Chapter 13

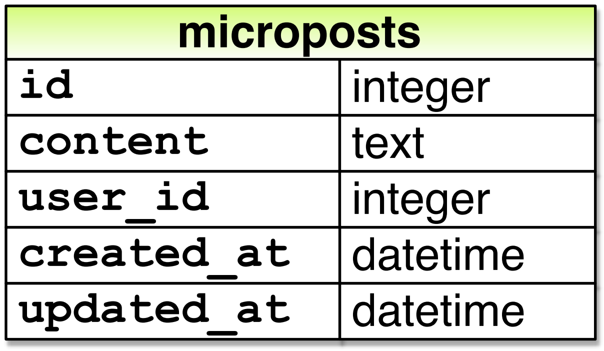
User microposts

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. Un-

[1]The name is motivated by the common description of Twitter as a *microblog*; since blogs have posts, microblogs should have *microposts*, which can be thought of as the generic equivalent of "tweets".



microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime

Figure 13.1: The Micropost data model.

like 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.

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

[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.

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[6.0]
  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 the exercises are available to all Rails Tutorial purchasers here.

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

1. Using **`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`

```ruby
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 reality 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

```ruby
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
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

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 the exercises are available to all Rails Tutorial purchasers here.

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

1. 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

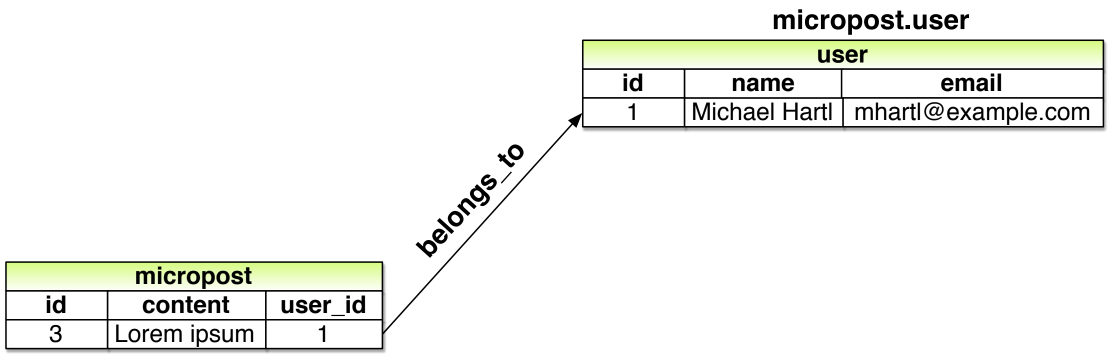


Figure 13.2: The **belongs_to** relationship between a micropost and its associated user.

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.

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

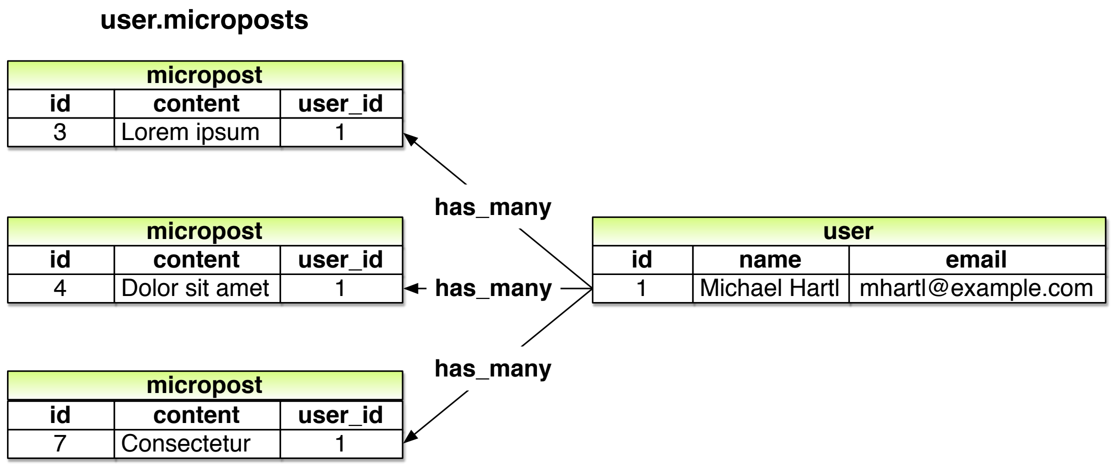


Figure 13.3: The **has_many** relationship between a user and its microposts.

```
@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.

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).

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.

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

```ruby
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`

```ruby
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`

```ruby
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 the exercises are available to all Rails Tutorial purchasers here.

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

1. 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`

```ruby
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 can define in analogy with the user fixtures, last seen in Listing 11.5. In addition to the

[4]We briefly encountered a similar issue in Section 10.5 in the context of the users index.

`content` attribute defined in Section 13.1.1, we also need define the associated **`user`**. Conveniently, Rails includes a 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
  .
  .
  .
```

The full micropost fixtures appear 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 %>
  user: michael

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

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

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

Note that Listing 13.15 explicitly sets the **`created_at`** column using embedded Ruby. Because it's a "magic" column automatically updated by Rails, setting **`created_at`** by hand isn't ordinarily possible, but it is possible in fixtures.[5]

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.[6] 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:

[5]In practice this might not be necessary, and in fact on many systems the fixtures are created in the order in which they appear in the file. 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.

[6]SQL is case-insensitive, but it is conventional to write SQL keywords (such as **`DESC`**) in all-caps.

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

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

1. 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.

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

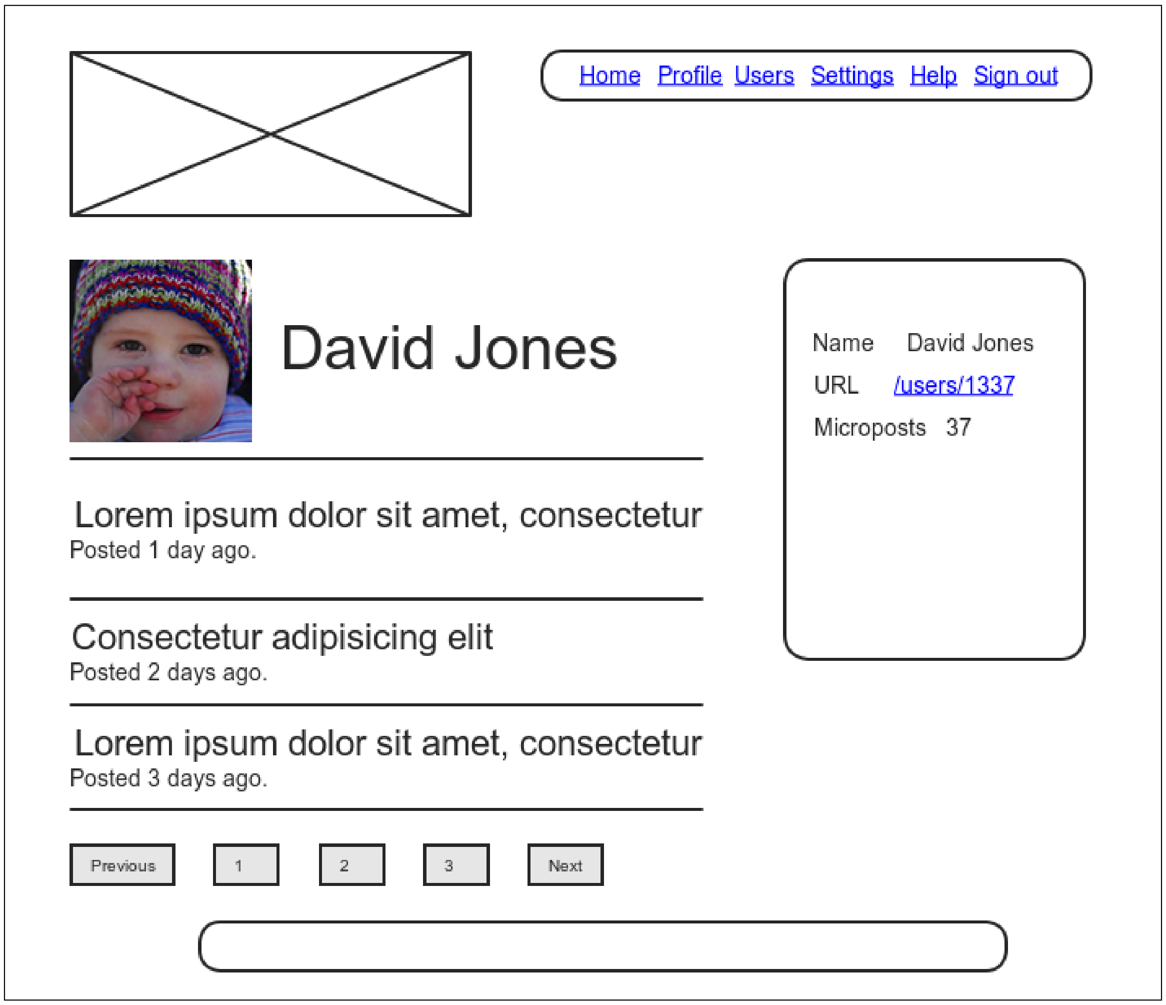

Home Profile Users Settings Help Sign out
David Jones
Name David Jones
URL /users/1337
Microposts 37
Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago.
Consectetur adipisicing elit
Posted 2 days ago.
Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago.
Previous
1
2
3
Next

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

```
<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:

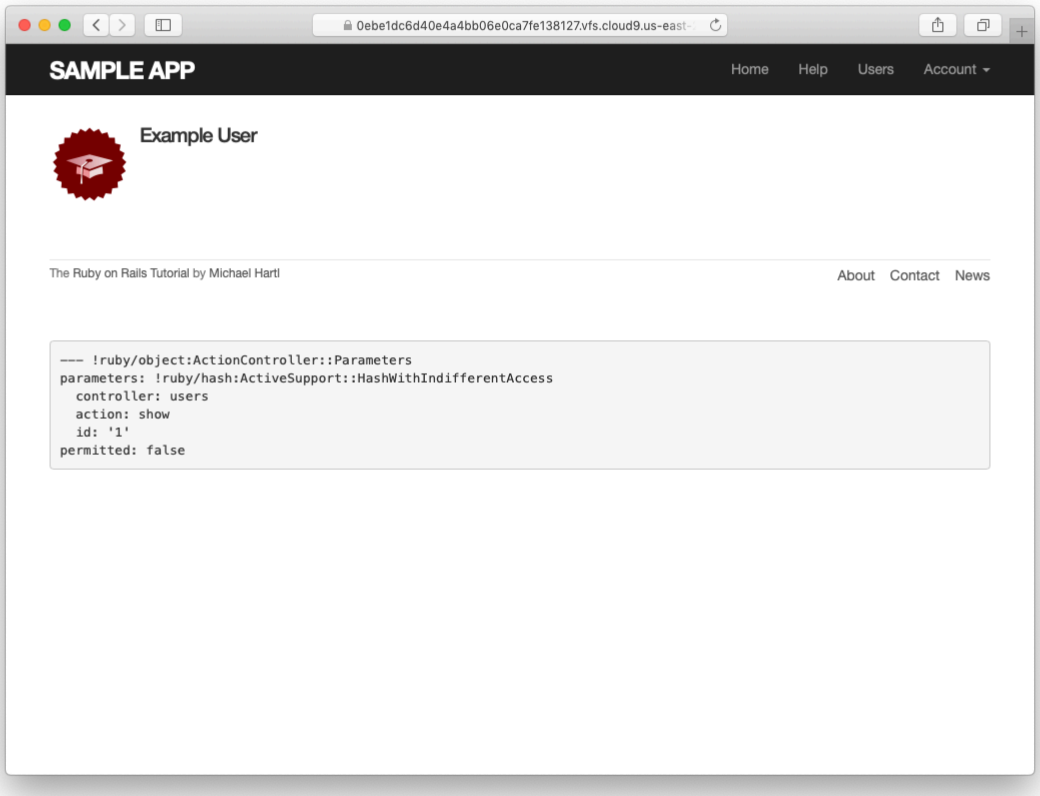
```
user.microposts.count
```

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>
```


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-
SAMPLE APP
Home
Help
Users
Account
Example User
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  controller: users
  action: show
  id: '1'
permitted: false
```

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

```
      <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.

Exercises

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

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

1. As 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.[7] The result is

[7] **`Faker::Lorem.sentence`** returns *lorem ipsum* text; as noted in Chapter 6, *lorem ipsum* has a fascinating back story.

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

```
.
.
.
# Generate microposts for a subset of users.
users = User.order(:created_at).take(6)
50.times do
  content = Faker::Lorem.sentence(word_count: 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.

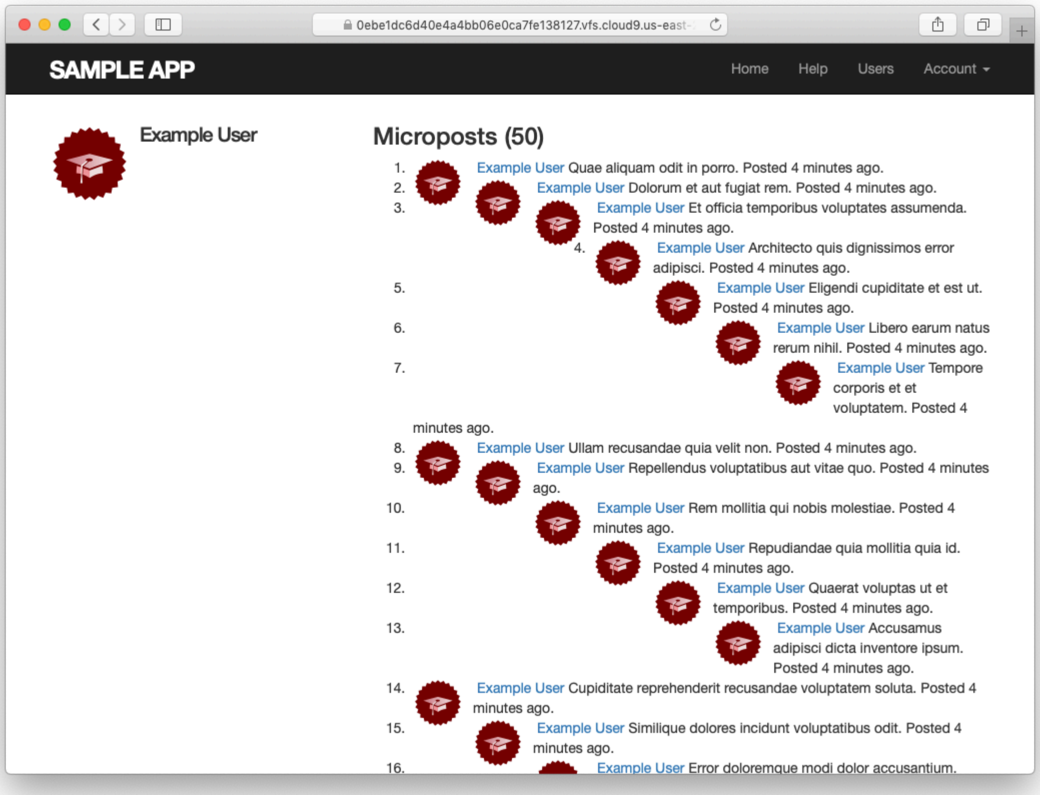
With that, we are in a position to enjoy the fruits of our Section 13.2.1 labors by displaying information for each micropost.[8] The preliminary results appear in Figure 13.6.

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.[9]

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

[8]By design, the Faker gem's *lorem ipsum* text is randomized, so the contents of your sample microposts will differ.

[9]For convenience, Listing 13.26 actually has *all* the CSS needed for this chapter.



Example User

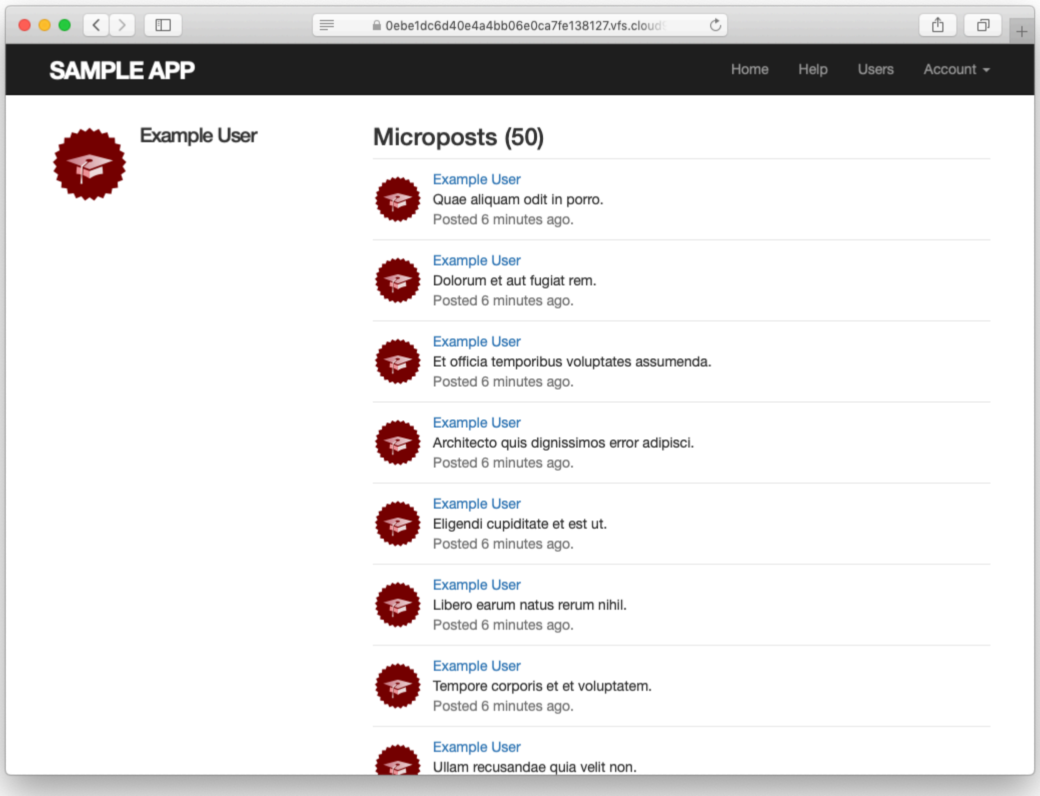
Microposts (50)

1. Example User Quae aliquam odit in porro. Posted 4 minutes ago.
2. Example User Dolorum et aut fugiat rem. Posted 4 minutes ago.
3. Example User Et officia temporibus voluptates assumenda. Posted 4 minutes ago.
4. Example User Architecto quis dignissimos error adipisci. Posted 4 minutes ago.
5. Example User Eligendi cupiditate et est ut. Posted 4 minutes ago.
6. Example User Libero earum natus rerum nihil. Posted 4 minutes ago.
7. Example User Tempore corporis et et voluptatem. Posted 4 minutes ago.
8. Example User Ullam recusandae quia velit non. Posted 4 minutes ago.
9. Example User Repellendus voluptatibus aut vitae quo. Posted 4 minutes ago.
10. Example User Rem mollitia qui nobis molestiae. Posted 4 minutes ago.
11. Example User Repudiandae quia mollitia quia id. Posted 4 minutes ago.
12. Example User Quaerat voluptas ut et temporibus. Posted 4 minutes ago.
13. Example User Accusamus adipisci dicta inventore ipsum. Posted 4 minutes ago.
14. Example User Cupiditate reprehenderit recusandae voluptatem soluta. Posted 4 minutes ago.
15. Example User Similique dolores incidunt voluptatibus odit. Posted 4 minutes ago.
16. Example User Error doloremque modi dolor accusantium.

Figure 13.6: The user profile with unstyled microposts.

