

Chapter 11 User microposts

11.1 A Micropost model

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

11.1.1 The basic model

The Micropost model needs only two attributes: a `content` attribute to hold the micropost's content,[2](#) and a `user_id` to associate a micropost with a particular user. As with the case of the User model ([Listing 6.1](#)), we generate it using `generate_rspec_model`:

```
$ script/generate rspec_model Micropost content:string user_id:integer
```

This produces a migration to create a `microposts` table in the database ([Listing 11.1](#)); compare it to the analogous migration for the `users` table from [Listing 6.2](#).

Listing 11.1. The Micropost migration. (Note the index on `user_id`.)

db/migrate/<timestamp>_create_microposts.rb



```
class CreateMicroposts < ActiveRecord::Migration
  def self.up
    create_table :microposts do |t|
      t.string :content
      t.integer :user_id

      t.timestamps
    end
    add_index :microposts, :user_id
  end

  def self.down
    drop_table :microposts
  end
end
```

Note that, since we expect to retrieve all the microposts associated with a given user id, [Listing 11.1](#) adds an index ([Box 6.2](#)) on the `user_id` column:

```
add_index :microposts, :user_id
```

Note also the `t.timestamps` line, which (as mentioned in [Section 6.1.1](#)) adds the magic `created_at` and `updated_at` columns. We'll put the `created_at` column to work in [Section 11.1.3](#) and [Section 11.2.1](#).

We can run the microposts migration as usual (taking care to prepare the test database since the data model has changed):

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

The result is a Micropost model with the structure shown in [Figure 11.1](#).



microposts	
id	integer
content	string
user_id	integer
created_at	datetime
updated_at	datetime


Figure 11.1: The Micropost data model.

Accessible attribute

Before fleshing out the Micropost model, it's important first to use `attr_accessible` to indicate the attributes editable through the web. As discussed in [Section 6.1.2.2](#) and [Section 10.4.1.1](#), failing to define accessible attributes means that anyone could change any aspect of a micropost object simply by using a command-line client to issue malicious requests. For example, a malicious user could change the `user_id` attributes on microposts, thereby associating microposts with the wrong users.

In the case of the Micropost model, there is only *one* attribute that needs to be editable through the web, namely, the `content` attribute ([Listing 11.2](#)).

Listing 11.2. Making the `content` attribute (and *only* the `content` attribute) accessible.

```
app/models/micropost.rb   
class Micropost < ActiveRecord::Base  
  attr_accessible :content  
end
```


Since `user_id` *isn't* listed as an `attr_accessible` parameter, it can't be edited through the web, because a `user_id` parameter in a mass assignment such as

```
Micropost.new(:content => "foo bar", :user_id => 17)
```

will simply be ignored.

The `attr_accessible` declaration in [Listing 11.2](#) is necessary for site security, but it introduces a problem in the default Micropost model spec ([Listing 11.3](#)).

Listing 11.3. The initial (lightly edited) Micropost spec.

```
spec/models/micropost_spec.rb   
require 'spec_helper'  
  
describe Micropost do  
  before(:each) do
```

```

    @attr = {
      :content => "value for content",
      :user_id => 1
    }
  end

  it "should create a new instance given valid attributes" do
    Micropost.create!(@attr)
  end
end

```

(As in [Listing 6.8](#), I've replaced `@valid_attributes` with the more succinct `@attr`.) This test currently passes, but there's something fishy about it. (See if you can figure out what before proceeding.)

The problem is that the `before(:each)` block in [Listing 11.3](#) assigns the user id through mass assignment, which is exactly what `attr_accessible` is designed to prevent; in particular, as noted above, the `:user_id => 1` part of the initialization hash is simply ignored. The solution is to avoid using `Micropost.new` directly; instead, we will create the new micropost through its *association* with the User model, which sets the user id automatically. Accomplishing this is the task of the next section.

11.1.2 User/Micropost associations

The goal of this section is to establish an *association* between the Micropost model and the User model—a relationship seen briefly in [Section 2.3.3](#) and shown schematically in [Figure 11.2](#) and [Figure 11.3](#). Along the way, we'll write tests for the Micropost model that, unlike [Listing 11.3](#), are compatible with the use of `attr_accessible` in [Listing 11.2](#).



Figure 11.2: The `belongs_to` relationship between a micropost and its user. ([full size](#))

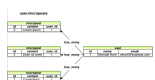


Figure 11.3: The `has_many` relationship between a user and its microposts. ([full size](#))

We start with tests for the Micropost model association. First, we want to replicate the `Micropost.create!` test shown in [Listing 11.3](#) without the invalid mass assignment. Second, we see from [Figure 11.2](#) that a `micropost` object should have a `user` method. Finally, `micropost.user` should be the user corresponding to the micropost's `user_id`. We can express these requirements in RSpec with the code in [Listing 11.4](#).

Listing 11.4. Tests for the micropost's user association.

