```

Listing 13.17 introduces the “stabby lambda” syntax for an object called a *Proc* (procedure) or *lambda*, which is an *anonymous function* (a function created without a name). The stabby lambda `->` takes in a block (Section 4.3.2) and returns a Proc, which can then be evaluated with the `call` method. We can see how it works at the console:

```
>> -> { puts "foo" }
=> #<Proc:0x007fab938d0108@(irb):1 (lambda)>
>> -> { puts "foo" }.call
foo
=> nil
```

(This is a somewhat advanced Ruby topic, so don’t worry if it doesn’t make sense right away.)

With the code in Listing 13.17, the tests should be **green**:

Listing 13.18: **green**

```
$ rails test
```

Dependent: destroy

Apart from proper ordering, there is a second refinement we’d like to add to microposts. Recall from Section 10.4 that site administrators have the power to *destroy* users. It stands to reason that, if a user is destroyed, the user’s microposts should be destroyed as well.

We can arrange for this behavior by passing an option to the `has_many` association method, as shown in Listing 13.19.

Listing 13.19: Ensuring that a user’s microposts are destroyed along with the user. `app/models/user.rb`

```
class User < ApplicationRecord
has_many :microposts, dependent: :destroy
.
.
.
end
```

Here the option `dependent: :destroy` arranges for the dependent microposts to be destroyed when the user itself is destroyed. This prevents userless microposts from being stranded in the database when admins choose to remove users from the system.

We can verify that Listing 13.19 is working with a test for the User model. All we need to do is save the user (so it gets an id) and create an associated micropost. Then we check that destroying the user reduces the micropost count by 1. The result appears in Listing 13.20. (Compare to the integration test for “delete” links in Listing 10.62.)

Listing 13.20: A test of `dependent: :destroy`. **green** `test/models/user_test.rb`

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

def setup
@user = User.new(name: "Example User", email: "user@example.com",
password: "foobar", password_confirmation: "foobar")
end
.
.
.
test "associated microposts should be destroyed" do
@user.save
@user.microposts.create!(content: "Lorem ipsum")
assert_difference 'Micropost.count', -1 do
@user.destroy
end
```

```
  end
end
```

If the code in Listing 13.19 is working correctly, the test suite should still be **green**:

Listing 13.21: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. How does the value of `Micropost.first.created_at` compare to `Micropost.last.created_at`?
2. What are the SQL queries for Micropost.first and Micropost.last? *Hint*: They are printed out by the console.
3. Let `user` be the first user in the database. What is the id of its first micropost? Destroy the first user in the database using the `destroy` method, then confirm using `Micropost.find` that the user's first micropost was also destroyed.

13.2 Showing microposts

Although we don't yet have a way to create microposts through the web—that comes in Section 13.3.2—this won't stop us from displaying them (and testing that display). Following Twitter's lead, we'll plan to display a user's microposts not on a separate microposts `index` page but rather directly on the user `show` page itself, as mocked up in Figure 13.4. We'll start with fairly simple ERb templates for adding a micropost display to the user profile, and then we'll add microposts to the seed data from Section 10.3.2 so that we have something to display.

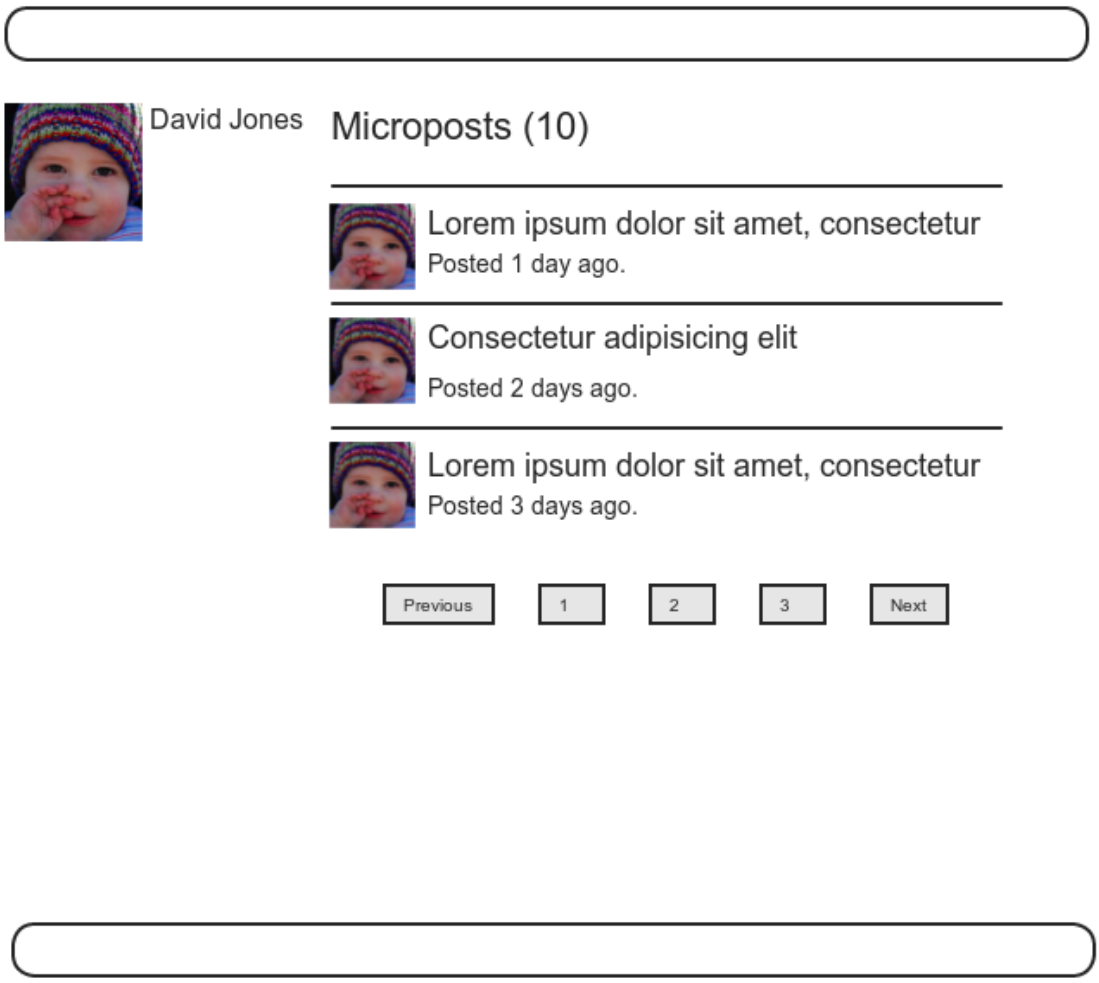

Figure 13.4: A mockup of a profile page with microposts.

13.2.1 Rendering microposts

Our plan is to display the microposts for each user on their respective profile page (`show.html.erb`), together with a running count of how many microposts they've made. As we'll see, many of the ideas are similar to our work in Section 10.3 on showing all users.

In case you've added some microposts in the exercises, it's a good idea to reset and reseed the database at this time:

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

Although we won't need the Microposts controller until Section 13.3, we will need the views directory in just a moment, so let's generate the controller now:

```
$ rails generate controller Microposts
```

Our primary purpose in this section is to render all the microposts for each user. We saw in Section 10.3.5 that the code

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

automatically renders each of the users in the `@users` variable using the `_user.html.erb` partial. We'll define an analogous `_micropost.html.erb` partial so that we can use the same technique on a collection of microposts as follows:

```
<ol class="microposts">
<%= render @microposts %>
</ol>
```

Note that we've used the *ordered list* tag `ol` (as opposed to an unordered list `ul`) because microposts are listed in a particular order (reverse-chronological). The corresponding partial appears in Listing 13.22.

Listing 13.22: A partial for showing a single micropost. `app/views/microposts/_micropost.html.erb`

```
<li id="micropost-<%= micropost.id %>">
<%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
<span class="user"><%= link_to micropost.user.name, micropost.user %></span>
<span class="content"><%= micropost.content %></span>
<span class="timestamp">
Posted <%= time_ago_in_words(micropost.created_at) %> ago.
</span>
</li>
```

This uses the awesome `time_ago_in_words` helper method, whose meaning is probably clear and whose effect we will see in Section 13.2.2. Listing 13.22 also adds a CSS id for each micropost using

```
<li id="micropost-<%= micropost.id %>">
```

This is a generally good practice, as it opens up the possibility of manipulating individual microposts at a future date (using JavaScript, for example).

The next step is to address the difficulty of displaying a potentially large number of microposts. We'll solve this problem the same way we solved it for users in Section 10.3.3, namely, using pagination. As before, we'll use the `will_paginate` method:

```
<%= will_paginate @microposts %>
```

If you compare this with the analogous line on the user index page, Listing 10.45, you'll see that before we had just

```
<%= will_paginate %>
```

This worked because, in the context of the Users controller, `will_paginate` *assumes* the existence of an instance variable called `@users` (which, as we saw in Section 10.3.3, should be of class `ActiveRecord::Relation`). In the present case, since we are still in the Users controller but want to paginate *microposts* instead, we'll pass an explicit `@microposts` variable to `will_paginate`. Of course, this means that we will have to define such a variable in the user `show` action (Listing 13.23).

Listing 13.23: Adding an `@microposts` instance variable to the user `show` action. `app/controllers/users_controller.rb`

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

Notice here how clever `paginate` is—it even works *through* the microposts association, reaching into the `microposts` table and pulling out the desired page of microposts.

Our final task is to display the number of microposts for each user, which we can do with the `count` method:

```
user.microposts.count
```

As with `paginate`, we can use the `count` method through the association. In particular, `count` does *not* pull all the microposts out of the database and then call `length` on the resulting array, as this would become inefficient as the number of microposts grew. Instead, it performs the calculation directly in the database, asking the database to count the microposts with the given `user_id` (an operation for which all databases are highly optimized). (In the unlikely event that finding the count is still a bottleneck in your application, you can make it even faster using a *counter cache*.)

Putting all the elements above together, we are now in a position to add microposts to the profile page, as shown in Listing 13.24. Note the use of `if @user.microposts.any?` (a construction we saw before in Listing 7.21), which makes sure that an empty list won't be displayed when the user has no microposts.

Listing 13.24: Adding microposts to the user `show` page. `app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
<aside class="col-md-4">
<section class="user_info">
<h1>
<%= gravatar_for @user %>
<%= @user.name %>
</h1>
</section>
</aside>
<div class="col-md-8">
<% if @user.microposts.any? %>
<h3>Microposts (<%= @user.microposts.count %>)</h3>
<ol class="microposts">
<%= render @microposts %>
</ol>
<%= will_paginate @microposts %>
<% end %>
</div>
</div>
```

At this point, we can get a look at our updated user profile page in Figure 13.5. It's rather…disappointing. Of course, this is because there are not currently any microposts. It's time to change that.

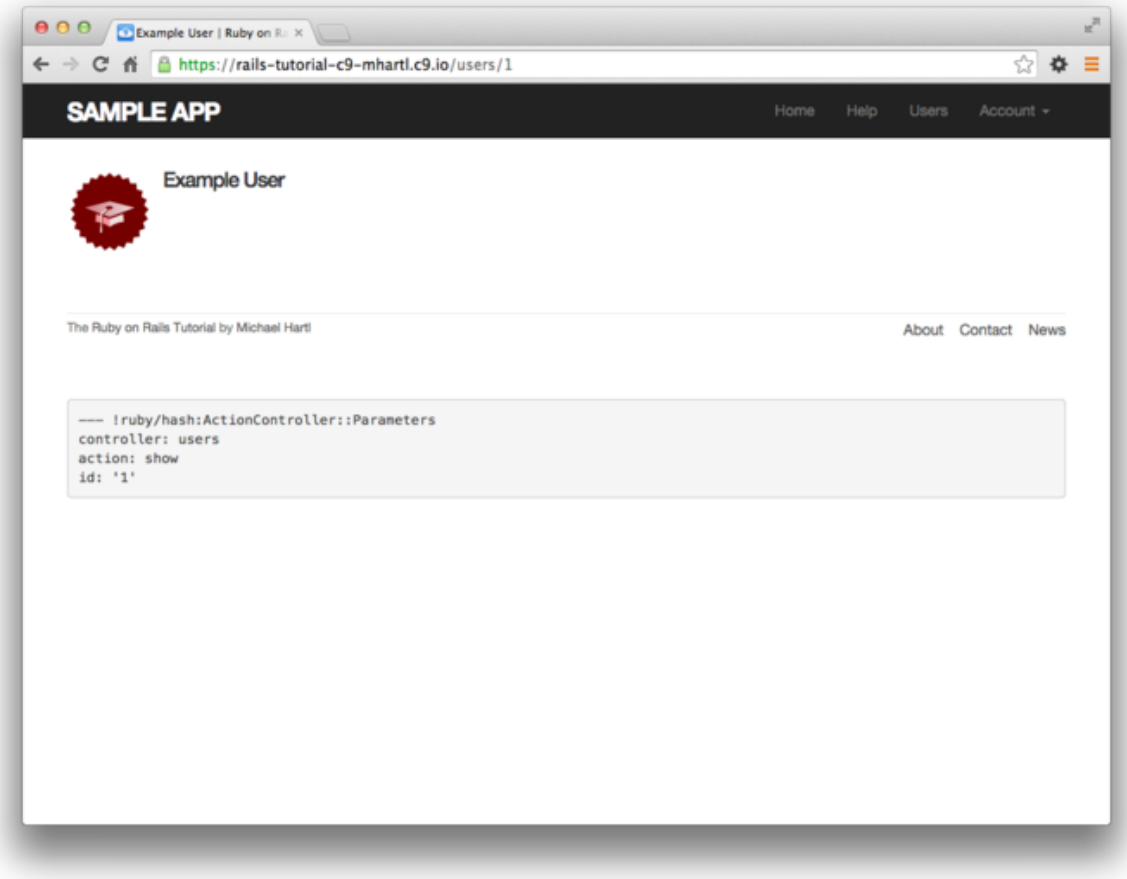

Figure 13.5: The user profile page with code for microposts—but no microposts.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. As mentioned briefly in Section 7.3.3, helper methods like `time_ago_in_words` are available in the Rails console via the `helper` object. Using `helper`, apply `time_ago_in_words` to `3.weeks.ago` and `6.months.ago`.
2. What is the result of `helper.time_ago_in_words(1.year.ago)`?
3. What is the Ruby class for a page of microposts? *Hint*: Use the code in Listing 13.23 as your model, and call the `class` method on `paginate` with the argument `page: nil`.

13.2.2 Sample microposts

With all the work making templates for user microposts in Section 13.2.1, the ending was rather anticlimactic. We can rectify this sad situation by adding microposts to the seed data from Section 10.3.2.

Adding sample microposts for *all* the users actually takes a rather long time, so first we'll select just the first six users (i.e., the five users with custom Gravatars, and one with the default Gravatar) using the `take` method:

```
User.order(:created_at).take(6)
```

The call to `order` ensures that we find the first six users that were created.

For each of the selected users, we'll make 50 microposts (plenty to overflow the pagination limit of 30). To generate sample content for each micropost, we'll use the Faker gem's handy `Lorem.sentence` method.[6] The result is the new seed data method shown in Listing 13.25. (The reason for the order of the loops in Listing 13.25 is to intermix the microposts for use in the status feed (Section 14.3). Looping over the users first gives feeds with big runs of microposts from the same user, which is visually unappealing.)

Listing 13.25: Adding microposts to the sample data. `db/seeds.rb`

.
.

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

At this point, we can reseed the development database as usual:

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

You should also quit and restart the Rails development server.

With that, we are in a position to enjoy the fruits of our Section 13.2.1 labors by displaying information for each micropost.[7] The preliminary results appear in Figure 13.6.

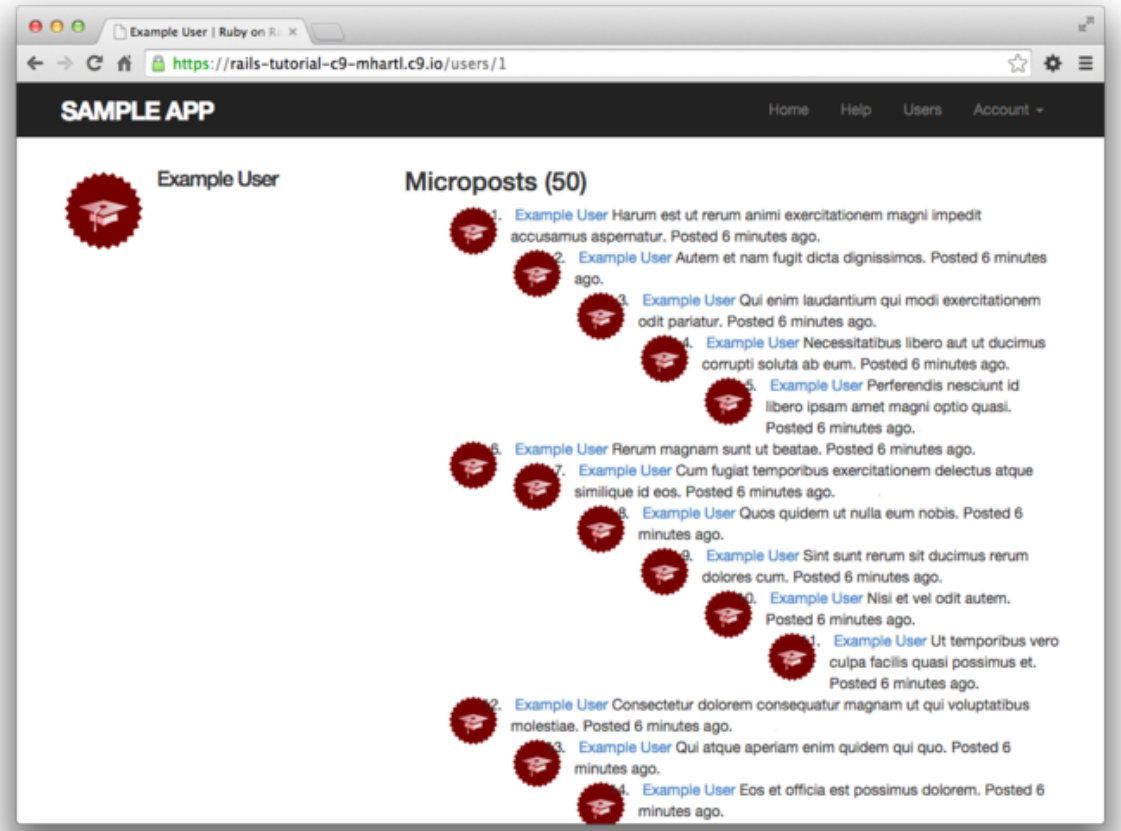

Figure 13.6: The user profile with unstyled microposts.

The page shown in Figure 13.6 has no micropost-specific styling, so let's add some (Listing 13.26) and take a look at the resulting pages.[8]

Listing 13.26: The CSS for microposts (including all the CSS for this chapter).
app/assets/stylesheets/custom.scss

```
.
.
.
/* microposts */

.microposts {
list-style: none;
padding: 0;
li {
padding: 10px 0;
border-top: 1px solid #e8e8e8;
}
```

```
.user {
margin-top: 5em;
padding-top: 0;
}
.content {
display: block;
margin-left: 60px;
img {
display: block;
padding: 5px 0;
}
}
.timestamp {
color: $gray-light;
display: block;
margin-left: 60px;
}
.gravatar {
float: left;
margin-right: 10px;
margin-top: 5px;
}
}

aside {
textarea {
height: 100px;
margin-bottom: 5px;
}
}

span.picture {
margin-top: 10px;
input {
border: 0;
}
}
```

Figure 13.7 shows the user profile page for the first user, while Figure 13.8 shows the profile for a second user. Finally, Figure 13.9 shows the *second* page of microposts for the first user, along with the pagination links at the bottom of the display. In all three cases, observe that each micropost display indicates the time since it was created (e.g., “Posted 1 minute ago.”); this is the work of the `time_ago_in_words` method from Listing 13.22. If you wait a couple of minutes and reload the pages, you’ll see how the text gets automatically updated based on the new time.

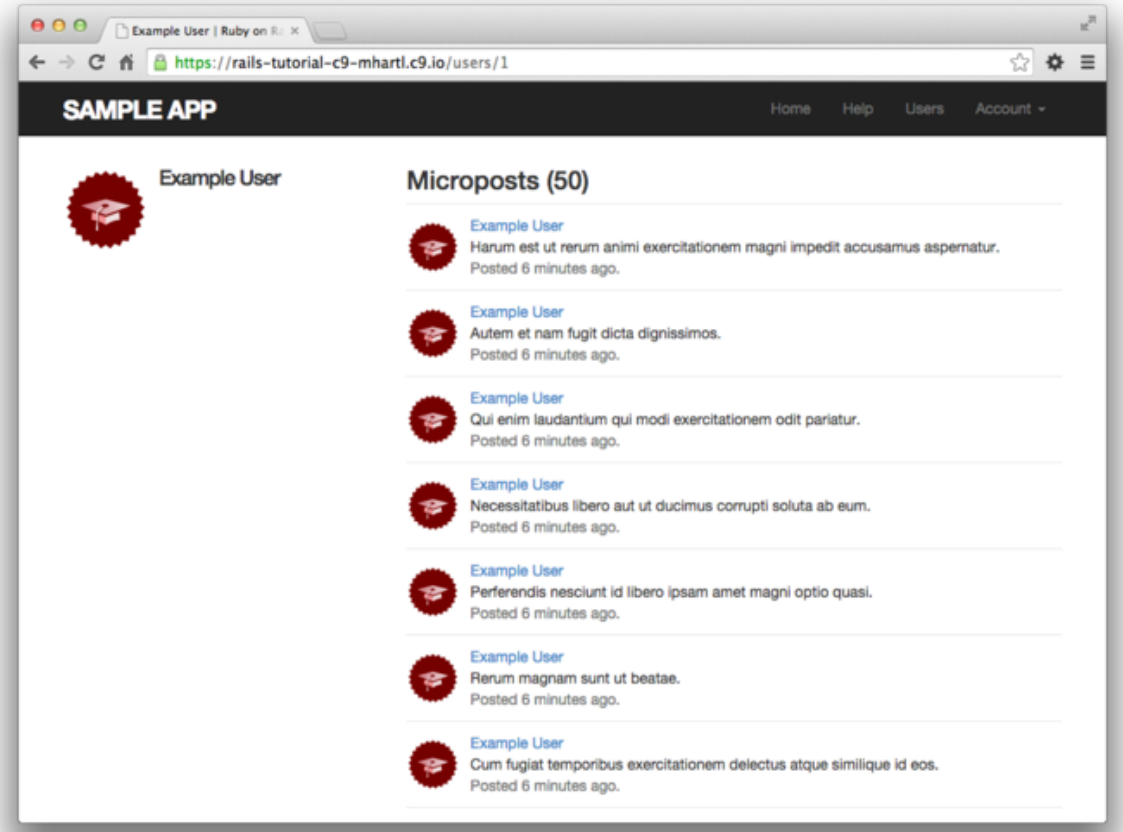

Figure 13.7: The user profile with microposts (/users/1).

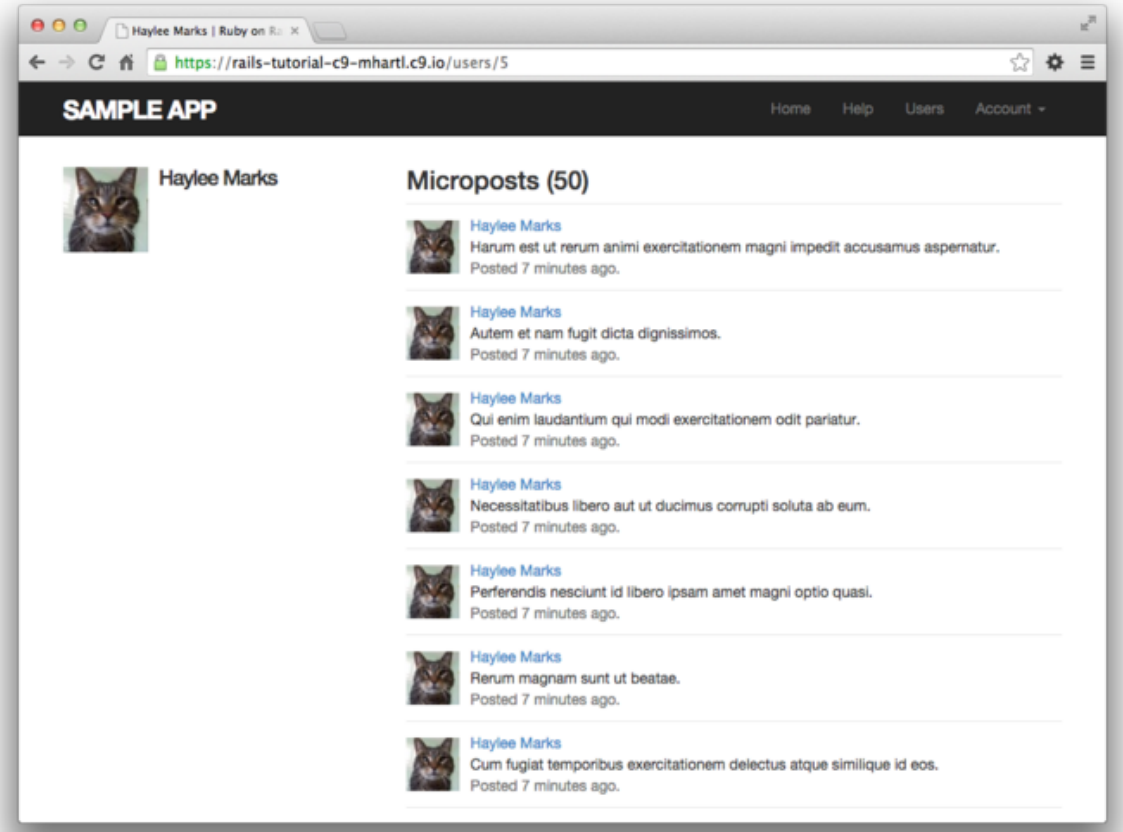

Figure 13.8: The profile of a different user, also with microposts (/users/5).

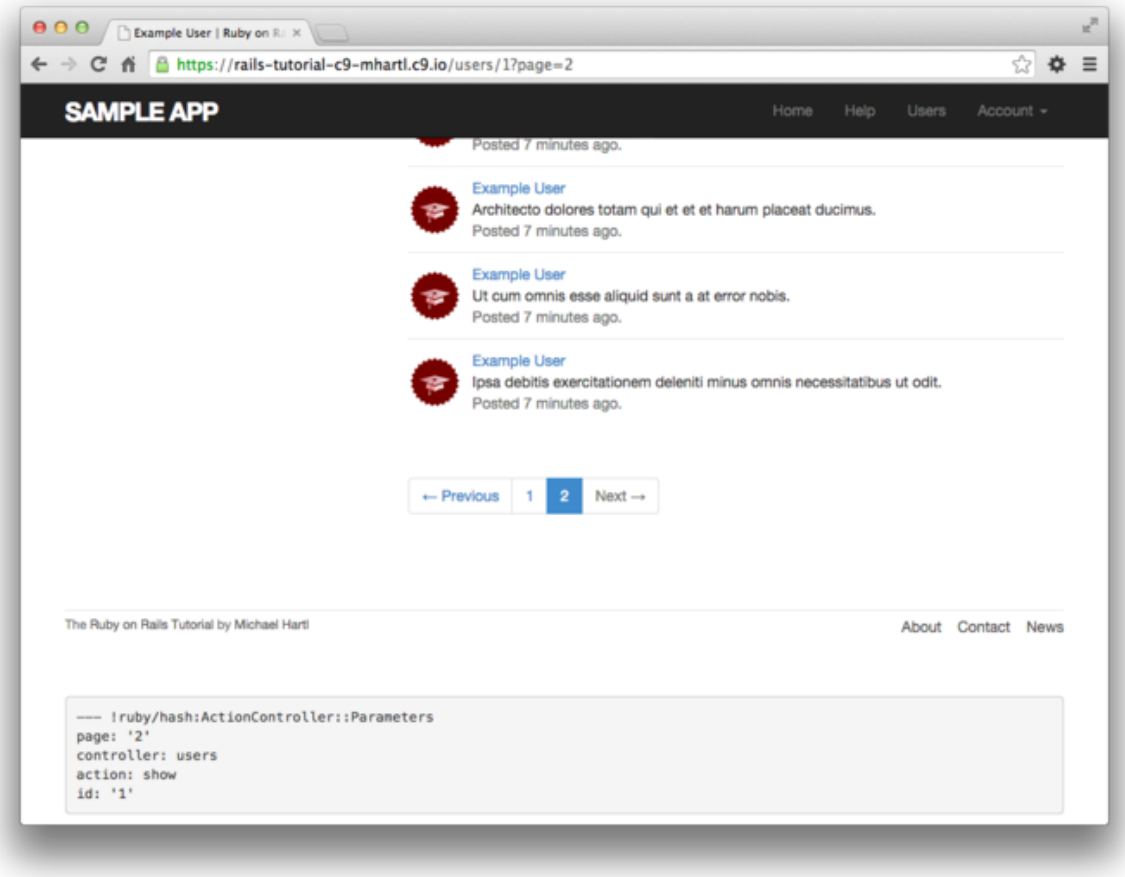

Figure 13.9: Micropost pagination links (/users/1?page=2).

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. See if you can guess the result of running `(1..10).to_a.take(6)`. Check at the console to see if your guess is right.
2. Is the `to_a` method in the previous exercise necessary?
3. Faker has a huge number of occasionally amusing applications. By consulting the Faker documentation, learn how to print out a fake university name, a fake phone number, a fake Hipster Ipsum sentence, and a fake Chuck Norris fact.

13.2.3 Profile micropost tests

Because newly activated users get redirected to their profile pages, we already have a test that the profile page renders correctly (Listing 11.33). In this section, we'll write a short integration test for some of the other elements on the profile page, including the work from this section. We'll start by generating an integration test for the profiles of our site's users:

```
$ rails generate integration_test users_profile
      invoke  test_unit
      create    test/integration/users_profile_test.rb
```

To test the micropost display on the profile, we need to associate the fixture microposts with a user. Rails includes a convenient way to build associations in fixtures, like this:

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

By identifying the `user` as `michael`, we tell Rails to associate this micropost with the corresponding user in the users fixture:

```
michael:
  name: Michael Example
  email: michael@example.com
.
.
.
```

To test micropost pagination, we'll also generate some additional micropost fixtures using the same embedded Ruby technique we used to make additional users in Listing 10.47:

```
<% 30.times do |n| %>
micropost_<%= n %>:
  content: <%= Faker::Lorem.sentence(5) %>
  created_at: <%= 42.days.ago %>
  user: michael
<% end %>
```

Putting all this together gives the updated micropost fixtures in Listing 13.27.

Listing 13.27: Micropost fixtures with user associations. `test/fixtures/microposts.yml`

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

tau_manifesto:
  content: "Check out the @tauday site by @mhartl: http://tauday.com"
  created_at: <%= 3.years.ago %>
  user: michael

cat_video:
  content: "Sad cats are sad: http://youtu.be/PKffm2uI4dk"
  created_at: <%= 2.hours.ago %>
  user: michael

most_recent:
  content: "Writing a short test"
  created_at: <%= Time.zone.now %>
  user: michael

<% 30.times do |n| %>
micropost_<%= n %>:
  content: <%= Faker::Lorem.sentence(5) %>
  created_at: <%= 42.days.ago %>
  user: michael
<% end %>
```

With the test data thus prepared, the test itself is fairly straightforward: we visit the user profile page and check for the page title and the user's name, Gravatar, micropost count, and paginated microposts. The result appears in Listing 13.28. Note the use of the `full_title` helper from Listing 4.2 to test the page's title, which we gain access to by including the Application Helper module into the test.[9]

Listing 13.28: A test for the user profile. **green** `test/integration/users_profile_test.rb`

```
require 'test_helper'

class UsersProfileTest < ActionDispatch::IntegrationTest
  include ApplicationHelper

  def setup
    @user = users(:michael)
  end

  test "profile display" do
    get user_path(@user)
    assert_template 'users/show'
    assert_select 'title', full_title(@user.name)
    assert_select 'h1', text: @user.name
    assert_select 'h1>img.gravatar'
    assert_match @user.microposts.count.to_s, response.body
```

```
  assert_select 'div.pagination'
@user.microposts.paginate(page: 1).each do |micropost|
  assert_match micropost.content, response.body
  end
 end
end
```

The micropost count assertion in Listing 13.28 uses `response.body`, which we saw briefly in the Chapter 12 exercises (Section 12.3.3.1). Despite its name, `response.body` contains the full HTML source of the page (and not just the page's body). This means that if all we care about is that the number of microposts appears *somewhere* on the page, we can look for a match as follows:

```
assert_match @user.microposts.count.to_s, response.body
```

This is a much less specific assertion than `assert_select`; in particular, unlike `assert_select`, using `assert_match` in this context doesn't require us to indicate which HTML tag we're looking for.

Listing 13.28 also introduces the nesting syntax for `assert_select`:

```
assert_select 'h1>img.gravatar'
```

This checks for an `img` tag with class `gravatar` *inside* a top-level heading tag (`h1`).

Because the application code was working, the test suite should be **green**:

Listing 13.29: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Comment out the application code needed to change the two `'h1'` lines in Listing 13.28 from **green** to **red**.
2. Update Listing 13.28 to test that `will_paginate` appears only *once*. *Hint*: Refer to Table 5.2.

13.3 Manipulating microposts

Having finished both the data modeling and display templates for microposts, we now turn our attention to the interface for creating them through the web. In this section, we'll also see the first hint of a *status feed*—a notion brought to full fruition in Chapter 14. Finally, as with users, we'll make it possible to destroy microposts through the web.

There is one break with past convention worth noting: the interface to the Microposts resource will run principally through the Profile and Home pages, so we won't need actions like `new` or `edit` in the Microposts controller; we'll need only `create` and `destroy`. This leads to the routes for the Microposts resource shown in Listing 13.30. The code in Listing 13.30 leads in turn to the RESTful routes shown in Table 13.2, which is a small subset of the full set of routes seen in Table 2.3. Of course, this simplicity is a sign of being *more* advanced, not less—we've come a long way since our reliance on scaffolding in Chapter 2, and we no longer need most of its complexity.

Listing 13.30: Routes for the Microposts resource. `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
  resources :account_activations, only: [:edit]
resources :password_resets,     only: [:new, :create, :edit, :update]
resources :microposts,          only: [:create, :destroy]
end
```

HTTP request URL	Action	Named route

POST	/microposts	create	microposts_path
DELETE	/microposts/1	destroy	micropost_path(micropost)

Table 13.2: RESTful routes provided by the Microposts resource in Listing 13.30.

13.3.1 Micropost access control

We begin our development of the Microposts resource with some access control in the Microposts controller. In particular, because we access microposts through their associated users, both the `create` and `destroy` actions must require users to be logged in.

Tests to enforce logged-in status mirror those for the Users controller (Listing 10.20 and Listing 10.61). We simply issue the correct request to each action and confirm that the micropost count is unchanged and the result is redirected to the login URL, as seen in Listing 13.31.

Listing 13.31: Authorization tests for the Microposts controller. **red**
`test/controllers/microposts_controller_test.rb`

```
require 'test_helper'

class MicropostsControllerTest < ActionDispatch::IntegrationTest

  def setup
    @micropost = microposts(:orange)
  end

  test "should redirect create when not logged in" do
    assert_no_difference 'Micropost.count' do
      post microposts_path, params: { micropost: { content: "Lorem ipsum" } }
    end
    assert_redirected_to login_url
  end

  test "should redirect destroy when not logged in" do
    assert_no_difference 'Micropost.count' do
      delete micropost_path(@micropost)
    end
    assert_redirected_to login_url
  end
end
```

Writing the application code needed to get the tests in Listing 13.31 to pass requires a little refactoring first. Recall from Section 10.2.1 that we enforced the login requirement using a before filter that called the `logged_in_user` method (Listing 10.15). At the time, we needed that method only in the Users controller, but now we find that we need it in the Microposts controller as well, so we'll move it into the Application controller, which is the base class of all controllers (Section 4.4.4).[10] The result appears in Listing 13.32.

Listing 13.32: Moving the `logged_in_user` method into the Application controller. **red**
`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper

  private

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

To avoid code repetition, you should also remove `logged_in_user` from the Users controller at this time (Listing 13.33).

Listing 13.33: The Users controller with the logged-in user filter removed. **red**
app/controllers/users_controller.rb

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

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

# Before filters

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

# Confirms an admin user.
def admin_user
redirect_to(root_url) unless current_user.admin?
end
end
```

With the code in Listing 13.32, the `logged_in_user` method is now available in the Microposts controller, which means that we can add `create` and `destroy` actions and then restrict access to them using a before filter, as shown in Listing 13.34.

Listing 13.34: Adding authorization to the Microposts controller actions. **green**
app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController
before_action :logged_in_user, only: [:create, :destroy]

def create
end

def destroy
end
end
```

At this point, the tests should pass:

Listing 13.35: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Why is it a bad idea to leave a copy of `logged_in_user` in the Users controller?

13.3.2 Creating microposts

In Chapter 7, we implemented user signup by making an HTML form that issued an HTTP `POST` request to the `create` action in the Users controller. The implementation of micropost creation is similar; the main difference is that, rather than using a separate page at /microposts/new, we will put the form on the Home page itself (i.e., the root path /), as mocked up in Figure 13.10.

Figure 13.10: A mockup of the Home page with a form for creating microposts.

When we last left the Home page, it appeared as in Figure 5.8—that is, it had a “Sign up now!” button in the middle. Since a micropost creation form makes sense only in the context of a particular logged-in user, one goal of this section will be to serve different versions of the Home page depending on a visitor’s login status. We’ll implement this in Listing 13.37 below.

We’ll start with the `create` action for microposts, which is similar to its user analogue (Listing 7.28); the principal difference lies in using the user/micropost association to `build` the new micropost, as seen in Listing 13.36. Note the use of strong parameters via `micropost_params`, which permits only the micropost’s `content` attribute to be modified through the web.

Listing 13.36: The Microposts controller `create` action. `app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
before_action :logged_in_user, only: [:create, :destroy]

def create
@micropost = current_user.microposts.build(micropost_params)
if @micropost.save
flash[:success] = "Micropost created!"
redirect_to root_url
else
render 'static_pages/home'
end
end

def destroy
end

private

def micropost_params
params.require(:micropost).permit(:content)
```

```
  end
end
```

To build a form for creating microposts, we use the code in Listing 13.37, which serves up different HTML based on whether the site visitor is logged in or not.

Listing 13.37: Adding microposts creation 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="micropost_form">
<%= render 'shared/micropost_form' %>
</section>
</aside>
</div>
<% else %>
<div class="center jumbotron">
<h1>Welcome to the Sample App</h1>

<h2>
This is the home page for the
<a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
sample application.
</h2>

<%= link_to "Sign up now!", signup_path, class: "btn btn-lg btn-primary" %>
</div>

<%= link_to image_tag("rails.png", alt: "Rails logo"),
'http://rubyonrails.org/' %>
<% end %>
```

(Having so much code in each branch of the `if-else` conditional is a bit messy, and cleaning it up using partials is left as an exercise (Section 13.3.2.1).)

To get the page defined in Listing 13.37 working, we need to create and fill in a couple of partials. The first is the new Home page sidebar, as shown in Listing 13.38.

Listing 13.38: The partial for the user info sidebar. `app/views/shared/_user_info.html.erb`

```
<%= link_to gravatar_for(current_user, size: 50), current_user %>
<h1><%= current_user.name %></h1>
<span><%= link_to "view my profile", current_user %></span>
<span><%= pluralize(current_user.microposts.count, "micropost") %></span>
```

Note that, as in the profile sidebar (Listing 13.24), the user info in Listing 13.38 displays the total number of microposts for the user. There's a slight difference in the display, though; in the profile sidebar, "Microposts" is a label, and showing "Microposts (1)" makes sense. In the present case, though, saying "1 microposts" is ungrammatical, so we arrange to display "1 micropost" and "2 microposts" using the `pluralize` method we saw in Section 7.3.3.

We next define the form for creating microposts (Listing 13.39), which is similar to the signup form in Listing 7.15.

Listing 13.39: The form partial for creating microposts. `app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost) do |f| %>
<%= render 'shared/error_messages', object: f.object %>
<div class="field">
<%= f.text_area :content, placeholder: "Compose new micropost..." %>
</div>
<%= f.submit "Post", class: "btn btn-primary" %>
<% end %>
```

We need to make two changes before the form in Listing 13.39 will work. First, we need to define `@micropost`, which (as before) we do through the association:

```
@micropost = current_user.microposts.build
```

The result appears in Listing 13.40.

Listing 13.40: Adding a micropost instance variable to the `home` action.
`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

def home
@micropost = current_user.microposts.build if logged_in?
end

def help
end

def about
end

def contact
end
end
```

Of course, `current_user` exists only if the user is logged in, so the `@micropost` variable should only be defined in this case.

The second change needed to get Listing 13.39 to work is to redefine the error-messages partial so the following code from Listing 13.39 works:

```
<%= render 'shared/error_messages', object: f.object %>
```

You may recall from Listing 7.20 that the error-messages partial references the `@user` variable explicitly, but in the present case we have an `@micropost` variable instead. To unify these cases, we can pass the form variable `f` to the partial and access the associated object through `f.object`, so that in

```
form_for(@user) do |f|
```

`f.object` is `@user`, and in

```
form_for(@micropost) do |f|
```

`f.object` is `@micropost`, etc.

To pass the object to the partial, we use a hash with value equal to the object and key equal to the desired name of the variable in the partial, which is what the second line in Listing 13.39 accomplishes. In other words, `object: f.object` creates a variable called `object` in the `error_messages` partial, and we can use it to construct a customized error message, as shown in Listing 13.41.

Listing 13.41: Error messages that work with other objects. **red** `app/views/shared/_error_messages.html.erb`

```
<% if object.errors.any? %>
<div id="error_explanation">
<div class="alert alert-danger">
The form contains <%= pluralize(object.errors.count, "error") %>.
</div>
<ul>
<% object.errors.full_messages.each do |msg| %>
<li><%= msg %></li>
<% end %>
</ul>
</div>
<% end %>
```

At this point, you should verify that the test suite is **red**:

Listing 13.42: **red**

```
$ rails test
```

This is a hint that we need to update the other occurrences of the error-messages partial, which we used when signing up users (Listing 7.20), resetting passwords (Listing 12.14), and editing users (Listing 10.2). The updated versions are shown in Listing 13.43, Listing 13.45, and Listing 13.44.

Listing 13.43: Updating the rendering of user signup errors. `app/views/users/new.html.erb`

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

<div class="row">
<div class="col-md-6 col-md-offset-3">
<%= form_for(@user) do |f| %>
<%= render 'shared/error_messages', object: f.object %>
<%= f.label :name %>
<%= f.text_field :name, class: 'form-control' %>

<%= f.label :email %>
<%= f.email_field :email, class: 'form-control' %>

<%= f.label :password %>
<%= f.password_field :password, class: 'form-control' %>

<%= f.label :password_confirmation, "Confirmation" %>
<%= f.password_field :password_confirmation, class: 'form-control' %>

<%= f.submit "Create my account", class: "btn btn-primary" %>
<% end %>
</div>
</div>
```

Listing 13.44: Updating the errors for editing users. `app/views/users/edit.html.erb`

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

<div class="row">
<div class="col-md-6 col-md-offset-3">
<%= form_for(@user) do |f| %>
<%= render 'shared/error_messages', object: f.object %>

<%= f.label :name %>
<%= f.text_field :name, class: 'form-control' %>

<%= f.label :email %>
<%= f.email_field :email, class: 'form-control' %>

<%= f.label :password %>
<%= f.password_field :password, class: 'form-control' %>

<%= f.label :password_confirmation, "Confirmation" %>
<%= f.password_field :password_confirmation, class: 'form-control' %>

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

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

Listing 13.45: Updating the errors for password resets. `app/views/password_resets/edit.html.erb`

```
<% provide(:title, 'Reset password') %>
<h1>Password reset</h1>

<div class="row">
<div class="col-md-6 col-md-offset-3">
<%= form_for(@user, url: password_reset_path(params[:id])) do |f| %>
<%= render 'shared/error_messages', object: f.object %>

<%= hidden_field_tag :email, @user.email %>

<%= f.label :password %>
<%= f.password_field :password, class: 'form-control' %>
```

```
<%= f.label :password_confirmation, "Confirmation" %>
<%= f.password_field :password_confirmation, class: 'form-control' %>

<%= f.submit "Update password", class: "btn btn-primary" %>
<% end %>
</div>
</div>
```

At this point, all the tests should be **green**:

```
$ rails test
```

Additionally, all the HTML in this section should render properly, showing the form as in Figure 13.11, and a form with a submission error as in Figure 13.12.