```
.
.
.
/* 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.image {
  margin-top: 10px;
  input {
    border: 0;
  }
}
```

Figure 13.7 shows the user profile page for the first user, while Figure 13.8



Microposts (50)

Example User
Quae aliquam odit in porro.
Posted 6 minutes ago.

Example User
Dolorum et aut fugiat rem.
Posted 6 minutes ago.

Example User
Et officia temporibus voluptates assumenda.
Posted 6 minutes ago.

Example User
Architecto quis dignissimos error adipisci.
Posted 6 minutes ago.

Example User
Eligendi cupiditate et est ut.
Posted 6 minutes ago.

Example User
Libero earum natus rerum nihil.
Posted 6 minutes ago.

Example User
Tempore corporis et et voluptatem.
Posted 6 minutes ago.

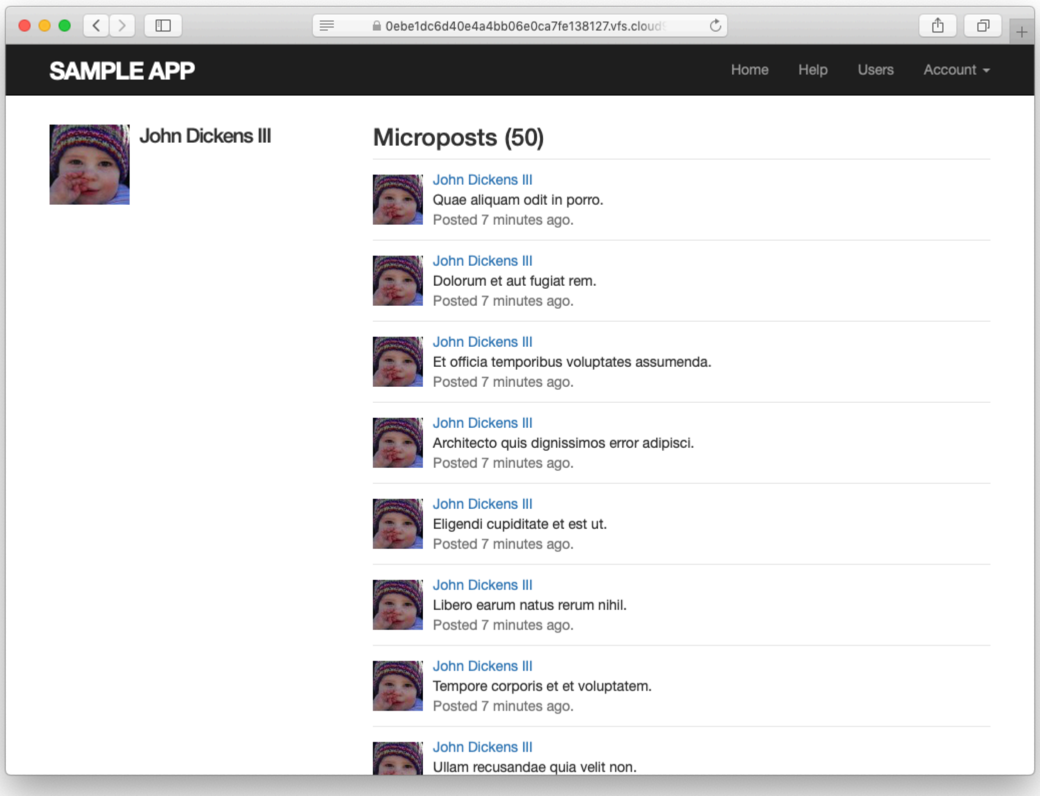
Example User
Ullam recusandae quia velit non.

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

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.

Exercises

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



John Dickens III

Microposts (50)

John Dickens III
Quae aliquam odit in porro.
Posted 7 minutes ago.

John Dickens III
Dolorum et aut fugiat rem.
Posted 7 minutes ago.

John Dickens III
Et officia temporibus voluptates assumenda.
Posted 7 minutes ago.

John Dickens III
Architecto quis dignissimos error adipisci.
Posted 7 minutes ago.

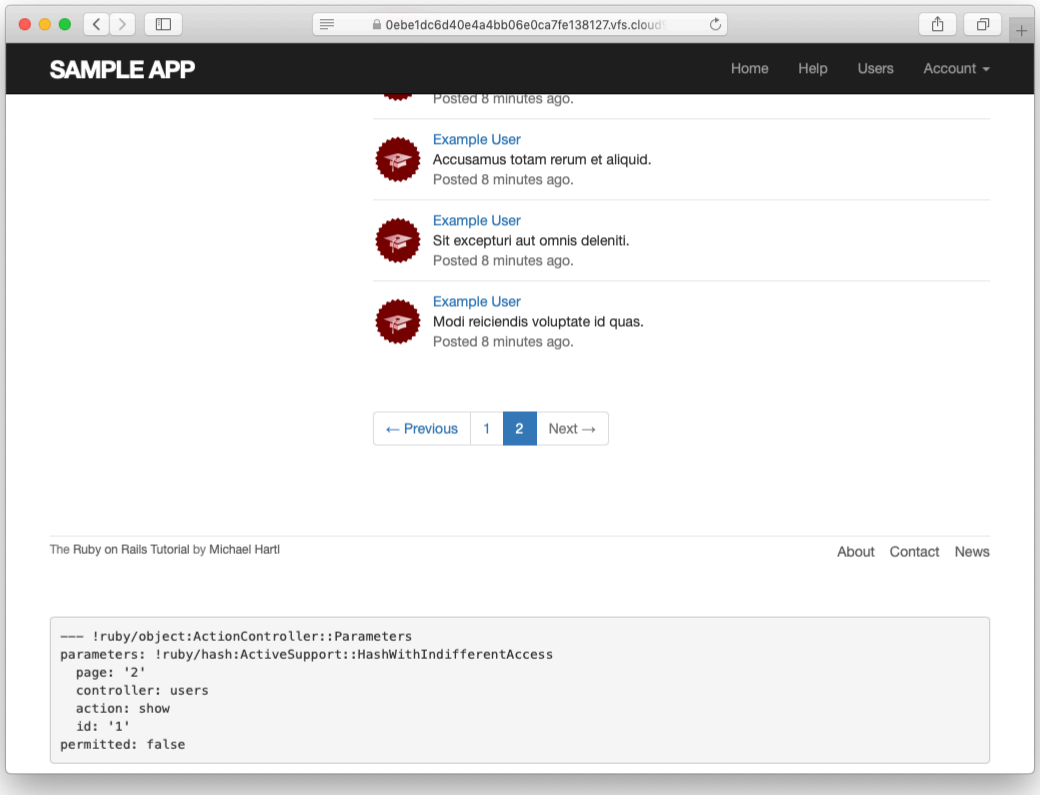
John Dickens III
Eligendi cupiditate et est ut.
Posted 7 minutes ago.

John Dickens III
Libero earum natus rerum nihil.
Posted 7 minutes ago.

John Dickens III
Tempore corporis et et voluptatem.
Posted 7 minutes ago.

John Dickens III
Ullam recusandae quia velit non.

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


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud
SAMPLE APP
Home
Help
Users
Account
Posted 8 minutes ago.
Example User
Accusamus totam rerum et aliquid.
Posted 8 minutes ago.
Example User
Sit excepturi aut omnis deleniti.
Posted 8 minutes ago.
Example User
Modi reiciendis voluptate id quas.
Posted 8 minutes ago.
← Previous
1
2
Next →
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  page: '2'
  controller: users
  action: show
  id: '1'
permitted: false
```

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

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

1. 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 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 %>
```

Adding this to the code from Listing 13.15 gives the updated micropost fixtures in Listing 13.27.

Listing 13.27: Micropost fixtures with generated micropsts.
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: https://tauday.com"
  created_at: <%= 3.years.ago %>
  user: michael

cat_video:
  content: "Sad cats are sad: https://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(word_count: 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.[10]

[10]If you'd like to refactor other tests to use **`full_title`** (such as those in Listing 3.32), you should include the Application Helper in **`test_helper.rb`** instead.

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

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

1. Comment out the 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

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.

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
```

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).[11] The result appears in Listing 13.32.

[11]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.

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

```
class ApplicationController < ActionController::Base
  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 the exercises are available to all Rails Tutorial purchasers here.

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

1. 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.

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 7.26); 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`

```ruby
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
```

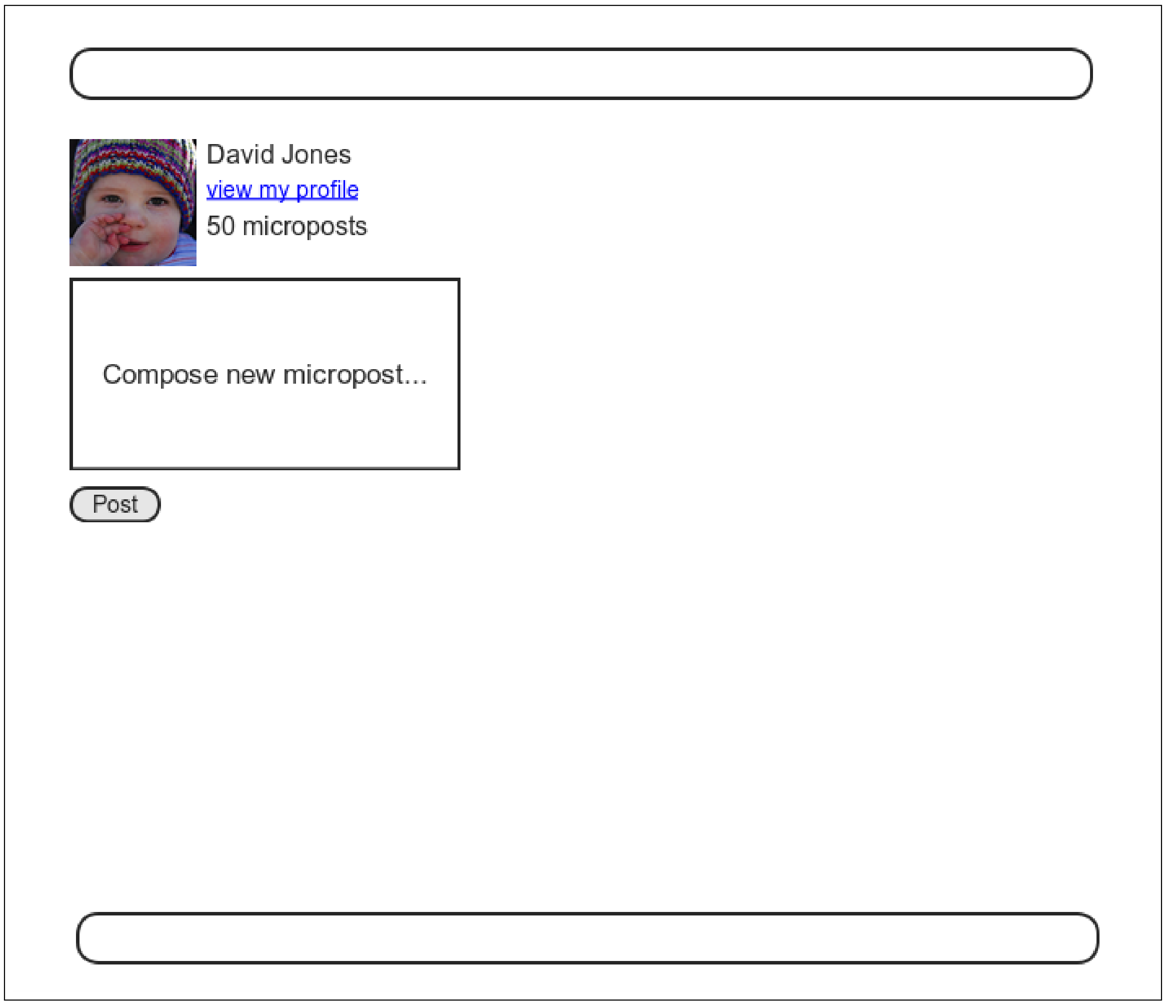


Figure 13.10: A mockup of the Home page with a form for creating microposts.

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="https://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.svg", alt: "Rails logo", width: "200"),
                         "https://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).)

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_with(model: @micropost, local: true) 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_with(model: @user, local: true) do |f|
```

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

```
form_with(model: @micropost, local: true) 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. RED
`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_with(model: @user, local: true) 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. RED
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_with(model: @user, local: true) 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="https://gravatar.com/emails">change</a>
    </div>
  </div>
</div>
```

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

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, url: password_reset_path(params[:id]),
                  local: true) 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:

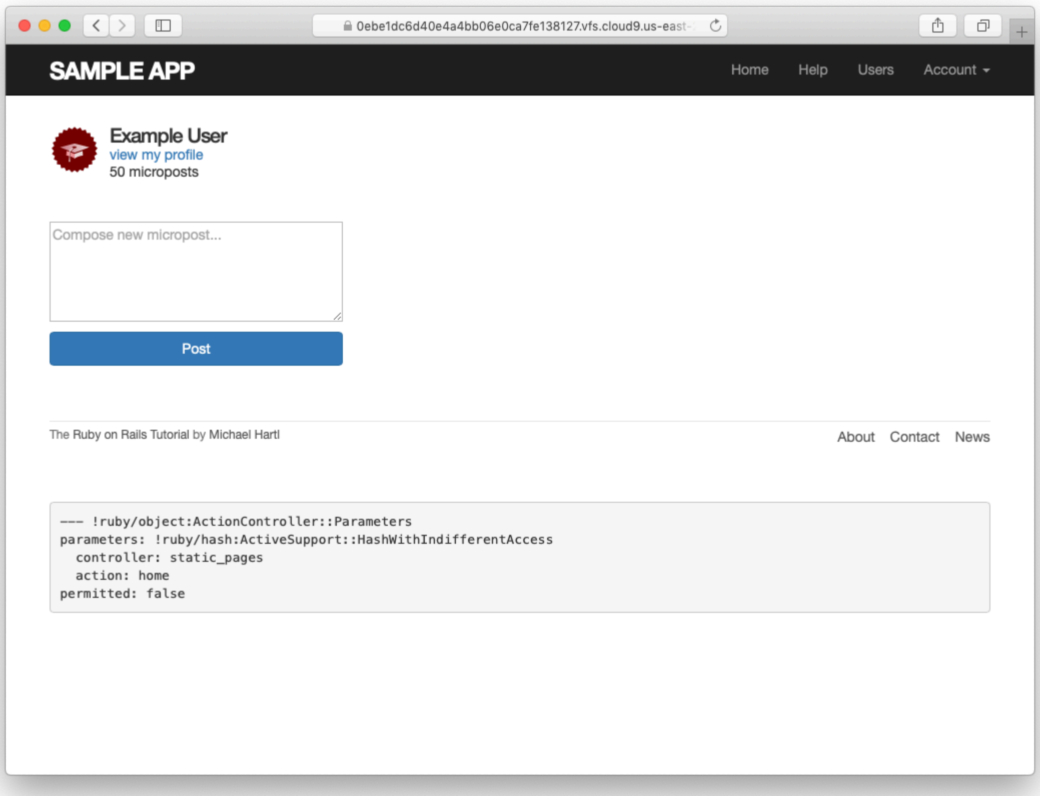
```
$ rails test
```

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.