`spec/models/micropost_spec.rb`



```
require 'spec_helper'
```

```
describe Micropost do
```

```

before(:each) do
  @user = Factory(:user)
  @attr = { :content => "value for content" }
end

it "should create a new instance given valid attributes" do
  @user.microposts.create!(@attr)
end

describe "user associations" do

  before(:each) do
    @micropost = @user.microposts.create(@attr)
  end

  it "should have a user attribute" do
    @micropost.should respond_to(:user)
  end

  it "should have the right associated user" do
    @micropost.user_id.should == @user.id
    @micropost.user.should == @user
  end
end
end

```

Note that, rather than using `Micropost.create` or `Micropost.create!` to create a micropost, [Listing 11.4](#) uses

```
@user.microposts.create(@attr)
```

and

```
@user.microposts.create!(@attr)
```

This pattern is the [canonical](#) way to create a micropost through its association with users. (We use a factory user because these tests are for the Micropost model, not the User model.) When created in this way, the micropost object *automatically* has its `user_id` set to the right value, which fixes the issue noted in [Section 11.1.1.1](#). In particular, the code

```

before(:each) do
  @attr = {
    :content => "value for content",
    :user_id => 1
  }
end

it "should create a new instance given valid attributes" do
  Micropost.create!(@attr)
end

```

from [Listing 11.3](#) is defective because `:user_id => 1` does nothing when `user_id` is not one of the Micropost model's accessible attributes. By going through the user association, on the other hand, the code


```
it "should create a new instance given valid attributes" do
```

```
@user.microposts.create!(@attr)
end
```

from [Listing 11.4](#) has the right `user_id` by construction.

These special `create` methods won't work yet; they require the proper `has_many` association in the `User` model. We'll defer the more detailed tests for this association to [Section 11.1.3](#); for now, we'll simply test for the presence of a `microposts` attribute ([Listing 11.5](#)).

Listing 11.5. A test for the user's `microposts` attribute.

```
spec/models/user_spec.rb 
require 'spec_helper'


describe User do
  :
  :
  :
  describe "micropost associations" do

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

    it "should have a microposts attribute" do
      @user.should respond_to(:microposts)
    end
  end
end
```


We can get the tests in both [Listing 11.4](#) and [Listing 11.5](#) to pass using the `belongs_to/has_many` association illustrated in [Figure 11.2](#) and [Figure 11.3](#), as shown in [Listing 11.6](#) and [Listing 11.7](#).

Listing 11.6. A micropost `belongs_to` a user.

```
app/models/micropost.rb 
class Micropost < ActiveRecord::Base
  attr_accessible :content

  belongs_to :user
end
```

Listing 11.7. A user `has_many` microposts.

```
app/models/user.rb 
class User < ActiveRecord::Base
  attr_accessor :password
  attr_accessible :name, :email, :password, :password_confirmation

  has_many :microposts
  :
  :
```

end

Using this `belongs_to/has_many` association, Rails constructs the methods shown in [Table 11.1](#). You should compare the entries in [Table 11.1](#) with the code in [Listing 11.4](#) and [Listing 11.5](#) to satisfy yourself that you understand the basic nature of the associations. (There is one method in [Table 11.1](#) we haven't used so far, the `build` method; it will be put to good use in [Section 11.1.4](#) and especially in [Section 11.3.2](#).)

Method	Purpose
<code>micropost.user</code>	Return the User object associated with the micropost.
<code>user.microposts</code>	Return an array of the user's microposts.
<code>user.microposts.create(arg)</code>	Create a micropost (<code>user_id = user.id</code>).
<code>user.microposts.create!(arg)</code>	Create a micropost (exception on failure).
<code>user.microposts.build(arg)</code>	Return a new Micropost object (<code>user_id = user.id</code>).


Table 11.1: A summary of user/micropost association methods.

[11.1.3 Micropost refinements](#)

The test in [Listing 11.5](#) of the `has_many` association doesn't test for much—it merely verifies the *existence* of a `microposts` attribute. In this section, we'll add *ordering* and *dependency* to microposts, while also testing that the `user.microposts` method actually returns an array of microposts

We will need to construct some microposts in the User model spec, which means that we should make a micropost factory at this point. To do this, we need a way to make an association in Factory Girl. Happily, this is easy—we just use the Factory Girl method `micropost.association`, as seen in [Listing 11.8.3](#)

Listing 11.8. The complete factory file, including a new factory for microposts.

```
spec/factories.rb   
  
# By using the symbol ':user', we get Factory Girl to simulate the User model.  
Factory.define :user do |user|  
  user.name "Michael Hartl"  
  user.email "mhartl@example.com"  
  user.password "foobar"  
  user.password_confirmation "foobar"  
end  
  
Factory.sequence :email do |n|  
  "person-#{n}@example.com"  
end  
  
Factory.define :micropost do |micropost|  
  micropost.content "Foo bar"  
  micropost.association :user  
end
```

Default scope


We can put the micropost factory to work in a test for the ordering of microposts. By default, using `user.microposts` to pull a user's microposts from the database 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, i.e., most recent first. To test this ordering, we first create a couple of microposts as follows:

```
@mp1 = Factory(:micropost, :user => @user, :created_at => 1.day.ago)
@mp2 = Factory(:micropost, :user => @user, :created_at => 1.hour.ago)
```

Here we indicate that the second post was created more recently, `1.hour.ago`, with the first post created `1.day.ago`. Note how convenient the use of Factory Girl is: not only can we assign the user using mass assignment (since factories bypass `attr_accessible`), we can also set `created_at` manually, which Active Record won't allow us to do.[4](#)

Most database adapters (including the one for SQLite) return the microposts in order of their ids, so we can arrange for an initial test that almost certainly fails using the code in [Listing 11.9](#).

