



[Home](#) | [Book](#) | [Help](#) | [Contact](#) | [News](#)  | [Follow](#) 

Please see the post [Do not buy the print version of the Ruby on Rails Tutorial \(yet\)](#)

[skip to content](#) | [view as single page](#)

---

# Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

## Contents

### **Chapter 1** From zero to deploy

#### 1.1 Introduction

##### 1.1.1 Comments for various readers

- 1.1.2 “Scaling” Rails
  - 1.1.3 Conventions in this book
- 1.2 Up and running
  - 1.2.1 Development environments
    - IDEs
    - Text editors and command lines
    - Browsers
    - A note about tools
  - 1.2.2 Ruby, RubyGems, Rails, and Git
    - Rails Installer (Windows)
    - Install Git
    - Install Ruby
    - Install RubyGems
    - Install Rails
  - 1.2.3 The first application
  - 1.2.4 Bundler
  - 1.2.5 rails server
  - 1.2.6 Model-view-controller (MVC)
- 1.3 Version control with Git
  - 1.3.1 Installation and setup
    - First-time system setup
    - First-time repository setup
  - 1.3.2 Adding and committing
  - 1.3.3 What good does Git do you?
  - 1.3.4 GitHub
  - 1.3.5 Branch, edit, commit, merge
    - Branch
    - Edit
    - Commit
    - Merge
    - Push
- 1.4 Deploying
  - 1.4.1 Heroku setup
  - 1.4.2 Heroku deployment, step one
  - 1.4.3 Heroku deployment, step two
  - 1.4.4 Heroku commands
- 1.5 Conclusion

## Chapter 2 A demo app

- 2.1 Planning the application
  - 2.1.1 Modeling demo users
  - 2.1.2 Modeling demo microposts
- 2.2 The Users resource
  - 2.2.1 A user tour
  - 2.2.2 MVC in action
  - 2.2.3 Weaknesses of this Users resource
- 2.3 The Microposts resource
  - 2.3.1 A micropost microtour
  - 2.3.2 Putting the *micro* in microposts
  - 2.3.3 A user has\_many microposts
  - 2.3.4 Inheritance hierarchies
  - 2.3.5 Deploying the demo app
- 2.4 Conclusion

## Chapter 3 Mostly static pages

- 3.1 Static pages
  - 3.1.1 Truly static pages
  - 3.1.2 Static pages with Rails
- 3.2 Our first tests
  - 3.2.1 Test-driven development
  - 3.2.2 Adding a page
    - Red
    - Green
    - Refactor
- 3.3 Slightly dynamic pages
  - 3.3.1 Testing a title change
  - 3.3.2 Passing title tests
  - 3.3.3 Embedded Ruby
  - 3.3.4 Eliminating duplication with layouts
- 3.4 Conclusion
- 3.5 Exercises
- 3.6 Advanced setup
  - 3.6.1 Eliminating bundle exec
    - RVM Bundler integration
    - binstubs

- 3.6.2 Automated tests with Guard
- 3.6.3 Speeding up tests with Spork  
Guard with Spork
- 3.6.4 Tests inside Sublime Text

## **Chapter 4 Rails-flavored Ruby**

- 4.1 Motivation
- 4.2 Strings and methods
  - 4.2.1 Comments
  - 4.2.2 Strings  
Printing  
Single-quoted strings
  - 4.2.3 Objects and message passing
  - 4.2.4 Method definitions
  - 4.2.5 Back to the title helper
- 4.3 Other data structures
  - 4.3.1 Arrays and ranges
  - 4.3.2 Blocks
  - 4.3.3 Hashes and symbols
  - 4.3.4 CSS revisited
- 4.4 Ruby classes
  - 4.4.1 Constructors
  - 4.4.2 Class inheritance
  - 4.4.3 Modifying built-in classes
  - 4.4.4 A controller class
  - 4.4.5 A user class
- 4.5 Conclusion
- 4.6 Exercises

## **Chapter 5 Filling in the layout**

- 5.1 Adding some structure
  - 5.1.1 Site navigation
  - 5.1.2 Bootstrap and custom CSS
  - 5.1.3 Partials
- 5.2 Sass and the asset pipeline
  - 5.2.1 The asset pipeline  
Asset directories

- Manifest files
  - Preprocessor engines
  - Efficiency in production
- 5.2.2 Syntactically awesome stylesheets
  - Nesting
  - Variables
- 5.3 Layout links
  - 5.3.1 Route tests
  - 5.3.2 Rails routes
  - 5.3.3 Named routes
  - 5.3.4 Pretty RSpec
- 5.4 User signup: A first step
  - 5.4.1 Users controller
  - 5.4.2 Signup URI
- 5.5 Conclusion
- 5.6 Exercises

## **Chapter 6 Modeling users**

- 6.1 User model
  - 6.1.1 Database migrations
  - 6.1.2 The model file
    - Model annotation
    - Accessible attributes
  - 6.1.3 Creating user objects
  - 6.1.4 Finding user objects
  - 6.1.5 Updating user objects
- 6.2 User validations
  - 6.2.1 Initial user tests
  - 6.2.2 Validating presence
  - 6.2.3 Length validation
  - 6.2.4 Format validation
  - 6.2.5 Uniqueness validation
    - The uniqueness caveat
- 6.3 Adding a secure password
  - 6.3.1 An encrypted password
  - 6.3.2 Password and confirmation
  - 6.3.3 User authentication

- 6.3.4 User has secure password
  - 6.3.5 Creating a user
- 6.4 Conclusion
- 6.5 Exercises

## **Chapter 7 Sign up**

- 7.1 Showing users
  - 7.1.1 Debug and Rails environments
  - 7.1.2 A Users resource
  - 7.1.3 Testing the user show page (with factories)
  - 7.1.4 A Gravatar image and a sidebar
- 7.2 Signup form
  - 7.2.1 Tests for user signup
  - 7.2.2 Using `form_for`
  - 7.2.3 The form HTML
- 7.3 Signup failure
  - 7.3.1 A working form
  - 7.3.2 Signup error messages
- 7.4 Signup success
  - 7.4.1 The finished signup form
  - 7.4.2 The flash
  - 7.4.3 The first signup
  - 7.4.4 Deploying to production with SSL
- 7.5 Conclusion
- 7.6 Exercises

## **Chapter 8 Sign in, sign out**

- 8.1 Sessions and signin failure
  - 8.1.1 Sessions controller
  - 8.1.2 Signin tests
  - 8.1.3 Signin form
  - 8.1.4 Reviewing form submission
  - 8.1.5 Rendering with a flash message
- 8.2 Signin success
  - 8.2.1 Remember me
  - 8.2.2 A working `sign_in` method
  - 8.2.3 Current user

- 8.2.4 Changing the layout links
  - 8.2.5 Signin upon signup
  - 8.2.6 Signing out
- 8.3 Introduction to Cucumber (optional)
  - 8.3.1 Installation and setup
  - 8.3.2 Features and steps
  - 8.3.3 Counterpoint: RSpec custom matchers
- 8.4 Conclusion
- 8.5 Exercises

## **Chapter 9 Updating, showing, and deleting users**

- 9.1 Updating users
  - 9.1.1 Edit form
  - 9.1.2 Unsuccessful edits
  - 9.1.3 Successful edits
- 9.2 Authorization
  - 9.2.1 Requiring signed-in users
  - 9.2.2 Requiring the right user
  - 9.2.3 Friendly forwarding
- 9.3 Showing all users
  - 9.3.1 User index
  - 9.3.2 Sample users
  - 9.3.3 Pagination
  - 9.3.4 Partial refactoring
- 9.4 Deleting users
  - 9.4.1 Administrative users
    - Revisiting `attr_accessible`
  - 9.4.2 The destroy action
- 9.5 Conclusion
- 9.6 Exercises

## **Chapter 10 User microposts**

- 10.1 A Micropost model
  - 10.1.1 The basic model
  - 10.1.2 Accessible attributes and the first validation
  - 10.1.3 User/Micropost associations
  - 10.1.4 Micropost refinements

- Default scope
    - Dependent: destroy
  - 10.1.5 Content validations
- 10.2 Showing microposts
  - 10.2.1 Augmenting the user show page
  - 10.2.2 Sample microposts
- 10.3 Manipulating microposts
  - 10.3.1 Access control
  - 10.3.2 Creating microposts
  - 10.3.3 A proto-feed
  - 10.3.4 Destroying microposts
- 10.4 Conclusion
- 10.5 Exercises