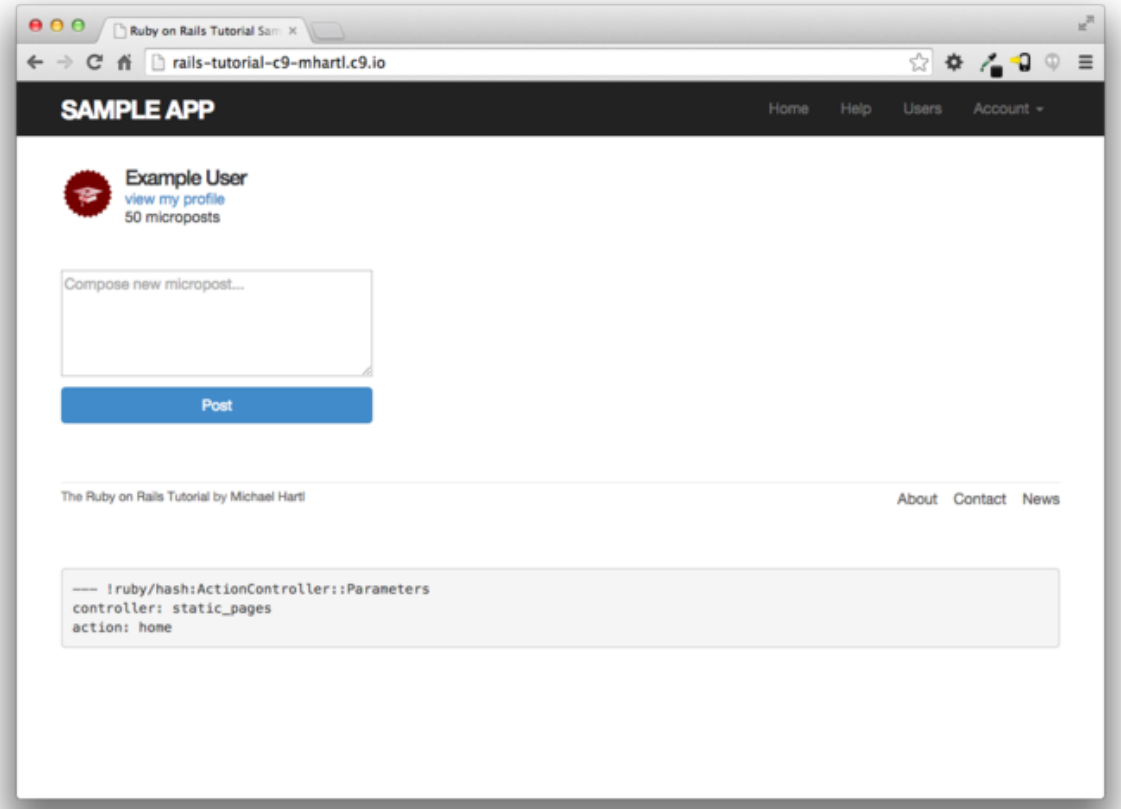

Figure 13.11: The Home page with a new micropost form.

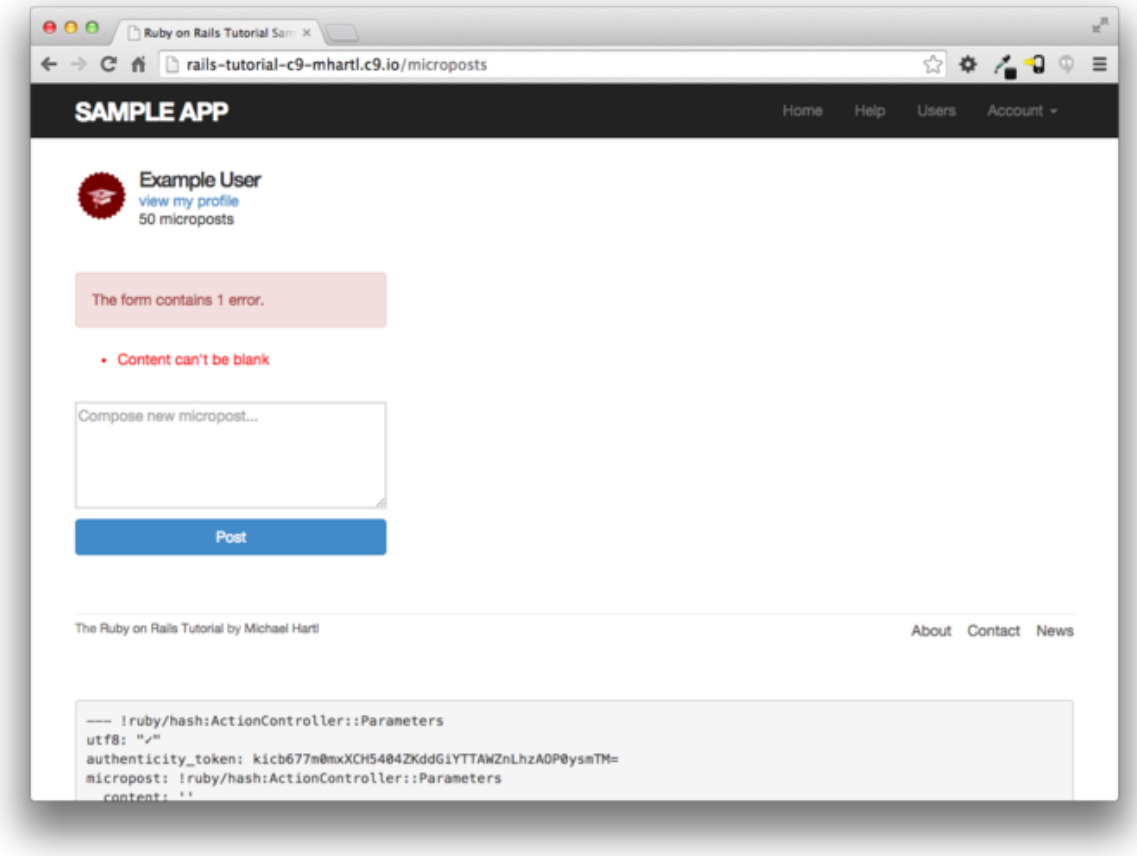

Figure 13.12: The Home page with a form error.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Refactor the Home page to use separate partials for the two branches of the `if-else` statement.

13.3.3 A proto-feed

Although the micropost form is actually now working, users can't immediately see the results of a successful submission because the current Home page doesn't display any microposts. If you like, you can verify that the form shown in Figure 13.11 is working by submitting a valid entry and then navigating to the profile page to see the post, but that's rather cumbersome. It would be far better to have a *feed* of microposts that includes the user's own posts, as mocked up in Figure 13.13. (In Chapter 14, we'll generalize this feed to include the microposts of users being *followed* by the current user, à la Twitter.)

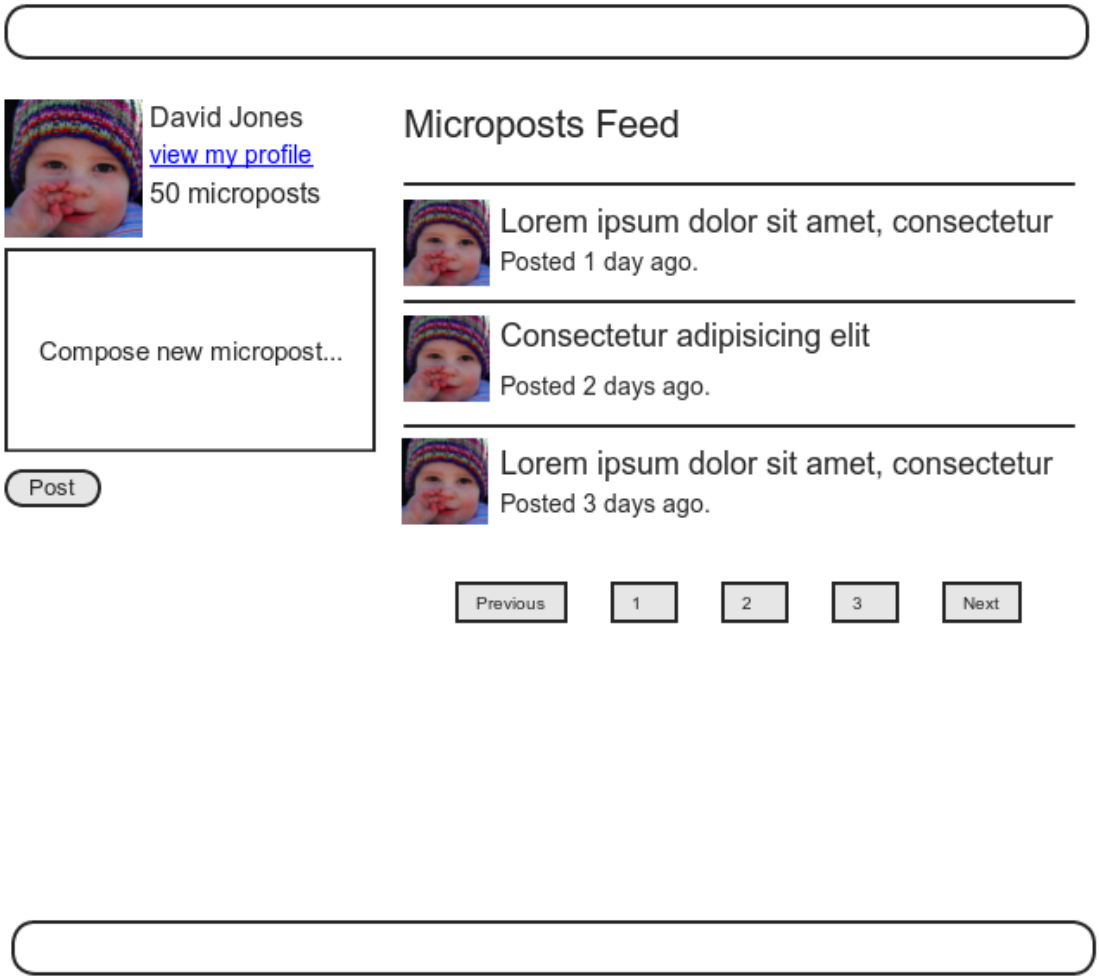

Figure 13.13: A mockup of the Home page with a proto-feed.

Since each user should have a feed, we are led naturally to a `feed` method in the User model, which will initially just select all the microposts belonging to the current user. We'll accomplish this using the `where` method on the `Micropost` model (seen briefly before in Section 11.3.3.1), as shown in Listing 13.46.[11]

Listing 13.46: A preliminary implementation for the micropost status feed. `app/models/user.rb`

```
class User < ApplicationRecord
                .
                .
                .
        # Defines a proto-feed.
# See "Following users" for the full implementation.
                def feed
        Micropost.where("user_id = ?", id)
                end

                private
                   .
                   .
                   .
                end
```

The question mark in

```
Micropost.where("user_id = ?", id)
```

ensures that `id` is properly *escaped* before being included in the underlying SQL query, thereby avoiding a serious security hole called *SQL injection*. The `id` attribute here is just an integer (i.e., `self.id`, the unique ID of the user), so there is no danger of SQL injection in this case, but it does no harm, and *always* escaping variables injected into SQL statements is a good habit to cultivate.

Alert readers might note at this point that the code in Listing 13.46 is essentially equivalent to writing

```
def feed
microposts
end
```

We've used the code in Listing 13.46 instead because it generalizes much more naturally to the full status feed needed in Chapter 14.

To use the feed in the sample application, we add an `@feed_items` instance variable for the current user's (paginated) feed, as in Listing 13.47, and then add a status feed partial (Listing 13.48) to the Home page (Listing 13.49). Note that, now that there are two lines that need to be run when the user is logged in, Listing 13.47 changes

```
@micropost = current_user.microposts.build if logged_in?
```

from Listing 13.40 to

```
if logged_in?
@micropost  = current_user.microposts.build
@feed_items = current_user.feed.paginate(page: params[:page])
end
```

thereby moving the conditional from the end of the line to an if-end statement.

Listing 13.47: Adding a feed instance variable to the `home` action. `app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

def home
if logged_in?
@micropost  = current_user.microposts.build
@feed_items = current_user.feed.paginate(page: params[:page])
end
end

def help
end

def about
end

def contact
end
end
```

Listing 13.48: The status feed partial. `app/views/shared/_feed.html.erb`

```
<% if @feed_items.any? %>
<ol class="microposts">
<%= render @feed_items %>
</ol>
<%= will_paginate @feed_items %>
<% end %>
```

The status feed partial defers the rendering to the micropost partial defined in Listing 13.22:

```
<%= render @feed_items %>
```

Here Rails knows to call the micropost partial because each element of `@feed_items` has class `Micropost`. This causes Rails to look for a partial with the corresponding name in the views directory of the given resource:

```
app/views/microposts/_micropost.html.erb
```

We can add the feed to the Home page by rendering the feed partial as usual (Listing 13.49). The result is a display of the feed on the Home page, as required (Figure 13.14).

Listing 13.49: Adding a status feed 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="micropost_form">
<%= render 'shared/micropost_form' %>
</section>
</aside>
<div class="col-md-8">
<h3>Micropost Feed</h3>
<%= render 'shared/feed' %>
</div>
</div>
<% else %>
.
.
.
<% end %>
```

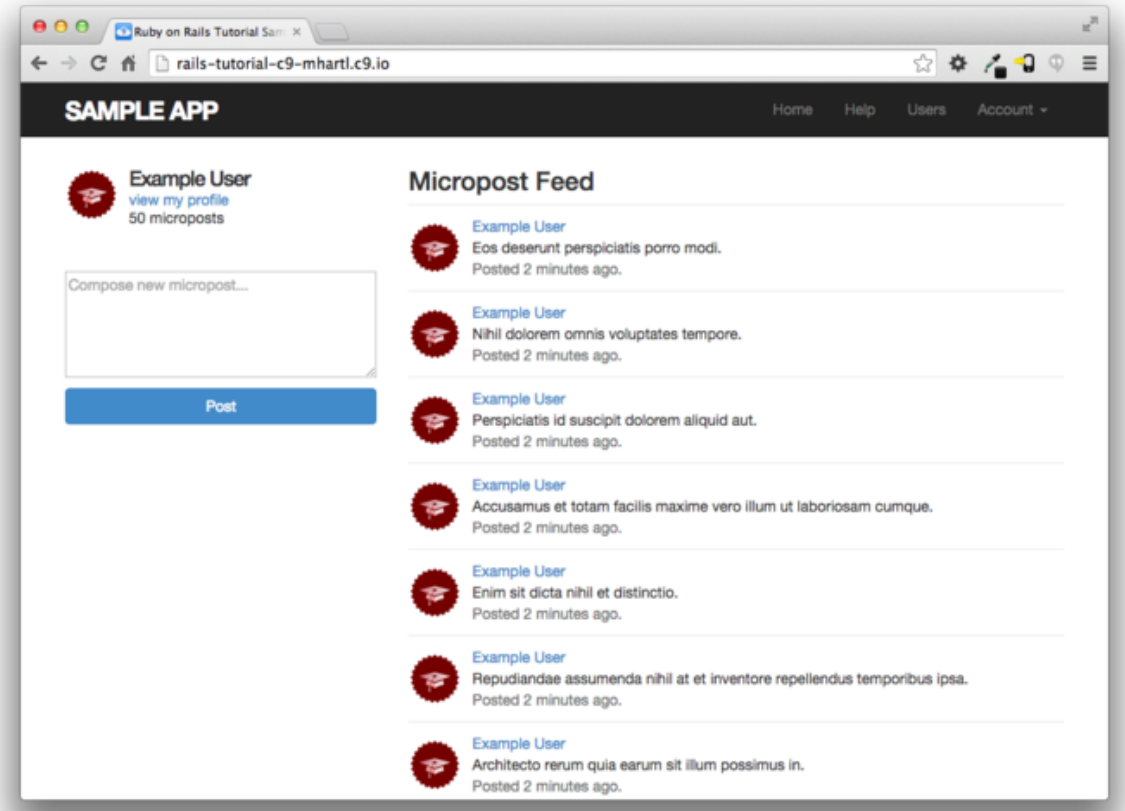

Figure 13.14: The Home page with a proto-feed.