Listing 11.9. Testing the order of a user's microposts.

`spec/models/user_spec.rb` 

```
require 'spec_helper'

describe User do
  .
  .
  .
  describe "micropost associations" do

    before(:each) do
      @user = User.create(@attr)
      @mp1 = Factory(:micropost, :user => @user, :created_at => 1.day.ago)
      @mp2 = Factory(:micropost, :user => @user, :created_at => 1.hour.ago)
    end

    it "should have a microposts attribute" do
      @user.should respond_to(:microposts)
    end

    it "should have the right microposts in the right order" do
      @user.microposts.should == [@mp2, @mp1]
    end
  end
end
```

The key line here is


```
@user.microposts.should == [@mp2, @mp1]
```

indicating that the posts should be ordered newest first. This should fail because by default the posts will be ordered by id, i.e., `[@mp1, @mp2]`. This test also verifies the basic correctness of the

`has_many` association itself, by checking (as indicated in [Table 11.1](#)) that `user.microposts` is an array of microposts.

To get the ordering test to pass, we use a Rails facility called `default_scope` with an `:order` parameter, as shown in [Listing 11.10](#). (This is our first example of the notion of *scope*. We will learn about scope in a more general context in [Chapter 12](#).)

Listing 11.10. Ordering the microposts with `default_scope`.

`app/models/micropost.rb` 


```
class Micropost < ActiveRecord::Base
  .
  .
  .
  default_scope :order => 'created_at DESC'
end
```

The order here is `'created_at DESC'`, where `DESC` is SQL for “descending”, i.e., in descending order from newest to oldest.

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 test for this by first destroying a micropost’s user and then verifying that the associated microposts are no longer in the database ([Listing 11.11](#)).

Listing 11.11. Testing that microposts are destroyed when users are.

`spec/models/user_spec.rb` 

```
describe User do
  .
  .
  .
  describe "micropost associations" do

    before(:each) do
      @user = User.create(@attr)
      @mp1 = Factory(:micropost, :user => @user, :created_at => 1.day.ago)
      @mp2 = Factory(:micropost, :user => @user, :created_at => 1.hour.ago)
    end
    .
    .
    .
    it "should destroy associated microposts" do
      @user.destroy
      [@mp1, @mp2].each do |micropost|
        Micropost.find_by_id(micropost.id).should be_nil
      end
    end
  end
end
```



```

.
.
.
end

```

Here we have used `Micropost.find_by_id`, which returns `nil` if the record is not found, whereas `Micropost.find` raises an exception on failure, which is a bit harder to test for. (In case you're curious,

```


lambda do
  Micropost.find(micropost.id)
end.should raise_error(ActiveRecord::RecordNotFound)

```

does the trick in this case.)

The application code to get [Listing 11.11](#) to pass is less than one line; in fact, it's just an option to the `has_many` association method, as shown in [Listing 11.12](#).

Listing 11.12. Ensuring that a user's microposts are destroyed along with the user.

`app/models/user.rb` 

```

class User < ActiveRecord::Base
.
.
.
  has_many :microposts, :dependent => :destroy
.
.
.
end


```

With that, the final form of the user/micropost association is in place.

11.1.4 Micropost validations

Before leaving the `Micropost` model, we'll tie off a couple loose ends by adding validations (following the example from [Section 2.3.2](#)). Both the `user_id` and `content` attributes are required, and `content` is further constrained to be shorter than 140 characters, which we test for using the code in [Listing 11.13](#).

Listing 11.13. Tests for the `Micropost` model validations.

`spec/models/micropost_spec.rb` 

```

require 'spec_helper'

describe Micropost do
  before(:each) do
    @user = Factory(:user)
    @attr = { :content => "value for content" }
  end
.

```



```

    default_scope :order => 'created_at DESC'
  end

```

This completes the data modeling for users and microposts. It's time now to build the web interface.

11.2 Showing microposts

Although we don't yet have a way to create microposts through the web—that comes in [Section 11.3.2](#)—that 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 11.4](#). 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 sample data populator from [Section 10.3.2](#) so that we have something to display.



Figure 11.4: A mockup of a profile page with microposts. ([full size](#))

As with the discussion of the signin machinery in [Section 9.3.2](#), [Section 11.2.1](#) will often push several elements onto the [stack](#) at a time, and then pop them off one by one. If you start getting bogged down, be patient; there's some nice payoff in [Section 11.2.2](#).