Exercises

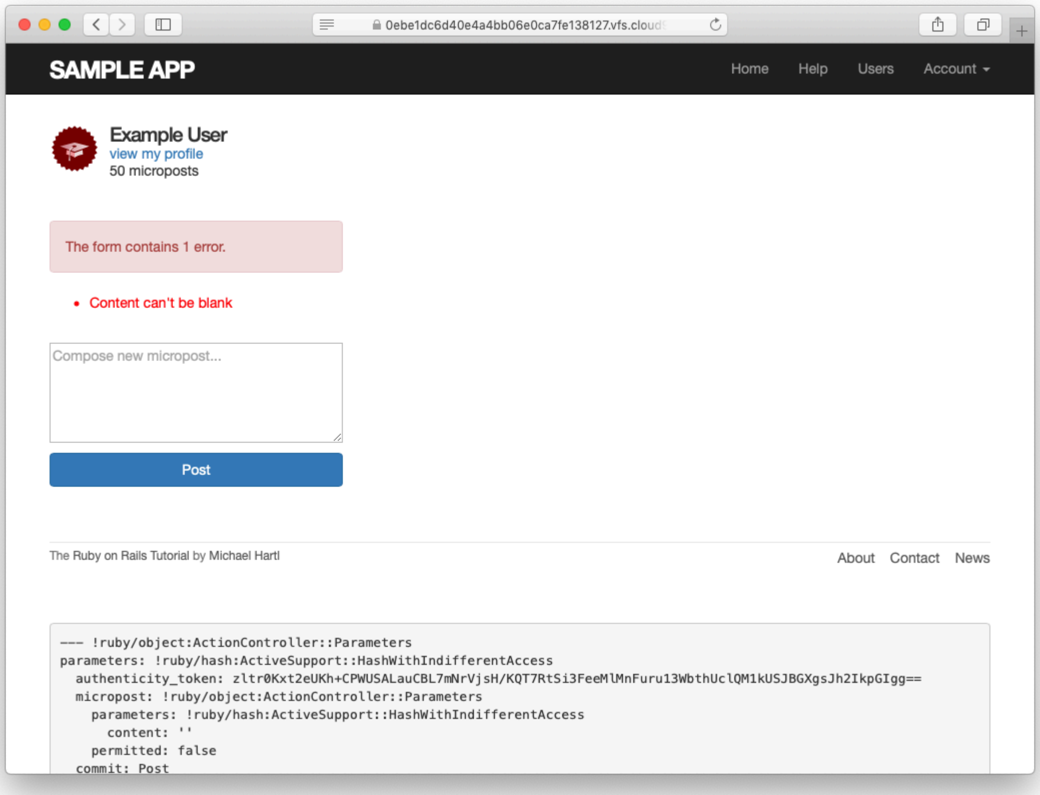
Solutions to the exercises are available to all Rails Tutorial purchasers here.

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



```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  controller: static_pages
  action: home
permitted: false
```

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



The form contains 1 error.

- Content can't be blank

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  authenticity_token: zltr0Kxt2eUKh+CPWUSALauCBL7mNrVjsH/KQT7RtSi3FeeMlMnFuru13WbthUclQM1kUSJBGXgsJh2IkpGIgg==
  micropost: !ruby/object:ActionController::Parameters
    parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
      content: ''
    permitted: false
  commit: Post
```

Figure 13.12: The Home page with a form error.

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.)

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), as shown in Listing 13.46.[12]

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

```ruby
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

[12]See the Rails Guide on the Active Record Query Interface for more on **where** and related methods.

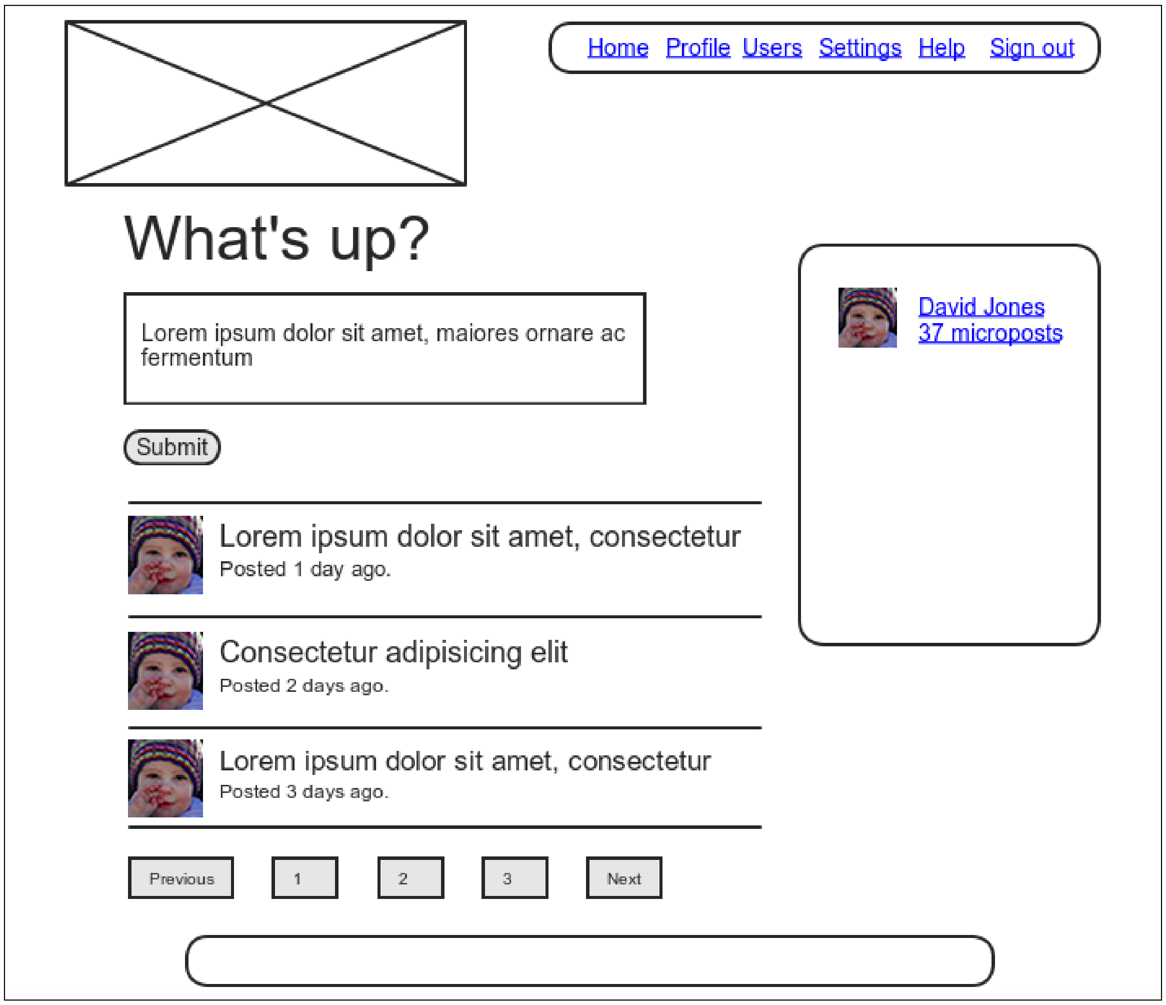

Home Profile Users Settings Help Sign out
What's up?
Lorem ipsum dolor sit amet, maiores ornare ac fermentum
Submit
David Jones
37 microposts
Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago.
Consectetur adipisicing elit
Posted 2 days ago.
Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago.
Previous
1
2
3
Next

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

```
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`

```ruby
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`

```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:

```erb
<%= 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
```

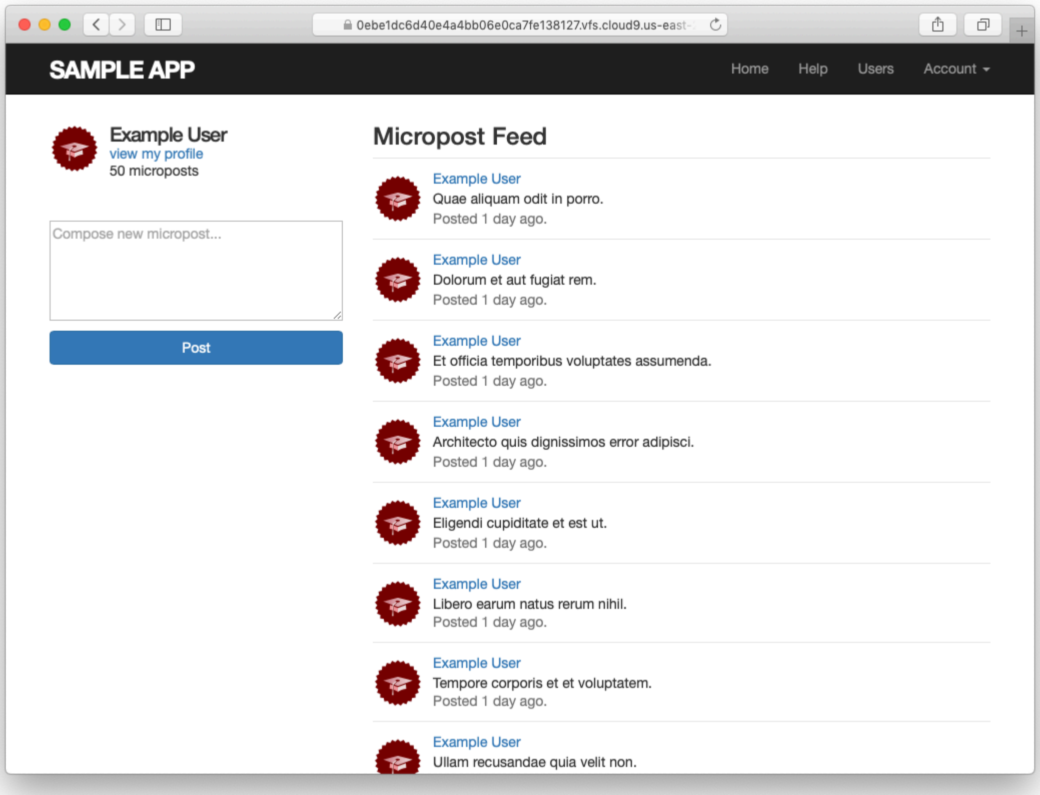
We can add the feed to the Home page by rendering the feed partial as usual (Listing 13.49). The result is a display of the feed on the Home page, as required (Figure 13.14).

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 %>
```

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 solution is to create the necessary feed variable in the branch for failed submissions in the Microposts controller **`create`** action, as shown in Listing 13.50.



Example User
view my profile
50 microposts

Micropost Feed

Example User
Quae aliquam odit in porro.
Posted 1 day ago.

Example User
Dolorum et aut fugiat rem.
Posted 1 day ago.

Example User
Et officia temporibus voluptates assumenda.
Posted 1 day ago.

Example User
Architecto quis dignissimos error adipisci.
Posted 1 day ago.

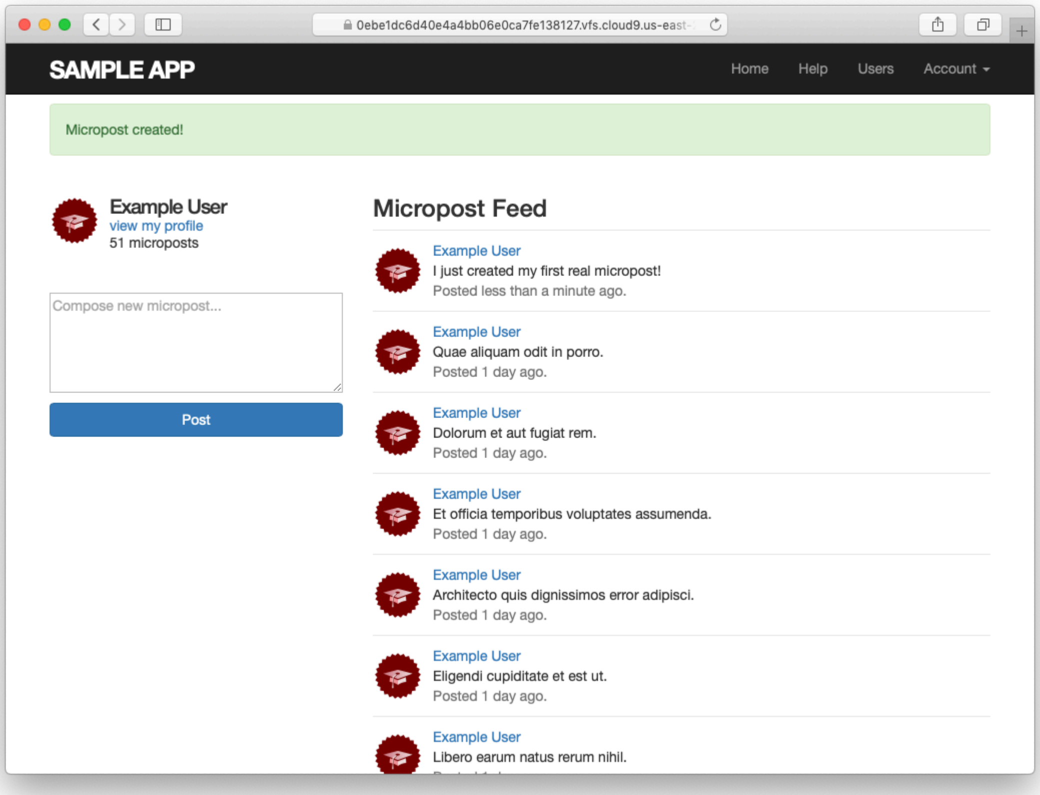
Example User
Eligendi cupiditate et est ut.
Posted 1 day ago.

Example User
Libero earum natus rerum nihil.
Posted 1 day ago.

Example User
Tempore corporis et et voluptatem.
Posted 1 day ago.

Example User
Ullam recusandae quia velit non.

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



Micropost created!

Example User
view my profile
51 microposts

Micropost Feed

Example User
I just created my first real micropost!
Posted less than a minute ago.

Example User
Quae aliquam odit in porro.
Posted 1 day ago.

Example User
Dolorum et aut fugiat rem.
Posted 1 day ago.

Example User
Et officia temporibus voluptates assumenda.
Posted 1 day ago.

Example User
Architecto quis dignissimos error adipisci.
Posted 1 day ago.

Example User
Eligendi cupiditate et est ut.
Posted 1 day ago.

Example User
Libero earum natus rerum nihil.

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`

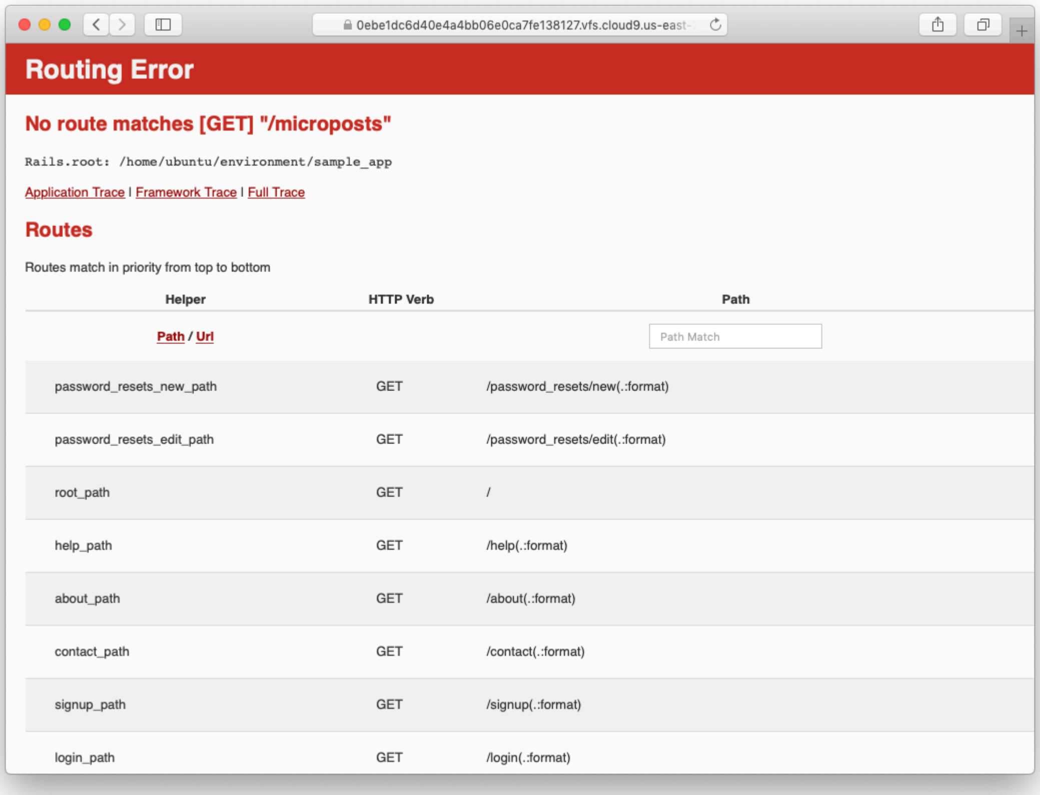
```ruby
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]

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

  def destroy
  end

  private

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

Unfortunately, pagination still doesn't quite work. We can see why by submitting an invalid micropost, say, one whose length is too long (Figure 13.16).

Scrolling down to the pagination links, we see links on both "2" and "Next" pointing to the next page (Figure 13.17). Because the **`create`** action is in the Microposts controller (Listing 13.50), the URL is /microposts?page=2, which tries to go to the nonexistent Microposts index action. As a result, clicking on either link gives a routing error (Figure 13.18).

We can solve this problem by giving **`will_paginate`** explicit **`controller`** and **`action`** parameters corresponding to the Home page, i.e., the **`static_pages`** controller and the **`home`** action.[13] The result appears in Listing 13.51.

[13]Thanks to reader Martin Francl for pointing out this solution.



0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-

Routing Error

No route matches [GET] "/microposts"