At this point, creating a new micropost works as expected, as seen in Figure 13.15. There is one subtlety, though: on *failed* micropost submission, the Home page expects an `@feed_items` instance variable, so failed submissions currently break. The easiest solution is to suppress the feed entirely by assigning it an empty array, as shown in Listing 13.50. (Unfortunately, returning a paginated feed doesn't work in this case. Implement it and click on a pagination link to see why.)

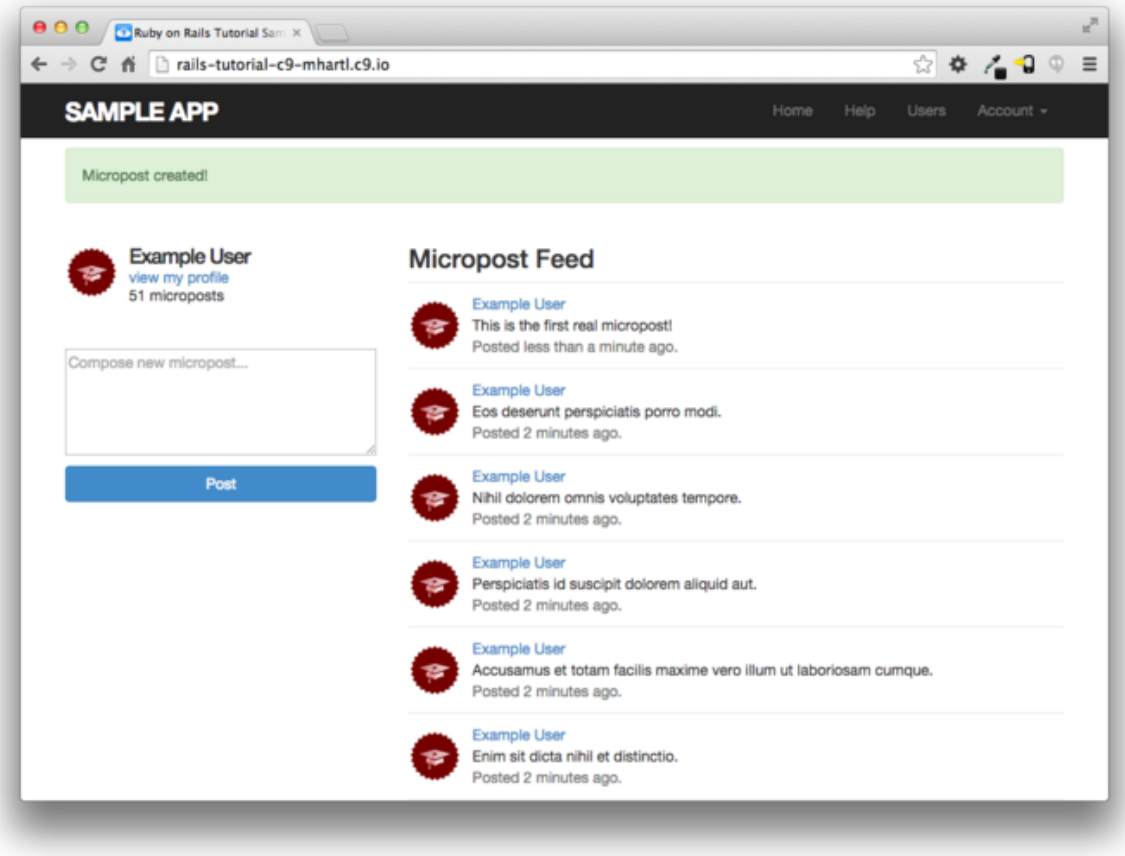

Figure 13.15: The Home page after creating a new micropost.

Listing 13.50: Adding an (empty) `@feed_items` instance variable to the `create` action.
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
before_action :logged_in_user, only: [:create, :destroy]

def create
@micropost = current_user.microposts.build(micropost_params)
if @micropost.save
flash[:success] = "Micropost created!"
redirect_to root_url
else
@feed_items = []
render 'static_pages/home'
end
end

def destroy
end

private

def micropost_params
params.require(:micropost).permit(:content)
end
end
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Use the newly created micropost UI to create the first real micropost. What are the contents of the `INSERT` command in the server log?
2. In the console, set `user` to the first user in the database. Confirm that the values of by `Micropost.where("user_id = ?", user.id)`, `user.microposts`, and `user.feed` are all the same. *Hint*:

It's probably easiest to compare directly using ==.

13.3.4 Destroying microposts

The last piece of functionality to add to the Microposts resource is the ability to destroy posts. As with user deletion (Section 10.4.2), we accomplish this with "delete" links, as mocked up in Figure 13.16. Unlike that case, which restricted user destruction to admin users, the delete links will work only for microposts created by the current user.

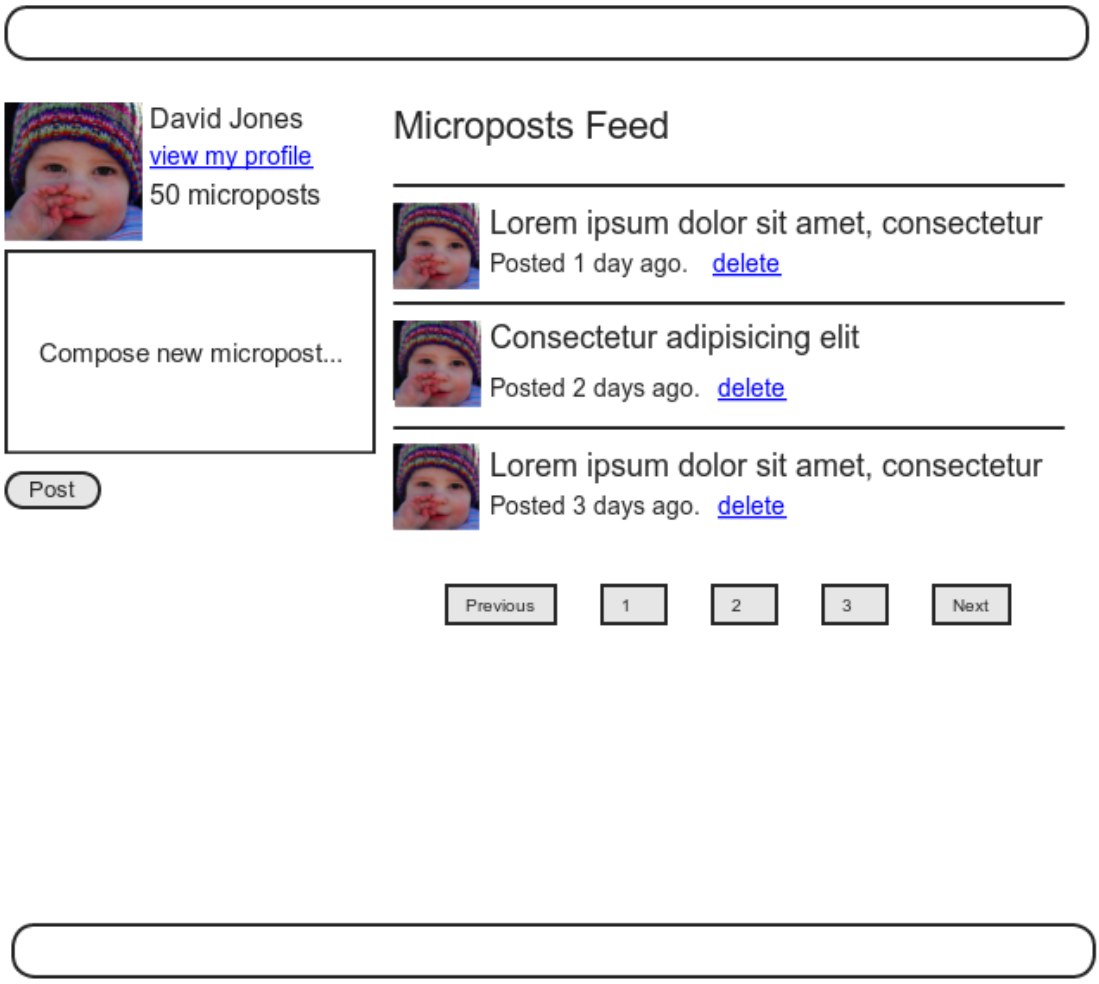

Figure 13.16: A mockup of the proto-feed with micropost delete links.

Our first step is to add a delete link to the micropost partial as in Listing 13.22. The result appears in Listing 13.51.

Listing 13.51: Adding a delete link to the micropost partial. `app/views/microposts/_micropost.html.erb`

```
<li id="<%= micropost.id %>">
<%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
<span class="user"><%= link_to micropost.user.name, micropost.user %></span>
<span class="content"><%= micropost.content %></span>
<span class="timestamp">
Posted <%= time_ago_in_words(micropost.created_at) %> ago.
<% if current_user?(micropost.user) %>
<%= link_to "delete", micropost, method: :delete,
data: { confirm: "You sure?" } %>
<% end %>
</span>
</li>
```

The next step is to define a `destroy` action in the Microposts controller, which is analogous to the user case in Listing 10.59. The main difference is that, rather than using an `@user` variable with an `admin_user` before filter, we'll find the micropost through the association, which will automatically fail if a user tries to delete another user's micropost. We'll put the resulting `find` inside a `correct_user` before filter, which checks that the current user actually has a micropost with the given id. The result appears in Listing 13.52.

Listing 13.52: The Microposts controller `destroy` action. `app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
before_action :logged_in_user, only: [:create, :destroy]
before_action :correct_user,   only: :destroy
.
.
.
def destroy
@micropost.destroy
flash[:success] = "Micropost deleted"
redirect_to request.referrer || root_url
end

private

def micropost_params
params.require(:micropost).permit(:content)
end

def correct_user
@micropost = current_user.microposts.find_by(id: params[:id])
redirect_to root_url if @micropost.nil?
end
end
```

Note that the `destroy` method in Listing 13.52 redirects to the URL

```
request.referrer || root_url
```

This uses the `request.referrer` method,[12] which is related to the `request.original_url` variable used in friendly forwarding (Section 10.2.3), and is just the previous URL (in this case, the Home page).[13] This is convenient because microposts appear on both the Home page and on the user's profile page, so by using `request.referrer` we arrange to redirect back to the page issuing the delete request in both cases. If the referring URL is `nil` (as is the case inside some tests), Listing 13.52 sets the `root_url` as the default using the `||` operator. (Compare to the default options defined in Listing 9.24.)

With the code as above, the result of destroying the second-most recent post appears in Figure 13.17.

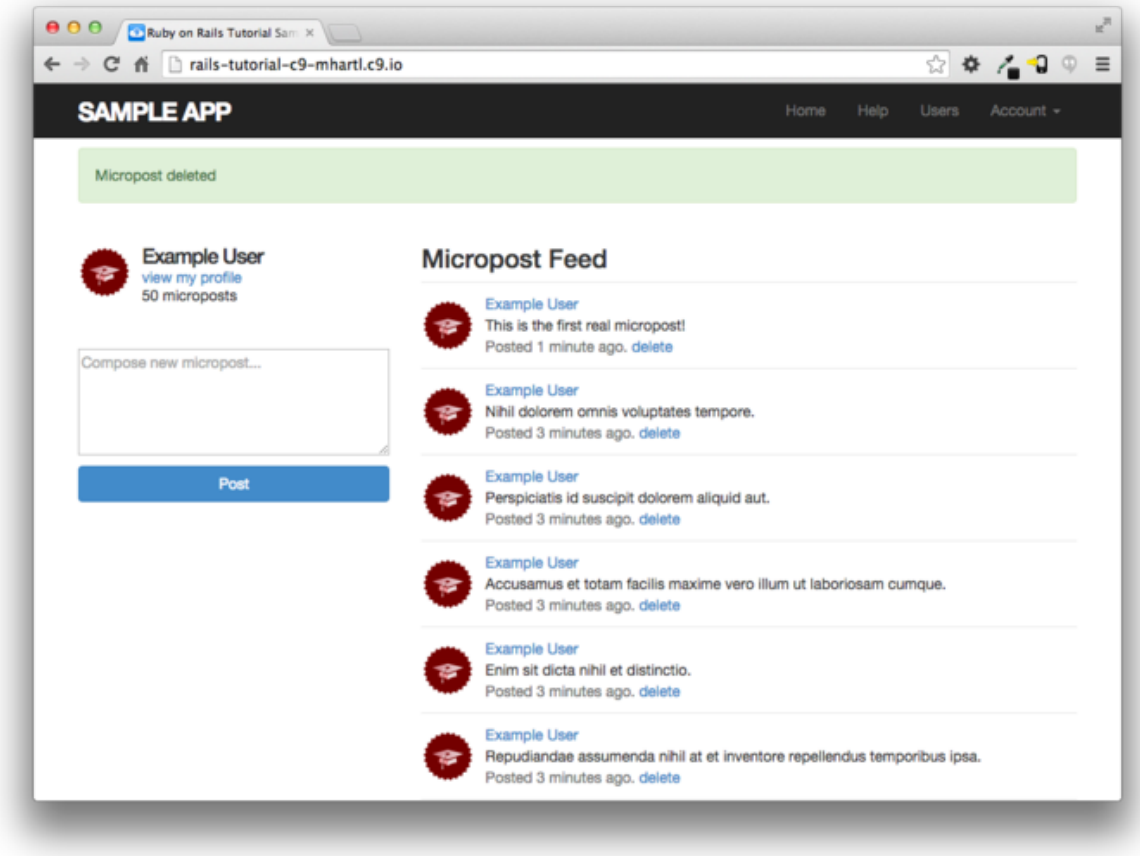

Figure 13.17: The Home page after deleting the second-most-recent micropost.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Create a new micropost and then delete it. What are the contents of the `DELETE` command in the server log?
2. Confirm directly in the browser that the line `redirect_to request.referrer || root_url` can be replaced with the line `redirect_back(fallback_location: root_url)`. (This method was added in Rails 5.)

13.3.5 Micropost tests

With the code in Section 13.3.4, the Micropost model and interface are complete. All that's left is writing a short Microposts controller test to check authorization and a micropost integration test to tie it all together.

We'll start by adding a few microposts with different owners to the micropost fixtures, as shown in Listing 13.53. (We'll be using only one for now, but we've put in the others for future reference.)

Listing 13.53: Adding a micropost with a different owner. `test/fixtures/microposts.yml`