## **Chapter 11 Following users**

- 11.1 The Relationship model
  - 11.1.1 A problem with the data model (and a solution)
  - 11.1.2 User/relationship associations
  - 11.1.3 Validations
  - 11.1.4 Followed users
  - 11.1.5 Followers
- 11.2 A web interface for following users
  - 11.2.1 Sample following data
  - 11.2.2 Stats and a follow form
  - 11.2.3 Following and followers pages
  - 11.2.4 A working follow button the standard way
  - 11.2.5 A working follow button with Ajax
- 11.3 The status feed
  - 11.3.1 Motivation and strategy
  - 11.3.2 A first feed implementation
  - 11.3.3 Subselects
  - 11.3.4 The new status feed
- 11.4 Conclusion
  - 11.4.1 Extensions to the sample application
    - Replies
    - Messaging
    - Follower notifications



## Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I've worked my way through many Rails books, this is the one that finally made me “get” it. Everything is done very much “the Rails way”—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it's like to do a real-world project. The tutorial's code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

[Derek Sivers \(sivers.org\)](http://sivers.org)

Formerly: Founder, [CD Baby](#)

Currently: Founder, [Thoughts Ltd.](#)

## Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor [Aurelius Prochazka](#). I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

## About the author

[Michael Hartl](#) is the author of the [Ruby on Rails Tutorial](#), the leading introduction to web development with [Ruby on Rails](#). His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a [Ruby Hero Award](#) for his contributions to the Ruby community. He is a graduate of [Harvard College](#), has a [Ph.D. in Physics](#) from [Caltech](#), and is an alumnus of the [Y Combinator](#) entrepreneur program.

## Copyright and license

*Ruby on Rails Tutorial: Learn Web Development with Rails*. Copyright © 2012 by Michael Hartl.

All source code in the *Ruby on Rails Tutorial* is available jointly under the [MIT License](#) and the [Beerware License](#).

The MIT License

Copyright (c) 2012 Michael Hartl

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
/*
 * -----
 * "THE BEER-WARE LICENSE" (Revision 42):
 * Michael Hartl wrote this code. As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.
 * -----
 */
```

# Chapter 10

## User microposts

[Chapter 9](#) saw the completion of the REST actions for the Users resource, so the time has finally come to add a second full resource: user *microposts*.<sup>1</sup> These are short messages associated with a particular user, first seen in larval form in [Chapter 2](#). 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. In [Chapter 11](#), we'll complete our tiny Twitter clone by adding the notion of *following* users in order to receive a *feed* of their microposts.

If you're using Git for version control, I suggest making a topic branch as usual:

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

### 10.1 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.

#### 10.1.1 The basic model



PDF/Screencasts



Rails 3.2 • Rails 3.0

f Like 2k

The Micropost model needs only two attributes: a **content** attribute to hold the micropost's content,<sup>2</sup> 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 model**:

```
$ rails generate model Micropost content:string user_id:integer
```

This produces a migration to create a **microposts** table in the database ([Listing 10.1](#)); compare it to the analogous migration for the **users** table from [Listing 6.2](#).

**Listing 10.1.** The Micropost migration. (Note the index on **user\_id** and **created\_at**.)

**db/migrate/[timestamp]\_create\_microposts.rb**

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

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

Note that, since we expect to retrieve all the microposts associated with a given user id in reverse order of creation, [Listing 10.1](#) 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. 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 10.1.4](#) and [Section 10.2.1](#).

We'll start with some minimal tests for the Micropost model based on the analogous tests for the User model ([Listing 6.8](#)). In particular, we verify that a micropost object responds to the `content` and `user_id` attributes, as shown in [Listing 10.2](#).

**Listing 10.2.** The initial Micropost spec.

`spec/models/micropost_spec.rb`

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before do
    # This code is wrong!
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
  end

  subject { @micropost }

  it { should respond_to(:content) }
  it { should respond_to(:user_id) }
end
```

We can get these tests to pass by running the microposts migration and preparing the test database:

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

The result is a Micropost model with the structure shown in [Figure 10.1](#).

microposts	
id	integer
content	string
user_id	integer
created_at	datetime
updated_at	datetime

Figure 10.1: The Micropost data model.

You should verify that the tests pass:

```
$ bundle exec rspec spec/models/micropost_spec.rb
```

Even though the tests are passing, you might have noticed this code:

```
let(:user) { FactoryGirl.create(:user) }  
before do  
  # This code is wrong!  
  @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)  
end
```

The comment indicates that the code in the **before** block is wrong. See if you can guess why. We'll see the answer in the next section.

## 10.1.2 Accessible attributes and the first validation

To see why the code in the **before** block is wrong, we first start with validation tests for the Micropost model ([Listing 10.3](#)). (Compare with the User model tests in [Listing 6.11](#).)

**Listing 10.3.** Tests for the validity of a new micropost.

**spec/models/micropost\_spec.rb**

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before do
    # This code is wrong!
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
  end

  subject { @micropost }

  it { should respond_to(:content) }
  it { should respond_to(:user_id) }

  it { should be_valid }

  describe "when user_id is not present" do
    before { @micropost.user_id = nil }
    it { should_not be_valid }
  end
end
```

This code requires that the micropost be valid and tests for the presence of the **user\_id** attribute. We can get these tests to pass with the simple presence validation shown in [Listing 10.4](#).

**Listing 10.4.** A validation for the micropost's **user\_id**.

**app/models/micropost.rb**

```
class Micropost < ActiveRecord::Base
```



```
attr_accessible :content, :user_id  
validates :user_id, presence: true  
end
```

Now we're prepared to see why

```
@micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)
```

is wrong. The problem is that by default (as of Rails 3.2.3) *all* of the attributes for our Micropost model are accessible. As discussed in [Section 6.1.2.2](#) and [Section 9.4.1.1](#), this 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. This means that we should remove `:user_id` from the `attr_accessible` list, and once we do, the code above will fail. We'll fix this issue in [Section 10.1.3](#).

### 10.1.3 User/Micropost associations

When constructing data models for web applications, it is essential to be able to make *associations* between individual models. In the present case, each micropost is associated with one user, and each user is associated with (potentially) many microposts—a relationship seen briefly in [Section 2.3.3](#) and shown schematically in [Figure 10.2](#) and [Figure 10.3](#). As part of implementing these associations, we'll write tests for the Micropost model that, unlike [Listing 10.2](#), are compatible with the use of `attr_accessible` in [Listing 10.7](#).

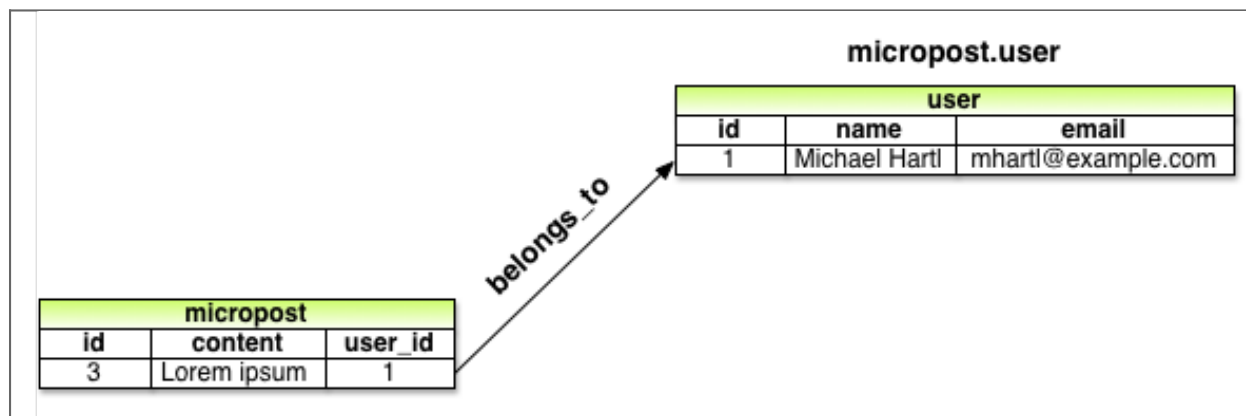


Figure 10.2: The **belongs\_to** relationship between a micropost and its user.

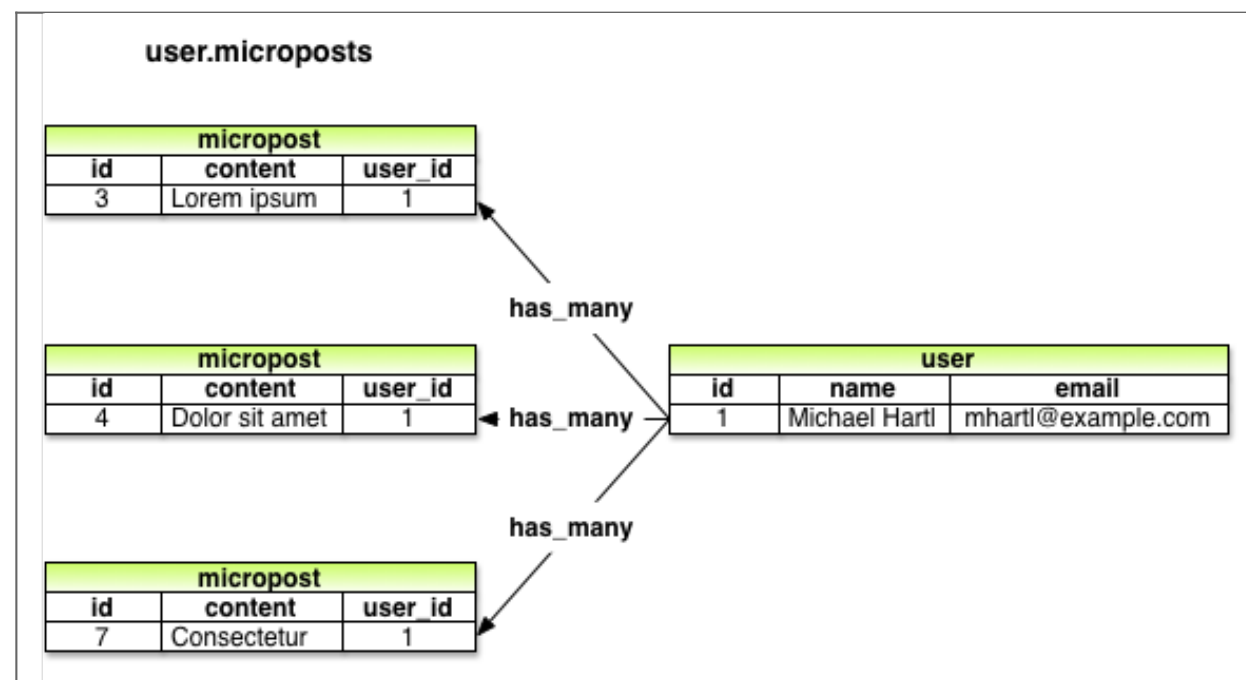


Figure 10.3: The **has\_many** relationship between a user and its microposts.

Using the `belongs_to/has_many` association defined in this section, Rails constructs the methods shown in [Table 10.1](#).

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 10.1: A summary of user/micropost association methods.

Note from [Table 10.1](#) that instead of

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

we have

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

This pattern is the canonical way to make a micropost: *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, which fixes the issue noted in [Section 10.1.2](#). In particular, we can replace the code