```
Rails.root: /home/ubuntu/environment/sample_app
```

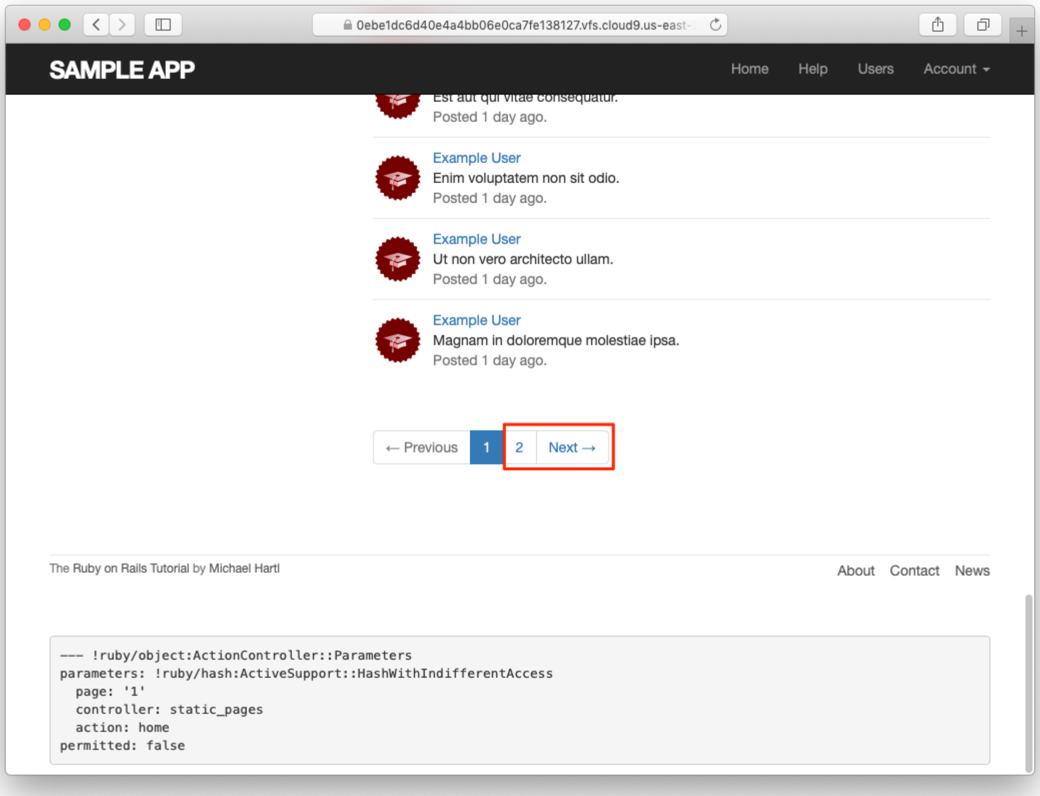
Application Trace | Framework Trace | Full Trace

Routes

Routes match in priority from top to bottom

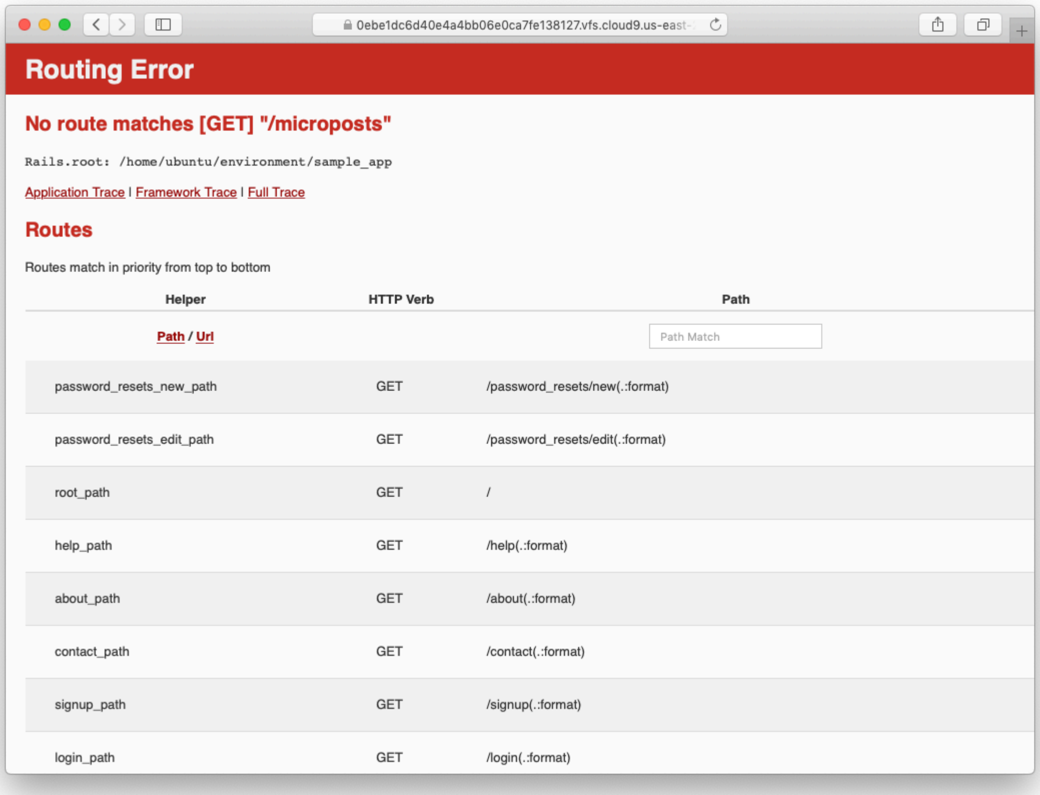
Helper	HTTP Verb	Path
Path / Url		Path Match
password_resets_new_path	GET	/password_resets/new(.:format)
password_resets_edit_path	GET	/password_resets/edit(.:format)
root_path	GET	/
help_path	GET	/help(.:format)
about_path	GET	/about(.:format)
contact_path	GET	/contact(.:format)
signup_path	GET	/signup(.:format)
login_path	GET	/login(.:format)

Figure 13.16: An invalid micropost on the Home page.


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-
SAMPLE APP
Home
Help
Users
Account
Est aut qui vitae consequatur.
Posted 1 day ago.
Example User
Enim voluptatem non sit odio.
Posted 1 day ago.
Example User
Ut non vero architecto ullam.
Posted 1 day ago.
Example User
Magnam in doloremque molestiae ipsa.
Posted 1 day ago.
← Previous
1
2
Next →
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  page: '1'
  controller: static_pages
  action: home
permitted: false
```

Figure 13.17: The next link on the Home page.



Routing Error

No route matches [GET] "/microposts"

```
Rails.root: /home/ubuntu/environment/sample_app
```

Application Trace | Framework Trace | Full Trace

Routes

Routes match in priority from top to bottom

Helper	HTTP Verb	Path
Path / Url		Path Match
password_resets_new_path	GET	/password_resets/new(.:format)
password_resets_edit_path	GET	/password_resets/edit(.:format)
root_path	GET	/
help_path	GET	/help(.:format)
about_path	GET	/about(.:format)
contact_path	GET	/contact(.:format)
signup_path	GET	/signup(.:format)
login_path	GET	/login(.:format)

Figure 13.18: A routing error on page 2.

Listing 13.51: Setting an explicit controller and action.
app/views/shared/_feed.html.erb

```
<% if @feed_items.any? %>
  <ol class="microposts">
    <%= render @feed_items %>
  </ol>
  <%= will_paginate @feed_items,
                    params: { controller: :static_pages, action: :home } %>
<% end %>
```

Now clicking on either of the pagination links in Figure 13.17 yields the expected second page, as shown in Figure 13.19.

Exercises

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

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

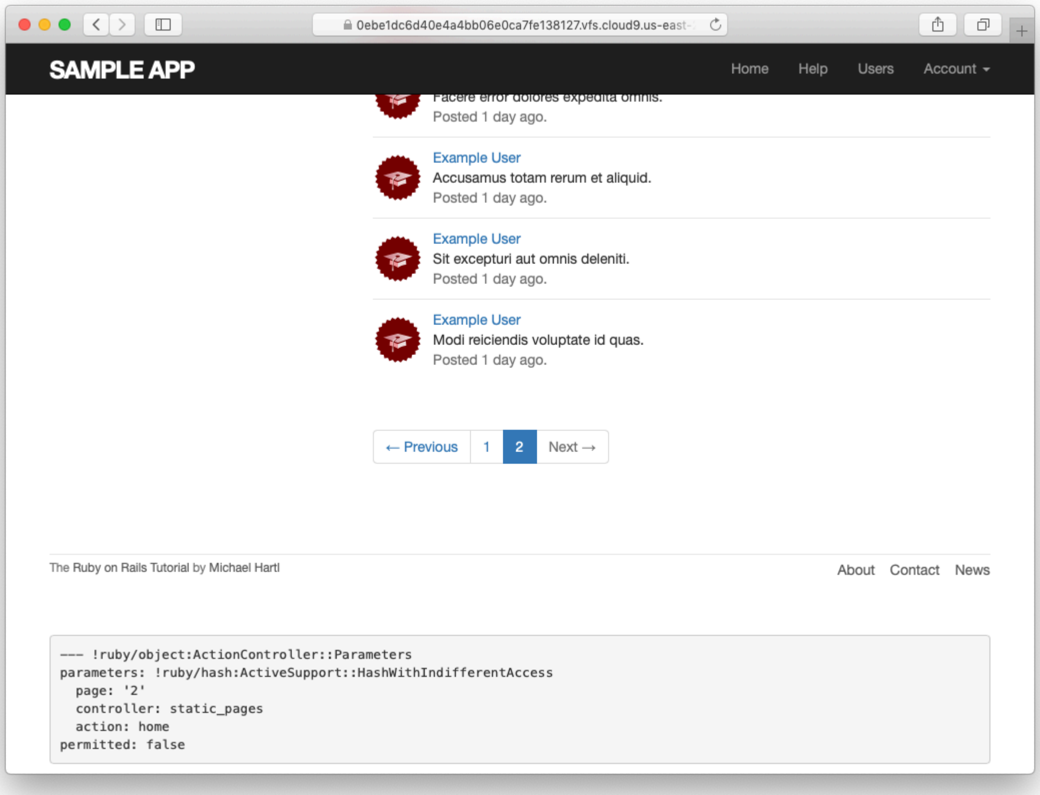
1. Use the newly created micropost UI to create the first real micropost. What are the contents of the `INSERT` command in the server log?

2. In the console, set `user` to the first user in the database. Confirm that the values of `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 13.20. Unlike that case, which restricted user destruction to admin users, the delete links will work only for microposts created by the current user.

Our first step is to add a delete link to the micropost partial as in Listing 13.22. The result appears in Listing 13.52.


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-
SAMPLE APP
Home
Help
Users
Account
← Previous
1
2
Next →
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

Facere error dolores expedita omnis.
Posted 1 day ago.

Example User
Accusamus totam rerum et aliquid.
Posted 1 day ago.

Example User
Sit excepturi aut omnis deleniti.
Posted 1 day ago.

Example User
Modi reiciendis voluptate id quas.
Posted 1 day ago.

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  page: '2'
  controller: static_pages
  action: home
permitted: false
```

Figure 13.19: The result of a working pagination link to the second page.

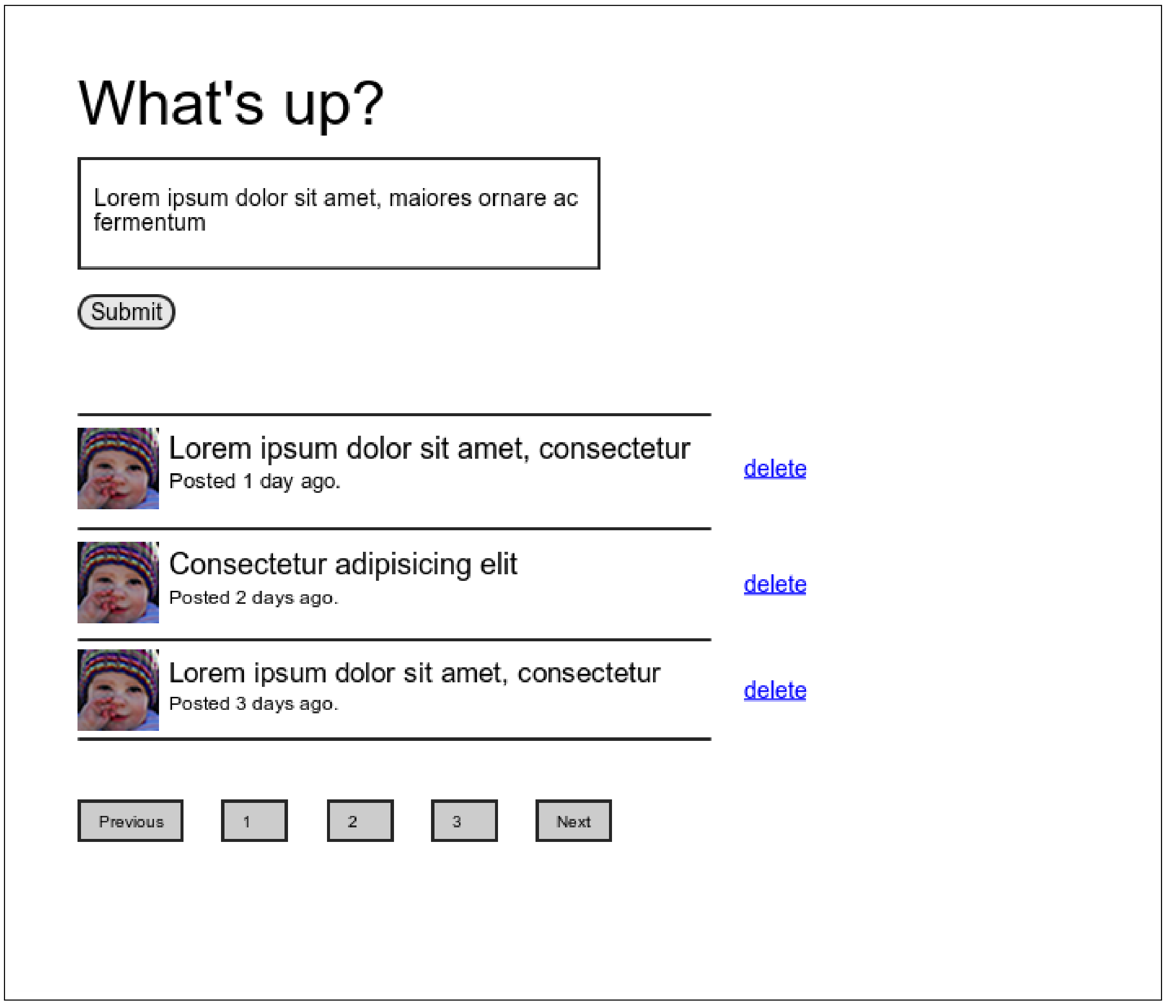

What's up?
Lorem ipsum dolor sit amet, maiores ornare ac fermentum
Submit
Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago.
delete
Consectetur adipisicing elit
Posted 2 days ago.
delete
Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago.
delete
Previous
1
2
3
Next

Figure 13.20: A mockup of the proto-feed with micropost delete links.

Listing 13.52: Adding a delete link to the micropost partial.
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.
    <% 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 13.53.

Listing 13.53: 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 13.53 redirects to the URL

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

This uses the **request.referrer** method,[14] 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).[15] 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 13.53 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 Home page has working delete links (Figure 13.21), which you can verify by deleting, e.g., the second post (Figure 13.22).

Exercises

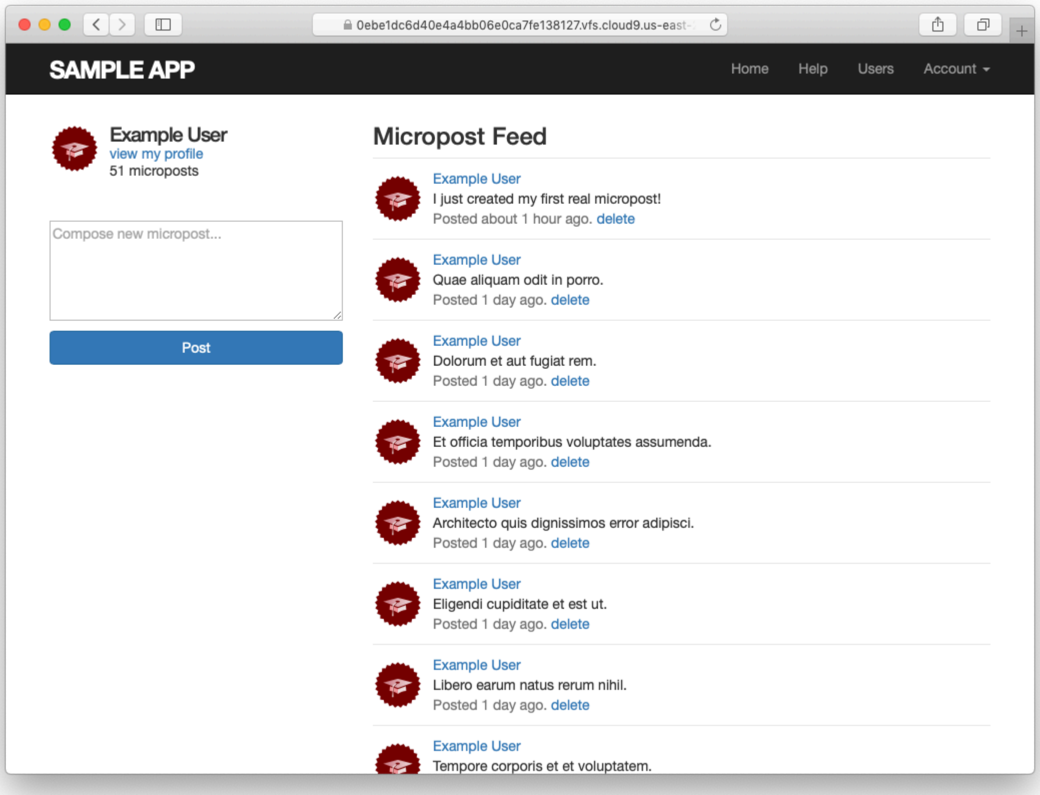
Solutions to the exercises are available to all Rails Tutorial purchasers here.

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

1. Create a new micropost and then delete it. What are the contents of the **DELETE** command in the server log?

[14]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.

[15]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.



Micropost Feed

Example User
I just created my first real micropost!
Posted about 1 hour ago. delete

Example User
Quae aliquam odit in porro.
Posted 1 day ago. delete

Example User
Dolorum et aut fugiat rem.
Posted 1 day ago. delete

Example User
Et officia temporibus voluptates assumenda.
Posted 1 day ago. delete

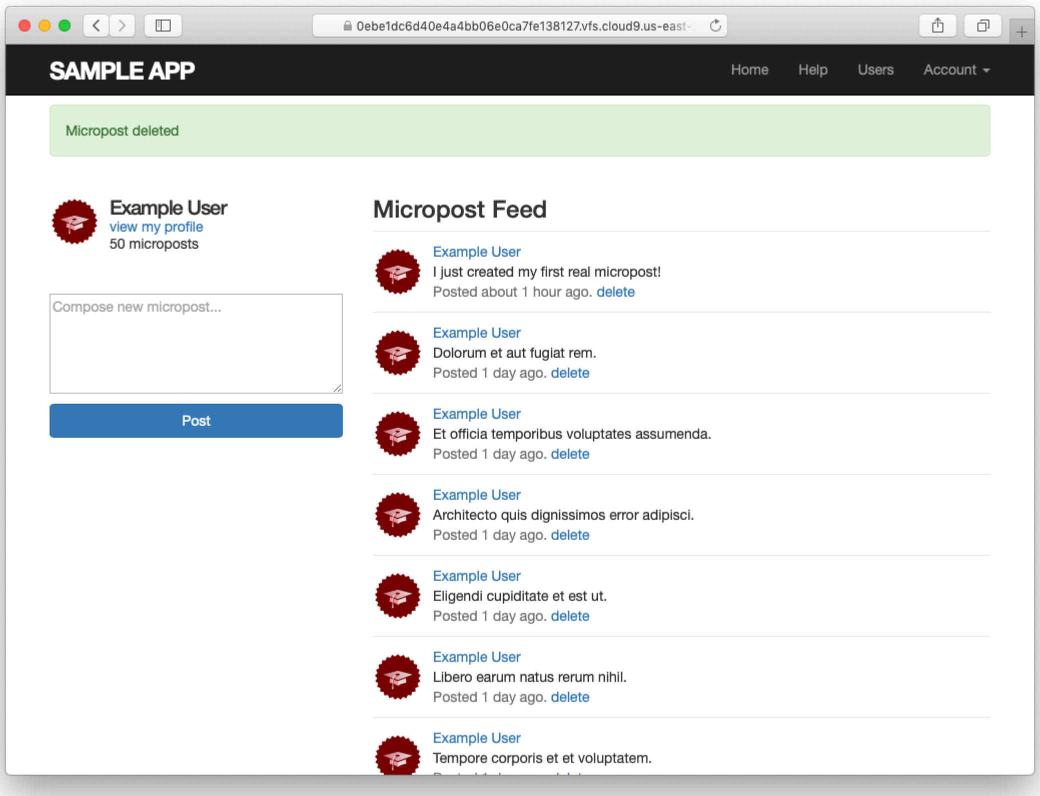
Example User
Architecto quis dignissimos error adipisci.
Posted 1 day ago. delete

Example User
Eligendi cupiditate et est ut.
Posted 1 day ago. delete

Example User
Libero earum natus rerum nihil.
Posted 1 day ago. delete

Example User
Tempore corporis et et voluptatem.

Figure 13.21: The Home page with delete links.



Micropost deleted

Example User
view my profile
50 microposts

Compose new micropost...

Post

Micropost Feed

Example User
I just created my first real micropost!
Posted about 1 hour ago. delete

Example User
Dolorum et aut fugiat rem.
Posted 1 day ago. delete

Example User
Et officia temporibus voluptates assumenda.
Posted 1 day ago. delete

Example User
Architecto quis dignissimos error adipisci.
Posted 1 day ago. delete

Example User
Eligendi cupiditate et est ut.
Posted 1 day ago. delete

Example User
Libero earum natus rerum nihil.
Posted 1 day ago. delete

Example User
Tempore corporis et et voluptatem.

Figure 13.22: The result of deleting the second post.

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 13.54. (We'll be using only one for now, but we've put in the others for future reference.)

Listing 13.54: 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 13.55.

Listing 13.55: 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 13.56. See if you can connect the lines in Listing 13.12 to the steps mentioned above.

Listing 13.56: 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'
    assert_select 'a[href=?]', '/?page=2'  # Correct pagination link
    # 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 13.57: GREEN