11.2.1 Augmenting the user show page

We begin with a test for displaying the user's microposts. We work in the Users controller spec, since it is the Users controller that contains the user `show` action. Our strategy is to create a couple of factory microposts associated with the user, and then verify that the show page has a `span` tag with CSS class "content" containing each post's content. The resulting RSpec example appears in [Listing 11.15](#).

Listing 11.15. A test for showing microposts on the user `show` page.
 spec/controllers/users_controller_spec.rb



```

require 'spec_helper'

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

    before(:each) do
      @user = Factory(:user)
      # Arrange for User.find(params[:id]) to find the right user.
      User.stub!(:find, @user.id).and_return(@user)
    end

    .
  end
end

```

```

.
.
it "should show the user's microposts" do
  mp1 = Factory(:micropost, :user => @user, :content => "Foo bar")
  mp2 = Factory(:micropost, :user => @user, :content => "Baz quux")
  get :show, :id => @user
  response.should have_tag("span.content", mp1.content)
  response.should have_tag("span.content", mp2.content)
end
end
.
.
.
end

```

Although these tests won't pass until [Listing 11.17](#), we'll get started on the application code by inserting a table of microposts into the user profile page, as shown in [Listing 11.16.5](#)

Listing 11.16. Adding microposts to the user show page.

app/views/users/show.html.erb



```

<table class="profile">
  <tr>
    <td class="main">
      .
      .
      .
      <% unless @user.microposts.empty? %>
        <table class="microposts">
          <%= render @microposts %>
        </table>
        <%= will_paginate @microposts %>
      <% end %>
    </td>
    <td class="sidebar round">
      <strong>Name</strong> <%= h @user.name %><br />
      <strong>URL</strong> <%= link_to user_path(@user), @user %><br />
      <strong>Microposts</strong> <%= @user.microposts.count %>
    </td>
  </tr>
</table>

```

We'll deal with the microposts **table** momentarily, but there are several other things to note first. One new idea is the use of **empty?** in the line

```
@user.microposts.empty?
```

This applies the **empty?** method, seen before in the context of strings (e.g., [Section 4.2.3](#)), to an array:

```

$ script/console
>> [1, 2].empty?
=> false
>> [].empty?
=> true

```

By using the conditional **unless** clause,

```
<% unless @user.microposts.empty? %>
```

we make sure that an empty table won't be displayed when the user has no microposts.

You'll also note from [Listing 11.16](#) that we've preemptively added pagination for microposts through

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

If you compare this with the analogous line on the user index page, [Listing 10.26](#), 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 `WillPaginate::Collection`). In the present case, since we are still in the Users controller but want to paginate *microposts* instead, we 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 11.18](#) below).

Finally, note that we have taken this opportunity to add a count of the current number of microposts to the profile sidebar:

```
<td class="sidebar round">
  <strong>Name</strong> <%= h @user.name %><br />
  <strong>URL</strong> <%= link_to user_path(@user), @user %><br />
  <strong>Microposts</strong> <%= @user.microposts.count %>
</td>
```

Here `@user.microposts.count` is the analogue of the `User.count` method, except that it counts the microposts belonging to a given user through the user/micropost association.[6](#)

Now for the microposts `table` itself:

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

This code is responsible for generating the table of microposts, but you can see that it just defers the heavy lifting to a micropost partial. We saw in [Section 10.3.4](#) that the code

```
<%= render @users %>
```

automatically renders each of the users in the `@users` variable using the `_user.html.erb` partial. Similarly, the code

```
<%= render @microposts %>
```

does exactly the same thing for microposts. This means that we must define a `_micropost.html.erb` partial, as shown in [Listing 11.17](#).

Listing 11.17. A partial for showing a single micropost.
app/views/microposts/_micropost.html.erb



```
<tr>
```


```

<td class="micropost">
  <span class="content"><%= h micropost.content %></span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
  </span>
</td>
</tr>

```

This uses the awesome `time_ago_in_words` helper method, whose effect we will see in [Section 11.2.2](#).

Thus far, despite defining all the relevant ERb templates, the test in [Listing 11.15](#) should have been failing for want of an `@microposts` variable. We can get it to pass with [Listing 11.18](#).

Listing 11.18. 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])
    @title = CGI.escapeHTML(@user.name)
  end
end

```

Notice here how clever `paginate` is—it even works with the microposts association, converting the array into a `WillPaginate::Collection` object on the fly.

Upon adding the CSS from [Listing 11.19](#) to our `custom.css` stylesheet,⁷ we can get a look at our new user profile page in [Figure 11.5](#). It's rather... disappointing. Of course, this is because there are not currently any microposts. It's time to change that.

Listing 11.19. The CSS for microposts (includes all the CSS for this chapter).
`public/stylesheets/custom.css` 

```

.
.
.
h2.micropost {
  margin-bottom: 0;
}

table.microposts tr {
  height: 70px;
}

table.microposts tr td.gravatar {
  border-top: 1px solid #ccc;
  vertical-align: top;
  width: 50px;
}

```