```
let(:user) { FactoryGirl.create(:user) }  
before do  
  # This code is wrong!  
  @micropost = Micropost.new(content: "Lorem ipsum", user_id: user.id)  
end
```

from [Listing 10.3](#) with

```
let(:user) { FactoryGirl.create(:user) }  
before { @micropost = user.microposts.build(content: "Lorem ipsum") }
```

Once we define the proper associations, the resulting `@micropost` variable will automatically have `user_id` equal to its associated user.

Building the micropost through the User association doesn't fix the security problem of having an accessible `user_id`, and because this is such an important security concern we'll add a failing test to catch it, as shown in [Listing 10.5](#).

**Listing 10.5.** A test to ensure that the `user_id` isn't accessible.

`spec/models/micropost_spec.rb`

```
require 'spec_helper'  
  
describe Micropost do  
  
  let(:user) { FactoryGirl.create(:user) }
```

```

before { @micropost = user.microposts.build(content: "Lorem ipsum") }

subject { @micropost }

.
.
.
describe "accessible attributes" do
  it "should not allow access to user_id" do
    expect do
      Micropost.new(user_id: user.id)
    end.to raise_error(ActiveModel::MassAssignmentSecurity::Error)
  end
end
end
end

```

This test verifies that calling `Micropost.new` with a nonempty `user_id` raises a mass assignment security error exception. This behavior is on by default as of Rails 3.2.3, but previous versions had it off, so you should make sure that your application is configured properly, as shown in [Listing 10.6](#).

**Listing 10.6.** Ensuring that Rails throws errors on invalid mass assignment.

`config/application.rb`

```

.
.
.
module SampleApp
  class Application < Rails::Application
    .
    .
    .
    config.active_record.whitelist_attributes = true
    .
    .
    .
  end
end
end

```

In the case of the Micropost model, there is only *one* attribute that needs to be editable through the web, namely, the `content` attribute, so we need to remove `:user_id` from the accessible list, as shown in [Listing 10.7](#).

**Listing 10.7.** Making the `content` attribute (and *only* the `content` attribute) accessible.

`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  attr_accessible :content

  validates :user_id, presence: true
end
```

As seen in [Table 10.1](#), another result of the user/micropost association is `micropost.user`, which simply returns the micropost's user. We can test this with the `it` and `its` methods as follows:

```
it { should respond_to(:user) }
its(:user) { should == user }
```

The resulting Micropost model tests are shown in [Listing 10.8](#).

**Listing 10.8.** Tests for the micropost's user association.

`spec/models/micropost_spec.rb`

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before { @micropost = user.microposts.build(content: "Lorem ipsum") }

  subject { @micropost }
```

```

it { should respond_to(:content) }
it { should respond_to(:user_id) }
it { should respond_to(:user) }
its(:user) { should == user }

it { should be_valid }

describe "accessible attributes" do
  it "should not allow access to user_id" do
    expect do
      Micropost.new(user_id: user.id)
    end.to raise_error(ActiveModel::MassAssignmentSecurity::Error)
  end
end

describe "when user_id is not present" do
  before { @micropost.user_id = nil }
  it { should_not be_valid }
end
end

```

On the User model side of the association, we'll defer the more detailed tests to [Section 10.1.4](#); for now, we'll simply test for the presence of a **microposts** attribute ([Listing 10.9](#)).

**Listing 10.9.** A test for the user's **microposts** attribute.

**spec/models/user\_spec.rb**

```

require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end

  subject { @user }
  .
  .

```

```
.
it { should respond_to(:authenticate) }
it { should respond_to(:microposts) }
.
.
.
end
```

After all that work, the code to implement the association is almost comically short: we can get the tests in both [Listing 10.8](#) and [Listing 10.9](#) to pass by adding just two lines: `belongs_to :user` ([Listing 10.10](#)) and `has_many :microposts` ([Listing 10.11](#)).

**Listing 10.10.** A micropost `belongs_to` a user.

`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  attr_accessible :content
  belongs_to :user

  validates :user_id, presence: true
end
```

**Listing 10.11.** A user `has_many` microposts.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  has_secure_password
  has_many :microposts
  .
  .
  .
end
```



At this point, you should compare the entries in [Table 10.1](#) with the code in [Listing 10.8](#) and [Listing 10.9](#) to satisfy yourself that you understand the basic nature of the associations. You should also check that the tests pass:

```
$ bundle exec rspec spec/models
```

## 10.1.4 Micropost refinements

The test in [Listing 10.9](#) 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 test, 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, as seen in [Listing 10.12](#).

**Listing 10.12.** The complete factory file, including a new factory for microposts.

`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    sequence(:name) { |n| "Person #{n}" }
    sequence(:email) { |n| "person_#{n}@example.com" }
    password "foobar"
    password_confirmation "foobar"

    factory :admin do
      admin true
    end
  end

  factory :micropost do
```

```
    content "Lorem ipsum"  
    user  
  end  
end
```

Here we tell Factory Girl about the micropost's associated user just by including a user in the definition of the factory:

```
factory :micropost do  
  content "Lorem ipsum"  
  user  
end
```

As we'll see in the next section, this allows us to define factory microposts as follows:

```
FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
```

## Default scope

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:

```
FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)  
FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
```

Here we indicate (using the time helpers discussed in [Box 8.1](#)) that the second post was created

more recently, i.e., `1.hour.ago`, while the first post was 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 ActiveRecord won't allow us to do. (Recall that `created_at` and `updated_at` are “magic” columns, automatically set to the proper creation and update timestamps, so any explicit initialization values are overwritten by the magic.)

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 10.13](#). This uses the `let!` (read “let bang”) method in place of `let`; the reason is that `let` variables are *lazy*, meaning that they only spring into existence when referenced. The problem is that we want the microposts to exist immediately, so that the timestamps are in the right order and so that `@user.microposts` isn't empty. We accomplish this with `let!`, which forces the corresponding variable to come into existence immediately.