```
$ rails test
```

Exercises

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

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

1. For each of the four scenarios indicated by comments in Listing 13.56 (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 13.58 will help get you started.

Listing 13.58: 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 13.23.[16]

13.4.1 Basic image upload

The most convenient way to upload files in Rails is to use a built-in feature called Active Storage.[17] Active Storage makes it easy to handle an uploaded image and associate it with a model of our choice (e.g., the Micropost model). Although we'll be using it only for uploading images, Active Storage is actually quite general, and can handle plain text and multiple kinds of binary files (such as PDF documents or recorded audio).

As described in the Active Storage documentation, adding Active Storage to our application is as easy as running a single command:

```
$ rails active_storage:install
```

This command generates a database migration that creates a data model for storing attached files. You're welcome to take a look at it, but this is an excellent

[16]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.

[17]Active Storage was added in Rails 5.2.

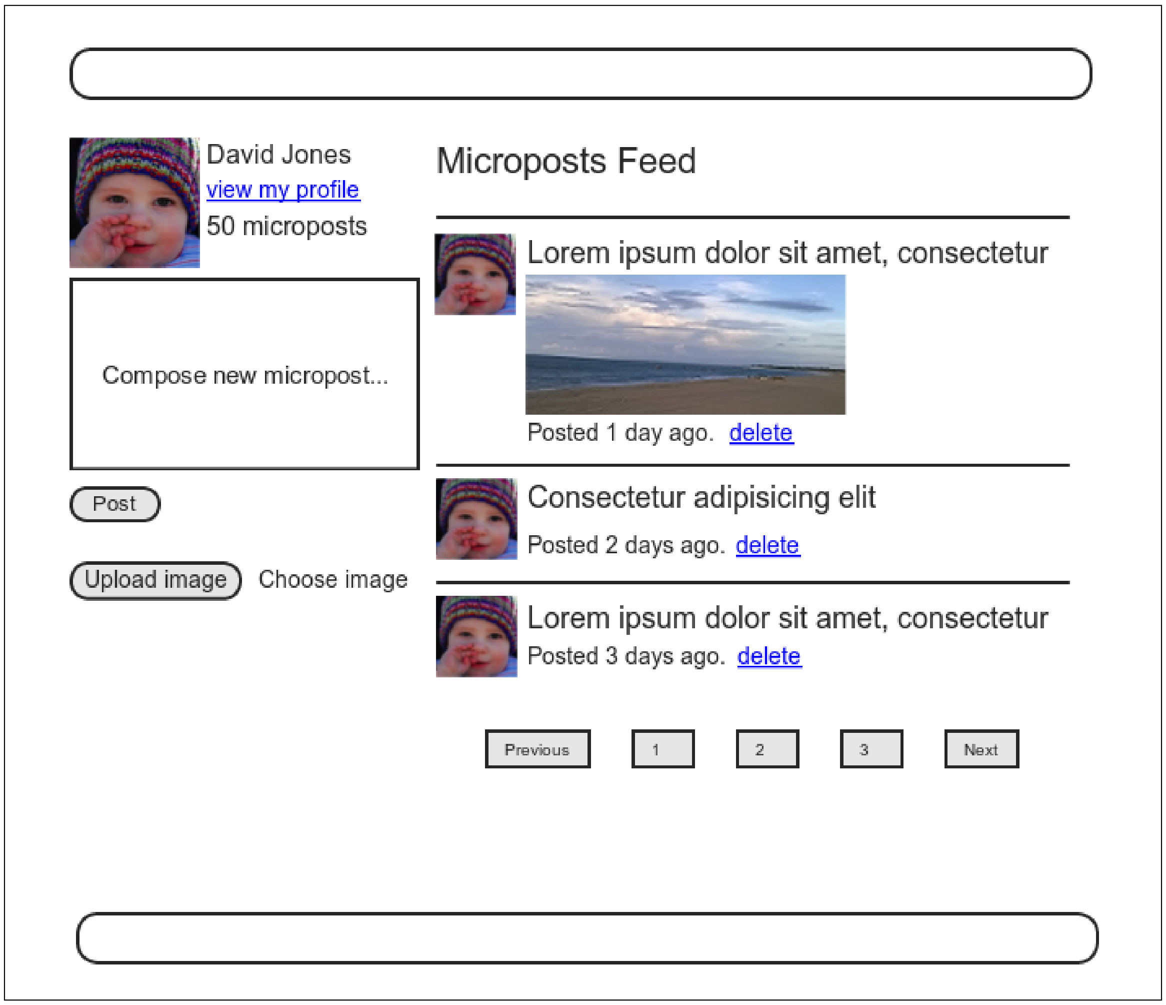

David Jones
view my profile
50 microposts
Compose new micropost...
Post
Upload image
Choose image
Microposts Feed
Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago. delete
Consectetur adipisicing elit
Posted 2 days ago. delete
Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago. delete
Previous
1
2
3
Next

Figure 13.23: A mockup of micropost image upload (with an uploaded image).

example of applying technical sophistication to know which details matter and which don't. In this case, what matters is the API for interacting with Active Storage, which we'll start covering in a moment; for our purposes, the implementation details are safe to ignore. All we need to do to set it up is run the migration:

```
$ rails db:migrate
```

The first part of the Active Storage API that we need is the **`has_one_attached`** method, which allows us to associate an uploaded file with a given model. In our case, we'll call it **`image`** and associate it with the Micropost model, as shown in Listing 13.59.

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

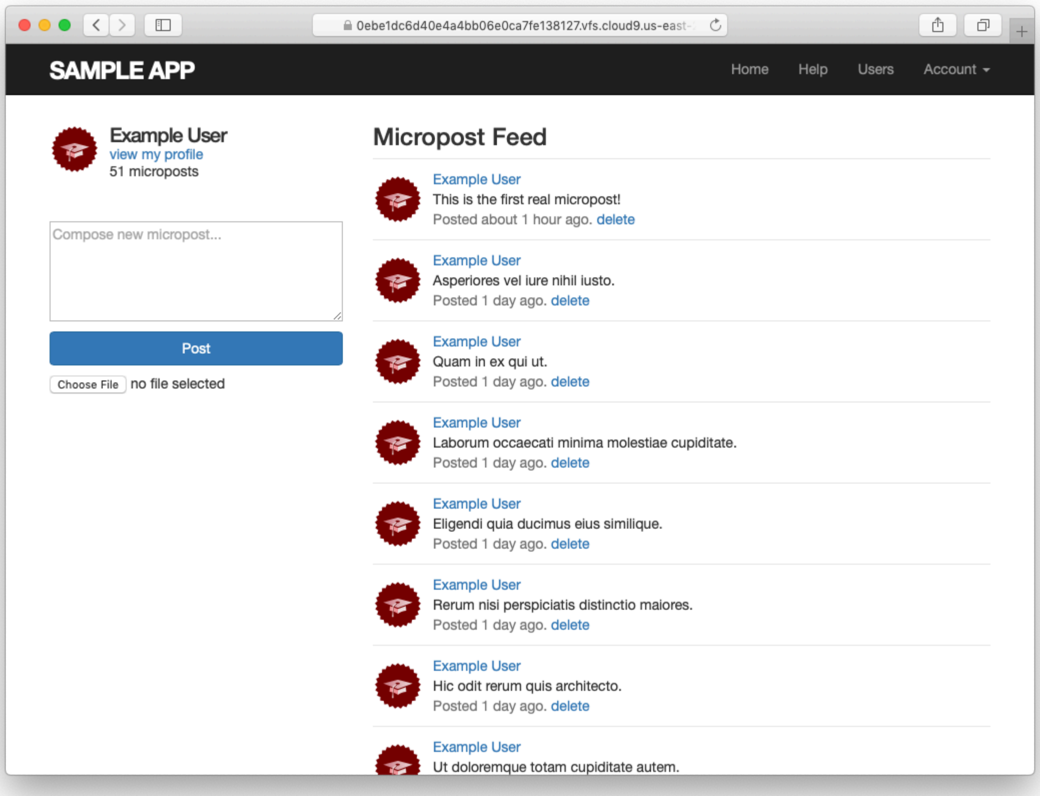
```
class Micropost < ApplicationRecord
  belongs_to       :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

We'll adopt a design of one image per micropost for our application, but Active Storage also offers a second option, **`has_many_attached`**, which allows for the attachment of multiple files to a single Active Record object.

To include image upload on the Home page as in Figure 13.23, we need to include a **`file_field`** tag in the micropost form, as shown in Listing 13.60 and Figure 13.24.

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

```
<%= form_with(model: @micropost, local: true) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
```



Example User
view my profile
51 microposts

Compose new micropost...

Micropost Feed

Example User
This is the first real micropost!
Posted about 1 hour ago. delete

Example User
Asperiores vel iure nihil iusto.
Posted 1 day ago. delete

Example User
Quam in ex qui ut.
Posted 1 day ago. delete

Example User
Laborum occaecati minima molestiae cupiditate.
Posted 1 day ago. delete

Example User
Eligendi quia ducimus eius similique.
Posted 1 day ago. delete

Example User
Rerum nisi perspiciatis distinctio maiores.
Posted 1 day ago. delete

Example User
Hic odit rerum quis architecto.
Posted 1 day ago. delete

Example User
Ut doloremque totam cupiditate autem.

Figure 13.24: Adding an image upload field.

```
    <div class="field">
      <%= f.text_area :content, placeholder: "Compose new micropost..." %>
    </div>
    <%= f.submit "Post", class: "btn btn-primary" %>
    <span class="image">
      <%= f.file_field :image %>
    </span>
<% end %>
```

Finally, we need to update the Microposts controller to add the image to the newly created micropost object. We can do this using the **`attach`** method provided by the Active Storage API, which attaches the uploaded image to the **`@micropost`** object in the Microposts controller's **`create`** action. To allow the upload to go through, we also need to update **`micropost_params`** method to add **`:image`** to the list of attributes permitted to be modified through the web. The result appears in Listing 13.61.

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

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

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

  def destroy
    @micropost.destroy
    flash[:success] = "Micropost deleted"
    redirect_to request.referrer || root_url
  end

  private

    def micropost_params
      params.require(:micropost).permit(:content, :image)
    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 the associated **`micro-post.image`** using the **`image_tag`** helper in the micropost partial, as shown in Listing 13.62. Notice the use of the **`attached?`** boolean method to prevent displaying an image tag when there isn't an image.

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.image if micropost.image.attached? %>
  </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 result of making a micropost with an image appears in Figure 13.25. I'm always amazed when things like this actually work, but there's the proof! (It's also a good idea to write at least a basic automated test for image upload, which is left as an exercise (Section 13.4.1).)