```

table.microposts tr td.micropost {
  border-top: 1px solid #ccc;
  vertical-align: top;
  padding-top: 10px;
}

table.microposts tr td.micropost span.timestamp {
  display: block;
  font-size: 85%;
  color: #666;
}

div.user_info img {
  padding-right: 0.1em;
}

div.user_info a {
  text-decoration: none;
}

div.user_info span.user_name {
  position: absolute;
}

div.user_info span.microposts {
  font-size: 80%;
}

form.new_micropost {
  margin-bottom: 2em;
}

form.new_micropost textarea {
  height: 4em;
  margin-bottom: 0;
}

```



Figure 11.5: The user profile page with code for microposts—but no microposts. ([full size](#))

11.2.2 Sample microposts


With all the work making templates for user microposts in [Section 11.2.1](#), the ending was rather anticlimactic. We can rectify this sad situation by adding microposts to the sample populator 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 users⁸ using the `:limit` option to the `User.all` method:⁹

```
User.all(:limit => 6)
```

We then make 50 microposts for each user (plenty to overflow the pagination limit of 30), generating sample content for each micropost using the Faker gem's handy [Lorem.sentence method](#).

(`Faker::Lorem.sentence` returns *lorem ipsum* text; as noted in [Chapter 6](#), *lorem ipsum* has a [fascinating back story](#).) The result is the new sample data populator shown in [Listing 11.20](#).

Listing 11.20. Adding microposts to the sample data.

```
lib/tasks/sample_data.rake   
require 'faker'  
  
namespace :db do  
  desc "Fill database with sample data"  
  task :populate => :environment do  
    .  
    .  
    .  
    User.all(:limit => 6).each do |user|  
      50.times do  
        user.microposts.create!(:content => Faker::Lorem.sentence(5))  
      end  
    end  
  end  
end  
end
```

Of course, to generate the new sample data we have to run the `db:populate` Rake task:

```
$ rake db:populate
```

With that, we are in a position to enjoy the fruits of our [Section 11.2.1](#) labors by displaying information for each micropost.¹⁰ [Figure 11.6](#) shows the user profile page for the first (signed-in) user, while [Figure 11.7](#) shows the profile for a second user. Finally, [Figure 11.8](#) 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 11.17](#). If you wait a couple minutes and reload the pages, you’ll see how the text gets automatically updated based on the new time.



Figure 11.6: The user profile (</users/1>) with microposts. ([full size](#))



Figure 11.7: The profile of a different user, also with microposts (</users/3>). ([full size](#))




Figure 11.8: A second of profile microposts, with pagination links (</users/1?page=2>). ([full size](#))

11.3 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. The result will be our third example of using an HTML form to create a resource—in this case, a Microposts resource.¹¹ In this section we'll also see the first hint of a *status feed*—a notion brought to full fruition in [Chapter 12](#). 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 Users and Pages controllers, rather than relying a controller of its own. This means that the routes for the Microposts resource are unusually simple, as seen in [Listing 11.21](#). The code in [Listing 11.21](#) leads in turn to the RESTful routes shown in [Table 11.2](#), 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 11.21. Routes for the Microposts resource.

config/routes.rb 

```

ActionController::Routing::Routes.draw do |map|
  map.resources :users
  map.resources :sessions, :only => [:new, :create, :destroy]
  map.resources :microposts, :only => [:create, :destroy]
  .
  .
  .
end
```


HTTP request	URL	Action	Purpose
POST	/microposts	create	create a new micropost
DELETE	/microposts/1	destroy	delete micropost with id 1

Table 11.2: RESTful routes provided by the Microposts resource in [Listing 11.21](#).

11.3.1 Access control

We begin our development of the Microposts resource with some access control in the Microposts controller. The idea is simple: both the `create` and `destroy` actions should require users to be signed in. The RSpec code to test for this appears in [Listing 11.22](#). (We'll test for and add a third protection—ensuring that only a micropost's user can destroy it—in [Section 11.3.4](#).)

Listing 11.22. Access control tests for the Microposts controller.

spec/controllers/microposts_controller_spec.rb 

```

require 'spec_helper'

describe MicropostsController do

  integrate_views
```

```

describe "access control" do


  it "should deny access to 'create'" do
    post :create
    response.should redirect_to(signin_path)
  end

  it "should deny access to 'destroy'" do
    delete :destroy
    response.should redirect_to(signin_path)
  end
end
end

```

Writing the application code needed to get the tests in [Listing 11.22](#) to pass requires a little refactoring first. Recall from [Section 10.2.1](#) that we enforced the signin requirement using a before filter that called the `authenticate` method ([Listing 10.9](#)). At the time, we only needed `authenticate` in the Users controller, but now we find that we need it in the Microposts controller as well, so we'll move `authenticate` into the Sessions helper, as shown in [Listing 11.23.12](#)

Listing 11.23. Moving the `authenticate` method into the Sessions helper.

`app/helpers/sessions_helper.rb` 