```
.
.
.
ants:
  content: "Oh, is that what you want? Because that's how you get ants!"
  created_at: <%= 2.years.ago %>
  user: archer

zone:
  content: "Danger zone!"
  created_at: <%= 3.days.ago %>
  user: archer

tone:
  content: "I'm sorry. Your words made sense, but your sarcastic tone did not."
```

```
  created_at: <%= 10.minutes.ago %>
  user: lana

  van:
  content: "Dude, this van's, like, rolling probable cause."
  created_at: <%= 4.hours.ago %>
  user: lana
```

We next write a short test to make sure one user can't delete the microposts of a different user, and we also check for the proper redirect, as seen in Listing 13.54.

Listing 13.54: Testing micropost deletion with a user mismatch. **green**
test/controllers/microposts_controller_test.rb

```
require 'test_helper'

class MicropostsControllerTest < ActionDispatch::IntegrationTest

  def setup
  @micropost = microposts(:orange)
  end

  test "should redirect create when not logged in" do
  assert_no_difference 'Micropost.count' do
  post microposts_path, params: { micropost: { content: "Lorem ipsum" } }
  end
  assert_redirected_to login_url
  end

  test "should redirect destroy when not logged in" do
  assert_no_difference 'Micropost.count' do
  delete micropost_path(@micropost)
  end
  assert_redirected_to login_url
  end

  test "should redirect destroy for wrong micropost" do
  log_in_as(users(:michael))
  micropost = microposts(:ants)
  assert_no_difference 'Micropost.count' do
  delete micropost_path(micropost)
  end
  assert_redirected_to root_url
  end
  end
```

Finally, we'll write an integration test to log in, check the micropost pagination, make an invalid submission, make a valid submission, delete a post, and then visit a second user's page to make sure there are no "delete" links. We start by generating a test as usual:

```
$ rails generate integration_test microposts_interface
  invoke  test_unit
  create    test/integration/microposts_interface_test.rb
```

The test appears in Listing 13.55. See if you can connect the lines in Listing 13.12 to the steps mentioned above.

Listing 13.55: An integration test for the micropost interface. **green**
test/integration/microposts_interface_test.rb

```
require 'test_helper'

class MicropostsInterfaceTest < ActionDispatch::IntegrationTest

  def setup
  @user = users(:michael)
  end

  test "micropost interface" do
  log_in_as(@user)
  get root_path
  assert_select 'div.pagination'
```

```
# Invalid submission
assert_no_difference 'Micropost.count' do
post microposts_path, params: { micropost: { content: "" } }
end
assert_select 'div#error_explanation'
# Valid submission
content = "This micropost really ties the room together"
assert_difference 'Micropost.count', 1 do
post microposts_path, params: { micropost: { content: content } }
end
assert_redirected_to root_url
follow_redirect!
assert_match content, response.body
# Delete post
assert_select 'a', text: 'delete'
first_micropost = @user.microposts.paginate(page: 1).first
assert_difference 'Micropost.count', -1 do
delete micropost_path(first_micropost)
end
# Visit different user (no delete links)
get user_path(users(:archer))
assert_select 'a', text: 'delete', count: 0
end
end
```

Because we wrote working application code first, the test suite should be **green**:

Listing 13.56: **green**

```
$ rails test
```

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. For each of the four scenarios indicated by comments in Listing 13.55 (starting with "Invalid submission"), comment out application code to get the corresponding test to **red**, then uncomment to get back to **green**.
2. Add tests for the sidebar micropost count (including proper pluralization). Listing 13.57 will help get you started.

Listing 13.57: A template for the sidebar micropost count test.
`test/integration/microposts_interface_test.rb`

```
require 'test_helper'

class MicropostInterfaceTest < ActionDispatch::IntegrationTest

def setup
@user = users(:michael)
end
.
.
.
test "micropost sidebar count" do
log_in_as(@user)
get root_path
assert_match "#{FILL_IN} microposts", response.body
# User with zero microposts
other_user = users(:malory)
log_in_as(other_user)
get root_path
assert_match "0 microposts", response.body
other_user.microposts.create!(content: "A micropost")
get root_path
assert_match FILL_IN, response.body
end
end
```

13.4 Micropost images

Now that we've added support for all relevant micropost actions, in this section we'll make it possible for microposts to include images as well as text. We'll start with a basic version good enough for development use, and then add a series of enhancements to make image upload production-ready.

Adding image upload involves two main visible elements: a form field for uploading an image and the micropost images themselves. A mockup of the resulting "Upload image" button and micropost photo appears in Figure 13.18.[14]

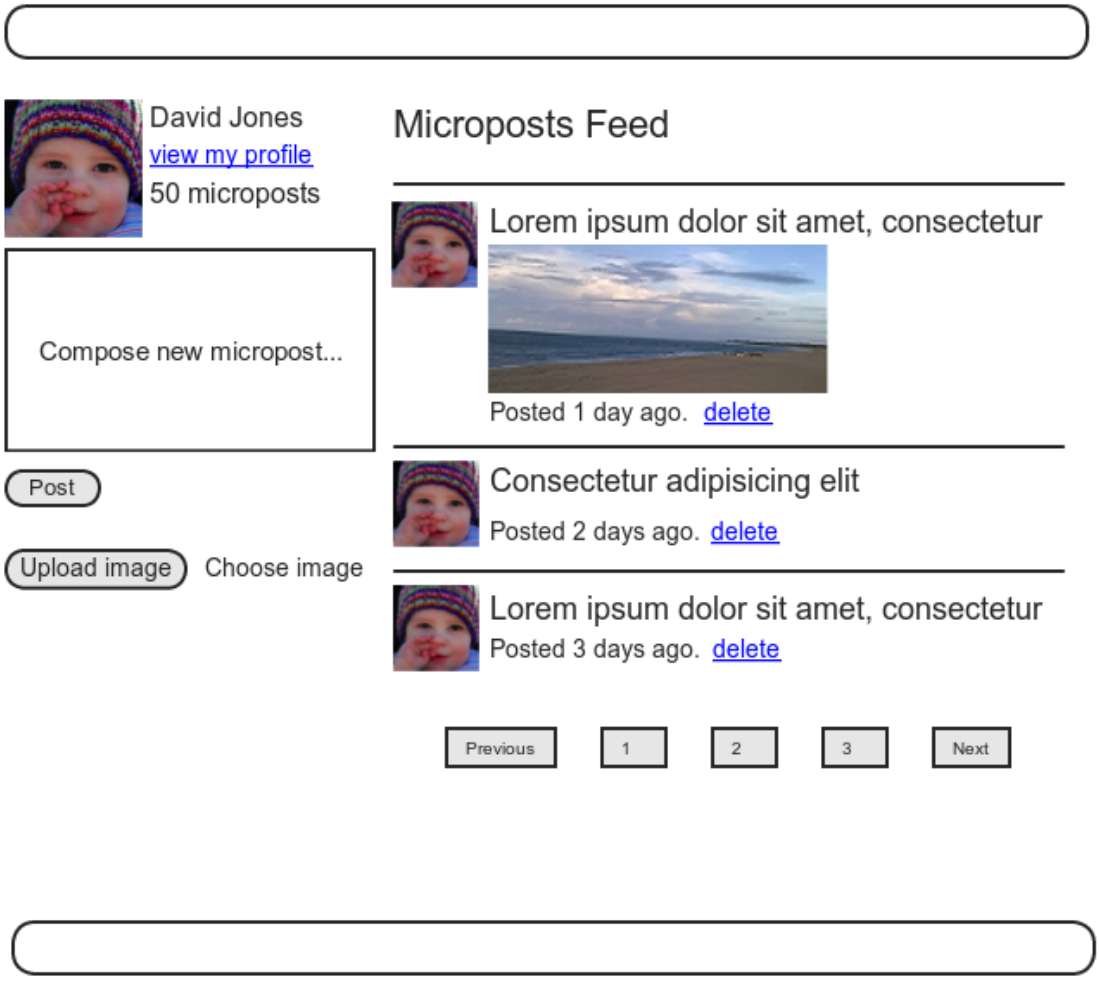

Figure 13.18: A mockup of micropost image upload (with an uploaded image).

13.4.1 Basic image upload

To handle an uploaded image and associate it with the Micropost model, we'll use the CarrierWave image uploader. To get started, we need to include the `carrierwave` and `mini_magick` gems in the `Gemfile` (Listing 13.58).[15] For completeness, Listing 13.58 also includes the `fog` gem needed for image upload in production (Section 13.4.4).

Listing 13.58: Adding CarrierWave to the `Gemfile`.

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

gem 'rails',                    '5.1.4'
gem 'bcrypt',                   '3.1.11'
gem 'faker',                    '1.7.3'
gem 'carrierwave',              '1.2.2'
gem 'mini_magick',              '4.7.0'
gem 'will_paginate',            '3.1.5'
gem 'bootstrap-will_paginate',  '1.0.0'
                    .
                    .
                    .
group :production do
  gem 'pg',   '0.20.0'
  gem 'fog', '1.42'
```

```
end
.
.
.
```

Then we install as usual:

```
$ bundle install
```

CarrierWave adds a Rails generator for creating an image uploader, which we'll use to make an uploader for an image called `picture`:[16]

```
$ rails generate uploader Picture
```

Images uploaded with CarrierWave should be associated with a corresponding attribute in an Active Record model, which simply contains the name of the image file in a string field. The resulting augmented data model for microposts appears in Figure 13.19.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime
picture	string

Figure 13.19: The Micropost data model with a `picture` attribute.

To add the required `picture` attribute to the Micropost model, we generate a migration and migrate the development database:

```
$ rails generate migration add_picture_to_microposts picture:string
$ rails db:migrate
```

The way to tell CarrierWave to associate the image with a model is to use the `mount_uploader` method, which takes as arguments a symbol representing the attribute and the class name of the generated uploader:

```
mount_uploader :picture, PictureUploader
```

(Here `PictureUploader` is defined in the file `picture_uploader.rb`, which we'll start editing in Section 13.4.2, but for now the generated default is fine.) Adding the uploader to the Micropost model gives the code shown in Listing 13.59.

Listing 13.59: Adding an image to the Micropost model. `app/models/micropost.rb`

```
class Micropost < ApplicationRecord
belongs_to :user
default_scope -> { order(created_at: :desc) }
mount_uploader :picture, PictureUploader
validates :user_id, presence: true
validates :content, presence: true, length: { maximum: 140 }
end
```

On some systems, you may need to restart the Rails server at this point to keep the test suite **green**. (If you're using Guard as described in Section 3.6.2, you may need to restart that as well, and it may even be necessary to exit the terminal shell and re-run Guard in a new one.)

To include the uploader on the Home page as in Figure 13.18, we need to include a `file_field` tag in the micropost form, as shown in Listing 13.60.[17]

Listing 13.60: Adding image upload to the micropost create form. `app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost) do |f| %>
<%= render 'shared/error_messages', object: f.object %>
<div class="field">
<%= f.text_area :content, placeholder: "Compose new micropost..." %>
```

```
</div>
<%= f.submit "Post", class: "btn btn-primary" %>
<span class="picture">
<%= f.file_field :picture %>
</span>
<% end %>
```

Finally, we need to add `picture` to the list of attributes permitted to be modified through the web. This involves editing the `micropost_params` method, as shown in Listing 13.61.

Listing 13.61: Adding `picture` to the list of permitted attributes. `app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
before_action :logged_in_user, only: [:create, :destroy]
before_action :correct_user,   only: :destroy
.
.
.
private

def micropost_params
params.require(:micropost).permit(:content, :picture)
end

def correct_user
@micropost = current_user.microposts.find_by(id: params[:id])
redirect_to root_url if @micropost.nil?
end
end
```

Once the image has been uploaded, we can render it using the `image_tag` helper in the micropost partial, as shown in Listing 13.62. Notice the use of the `picture?` boolean method to prevent displaying an image tag when there isn't an image. This method is created automatically by CarrierWave based on the name of the image attribute. The result of making a successful submission by hand appears in Figure 13.20. Writing an automated test for image upload is left as an exercise (Section 13.4.1.1).