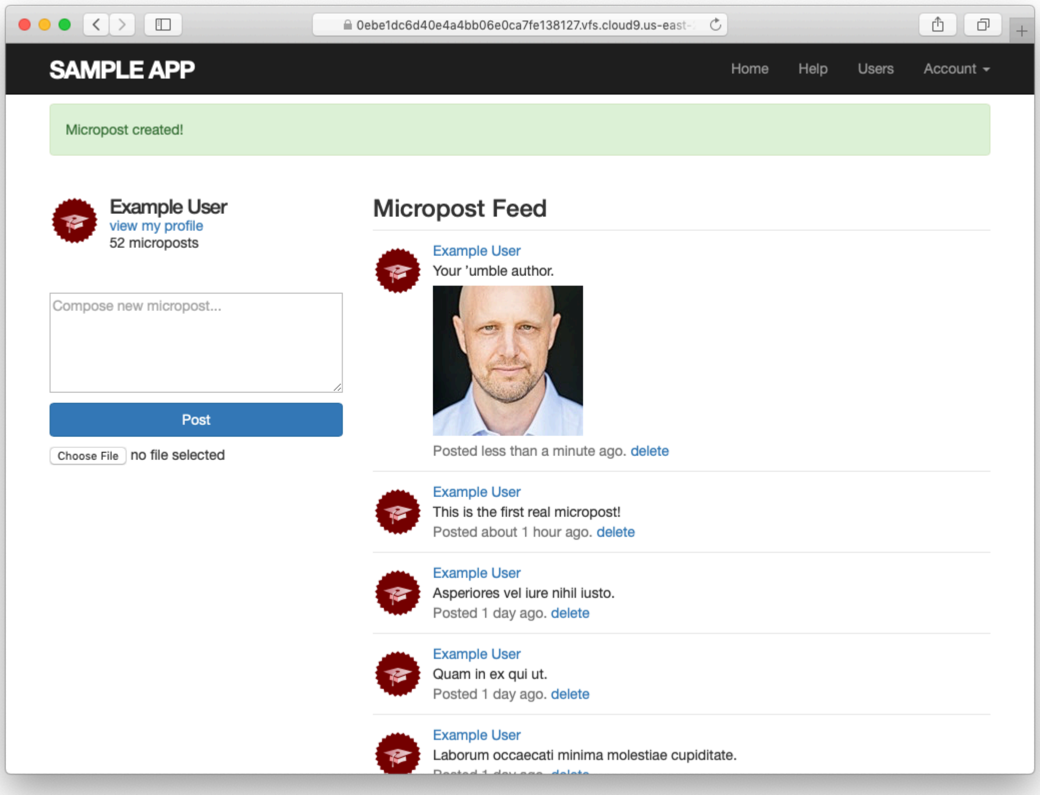
Exercises

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

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

1. 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 13.64, write a test of the image uploader in Section 13.4. As preparation, you should add an image to the fixtures directory using Listing 13.63. The additional assertions in Listing 13.64 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



Micropost created!

Example User
view my profile
52 microposts

Micropost Feed

Example User
Your 'umble author.
Posted less than a minute ago. delete

Example User
This is the first real micropost!
Posted about 1 hour ago. delete

Example User
Asperiores vel iure nihil iusto.
Posted 1 day ago. delete

Example User
Quam in ex qui ut.
Posted 1 day ago. delete

Example User
Laborum occaecati minima molestiae cupiditate.

Figure 13.25: The result of submitting a micropost with an image.

fixtures inside tests.[18] *Hint*: To check for a valid **`image`** 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: Downloading a fixture image for use in the tests.

```
$ curl -o test/fixtures/kitten.jpg -OL https://cdn.learnenough.com/kitten.jpg
```

Listing 13.64: 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
    assert_no_difference 'Micropost.count' do
      post microposts_path, params: { micropost: { content: "" } }
    end
    assert_select 'div#error_explanation'
    assert_select 'a[href=?]', '/?page=2'  # Correct pagination link
    # Valid submission
    content = "This micropost really ties the room together"
    image = fixture_file_upload('test/fixtures/kitten.jpg', 'image/jpeg')
    assert_difference 'Micropost.count', 1 do
      post microposts_path, params: { micropost:
                                      { content: content, image: image } }
    end
    assert FILL_IN.image.attached?
    follow_redirect!
    assert_match content, response.body
    # Delete a post.
    assert_select 'a', text: 'delete'
    first_micropost = @user.microposts.paginate(page: 1).first
```

[18]Windows users should add a **`:binary`** parameter: **`fixture_file_upload(file, type, :binary)`**.

```
    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 image upload code in Section 13.4.1 is a good start, but it has significant limitations. Among other things, 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.

As of this writing, Active Storage (somewhat surprisingly) doesn't offer native support for things like format and size validations, but as is so often the case there is a gem that adds it for us (Listing 13.65).

Listing 13.65: Adding a gem for Active Storage validations.
Gemfile

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                         '6.0.1'
gem 'active_storage_validations', '0.8.2'
gem 'bcrypt',                        '3.1.13'
.
.
.
```

Then **bundle install**:

```
$ bundle install
```

Following the gem documentation, we see that we can validate the image format by examining the `content_type` as follows:

```
content_type: { in: %w[image/jpeg image/gif image/png],
                message: "must be a valid image format" }
```

This checks that the MIME type of the image corresponds to a supported image format. (Recall the `%w[]` array-building syntax from Section 6.2.4.)

Similarly, we can validate the file size like this:

```
size: { less_than: 5.megabytes,
        message:   "should be less than 5MB" }
```

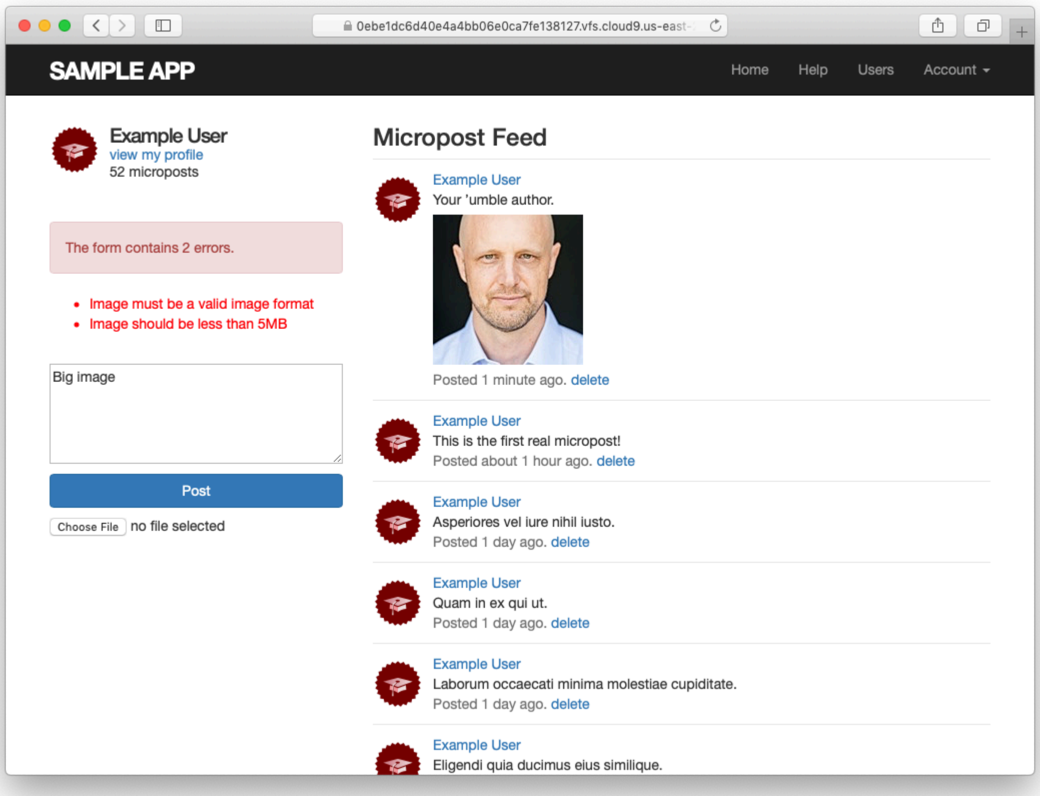
This sets a limit of 5 megabytes using a syntax we saw before in the context of time helpers (Box 9.1).

Adding these validations to the Micropost model gives the code in Listing 13.66.

Listing 13.66: Adding validations to images.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to        :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
  validates :image,   content_type: { in: %w[image/jpeg image/gif image/png],
                                      message: "must be a valid image format" },
                      size:         { less_than: 5.megabytes,
                                      message:   "should be less than 5MB" }
end
```

The result of trying to upload a large, invalid image then appears as in Figure 13.26. (You may have to restart the Rails server first.)



Example User
view my profile
52 microposts

The form contains 2 errors.

- Image must be a valid image format
- Image should be less than 5MB

Big image

Post

Choose File no file selected

Micropost Feed

Example User
Your 'umble author.
Posted 1 minute ago. delete

Example User
This is the first real micropost!
Posted about 1 hour ago. delete

Example User
Asperiores vel iure nihil iusto.
Posted 1 day ago. delete

Example User
Quam in ex qui ut.
Posted 1 day ago. delete

Example User
Laborum occaecati minima molestiae cupiditate.
Posted 1 day ago. delete

Example User
Eligendi quia ducimus eius similique.

Figure 13.26: Trying to upload a large, invalid image.

To go along with the validations in Listing 13.66, we'll add client-side (in-browser) checks on the uploaded image size and format. We'll start by including 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). The result appears in Listing 13.67.[19]

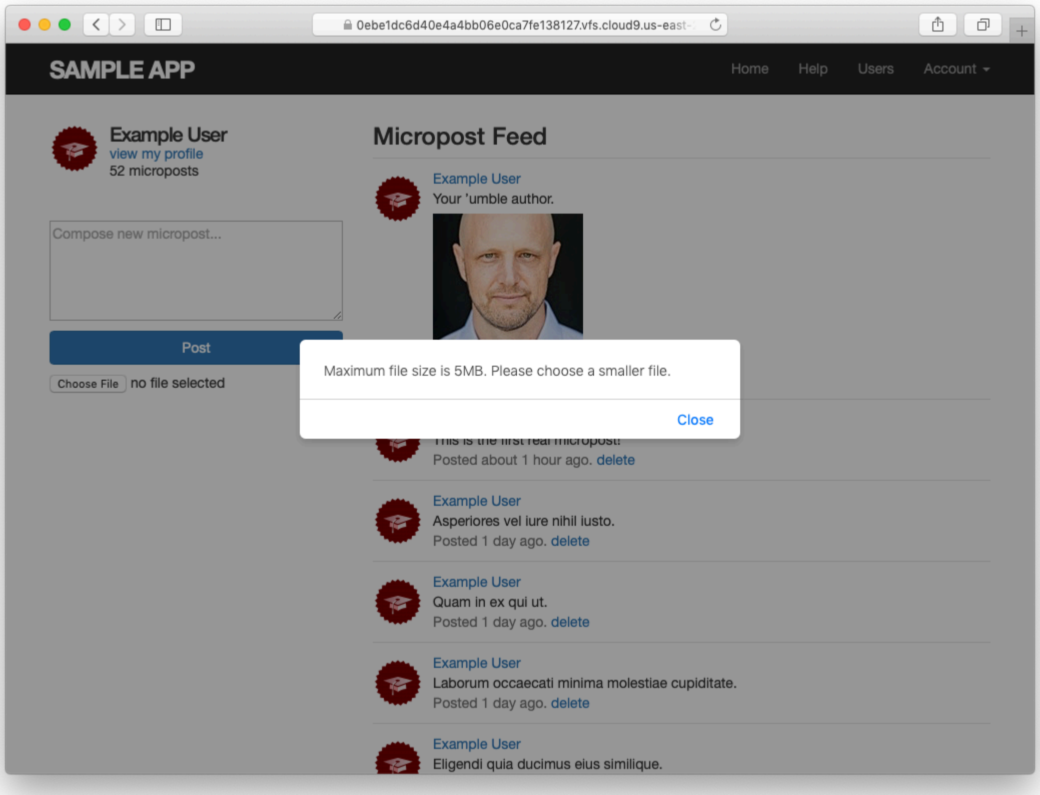
Listing 13.67: Checking the file size with jQuery.
app/views/shared/_micropost_form.html.erb

```
<%= form_with(model: @micropost, local: true) 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="image">
    <%= f.file_field :image %>
  </span>
<% end %>

<script type="text/javascript">
  $("#micropost_image").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.");
      $("#micropost_image").val("");
    }
  });
</script>
```

Although JavaScript isn't the focus of this book, you might be able to figure out that Listing 13.67 monitors the page element containing the CSS id **`micropost_image`** (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, as seen in Figure 13.27.[20]

[19]More advanced users of JavaScript would probably put the size check in its own function, but since this isn't a JavaScript tutorial the code in Listing 13.67 is fine for our purposes.

[20]To learn how to do things like this, you can do what I did: Google for things like "javascript maximum file size" until you find something on Stack Overflow.



Maximum file size is 5MB. Please choose a smaller file.

Example User
Your 'umble author.

This is the first real micropost!
Posted about 1 hour ago. delete

Example User
Asperiores vel iure nihil iusto.
Posted 1 day ago. delete

Example User
Quam in ex qui ut.
Posted 1 day ago. delete

Example User
Laborum occaecati minima molestiae cupiditate.
Posted 1 day ago. delete

Example User
Eligendi quia ducimus eius similique.

Figure 13.27: A JavaScript alert for a large file.

Finally, by using the **`accept`** parameter in the **`file_field`** input tag, we can specify that only valid formats should be allowed (Listing 13.68).

Listing 13.68: Allowing only valid image formats.
`app/views/shared/_micropost_form.html.erb`

```erb
<%= form_with(model: @micropost, local: true) 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="image">
    <%= f.file_field :image, accept: "image/jpeg,image/gif,image/png" %>
  </span>
<% end %>

<script type="text/javascript">
  $("#micropost_image").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.");
      $("#micropost_image").val("");
    }
  });
</script>
```

Listing 13.68 arranges to allow only valid image types to be selected in the first place, graying out any other file types (Figure 13.28).

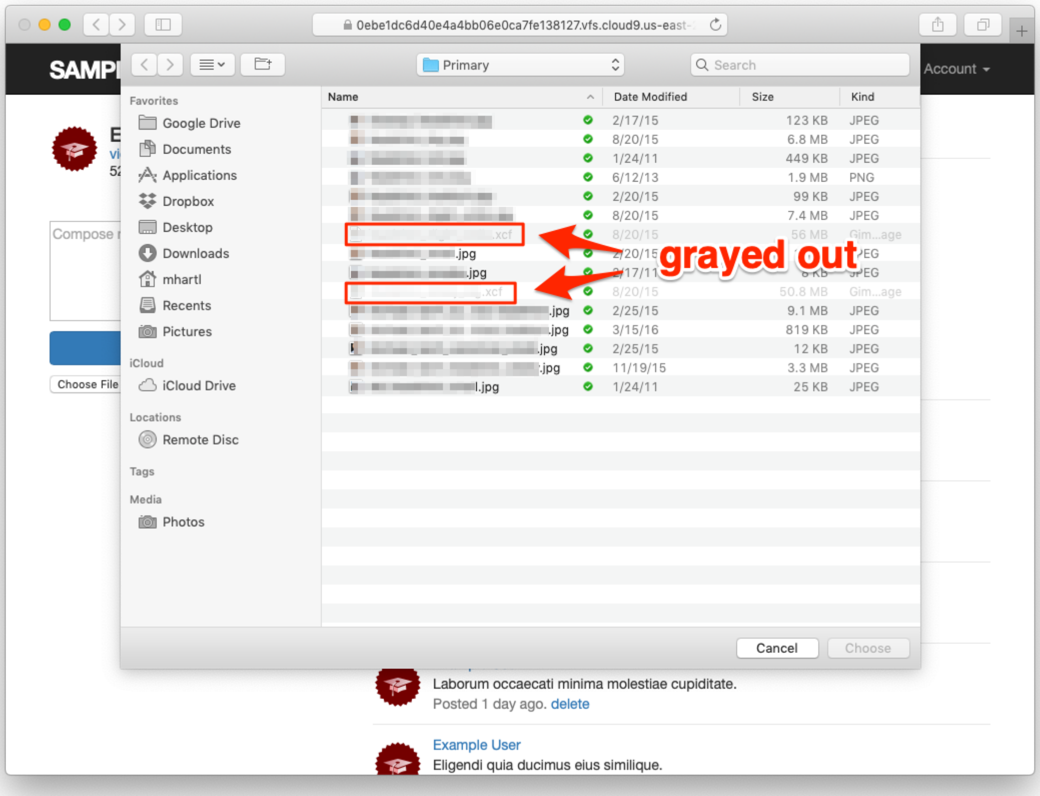
Preventing invalid images from being uploaded in a browser is a nice touch, but it's important to understand that this sort of code can only make it *difficult* to upload an invalid format or large file; a user determined to upload an invalid file can always issue a direct `POST` request using, e.g., **`curl`**. It is thus essential to include server-side validations of the type shown in Listing 13.66.

Exercises

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

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

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



Name	Date Modified	Size	Kind
[illegible]	2/17/15	123 KB	JPEG
[illegible]	8/20/15	6.8 MB	JPEG
[illegible]	1/24/11	449 KB	JPEG
[illegible]	6/12/13	1.9 MB	PNG
[illegible]	2/20/15	99 KB	JPEG
[illegible]	8/20/15	7.4 MB	JPEG
[illegible].xcf	8/20/15	56 MB	Gim...age
[illegible].jpg	2/20/15	[illegible]	[illegible]EG
[illegible].jpg	2/17/11	[illegible] KB	[illegible]EG
[illegible].xcf	8/20/15	50.8 MB	Gim...age
[illegible].jpg	2/25/15	9.1 MB	JPEG
[illegible].jpg	3/15/16	819 KB	JPEG
[illegible].jpg	2/25/15	12 KB	JPEG
[illegible].jpg	11/19/15	3.3 MB	JPEG
[illegible]l.jpg	1/24/11	25 KB	JPEG

Laborum occaecati minima molestiae cupiditate.
Posted 1 day ago. delete

Example User
Eligendi quia ducimus eius similique.

Figure 13.28: Grayed-out invalid file types.