**Listing 10.13.** Testing the order of a user's microposts.

`spec/models/user_spec.rb`

```
require 'spec_helper'

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

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end

    it "should have the right microposts in the right order" do
```

```
@user.microposts.should == [newer_micropost, older_micropost]
end
end
end
```

The key line here is

```
@user.microposts.should == [newer_micropost, older_micropost]
```

indicating that the posts should be ordered newest first. This should fail because by default the posts will be ordered by id, i.e., `[older_micropost, newer_micropost]`. This test also verifies the basic correctness of the `has_many` association itself, by checking (as indicated in [Table 10.1](#)) 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 10.14](#). (This is our first example of the notion of *scope*. We will learn about scope in a more general context in [Chapter 11](#).)

**Listing 10.14.** Ordering the microposts with `default_scope`.

`app/models/micropost.rb`

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

The order here is `'microposts.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 9.4](#) 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 10.15](#)).

**Listing 10.15.** Testing that microposts are destroyed when users are.

`spec/models/user_spec.rb`

```
require 'spec_helper'

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

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end

    .
    .
    .
    it "should destroy associated microposts" do
      microposts = @user.microposts
      @user.destroy
      microposts.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 10.15](#) to pass is less than one line; in fact, it's just an option to the `has_many` association method, as shown in [Listing 10.16](#).

**Listing 10.16.** Ensuring that a user's microposts are destroyed along with the user.

`app/models/user.rb`

```
class User < ActiveRecord::Base  
  attr_accessible :name, :email, :password, :password_confirmation  
  has_secure_password  
  has_many :microposts, dependent: :destroy  
  .  
  .  
  .  
end
```

Here the option `dependent: :destroy` in

```
has_many :microposts, dependent: :destroy
```

arranges for the dependent microposts (i.e., the ones belonging to the given user) 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.

With that, the final form of the user/micropost association is in place, and all the tests should be passing:

```
$ bundle exec rspec spec/
```

## 10.1.5 Content validations

Before leaving the Micropost model, we'll add validations for the micropost **content** (following the example from [Section 2.3.2](#)). Like the **user\_id**, the **content** attribute must be present, and it is further constrained to be no longer than 140 characters, making it an honest *micropost*. The tests generally follow the examples from the User model validation tests in [Section 6.2](#), as shown in [Listing 10.17](#).

**Listing 10.17.** Tests for the Micropost model validations.

**spec/models/micropost\_spec.rb**

```
require 'spec_helper'

describe Micropost do

  let(:user) { FactoryGirl.create(:user) }
  before { @micropost = user.microposts.build(content: "Lorem ipsum") }

  .
  .
  .
```

```

describe "when user_id is not present" do
  before { @micropost.user_id = nil }
  it { should_not be_valid }
end

describe "with blank content" do
  before { @micropost.content = " " }
  it { should_not be_valid }
end

describe "with content that is too long" do
  before { @micropost.content = "a" * 141 }
  it { should_not be_valid }
end
end

```

As in [Section 6.2](#), the code in [Listing 10.17](#) uses string multiplication to test the micropost length validation:

```

$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

```

The application code is a one-liner:

```

validates :content, presence: true, length: { maximum: 140 }

```

The resulting Micropost model is shown in [Listing 10.18](#).

**Listing 10.18.** The Micropost model validations.



app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  attr_accessible :content

  belongs_to :user

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

  default_scope order: 'microposts.created_at DESC'
end
```

## 10.2 Showing microposts

Although we don't yet have a way to create microposts through the web—that comes in [Section 10.3.2](#)—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 10.4](#). 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 9.3.2](#) so that we have something to display.



David Jones

## Microposts (3)

---

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

As with the discussion of the signin machinery in [Section 8.2.1](#), [Section 10.2.1](#) 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 10.2.2](#).

## 10.2.1 Augmenting the user show page

We begin with tests for displaying the user's microposts, which we'll create in the request spec for Users. Our strategy is to create a couple of factory microposts associated with the user, and then verify that the show page contains each post's content. We'll also verify that, as in [Figure 10.4](#), the total number of microposts also gets displayed.

We can create the posts with the `let` method, but as in [Listing 10.13](#) we want the association to exist immediately so that the posts appear on the user show page. To accomplish this, we use the `let!` variant:

```
let(:user) { FactoryGirl.create(:user) }
let!(:m1) { FactoryGirl.create(:micropost, user: user, content: "Foo") }
let!(:m2) { FactoryGirl.create(:micropost, user: user, content: "Bar") }

before { visit user_path(user) }
```

With the microposts so defined, we can test for their appearance on the profile page using the code in [Listing 10.19](#).

**Listing 10.19.** A test for showing microposts on the user `show` page.

`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
```

```

describe "profile page" do
  let(:user) { FactoryGirl.create(:user) }
  let!(:m1) { FactoryGirl.create(:micropost, user: user, content: "Foo") }
  let!(:m2) { FactoryGirl.create(:micropost, user: user, content: "Bar") }

  before { visit user_path(user) }

  it { should have_selector('h1', text: user.name) }
  it { should have_selector('title', text: user.name) }

  describe "microposts" do
    it { should have_content(m1.content) }
    it { should have_content(m2.content) }
    it { should have_content(user.microposts.count) }
  end
end
.
.
.
end

```

Note here that we can use the `count` method *through* the association:

```
user.microposts.count
```

The association `count` method is smart, and performs the count directly in the database. In particular, it does *not* pull all the microposts out of the database and then call `length` on the resulting array, as this could become inefficient as the number of microposts grew. Instead, it asks the database to count the microposts with the given `user_id`. In the unlikely event that finding the count is still a bottleneck in your application, you can make it even faster with a [counter cache](#).

Although the tests in [Listing 10.19](#) won't pass until [Listing 10.21](#), we'll get started on the application code by inserting a list of microposts into the user profile page, as shown in [Listing 10.20](#).

**Listing 10.20.** Adding microposts to the user `show` page.

`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
  .
  .
  .
  <aside>
    .
    .
    .
  </aside>
  <div class="span8">
    <% if @user.microposts.any? %>
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
  </div>
</div>
```

We'll deal with the microposts list momentarily, but there are several other things to note first. In [Listing 10.20](#), the use of `if @user.microposts.any?` (a construction we saw before in [Listing 7.23](#)) makes sure that an empty list won't be displayed when the user has no microposts.

Also note from [Listing 10.20](#) 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 9.34](#), 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 9.3.3](#), 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 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 10.22](#)).

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

```
<h3>Microposts (<%= @user.microposts.count %>)</h3>
```

As noted, `@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.

We come finally to the micropost list itself:

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

This code, which uses the *ordered list* tag `ol`, is responsible for generating the list of microposts, but you can see that it just defers the heavy lifting to a micropost partial. We saw in [Section 9.3.4](#) 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 (along with a `micropost` views directory), as shown in [Listing 10.21](#).