```

module SessionsHelper
  .
  .
  .
  def authenticate
    deny_access unless signed_in?
  end


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

```

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

With the code in [Listing 11.23](#), the `authenticate` method is now available in the Microposts controller, which means that we can restrict access to the `create` and `destroy` actions with the before filter shown in [Listing 11.24](#). (Since we didn't generate it at the command line, you will have to create the Microposts controller file by hand.)

Listing 11.24. Adding authentication to the Microposts controller actions.

`app/controllers/microposts_controller.rb` 

```
class MicropostsController < ApplicationController
  before_filter :authenticate

  def create
  end

  def destroy
  end
end
```

Note that we haven't restricted the actions the before filter applies to, since presently it applies to them both. If we were to add, say, an `index` action accessible even to non-signed-in users, we would need to specify the protected actions explicitly:

```
class MicropostsController < ApplicationController
  before_filter :authenticate, :only => [:create, :destroy]

  def create
  end

  def destroy
  end
end
```

11.3.2 Creating microposts

In [Chapter 8](#), 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 (following Twitter's convention) put the form on the Home page itself (i.e., the root path `/`), as mocked up in [Figure 11.9](#).



Figure 11.9: A mockup of the Home page with a form for creating microposts. ([full size](#))

When we last left the Home page, it appeared as in [Figure 5.7](#)—that is, it had a big, fat “Sign up now!” button in the middle. Since a micropost creation form only makes sense in the context of a particular signed-in user, one goal of this section will be to serve different versions of the Home page depending on a visitor's signin status. We'll implement this in [Listing 11.27](#) below, but for now the only implication is that the tests for the Microposts controller `create` action should sign a (factory) user in before attempting to make a post.

With that caveat in mind, the micropost creation tests parallel those for user creation from [Listing 8.6](#) and [Listing 8.13](#); the result appears in [Listing 11.25](#).

Listing 11.25. Tests for the Microposts controller `create` action.
`spec/controllers/microposts_controller_spec.rb`
`require 'spec_helper'`



```

describe MicropostsController do
  .
  .
  .
  describe "POST 'create'" do

    before(:each) do
      @user = test_sign_in(Factory(:user))
      @attr = { :content => "Lorem ipsum" }
      @micropost = Factory(:micropost, @attr.merge(:user => @user))

      @user.microposts.stub!(:build).and_return(@micropost)
    end

    describe "failure" do

      before(:each) do
        @micropost.should_receive(:save).and_return(false)
      end

      it "should render the home page" do
        post :create, :micropost => @attr
        response.should render_template('pages/home')
      end
    end

    describe "success" do

      before(:each) do
        @micropost.should_receive(:save).and_return(true)
      end

      it "should redirect to the home page" do
        post :create, :micropost => @attr
        response.should redirect_to(root_path)
      end

      it "should have a flash message" do
        post :create, :micropost => @attr
        flash[:success].should =~ /micropost created/i
      end
    end
  end
end

```

Here we've stubbed out `@user.microposts.build` instead of `User.new` as in, e.g., [Listing 8.6](#), but the basic idea is the same: the test intercepts calls to `build` and returns instead the factory micropost created by

```
@micropost = Factory(:micropost, @attr.merge(:user => @user))
```

The micropost creation failure and success tests then use the message expectations

```
@micropost.should_receive(:save)
```

to ensure that the code attempts to save the micropost, in each case either rendering the Home page (on

failure) or redirecting (on success).[13](#)

The `create` action for microposts is similar to its user analogue ([Listing 8.14](#)); the principal difference lies in using the user/micropost association to `build` the new micropost, as seen in [Listing 11.26](#).

Listing 11.26. The Microposts controller `create` action.

`app/controllers/microposts_controller.rb`



```
class MicropostsController < ApplicationController
  .
  .
  .
  def create
    @micropost = current_user.microposts.build(params[:micropost])
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_path
    else
      render 'pages/home'
    end
  end
end
.
.
.
end
```

At this point, the tests in [Listing 11.25](#) should all be passing, but of course we still don't have a form to create microposts. We can rectify this with [Listing 11.27](#), which serves up different HTML based on whether the site visitor is signed in or not.

Listing 11.27. Adding microposts creation to the Home page ([/](#)).

`app/views/pages/home.html.erb`



```
<% if signed_in? %>
  <table class="front">
    <tr>
      <td class="main">
        <h2 class="micropost">What's up?</h2>
        <%= render 'pages/micropost_form' %>
      </td>
      <td class="sidebar round">
        <%= render 'pages/user_info' %>
      </td>
    </tr>
  </table>
<% else %>
  <h1>Sample App</h1>

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

```
<%= link_to "Sign up now!", signup_path, :class => "signup_button round" %>
<% 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 11.5](#)). Filling in the necessary partials from [Listing 11.27](#) isn't an exercise, though; we fill in the micropost form partial in [Listing 11.28](#) and the new Home page sidebar in [Listing 11.29](#).

Listing 11.28. The form partial for creating microposts.
 app/views/pages/_micropost_form.html.erb