Listing 13.62: Adding image display to microposts. `app/views/microposts/_micropost.html.erb`

```
<li id="micropost-<%= micropost.id %>">
<%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
<span class="user"><%= link_to micropost.user.name, micropost.user %></span>
<span class="content">
<%= micropost.content %>
<%= image_tag micropost.picture.url if micropost.picture? %>
</span>
<span class="timestamp">
Posted <%= time_ago_in_words(micropost.created_at) %> ago.
<% if current_user?(micropost.user) %>
<%= link_to "delete", micropost, method: :delete,
data: { confirm: "You sure?" } %>
<% end %>
</span>
</li>
```

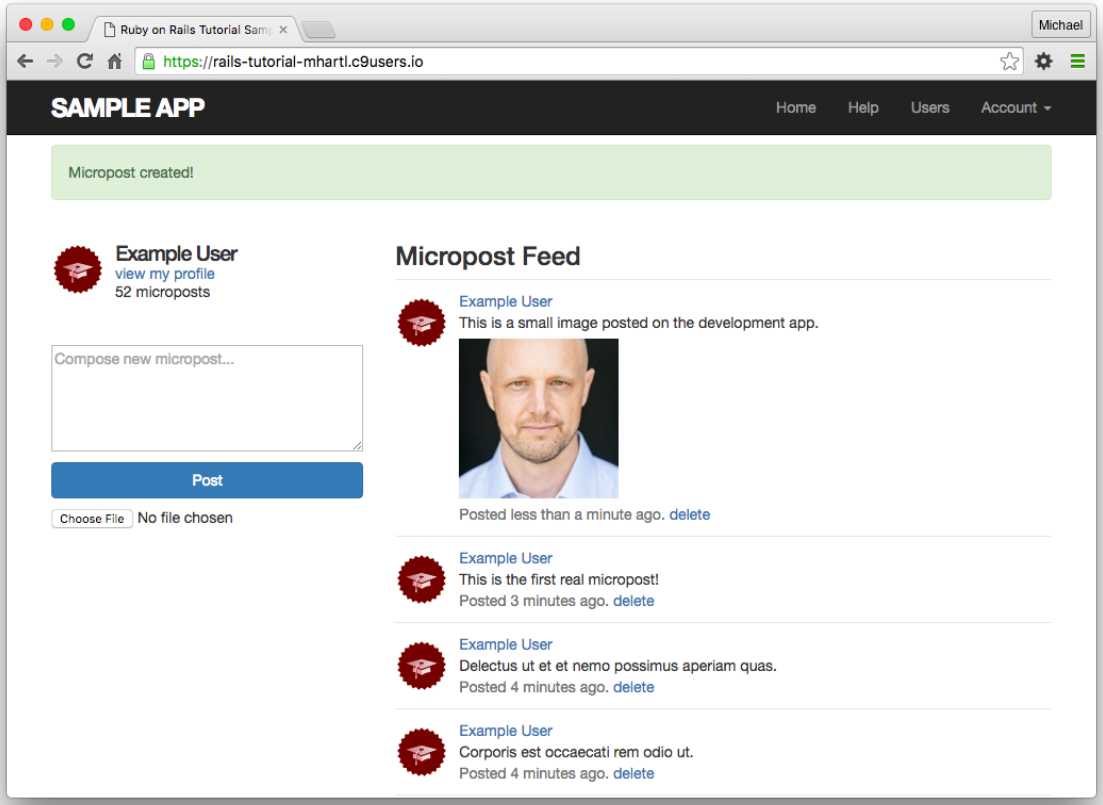

Figure 13.20: The result of submitting a micropost with an image.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Upload a micropost with attached image. Does the result look too big? (If so, don't worry; we'll fix it in Section 13.4.3).
2. Following the template in Listing 13.63, write a test of the image uploader in Section 13.4. As preparation, you should add an image to the fixtures directory (using, e.g, `cp app/assets/images/rails.png test/fixtures/`). The additional assertions in Listing 13.63 check both for a file upload field on the Home page and for a valid image attribute on the micropost resulting from valid submission. Note the use of the special `fixture_file_upload` method for uploading files as fixtures inside tests.[18] *Hint*: To check for a valid `picture` attribute, use the `assigns` method mentioned in Section 11.3.3 to access the micropost in the `create` action after valid submission.

Listing 13.63: A template for testing image upload. `test/integration/microposts_interface_test.rb`

```
require 'test_helper'

class MicropostInterfaceTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "micropost interface" do
    log_in_as(@user)
    get root_path
    assert_select 'div.pagination'
    assert_select 'input[type=FILL_IN]'
    # Invalid submission
    post microposts_path, micropost: { content: "" }
    assert_select 'div#error_explanation'
    # Valid submission
    content = "This micropost really ties the room together"
```

```
picture = fixture_file_upload('test/fixtures/rails.png', 'image/png')
              assert_difference 'Micropost.count', 1 do
post microposts_path, micropost: { content: content, picture: FILL_IN }
                                end
                        assert FILL_IN.picture?
                            follow_redirect!
                  assert_match content, response.body
                            # Delete a post.
                      assert_select 'a', 'delete'
      first_micropost = @user.microposts.paginate(page: 1).first
              assert_difference 'Micropost.count', -1 do
                 delete micropost_path(first_micropost)
                                end
                       # Visit a different user.
                     get user_path(users(:archer))
            assert_select 'a', { text: 'delete', count: 0 }
                               end
                                .
                                .
                                .
                              end
```

13.4.2 Image validation

The uploader in Section 13.4.1 is a good start, but it has significant limitations. In particular, it doesn't enforce any constraints on the uploaded file, which can cause problems if users try to upload large files or invalid file types. To remedy this defect, we'll add validations for the image size and format, both on the server and on the client (i.e., in the browser).

The first image validation, which restricts uploads to valid image types, appears in the CarrierWave uploader itself. The resulting code (which appears as a commented-out suggestion in the generated uploader) verifies that the image filename ends with a valid image extension (PNG, GIF, and both variants of JPEG), as shown in Listing 13.64.

Listing 13.64: The picture format validation. `app/uploaders/picture_uploader.rb`

```
class PictureUploader < CarrierWave::Uploader::Base
                        storage :file

    # Override the directory where uploaded files will be stored.
# This is a sensible default for uploaders that are meant to be mounted:
                        def store_dir
  "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
                               end

  # Add a white list of extensions which are allowed to be uploaded.
                      def extension_white_list
                         %w(jpg jpeg gif png)
                               end
                              end
```

The second validation, which controls the size of the image, appears in the Micropost model itself. In contrast to previous model validations, file size validation doesn't correspond to a built-in Rails validator. As a result, validating images requires defining a custom validation, which we'll call `picture_size` and define as shown in Listing 13.65. Note the use of `validate` (as opposed to `validates`) to call a custom validation.

Listing 13.65: Adding validations to images. `app/models/micropost.rb`

```
             class Micropost < ApplicationRecord
                       belongs_to :user
        default_scope -> { order(created_at: :desc) }
           mount_uploader :picture, PictureUploader
              validates :user_id, presence: true
validates :content, presence: true, length: { maximum: 140 }
                    validate  :picture_size

                            private

          # Validates the size of an uploaded picture.
                        def picture_size
                  if picture.size > 5.megabytes
         errors.add(:picture, "should be less than 5MB")
```

```
  end
end
end
```

This custom validation arranges to call the method corresponding to the given symbol (`:picture_size`). In `picture_size` itself, we add a custom error message to the `errors` collection (seen before briefly in Section 6.2.2), in this case a limit of 5 megabytes (using a syntax seen before in Box 9.1).

To go along with the validations in Listing 13.64 and Listing 13.65, we'll add two client-side checks on the uploaded image. We'll first mirror the format validation by using the `accept` parameter in the `file_field` input tag:

```
<%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
```

The valid formats consist of the MIME types accepted by the validation in Listing 13.64.

Next, we'll include a little JavaScript (or, more specifically, jQuery) to issue an alert if a user tries to upload an image that's too big (which prevents accidental time-consuming uploads and lightens the load on the server):

```
$('#micropost_picture').bind('change', function() {
var size_in_megabytes = this.files[0].size/1024/1024;
if (size_in_megabytes > 5) {
alert('Maximum file size is 5MB. Please choose a smaller file.');
}
});
```

Although jQuery isn't the focus of this book, you might be able to figure out that the code above monitors the page element containing the CSS id `micropost_picture` (as indicated by the hash mark #), which is the id of the micropost form in Listing 13.60. (The way to figure this out is to Ctrl-click and use your browser's web inspector.) When the element with that CSS id changes, the jQuery function fires and issues the `alert` method if the file is too big.[19]

The result of adding these additional checks appears in Listing 13.66.[20]

Listing 13.66: Checking the file size with jQuery. `app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost) do |f| %>
<%= render 'shared/error_messages', object: f.object %>
<div class="field">
<%= f.text_area :content, placeholder: "Compose new micropost..." %>
</div>
<%= f.submit "Post", class: "btn btn-primary" %>
<span class="picture">
<%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
</span>
<% end %>

<script type="text/javascript">
$('#micropost_picture').bind('change', function() {
var size_in_megabytes = this.files[0].size/1024/1024;
if (size_in_megabytes > 5) {
alert('Maximum file size is 5MB. Please choose a smaller file.');
}
});
</script>
```

As you can see by trying to upload a file that's too big, the code in Listing 13.66 doesn't actually clear the file input field, so a user can just dismiss the alert box and continue trying to upload the file. If this were a book on jQuery, we would probably polish this slight blemish by clearing the field, but it's important to understand that front-end code like that shown in Listing 13.66 can't prevent a user from trying to upload a file that's too big. Even if our code cleared the file input field, there would be no way to prevent a user from editing the JavaScript with a web inspector or issue a direct `POST` request using, e.g., `curl`. To prevent users from uploading arbitrarily large files, it is thus essential to include a server-side validation, as in Listing 13.65.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. What happens if you try uploading an image bigger than 5 megabytes?

2. What happens if you try uploading a file with an invalid extension?

13.4.3 Image resizing

The image size validations in Section 13.4.2 are a good start, but they still allow the uploading of images large enough to break our site's layout, sometimes with frightening results (Figure 13.21). Thus, while it's convenient to allow users to select fairly large images from their local disk, it's also a good idea to resize the images before displaying them.[21]

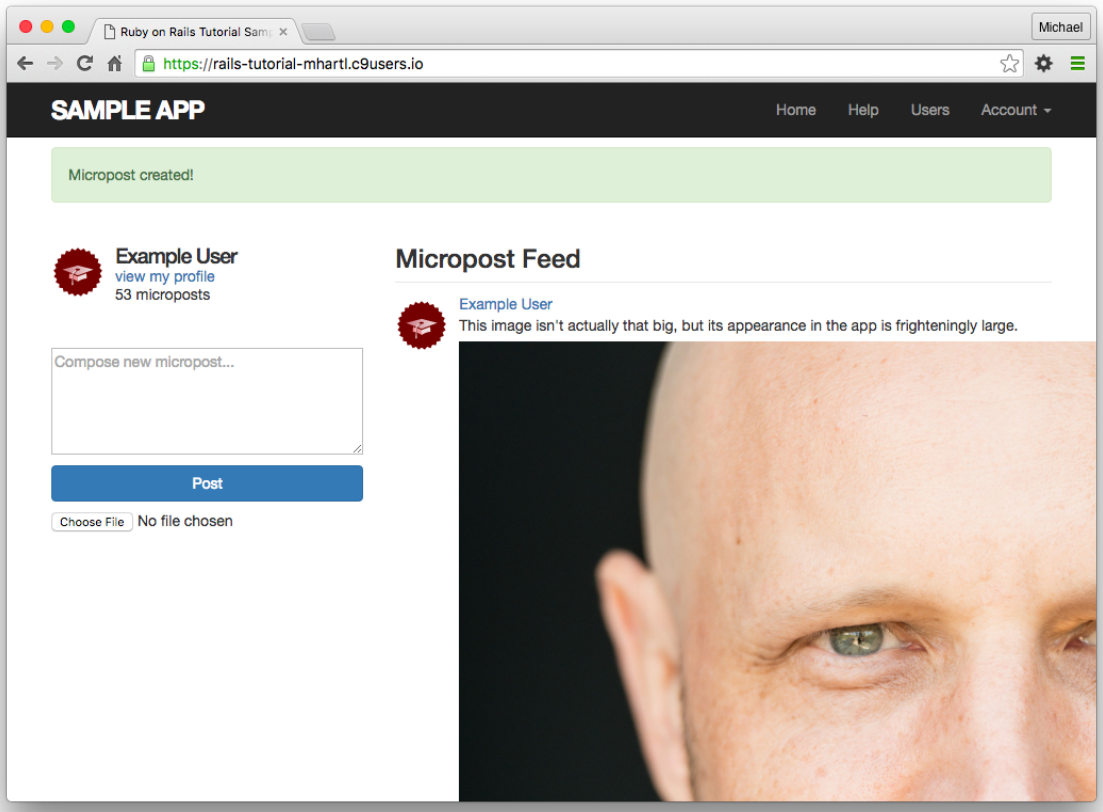

Figure 13.21: A frighteningly large uploaded image.

We'll be resizing images using the image manipulation program ImageMagick, which we need to install on the development environment. (As we'll see in Section 13.4.4, when using Heroku for deployment ImageMagick comes pre-installed in production.) On the cloud IDE, we can do this as follows:[22]

```
$ sudo yum install -y ImageMagick
```

(If you're developing on your local machine, you may have to install ImageMagick a different way, such as using Homebrew on a Mac via `brew install imagemagick`. Use your technical sophistication (Box 1.1) if you get stuck.)

Next, we need to include CarrierWave's MiniMagick interface for ImageMagick, together with a resizing command. For the resizing command, there are several possibilities listed in the MiniMagick documentation, but the one we want is `resize_to_limit: [400, 400]`, which resizes large images so that they aren't any bigger than 400px in either dimension, while simultaneously leaving smaller images alone. (The other main possibilities listed in the CarrierWave documentation on MiniMagick *stretch* images if they're too small, which isn't what we want in this case.) With the code as shown in Listing 13.67, large images are now resized nicely (Figure 13.22).

Listing 13.67: Configuring the image uploader for image resizing. `app/uploaders/picture_uploader.rb`

```
class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
  process resize_to_limit: [400, 400]

  storage :file

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounted:
```

```
  def store_dir
"uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

# Add a white list of extensions which are allowed to be uploaded.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

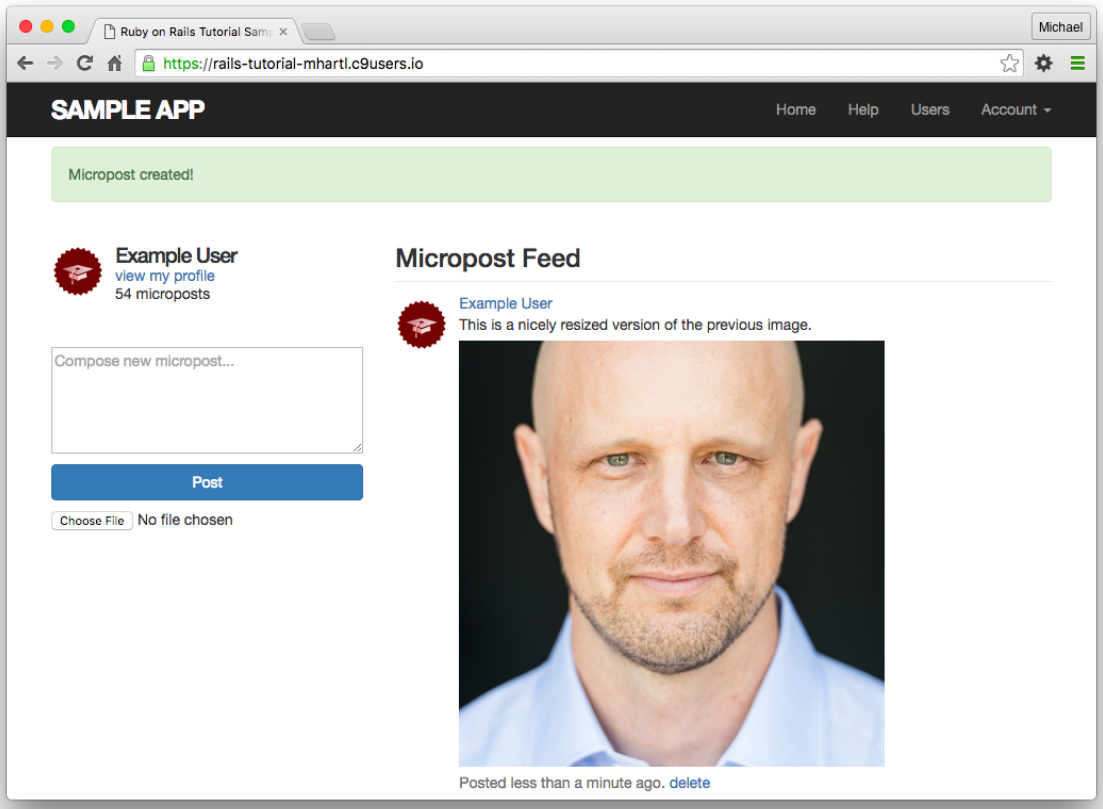

Figure 13.22: A nicely resized image.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Upload a large image and confirm directly that the resizing is working. Does the resizing work even if the image isn't square?
2. If you completed the image upload test in Listing 13.63, at this point your test suite may be giving you a confusing error message. Fix this issue by configuring CarrierWave to skip image resizing in tests using the initializer file shown in Listing 13.68.