**Listing 10.21.** A partial for showing a single micropost.

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

```
<li>
  <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 effect we will see in [Section 10.2.2](#).

Thus far, despite defining all the relevant ERb templates, the test in [Listing 10.19](#) should have been failing for want of an `@microposts` variable. We can get it to pass with [Listing 10.22](#).

**Listing 10.22.** 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.

At this point, we can get a look at our new user profile page in [Figure 10.5](#). It's rather... disappointing. Of course, this is because there are not currently any microposts. It's time to change that.



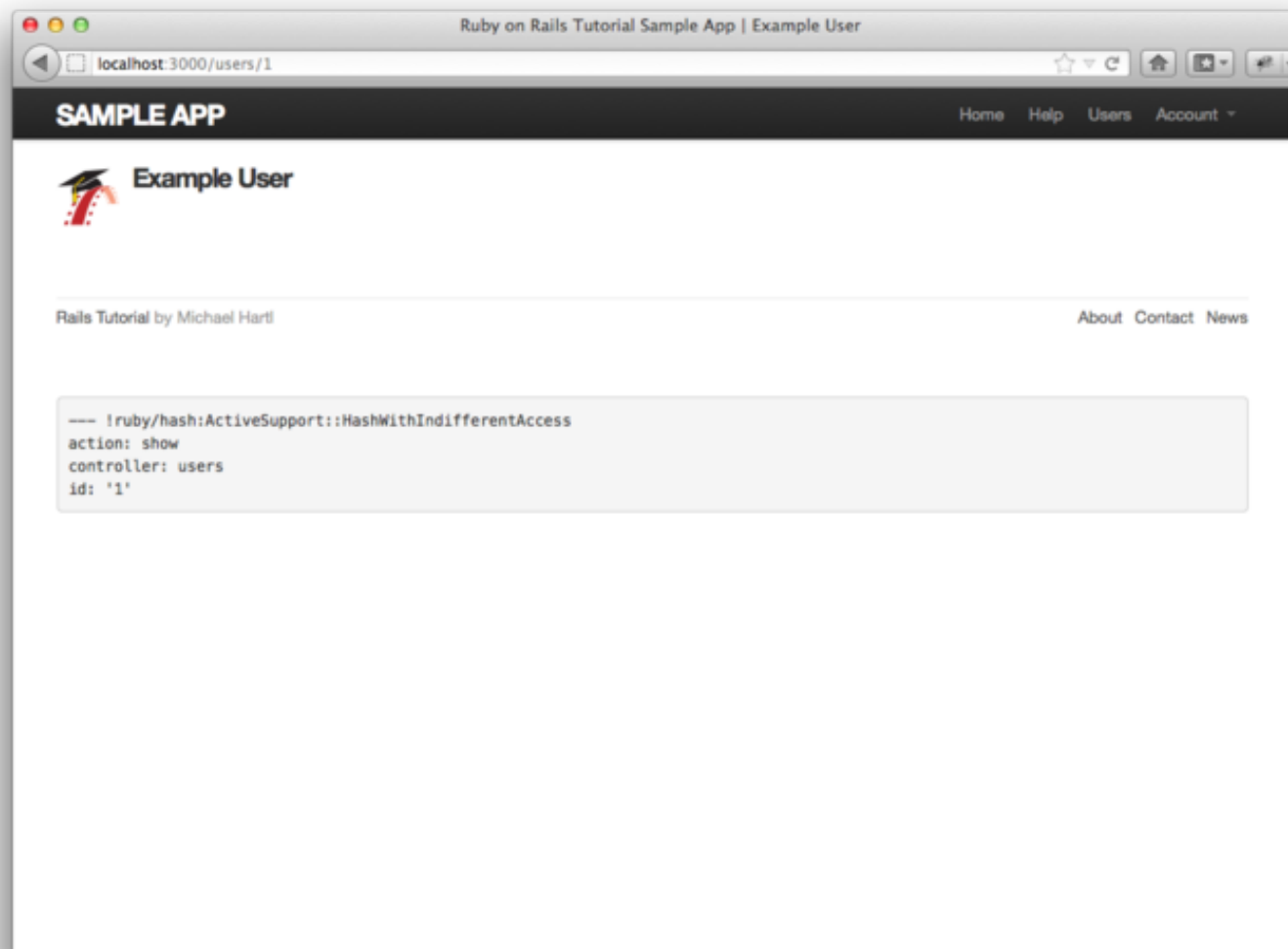


Figure 10.5: The user profile page with code for microposts—but no microposts. ([full size](#))

## 10.2.2 Sample microposts

With all the work making templates for user microposts in [Section 10.2.1](#), the ending was rather anticlimactic. We can rectify this sad situation by adding microposts to the sample populator from

[Section 9.3.2](#). Adding sample microposts for *all* the users actually takes a rather long time, so first we'll select just the first six users<sup>3</sup> using the `:limit` option to the `User.all` method:<sup>4</sup>

```
users = 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 10.23](#).

**Listing 10.23.** Adding microposts to the sample data.

`lib/tasks/sample_data.rake`

```
namespace :db do
  desc "Fill database with sample data"
  task :populate => :environment do
    .
    .
    .
    users = User.all(limit: 6)
    50.times do
      content = Faker::Lorem.sentence(5)
      users.each { |user| user.microposts.create!(content: content) }
    end
  end
end
```

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

```
$ bundle exec rake db:reset
$ bundle exec rake db:populate
```

```
$ bundle exec rake db:test:prepare
```

With that, we are in a position to enjoy the fruits of our [Section 10.2.1](#) labors by displaying information for each micropost.<sup>5</sup> The preliminary results appear in [Figure 10.6](#).

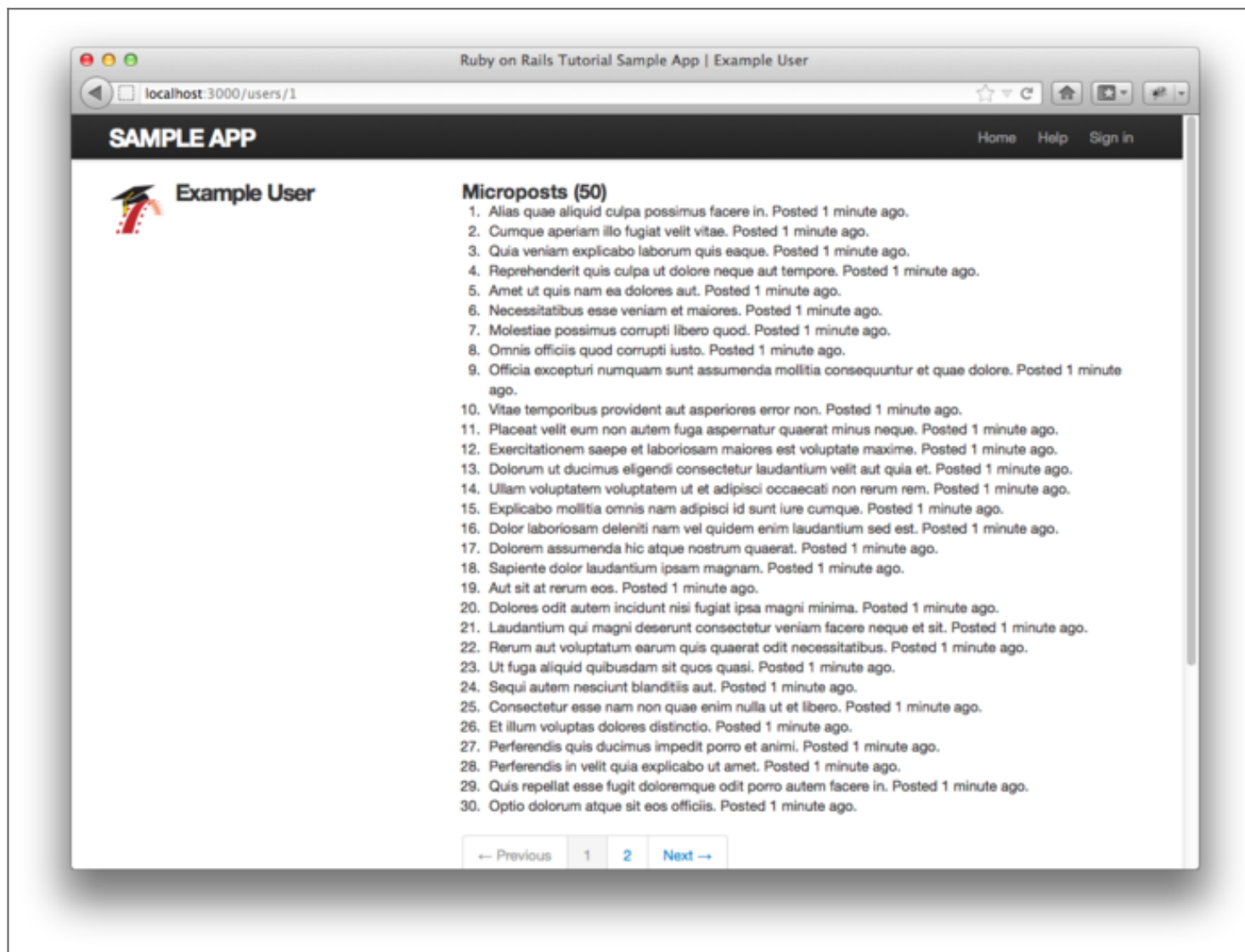


Figure 10.6: The user profile (</users/1>) with unstyled microposts. ([full size](#))

The page shown in [Figure 10.6](#) has no micropost-specific styling, so let's add some ([Listing 10.24](#)) and take a look the resulting pages.<sup>6</sup> [Figure 10.7](#), which displays the user profile page for the first (signed-in) user, while [Figure 10.8](#) shows the profile for a second user. Finally, [Figure 10.9](#) 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 10.21](#). If you wait a couple minutes and reload the pages, you'll see how the text gets automatically updated based on the new time.