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 13.29). 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]

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:

```
$ sudo apt-get -y install imagemagick
```

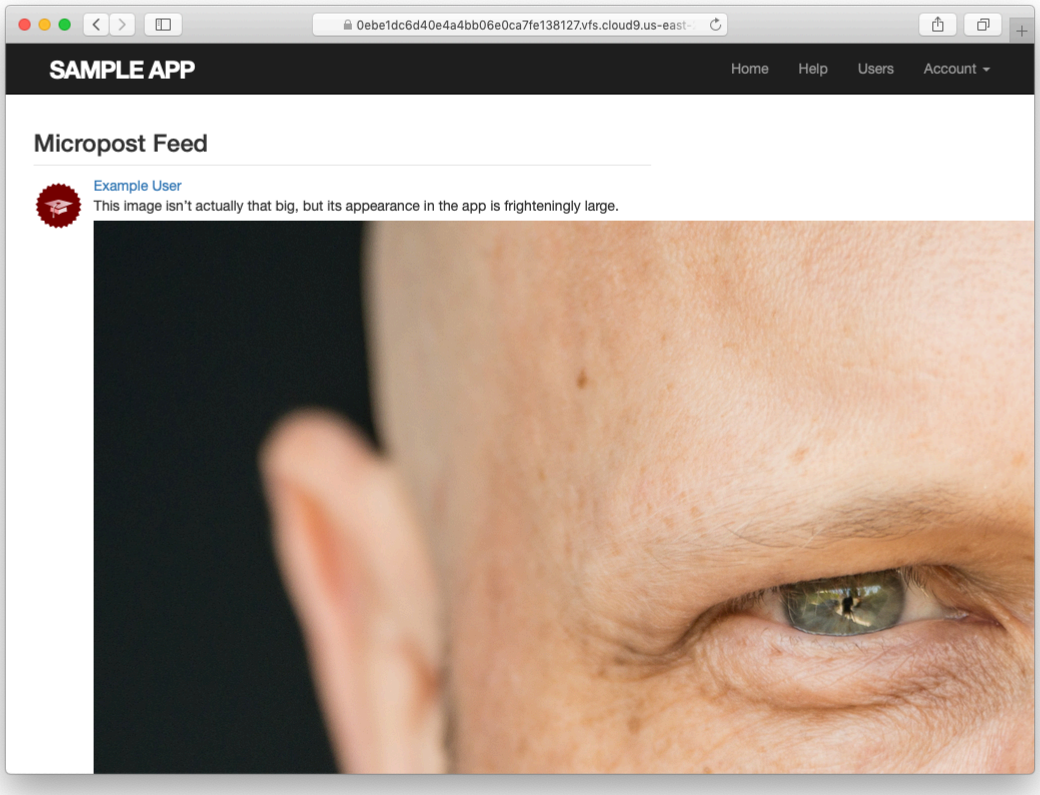
(If you're not using the cloud IDE or an equivalent Linux system, do a Google search for "imagemagick <your platform>". On macOS, **`brew install imagemagick`** should work if you have Homebrew installed. Use your technical sophistication (Box 1.2) if you get stuck.)

Next, we need to add a couple of gems for image processing, including the aptly named `image_processing` gem and `mini_magick`, a Ruby processor for ImageMagick (Listing 13.69).

Listing 13.69: Adding gems for image processing.
`Gemfile`

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                        '6.0.1'
gem 'image_processing',             '1.9.3'
```

[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.)



Micropost Feed

Example User

This image isn't actually that big, but its appearance in the app is frighteningly large.

Figure 13.29: A frighteningly large uploaded image.

```
gem 'mini_magick',                '4.9.5'
gem 'active_storage_validations', '0.8.2'
.
.
.
```

Then install as usual:

```
$ bundle install
```

You will probably need to restart the Rails server as well.

With the necessary software installed, we're now ready to use the **`variant`** method supplied by Active Storage for creating transformed images. In particular, we'll use the **`resize_to_limit`** option to ensure that neither the width nor the height of the image is greater than 500 pixels, as follows:

```
image.variant(resize_to_limit: [500, 500])
```

For convenience, we'll put this code in a separate **`display_image`** method, as shown in Listing 13.70.

Listing 13.70: Adding a resized display image.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to       :user
  has_one_attached :image
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
  validates :image,   content_type: { in: %w[image/jpeg image/gif image/png],
                                      message: "must be a valid image format" },
                      size: { less_than: 5.megabytes,
                              message:     "should be less than 5MB" }

  # Returns a resized image for display.
  def display_image
    image.variant(resize_to_limit: [500, 500])
  end
end
```

Finally, we can use **`display_image`** in the micropost partial, as shown in Listing 13.71.

Listing 13.71: Using the resized **`display_image`**.
`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.display_image if micropost.image.attached? %>
  </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 **`variant`** resizing in Listing 13.70 will happen on demand when the method is first called in Listing 13.71, and will be cached for efficiency in subsequent uses.[22] The result is a properly resized display image, as seen in Figure 13.30.

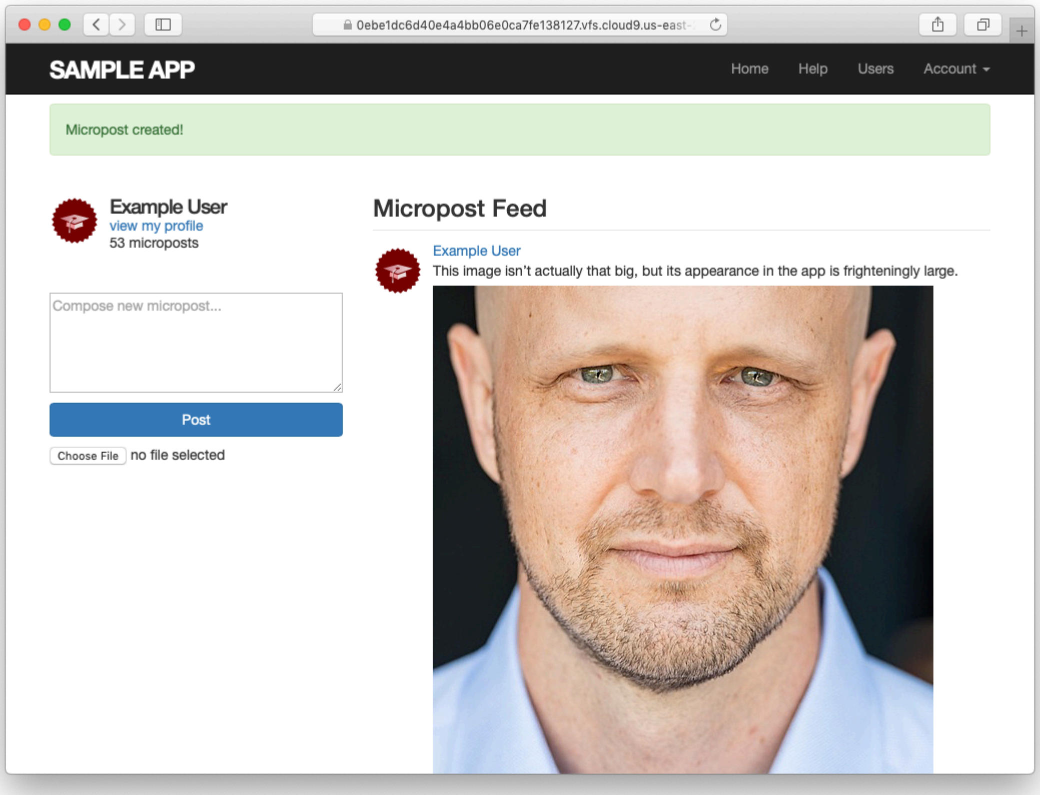
Exercises

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

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

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

[22]For larger sites, it's probably better to defer such processing to a background process; this method is beyond the scope of this tutorial, but investigating Active Job will get you started if you need to go this route.



Micropost created!

Example User
view my profile
53 microposts

Compose new micropost...

Micropost Feed

Example User
This image isn't actually that big, but its appearance in the app is frighteningly large.

Figure 13.30: A nicely resized image.

13.4.4 Image upload in production

The image uploading developed in Section 13.4.3 is good enough for development, but (as seen below in Listing 13.73) it uses the local disk for storing the images, which isn't a good practice in production. (Among other things, file storage on Heroku is temporary, so uploaded images will be deleted every time you deploy.) Instead, we'll use a cloud storage service to store images separately from our application.

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), part of Amazon Web Services (AWS).[23]

To configure our application to use cloud storage in production, we'll add the `aws-sdk-s3` gem to the **`:production`** environment, as shown in Listing 13.72.

Listing 13.72: Adding a gem for Amazon Web Services (AWS).
`Gemfile`

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                        '6.0.1'
gem 'image_processing',             '1.9.3'
gem 'mini_magick',                  '4.9.5'
gem 'active_storage_validations', '0.8.2'
.
.
.
group :production do
  gem 'pg',         '1.1.4'
  gem 'aws-sdk-s3', '1.46.0', require: false
end
.
.
.
```

Then **`bundle`** one more time:

[23] 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.