Listing 13.68: An initializer to skip image resizing in tests. `config/initializers/skip_image_resizing.rb`

```
if Rails.env.test?
CarrierWave.configure do |config|
  config.enable_processing = false
  end
end
```

13.4.4 Image upload in production

The image uploader developed in Section 13.4.3 is good enough for development, but (as seen in the `storage :file` line in Listing 13.67) it uses the local filesystem for storing the images, which isn't a good practice in production.[23] Instead, we'll use a cloud storage service to store images separately from our application.[24]

To configure our application to use cloud storage in production, we'll use the `fog` gem, as shown in Listing 13.69.

Listing 13.69: Configuring the image uploader for production. `app/uploaders/picture_uploader.rb`

```
class PictureUploader < CarrierWave::Uploader::Base
include CarrierWave::MiniMagick
process resize_to_limit: [400, 400]

if Rails.env.production?
storage :fog
else
storage :file
end

# Override the directory where uploaded files will be stored.
# This is a sensible default for uploaders that are meant to be mounted:
def store_dir
"uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
end

# Add a white list of extensions which are allowed to be uploaded.
def extension_white_list
%w(jpg jpeg gif png)
end
end
```

Listing 13.69 uses the `production?` boolean from Box 7.1 to switch storage method based on the environment:

```
if Rails.env.production?
storage :fog
else
storage :file
end
```

There are many choices for cloud storage, but we'll use one of the most popular and well-supported, Amazon.com's Simple Storage Service (S3).[25] Here are the essential steps to getting set up:

1. Sign up for an Amazon Web Services account if you don't have one already. If you signed up for the Cloud9 IDE in Section 1.2.1, you already have an AWS account and can skip this step.
2. Create a user via AWS Identity and Access Management (IAM) and record the access key and secret key.
3. Create an S3 bucket (with a name of your choice) using the AWS Console, and then grant read and write permission to the user created in the previous step.

You will likely find setting up S3 to be a challenging exercise in technical sophistication (Box 1.1); for further details on the steps above, consult the S3 documentation[26] (and, if necessary, Google or Stack Overflow).

Once you've created and configured your S3 account, you should create and fill the CarrierWave configuration file as shown in Listing 13.70.

Note: If your setup isn't working, your region location may be the issue. Some users may have to add `:region => ENV['S3_REGION']` to the fog credentials, followed by `heroku config:set S3_REGION=<bucket region>` at the command line, where the bucket region should be something like `'eu-central-1'`, depending on your location. To determine the correct region, consult the list of Regions and Endpoints at Amazon.

Listing 13.70: Configuring CarrierWave to use S3. `config/initializers/carrier_wave.rb`

```
if Rails.env.production?
CarrierWave.configure do |config|
config.fog_credentials = {
# Configuration for Amazon S3
:provider => 'AWS',
:aws_access_key_id => ENV['S3_ACCESS_KEY'],
:aws_secret_access_key => ENV['S3_SECRET_KEY']
}
config.fog_directory = ENV['S3_BUCKET']
end
end
```

As with production email configuration (Listing 11.41), Listing 13.70 uses Heroku `ENV` variables to avoid hard-coding sensitive information. In Section 11.4 and Section 12.4, these variables were defined automatically via the SendGrid

add-on, but in this case we need to define them explicitly, which we can accomplish using `heroku config:set` as follows:

```
$ heroku config:set S3_ACCESS_KEY=<access key>
$ heroku config:set S3_SECRET_KEY=<secret key>
$ heroku config:set S3_BUCKET=<bucket name>
```

With the configuration above, we are ready to commit our changes and deploy. I recommend updating your `.gitignore` file as shown in Listing 13.71 so that the image uploads directory is ignored.

Listing 13.71: Adding the uploads directory to the `.gitignore` file.

```
.
.
.
# Ignore uploaded test images.
/public/uploads
```

We're now ready to commit the changes on our topic branch and merge back to master:

```
$ rails test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

Then we deploy, reset the database, and reseed the sample data:

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

Because Heroku comes with an installation of ImageMagick, the result is successful image resizing and upload in production, as seen in Figure 13.23.

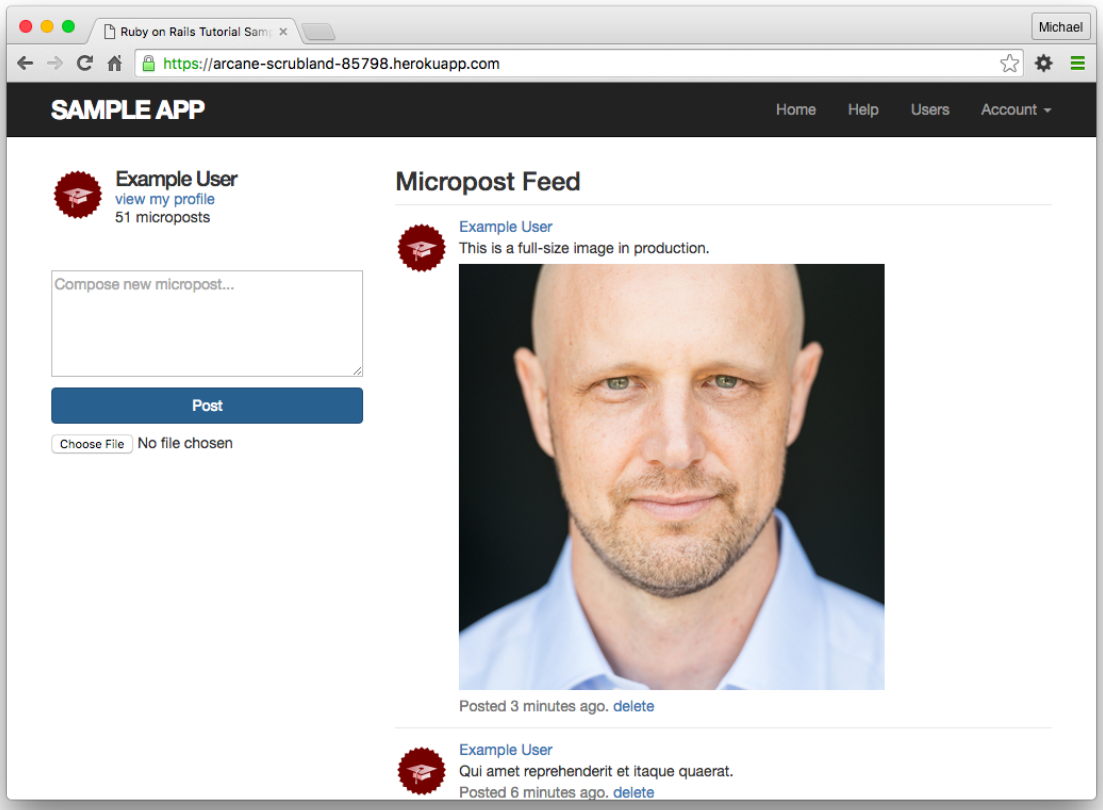

Figure 13.23: An uploaded image in production.

Exercises

Solutions to exercises are available for free at railstutorial.org/solutions with any Rails Tutorial purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. Upload a large image and confirm directly that the resizing is working in production. Does the resizing work even if the image isn't square?

13.5 Conclusion

With the addition of the Microposts resource, we are nearly finished with our sample application. All that remains is to add a social layer by letting users follow each other. We'll learn how to model such user relationships, and see the implications for the microposts feed, in Chapter 14.

If you skipped Section 13.4.4, be sure to commit and merge your changes:

```
$ rails test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

Then deploy to production:

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

It's worth noting that this chapter saw the last of the necessary gem installations. For reference, the final `Gemfile` is shown in Listing 13.72.[27]

Listing 13.72: The final `Gemfile` for the sample application.

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

gem 'rails',                    '5.1.4'
gem 'bcrypt',                   '3.1.11'
gem 'faker',                    '1.7.3'
gem 'carrierwave',              '1.2.2'
gem 'mini_magick',              '4.7.0'
gem 'will_paginate',            '3.1.6'
gem 'bootstrap-will_paginate',  '1.0.0'
gem 'bootstrap-sass',           '3.3.7'
gem 'puma',                     '3.9.1'
gem 'sass-rails',               '5.0.6'
gem 'uglifier',                 '3.2.0'
gem 'coffee-rails',             '4.2.2'
gem 'jquery-rails',             '4.3.1'
gem 'turbolinks',               '5.0.1'
gem 'jbuilder',                 '2.7.0'

group :development, :test do
    gem 'sqlite3', '1.3.13'
gem 'byebug',  '9.0.6', platform: :mri
end

group :development do
gem 'web-console',           '3.5.1'
gem 'listen',                '3.1.5'
gem 'spring',                '2.0.2'
gem 'spring-watcher-listen', '2.0.1'
end

group :test do
gem 'rails-controller-testing', '1.0.2'
gem 'minitest-reporters',       '1.1.14'
gem 'guard',                    '2.14.1'
gem 'guard-minitest',           '2.4.6'
```

```
end

group :production do
  gem 'pg',  '0.20.0'
  gem 'fog', '1.42'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

13.5.1 What we learned in this chapter

- Microposts, like Users, are modeled as a resource backed by an Active Record model.
- Rails supports multiple-key indices.
- We can model a user having many microposts using the `has_many` and `belongs_to` methods in the User and Micropost models, respectively.
- The `has_many`/`belongs_to` combination gives rise to methods that work through the association.
- The code `user.microposts.build(...)` returns a new Micropost object automatically associated with the given user.
- Rails supports default ordering via `default_scope`.
- Scopes take anonymous functions as arguments.
- The `dependent: :destroy` option causes objects to be destroyed at the same time as associated objects.
- Pagination and object counts can both be performed through associations, leading to automatically efficient code.
- Fixtures support the creation of associations.
- It is possible to pass variables to Rails partials.
- The `where` method can be used to perform Active Record selections.
- We can enforce secure operations by always creating and destroying dependent objects through their association.
- We can upload and resize images using CarrierWave.

1. The name is motivated by the common description of Twitter as a *microblog*; since blogs have posts, microblogs should have microposts. ↑
2. www.postgresql.org/docs/9.1/static/datatype-character.html ↑
3. The foreign key reference is a database-level constraint indicating that the user id in the microposts table refers to the id column in the users table. This detail will never be important in this tutorial, and the foreign key constraint isn't even supported by all databases. (It's supported by PostgreSQL, which we use in production, but not by the development SQLite database adapter.) We'll learn more about foreign keys in Section 14.1.2. ↑
4. We briefly encountered a similar issue in Section 10.5 in the context of the users index. ↑
5. SQL is case-insensitive, but it is conventional to write SQL keywords (such as `DESC`) in all-caps. ↑
6. `Faker::Lorem.sentence` returns *lorem ipsum* text; as noted in Chapter 6, *lorem ipsum* has a fascinating back story. ↑
7. By design, the Faker gem's *lorem ipsum* text is randomized, so the contents of your sample microposts will differ. ↑
8. For convenience, Listing 13.26 actually has *all* the CSS needed for this chapter. ↑
9. If you'd like to refactor other tests to use `full_title` (such as those in Listing 3.30), you should include the Application Helper in `test_helper.rb` instead. ↑
10. Note that, unlike the behavior in languages like Java or C++, private methods in Ruby can be called from derived classes. Thanks to reader Vishal Antony for bringing this difference to my attention. ↑
11. See the Rails Guide on the Active Record Query Interface for more on `where` and related methods. ↑
12. This corresponds to `HTTP_REFERER`, as defined by the specification for HTTP. Note that "referer" is not a typo—the word is actually misspelled in the spec. Rails corrects this error by writing "referrer" instead. ↑
13. I didn't remember offhand how to get this URL inside a Rails application, so I Googled "rails request previous url" and found a Stack Overflow thread with the answer. ↑
14. Image retrieved from https://www.flickr.com/photos/grungepunk/14026922186 on 2014-09-19. Copyright © 2014 by Jussie D. Brito and used unaltered under the terms of the Creative Commons Attribution-ShareAlike 2.0 Generic license. ↑
15. As always, you should use the version numbers listed at gemfiles-4th-ed.railstutorial.org instead of the ones listed here. ↑
16. Initially, I called the new attribute `image`, but that name was so generic it ended up being confusing. ↑
17. When using `form_tag` (discussed briefly in Section 8.1.2), it's necessary to add the option `html: { multipart: true }` to handle file uploads, but when using `form_for` the necessary multipart form-data encoding type is automatically added by Rails. Thanks to reader Alan Cruz for bringing this detail to my attention. ↑
18. Windows users should add a `:binary` parameter: `fixture_file_upload(file, type, :binary)`. ↑

19. To learn how to do things like this, you can do what I did: Google around for things like “javascript maximum file size” until you find something on Stack Overflow. ↑
20. More advanced users of jQuery would probably put the size check in its own JavaScript function, but since this isn’t a JavaScript tutorial the code in Listing 13.66 is fine for our purposes. ↑
21. It’s possible to constrain the *display* size with CSS, but this doesn’t change the image size. In particular, large images would still take a while to load. (You’ve probably visited websites where “small” images seemingly take forever to load. This is why.) ↑
22. If you’re not using the cloud IDE or an equivalent Linux system, do a Google search for “imagemagick <your platform>”. On macOS, `brew install imagemagick` should work if you have Homebrew installed. ↑
23. Among other things, file storage on Heroku is temporary, so uploaded images will be deleted every time you deploy. ↑
24. This is a challenging section and can be skipped without loss of continuity. ↑
25. S3 is a paid service, but the storage needed to set up and test the Rails Tutorial sample application costs less than a cent per month. ↑
26. aws.amazon.com/documentation/s3/ ↑
27. As always, you should use the version numbers listed at gemfiles-4th-ed.railstutorial.org instead of the ones listed here. ↑