**Listing 10.24.** The CSS for microposts (including all the CSS for this chapter).

`app/assets/stylesheets/custom.css.scss`

```
.  
.br/>.br/>  
/* microposts */  
  
.microposts {  
  list-style: none;  
  margin: 10px 0 0 0;  
  
  li {  
    padding: 10px 0;  
    border-top: 1px solid #e8e8e8;  
  }  
}  
  
.content {  
  display: block;  
}  
  
.timestamp {  
  color: $grayLight;  
}  
  
.gravatar {  
  float: left;  
  margin-right: 10px;
```

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

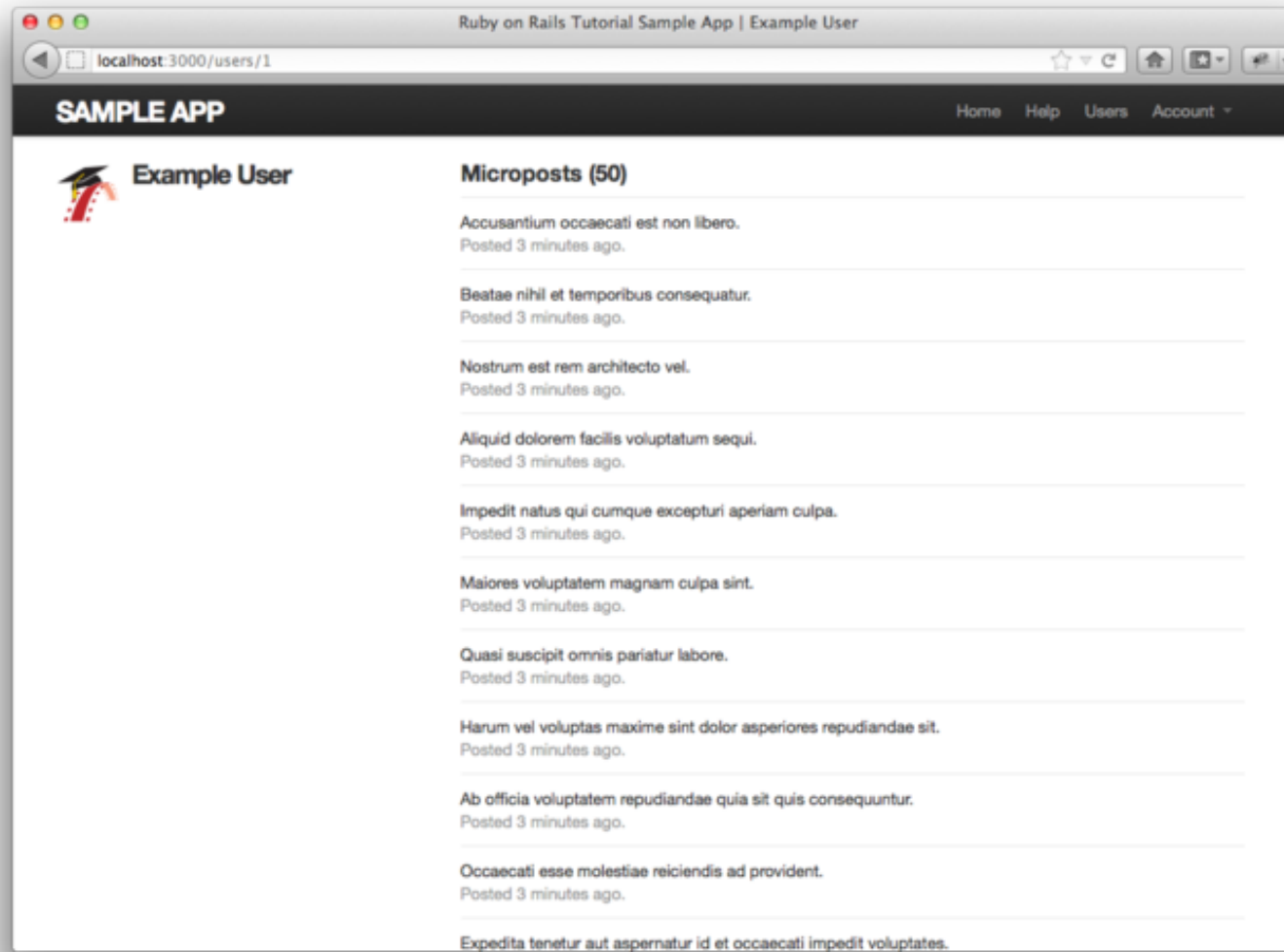


Figure 10.7: The user profile (</users/1>) with microposts. ([full size](#))

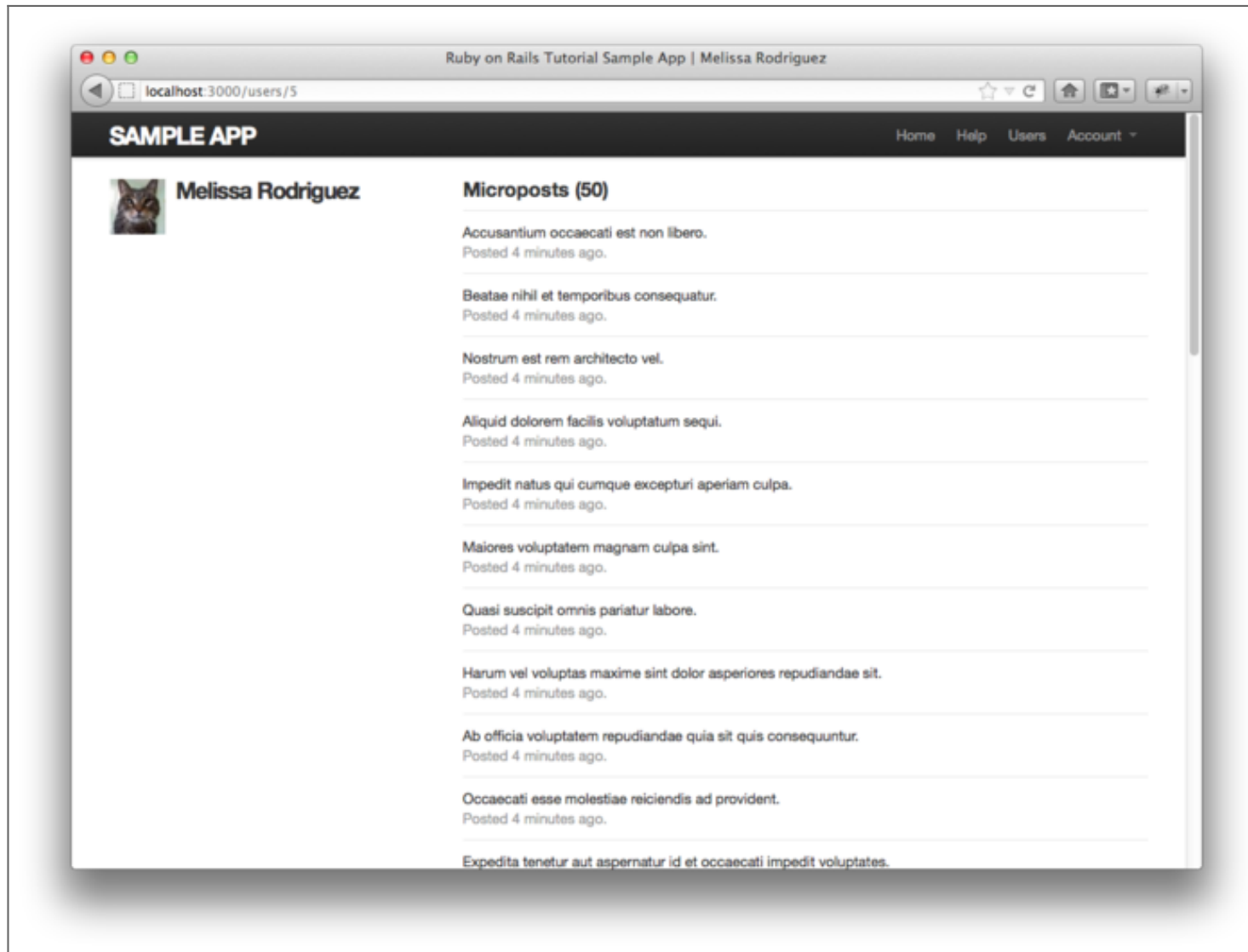


Figure 10.8: The profile of a different user, also with microposts (</users/5>). ([full size](#))