```
$ bundle install
```

AWS configuration

At this point, you'll need to configure your AWS system to use S3. Here are the basic steps:[24]

1. Sign up for an Amazon Web Services account if you don't have one already (Figure 13.31). (If you signed up for the Cloud9 IDE in Section 1.1.1, you already have an AWS account and can skip this step.)

2. Create a user via AWS Identity and Access Management (IAM). This involves using the IAM users interface (Figure 13.32) to navigate to the "Add user" page (Figure 13.33), where you should create a user while enabling "programmatic access" (Figure 13.34), grant the user administrator access (Figure 13.35), and then skip the optional user tags (Figure 13.36).

3. After clicking "Create user", you should see the name of the user together with the access key ID and the secret access key (Figure 13.37). Copy these keys and store them some place safe.

4. Create an S3 bucket using the AWS Console (Figure 13.38). S3 buckets exist in a global namespace, so the name has to be unique, but otherwise the default bucket settings should be fine.

You may find setting up S3 to be a challenging exercise in technical sophistication (Box 1.2); for further details on the steps above, consult the S3 documentation, the article "Setting up Rails 5 [or higher] with Active Storage with Amazon S3", and, if necessary, Google or Stack Overflow.

[24]The steps are current as of this writing, but services like AWS are constantly evolving, so the user interface may have changed in the interim. Use your technical sophistication to resolve any discrepancies.

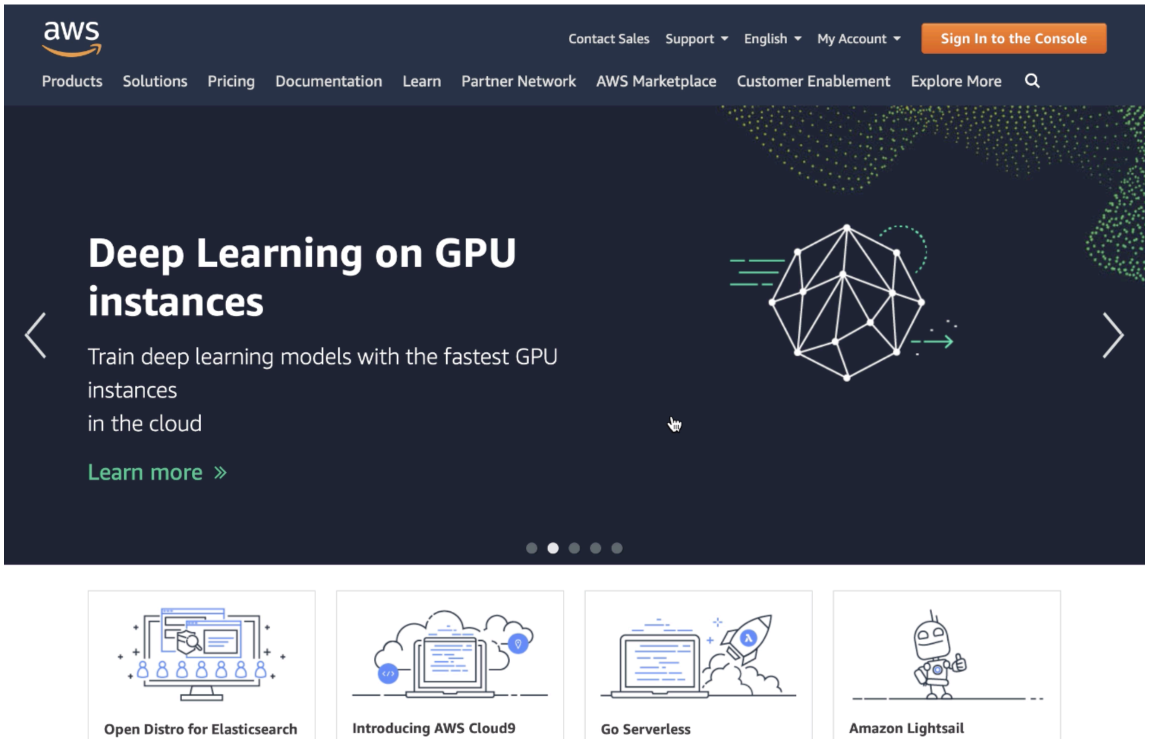


Figure 13.31: Signing up for AWS.

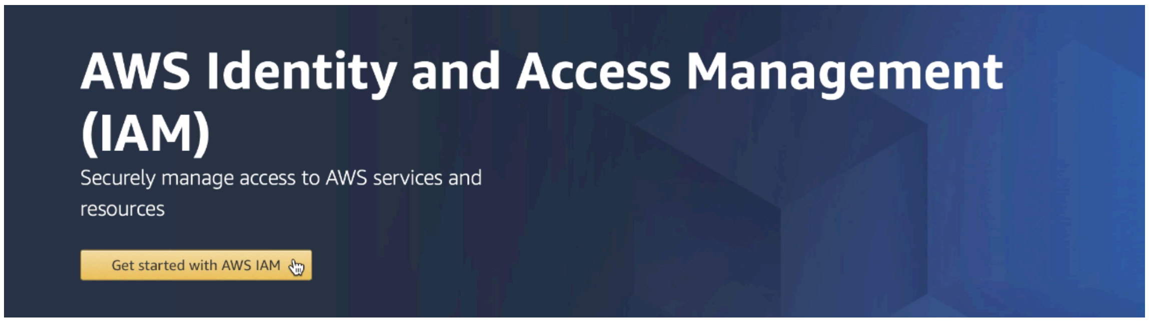

AWS Identity and Access Management (IAM)
Securely manage access to AWS services and resources
Get started with AWS IAM

Figure 13.32: The AWS IAM interface.

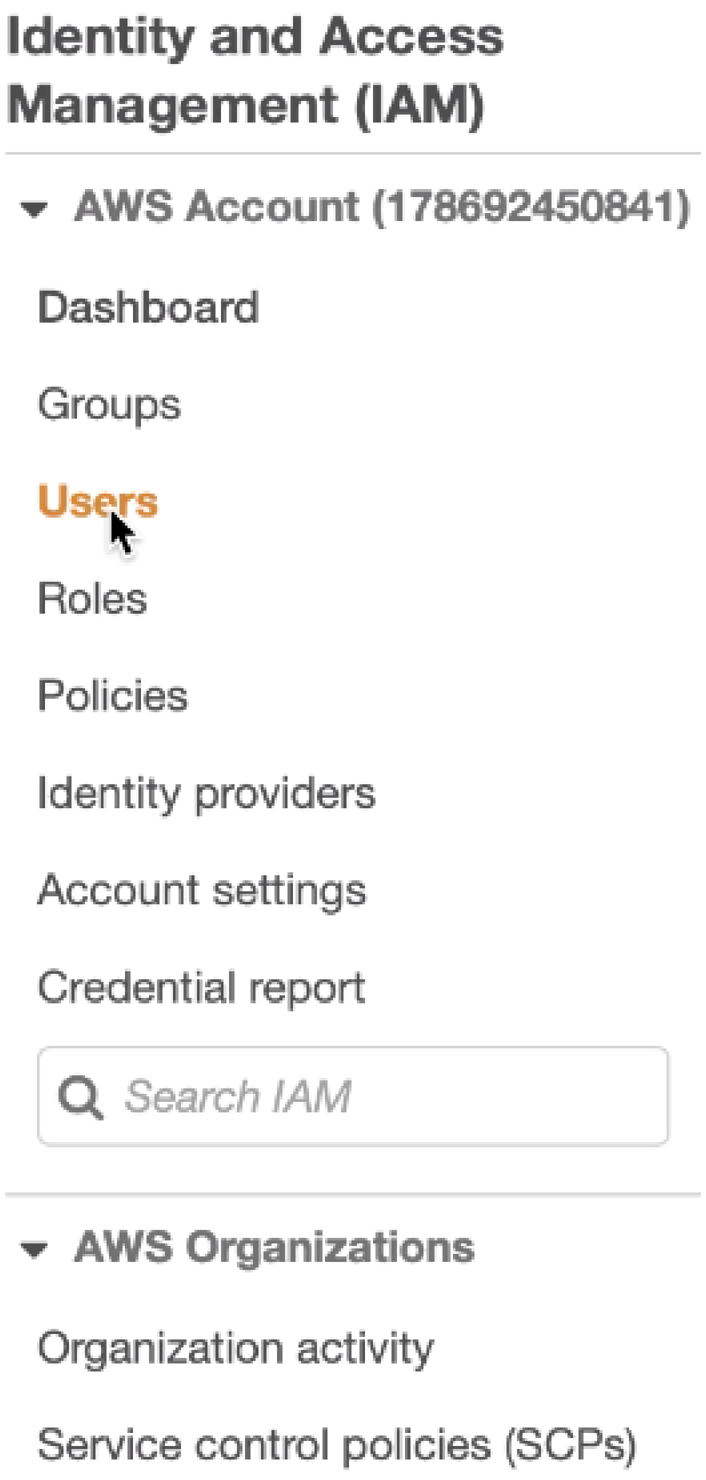
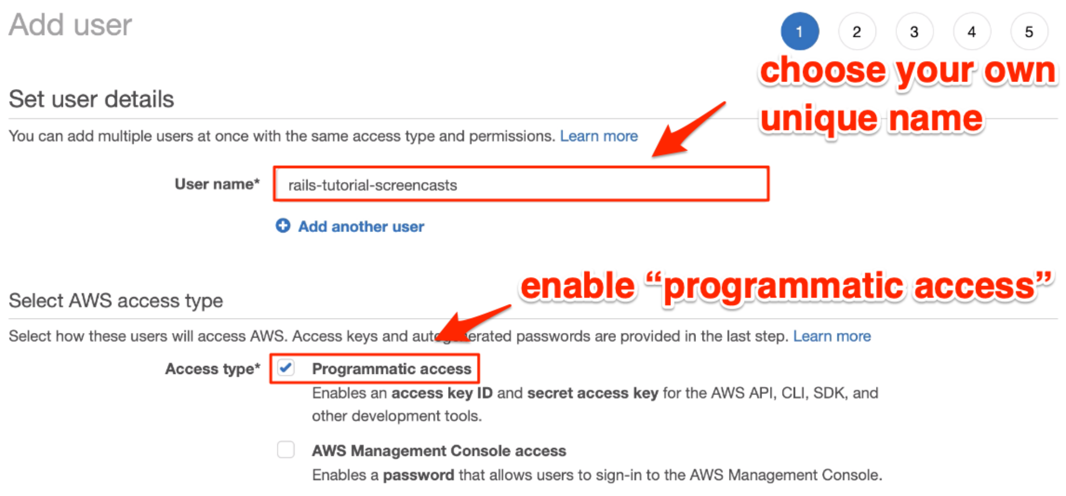


Figure 13.33: Navigating to the “Add user” page.



Add user

1 2 3 4 5

choose your own unique name

Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name* rails-tutorial-screencasts

Add another user

enable "programmatic access"

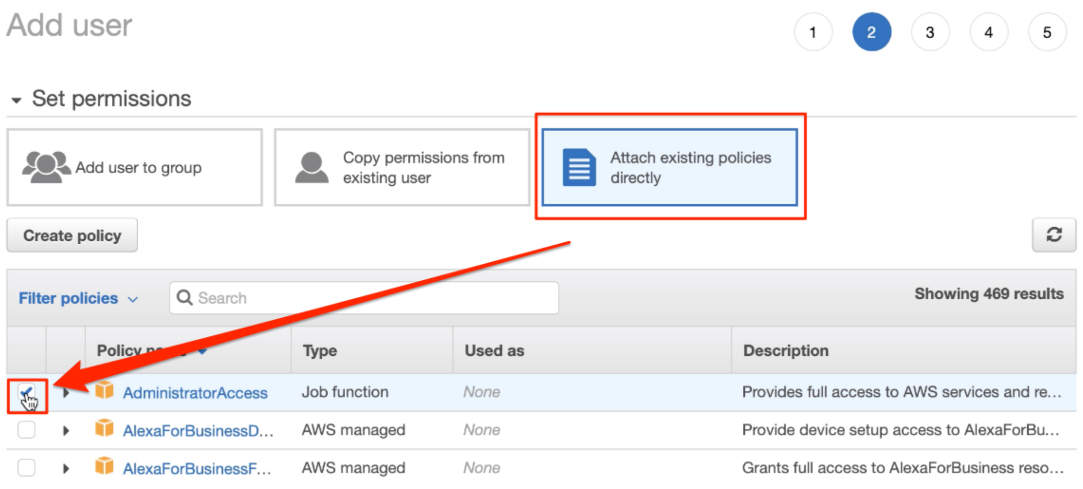
Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type* [✓] Programmatic access
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

[ ] AWS Management Console access
Enables a **password** that allows users to sign-in to the AWS Management Console.

Figure 13.34: Creating a user with "programmatic access".



Add user

1 2 3 4 5

Set permissions

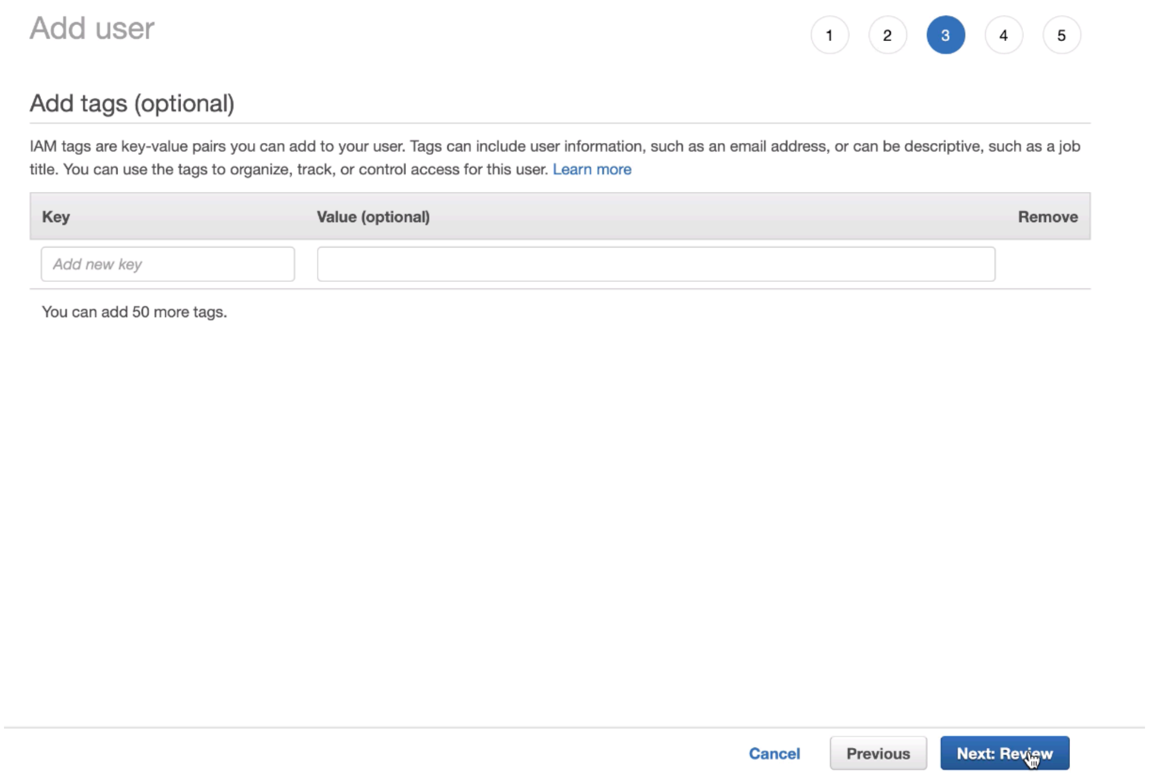
Add user to group | Copy permissions from existing user | Attach existing policies directly

Create policy

Filter policies | Search | Showing 469 results

		Policy name	Type	Used as	Description
✓	▸	AdministratorAccess	Job function	None	Provides full access to AWS services and re...
	▸	AlexaForBusinessD...	AWS managed	None	Provide device setup access to AlexaForBu...
	▸	AlexaForBusinessF...	AWS managed	None	Grants full access to AlexaForBusiness reso...

Figure 13.35: Granting the user administrator access.


Add user

1 2 3 4 5

Add tags (optional)

IAM tags are key-value pairs you can add to your user. Tags can include user information, such as an email address, or can be descriptive, such as a job title. You can use the tags to organize, track, or control access for this user. Learn more

Key	Value (optional)	Remove
Add new key		

You can add 50 more tags.

Figure 13.36: Skipping optional user tags.

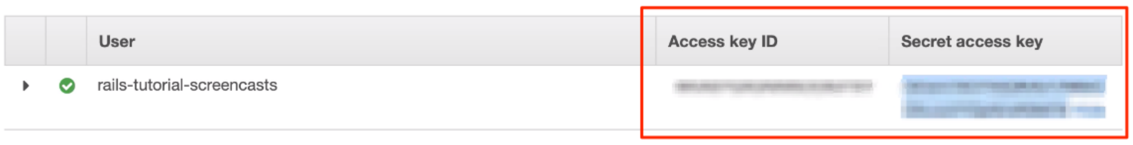
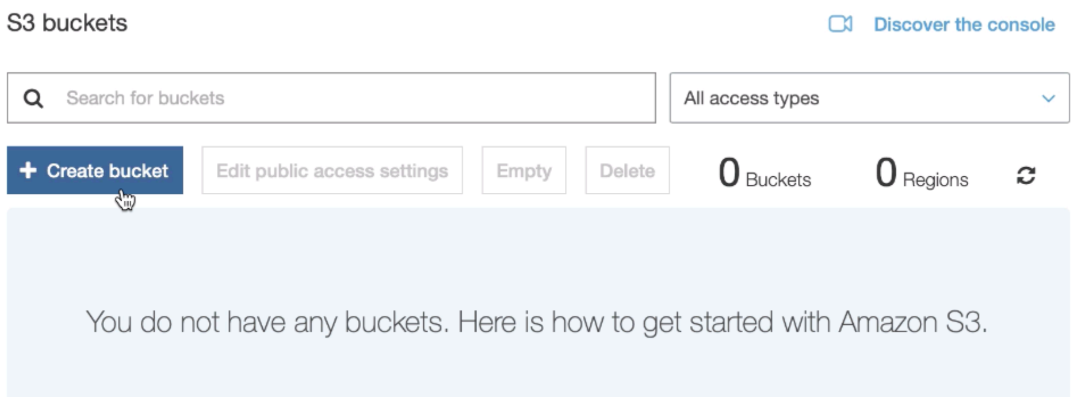

		User	Access key ID	Secret access key
▸	✓	rails-tutorial-screencasts		

Figure 13.37: Displaying the access key ID and the secret access key.



You do not have any buckets. Here is how to get started with Amazon S3.

Figure 13.38: Creating an AWS bucket.

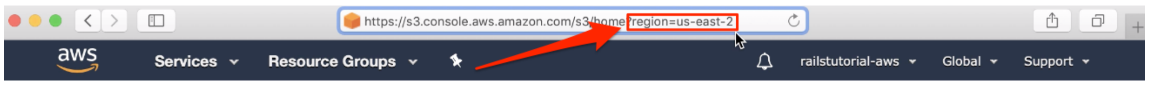


Figure 13.39: Getting the AWS region from the S3 console URL.

Production AWS

As with production email configuration (Listing 11.41), we'll be using Heroku **`ENV`** variables to avoid hard-coding sensitive information like AWS keys. 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. Per the gem configuration documentation, these variables should be named using the prefix **`AWS`**, which we can accomplish using **`heroku config:set`** as follows:

```
$ heroku config:set AWS_ACCESS_KEY=<access key>
$ heroku config:set AWS_SECRET_KEY=<secret key>
$ heroku config:set AWS_REGION=<region>
$ heroku config:set AWS_BUCKET=<bucket name>
```

You should paste in the values for your configuration in place of the placeholder values above. To get the region, you can inspect the URL on the S3 console page (Figure 13.39).

Once the Heroku variables are set, the next step is to use them in a special YAML file for configuring storage options called **`storage.yml`**. We can create a storage option for Amazon using the code in Listing 13.73.

Listing 13.73: Adding Amazon AWS as a storage option.
`config/storage.yml`

```
test:
  service: Disk
  root: <%= Rails.root.join("tmp/storage") %>

local:
  service: Disk
  root: <%= Rails.root.join("storage") %>

amazon:
  service: S3
  access_key_id:      <%= ENV['AWS_ACCESS_KEY_ID'] %>
  secret_access_key: <%= ENV['AWS_SECRET_ACCESS_KEY'] %>
  region:             <%= ENV['AWS_REGION'] %>
  bucket:             <%= ENV['AWS_BUCKET'] %>
```

Finally, we can put the option defined in Listing 13.73 to use in a production environment by adding the Active Storage service configuration parameter in **`production.rb`**. The result appears in Listing 13.74.

Listing 13.74: Configuring the production environment to use Amazon AWS (S3).
`config/environments/production.rb`

```
Rails.application.configure do
  .
  .
  .
  # Store uploaded files on Amazon AWS.
  config.active_storage.service = :amazon
  .
  .
  .
end
```

With the configuration above, we are ready to commit our changes and deploy:

```
$ rails test
$ git add -A
$ git commit -m "Add user microposts"
```

Because so many things can go wrong with the configuration, we'll deploy the app directly from our current topic branch, making sure it's working before merging into **`master`**. We can do this by including the branch name in the push to Heroku as follows:

```
$ git push heroku user-microposts:master
```

As usual, we then reset the database and reseed the sample data:

```
$ 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 13.40.

Exercises

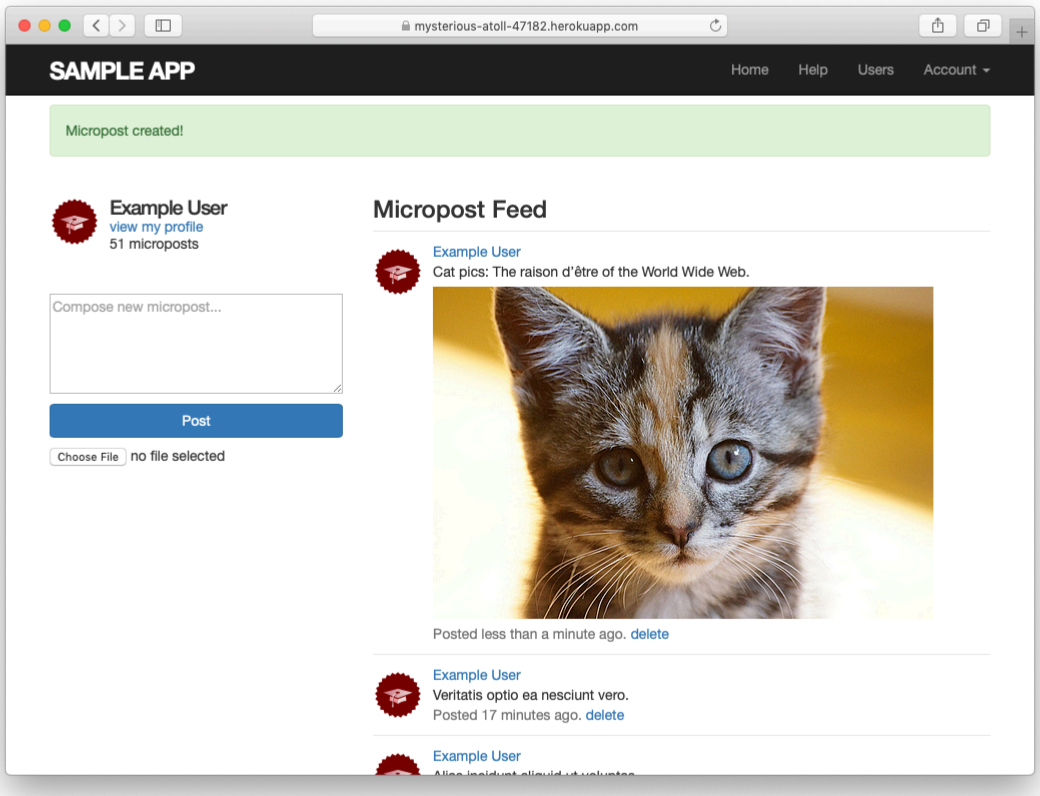
Solutions to the exercises are available to all Rails Tutorial purchasers here.

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

1. 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



Micropost created!

Example User
view my profile
51 microposts

Compose new micropost...

Micropost Feed

Example User
Cat pics: The raison d'être of the World Wide Web.
Posted less than a minute ago. delete

Example User
Veritatis optio ea nesciunt vero.
Posted 17 minutes ago. delete

Example User

Figure 13.40: An uploaded image in production.

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 your changes:

```
$ rails test
$ git add -A
$ git commit -m "Add user microposts"
```

Then merge into `master`:

```
$ git checkout master
$ git merge user-microposts
$ git push
```

And finally 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 13.75.[25]

Listing 13.75: The final `Gemfile` for the sample application.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',                        '6.0.1'
gem 'image_processing',             '1.9.3'
gem 'mini_magick',                  '4.9.5'
gem 'active_storage_validations',   '0.8.2'
gem 'bcrypt',                       '3.1.13'
gem 'faker',                        '2.1.2'
gem 'will_paginate',                '3.1.8'
```

[25]As always, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed here.

```ruby
gem 'bootstrap-will_paginate',    '1.0.0'
gem 'bootstrap-sass',             '3.4.1'
gem 'puma',                       '3.12.1'
gem 'sass-rails',                 '5.1.0'
gem 'webpacker',                  '4.0.7'
gem 'turbolinks',                 '5.2.0'
gem 'jbuilder',                   '2.9.1'
gem 'bootsnap',                   '1.4.4', require: false

group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.0.1'
  gem 'listen',                '3.1.5'
  gem 'spring',                '2.1.0'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',                 '3.28.0'
  gem 'selenium-webdriver',       '3.142.4'
  gem 'webdrivers',               '4.1.2'
  gem 'rails-controller-testing', '1.0.4'
  gem 'minitest',                 '5.11.3'
  gem 'minitest-reporters',       '1.3.8'
  gem 'guard',                    '2.15.0'
  gem 'guard-minitest',           '2.4.6'
end

group :production do
  gem 'pg',         '1.1.4'
  gem 'aws-sdk-s3', '1.46.0', require: false
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 images using Active Storage.