```
<% form_for @micropost do |f| %>
  <%= f.error_messages %>
  <div class="field">
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit "Submit" %>
  </div>
<% end %>
```

Listing 11.29. The partial for the user info sidebar.
 app/views/pages/_user_info.html.erb



```
<div class="user_info">
  <a href="<%= user_path(current_user) %>">
    <%= gravatar_for current_user, :size => 30 %>
    <span class="user_name">
      <%= h current_user.name %>
    </span>
    <span class="microposts">
      <%= pluralize(current_user.microposts.count, "micropost") %>
    </span>
  </a>
</div>
```

Note that, as in the profile sidebar ([Listing 11.16](#)), the user info in [Listing 11.29](#) 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 perfect sense. In the present case, though, saying “1 microposts” is ungrammatical, so we arrange to display “1 micropost” (but “2 microposts”) using the convenient `pluralize` helper method.

The form defined in [Listing 11.28](#) is an exact analogue of the signup form in [Listing 8.2](#), which means that it needs an `@micropost` instance variable. This is supplied in [Listing 11.30](#)—but only when the user is signed in.

Listing 11.30. Adding a micropost instance variable to the `home` action.
 app/controllers/pages_controller.rb



```
class PagesController < ApplicationController
```

```

def home
  @title = "Home"
  @micropost = Micropost.new if signed_in?
end
.
.
.
end

```

Now the HTML should render properly, showing the form as in [Figure 11.10](#), and a form with a submission error as in [Figure 11.11](#). You are invited at this point to create a new post for yourself and verify that everything is working—but you should probably wait until after [Section 11.3.3](#).



Figure 11.10: The Home page ([/](#)) with a new micropost form. ([full size](#))



Figure 11.11: The home page with form errors. ([full size](#))


11.3.3 A proto-feed

The comment at the end of [Section 11.3.2](#) alluded to a problem: the current Home page doesn't display any microposts. If you like, you can verify that the form shown in [Figure 11.10](#) 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 11.12](#). (In [Chapter 12](#), we'll generalize this feed to include the microposts of users being *followed* by the current user.)



Figure 11.12: A mockup of the Home page with a proto-feed. ([full size](#))

Since each user should have a feed, we are led naturally to a `feed` method in the `User` model. Eventually, we will test that the feed returns the microposts of the users being followed, but for now we'll just test that the `feed` method *includes* the current user's microposts but *excludes* the posts of a different user. We can express these requirements in code with [Listing 11.31](#).

Listing 11.31. Tests for the (proto-)status feed.
`spec/models/user_spec.rb` 

```

require 'spec_helper'

describe User do
  .
  .
  .

```

```

describe "micropost associations" do
  before(:each) do
    @user = User.create(@attr)
    @mp1 = Factory(:micropost, :user => @user, :created_at => 1.day.ago)
    @mp2 = Factory(:micropost, :user => @user, :created_at => 1.hour.ago)
  end
  .
  .
  .
  describe "status feed" do
    it "should have a feed" do
      @user.should respond_to(:feed)
    end

    it "should include the user's microposts" do
      @user.feed.include?(@mp1).should be_true
      @user.feed.include?(@mp2).should be_true
    end

    it "should not include a different user's microposts" do
      mp3 = Factory(:micropost,
                    :user => Factory(:user, :email => Factory.next(:email)))
      @user.feed.include?(mp3).should be_false
    end
  end
end
end
end

```

These tests introduce the array `include?` method, which simply checks if an array includes the given element:[14](#)

```

$ script/console
>> a = [1, "foo", :bar]
>> a.include?("foo")
=> true
>> a.include?(:bar)
=> true
>> a.include?("baz")
=> false

```

We can arrange for an appropriate micropost `feed` by selecting all the microposts with `user_id` equal to the current user's id, which we accomplish using a `:conditions` option to `Micropost.all`, as shown in [Listing 11.32.15](#)

Listing 11.32. A preliminary implementation for the micropost status feed.
 app/models/user.rb 

```

class User < ActiveRecord::Base
  .
  .
  .
  def feed
    # This is preliminary. See Chapter 12 for the full implementation.
    Micropost.all(:conditions => ["user_id = ?", id])
  end
end

```



```

end
.
.
.
end

```

The question mark in

```
:conditions => ["user_id = ?", id]
```

ensures that `id` is properly *escaped* before being included in the underlying SQL query, thereby avoiding a serious security breach called [SQL injection](#). (The `id` attribute here is just an integer, so there is no danger in this case, but *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 11.32](#) is essentially equivalent to writing

```


def feed
  microposts
end

```

We've used the code in [Listing 11.32](#) instead because it generalizes much more naturally to the full status feed needed in [Chapter 12](#).

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 11.33](#), and then add a feed partial ([Listing 11.34](#)) to the Home page ([Listing 11.36](#)).

Listing 11.33. Adding a feed instance variable to the `home` action.

`app/controllers/pages_controller.rb` 