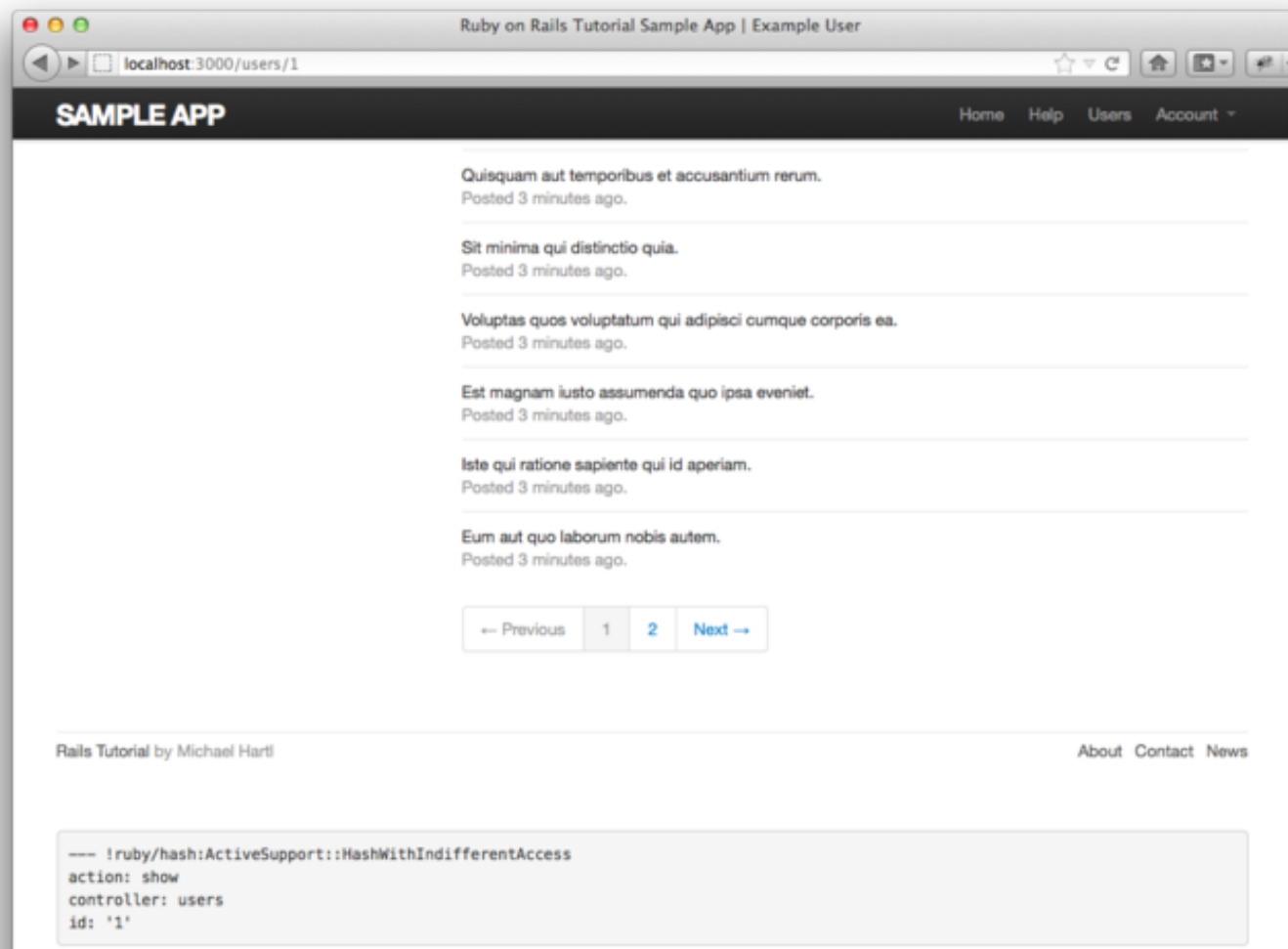


Figure 10.9: Micropost pagination links (</users/1?page=2>). [\(full size\)](#)

## 10.3 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.<sup>7</sup> In this section, we’ll also see the first hint of a *status feed*—a notion brought to full fruition in [Chapter 11](#). 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 StaticPages controllers, rather than relying on a controller of its own. This means that the routes for the Microposts resource are unusually simple, as seen in [Listing 10.25](#). The code in [Listing 10.25](#) leads in turn to the RESTful routes shown in [Table 10.2](#), 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 10.25.** Routes for the Microposts resource.

`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users
  resources :sessions, only: [:new, :create, :destroy]
  resources :microposts, only: [:create, :destroy]
  .
  .
  .
end
```

HTTP request	URI	Action	Purpose
POST	/microposts	<code>create</code>	create a new micropost
DELETE	/microposts/1	<code>destroy</code>	delete micropost with id <code>1</code>

Table 10.2: RESTful routes provided by the Microposts resource in [Listing 10.25](#).



## 10.3.1 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 10.26](#). (We'll test for and add a third protection—ensuring that only a micropost's user can destroy it—in [Section 10.3.4](#).)

**Listing 10.26.** Access control tests for microposts.

`spec/requests/authentication_pages_spec.rb`

```
require 'spec_helper'

describe "Authentication" do
  .
  .
  .
  describe "authorization" do

    describe "for non-signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      .
      .
      .
      describe "in the Microposts controller" do

        describe "submitting to the create action" do
          before { post microposts_path }
          specify { response.should redirect_to(signin_path) }
        end

        describe "submitting to the destroy action" do
          before { delete micropost_path(FactoryGirl.create(:micropost)) }
          specify { response.should redirect_to(signin_path) }
        end
      end
    end
  end
end
```

```
end
end
end
```

Rather than using the (yet-to-be-built) web interface for microposts, the code in [Listing 10.26](#) operates at the level of the individual micropost actions, a strategy we first saw in [Listing 9.14](#). In this case, a non-signed-in user is redirected upon submitting a POST request to /microposts (**post microposts\_path**, which hits the **create** action) or submitting a DELETE request to /microposts/1 (**delete micropost\_path(micropost)**, which hits the **destroy** action).

Writing the application code needed to get the tests in [Listing 10.26](#) to pass requires a little refactoring first. Recall from [Section 9.2.1](#) that we enforced the signin requirement using a before filter that called the **signed\_in\_user** method ([Listing 9.12](#)). At the time, we only needed that method 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 Sessions helper, as shown in [Listing 10.27](#).<sup>8</sup>

**Listing 10.27.** Moving the **signed\_in\_user** method into the Sessions helper.

**app/helpers/sessions\_helper.rb**

```
module SessionsHelper
  .
  .
  .
  def current_user?(user)
    user == current_user
  end

  def signed_in_user
    unless signed_in?
      store_location
      redirect_to signin_url, notice: "Please sign in."
    end
  end
end
.
```

```
.  
.  
end
```

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

With the code in [Listing 10.27](#), the `signed_in_user` 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 10.28](#). (Since we didn't generate it at the command line, you will have to create the Microposts controller file by hand.)

**Listing 10.28.** Adding authentication to the Microposts controller actions.

`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController  
  before_filter :signed_in_user  
  
  def create  
  end  
  
  def destroy  
  end  
end
```

Note that we haven't restricted the actions the before filter applies to since it applies to them both by default. 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 :signed_in_user, only: [:create, :destroy]  
  
  def index
```

```
end

def create
end


def destroy
end
end
```

At this point, the tests should pass:

```
$ bundle exec rspec spec/requests/authentication_pages_spec.rb
```

## 10.3.2 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 (following Twitter's convention) put the form on the Home page itself (i.e., the root path `/`), as mocked up in [Figure 10.10](#).



David Jones  
[view my profile](#)  
50 microposts

Compose new micropost...

Post

Figure 10.10: 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 5.6](#)—that is, it had a “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 10.31](#) below, but we can still write the tests now. As with the Users resource, we'll use an integration test:

```
$ rails generate integration_test micropost_pages
```

The micropost creation tests then parallel those for user creation from [Listing 7.16](#); the result appears in [Listing 10.29](#).

**Listing 10.29.** Tests for creating microposts.

`spec/requests/micropost_pages_spec.rb`

```
require 'spec_helper'

describe "Micropost pages" do

  subject { page }

  let(:user) { FactoryGirl.create(:user) }
  before { sign_in user }

  describe "micropost creation" do
    before { visit root_path }

    describe "with invalid information" do

      it "should not create a micropost" do
        expect { click_button "Post" }.not_to change(Micropost, :count)
      end

      describe "error messages" do
        before { click_button "Post" }
        it { should have_content('error') }
      end
    end
  end
end
```