```


class PagesController < ApplicationController

  def home
    @title = "Home"
    if signed_in?
      @micropost = Micropost.new
      @feed_items = current_user.feed.paginate(:page => params[:page])
    end
  end

  .
  .
  .
end

```

Listing 11.34. The status feed partial.

`app/views/pages/_feed.html.erb` 

```

<% unless @feed_items.empty? %>
  <table class="microposts">
    <%= render :partial => 'feed_item', :collection => @feed_items %>
  </table>
  <%= will_paginate @feed_items %>
<% end %>

```

The status feed partial defers the feed item rendering to a feed item partial using the code

```
<%= render :partial => 'feed_item', :collection => @feed_items %>
```

Here we pass a `:collection` parameter with the feed items, which causes `render` to use the given partial (`'feed_item'` in this case) to render each item in the collection. (We have omitted the `:partial` parameter in previous renderings, writing, e.g., `render 'pages/micropost'`, but with a `:collection` parameter that syntax doesn't work.) The feed item partial itself appears in [Listing 11.35](#); note the addition of a delete link to the feed item partial, following the example from [Listing 10.37](#).

Listing 11.35. A partial for a single feed item.

app/views/pages/_feed_item.html.erb



```
<tr>
  <td class="gravatar">
    <%= link_to gravatar_for(feed_item.user), feed_item.user %>
  </td>
  <td class="micropost">
    <span class="user">
      <%= link_to h(feed_item.user.name), feed_item.user %>
    </span>
    <span class="content"><%= h feed_item.content %></span>
    <span class="timestamp">
      Posted <%= time_ago_in_words(feed_item.created_at) %> ago.
    </span>
  </td>
  <%= if current_user?(feed_item.user) %>
  <td>
    <%= link_to "delete", feed_item, :method => :delete,
      :confirm => "You sure?" %>
  </td>
  <%= end %>
</tr>
```

We can then add the feed to the Home page by rendering the feed partial as usual ([Listing 11.36](#)). The result is a display of the feed on the Home page, as required ([Figure 11.13](#)).

Listing 11.36. Adding a status feed to the Home page.

app/views/pages/home.html.erb



```
<%= if signed_in? %>
<table class="front">
  <tr>
    <td class="main">
      <h2 class="micropost">What's up?</h2>
      <%= render 'pages/micropost_form' %>
      <%= render 'pages/feed' %>
    </td>
    .
    .
  </tr>
</table>
```

```

<% else %>
.
.
.
<% end %>

```



Figure 11.13: The Home page ([/](#)) with a proto-feed. ([full size](#))

At this point, creating a new micropost works as expected, as seen in [Figure 11.14](#). (We'll write an integration test to this effect in [Section 11.3.5](#).) There is one subtlety, though: on *failed* micropost submission, the Home page expects an `@feed_items` instance variable, so failed submissions currently break (as you should be able to verify by running your test suite). The easiest solution is to suppress the feed entirely by assigning it an empty array, as shown in [Listing 11.37.16](#)



Figure 11.14: The Home page after creating a new micropost. ([full size](#))

Listing 11.37. Adding an (empty) `@feed_items` instance variable to the `create` action.

`app/controllers/microposts_controller.rb`



```

class MicropostsController < ApplicationController
.
.
.
  def create
    @micropost = current_user.microposts.build(params[:micropost])
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_path
    else
      @feed_items = []
      render 'pages/home'
    end
  end
end
.
.
.
end

```

[11.3.4 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 11.15](#). Unlike that case, which restricted user destruction to admin users, the delete links will work only for microposts created by the current user.



Figure 11.15: A mockup of the proto-feed with micropost delete links. ([full size](#))

Our first step is to add a delete link to the micropost partial as in [Listing 11.35](#). The result appears in [Listing 11.38](#).

Listing 11.38. A partial for showing a single micropost.

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



```
<tr>
  <td class="micropost">
    <span class="content"><%= h micropost.content %></span>
    <span class="timestamp">
      Posted <%= time_ago_in_words(micropost.created_at) %> ago.
    </span>
  </td>
  <% if current_user?(micropost.user) %>
  <td>
    <%= link_to "delete", micropost, :method => :delete,
      :confirm => "You sure?" %>
  </td>
  <% end %>
</tr>
```

The tests for the `destroy` action are straightforward generalizations of the similar tests for destroying users ([Listing 10.38](#)), as seen in [Listing 11.39](#).

Listing 11.39. Tests for the Microposts controller `destroy` action.

`spec/controllers/microposts_controller_spec.rb`



```
describe MicropostsController do
  .
  .
  .
  describe "DELETE 'destroy'" do

    describe "for an unauthorized user" do

      before(:each) do
        @user = Factory(:user)
        wrong_user = Factory(:user, :email => Factory.next(:email))
        test_sign_in(wrong_user)
        @micropost = Factory(:micropost, :user => @user)
      end

      it "should deny access" do
        @micropost.should_not_receive(:destroy)
        delete :destroy, :id => @micropost
        response.should redirect_to(root_path)
      end
    end
  end
end
```