```

describe "with valid information" do

  before { fill_in 'micropost_content', with: "Lorem ipsum" }
  it "should create a micropost" do
    expect { click_button "Post" }.to change(Micropost, :count).by(1)
  end
end
end
end

```

We'll start with the **create** action for microposts, which is similar to its user analogue ([Listing 7.25](#)); the principal difference lies in using the user/micropost association to **build** the new micropost, as seen in [Listing 10.30](#).

**Listing 10.30.** The Microposts controller **create** action.  
**app/controllers/microposts\_controller.rb**

```

class MicropostsController < ApplicationController
  before_filter :signed_in_user

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

  def destroy
  end
end

```

To build a form for creating microposts, we use the code in [Listing 10.31](#), which serves up different HTML based on whether the site visitor is signed in or not.

**Listing 10.31.** Adding microposts creation to the Home page ([/](#)).

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

```
<% if signed_in? %>
  <div class="row">
    <aside class="span4">
      <section>
        <%= render 'shared/user_info' %>
      </section>
      <section>
        <%= render 'shared/micropost_form' %>
      </section>
    </aside>
  </div>
<% else %>
  <div class="center hero-unit">
    <h1>Welcome to the Sample App</h1>

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

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

  </div>

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

Having so much code in each branch of the `if-else` conditional is a bit messy, and cleaning it up using partials is left as an exercise ([Section 10.5](#)). Filling in the necessary partials from [Listing 10.31](#) isn't an exercise, though; we fill in the new Home page sidebar in [Listing 10.32](#) and the micropost form partial in [Listing 10.33](#).

**Listing 10.32.** The partial for the user info sidebar.



app/views/shared/\_user\_info.html.erb

```
<a href="<%= user_path(current_user) %>">
  <%= gravatar_for current_user, size: 52 %>
</a>
<h1>
  <%= current_user.name %>
</h1>
<span>
  <%= link_to "view my profile", current_user %>
</span>
<span>
  <%= pluralize(current_user.microposts.count, "micropost") %>
</span>
```

As in [Listing 9.25](#), the code in [Listing 10.32](#) uses the version of the `gravatar_for` helper defined in [Listing 7.29](#).

Note that, as in the profile sidebar ([Listing 10.20](#)), the user info in [Listing 10.32](#) 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" (but "2 microposts") using `pluralize`.

We next define the form for creating microposts ([Listing 10.33](#)), which is similar to the signup form in [Listing 7.17](#).

**Listing 10.33.** The form partial for creating microposts.

app/views/shared/\_micropost\_form.html.erb

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

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

We need to make two changes before the form in [Listing 10.33](#) 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 10.34](#).

**Listing 10.34.** 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 signed_in?
  end
  .
  .
  .
end
```

The code in [Listing 10.34](#) has the advantage that it will break the test suite if we forget to require the user to sign in.

The second change needed to get [Listing 10.33](#) to work is to redefine the error messages partial so that

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

works. You may recall from [Listing 7.22](#) that the error messages partial references the `@user` variable explicitly, but in the present case we have an `@micropost` variable instead. We should define an error messages partial that works regardless of the kind of object passed to it. Happily, the form variable `f` can access the associated object through `f.object`, so that in

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

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

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

`f.object` is `@micropost`.

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 this code accomplishes:

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

In other words, `object: f.object` creates a variable called `object` in the `error_messages` partial. We can use this object to construct a customized error message, as shown in [Listing 10.35](#).

**Listing 10.35.** Updating the error-messages partial from [Listing 7.23](#) to work with other objects.  
`app/views/shared/_error_messages.html.erb`

```
<% if object.errors.any? %>
```

```

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

```

As this point, the tests in [Listing 10.29](#) should be passing:

```
$ bundle exec rspec spec/requests/micropost_pages_spec.rb
```

Unfortunately, the User request spec is now broken because the signup and edit forms use the old version of the error messages partial. To fix them, we'll update them with the more general version, as shown in [Listing 10.36](#) and [Listing 10.37](#). (Note: Your code will differ if you implemented [Listing 9.50](#) and [Listing 9.51](#) from the exercises in [Section 9.6](#). *Mutatis mutandis*.)

**Listing 10.36.** Updating the rendering of user signup errors.

`app/views/users/new.html.erb`

```

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

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>
      .
      .
      .
    <% end %>
  </div>
</div>

```

```
</div>
</div>
```

**Listing 10.37.** Updating the errors for editing users.

`app/views/users/edit.html.erb`

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

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>
      .
      .
      .
    <% end %>

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

At this point, all the tests should be passing:

```
$ bundle exec rspec spec/
```

Additionally, all the HTML in this section should render properly, showing the form as in [Figure 10.11](#), and a form with a submission error as in [Figure 10.12](#). 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 10.3.3](#).

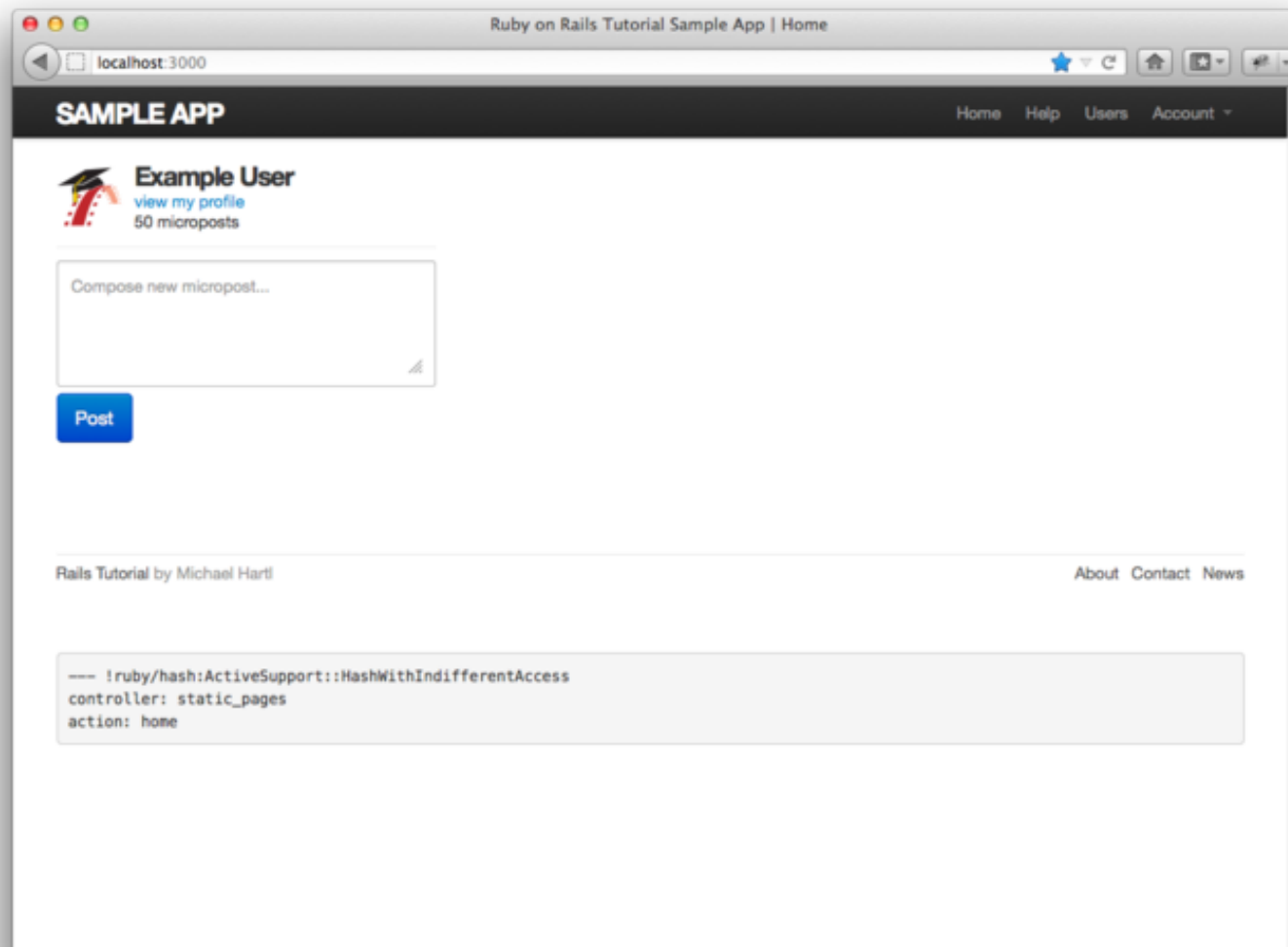


Figure 10.11: The Home page (/) with a new micropost form. [\(full size\)](#)

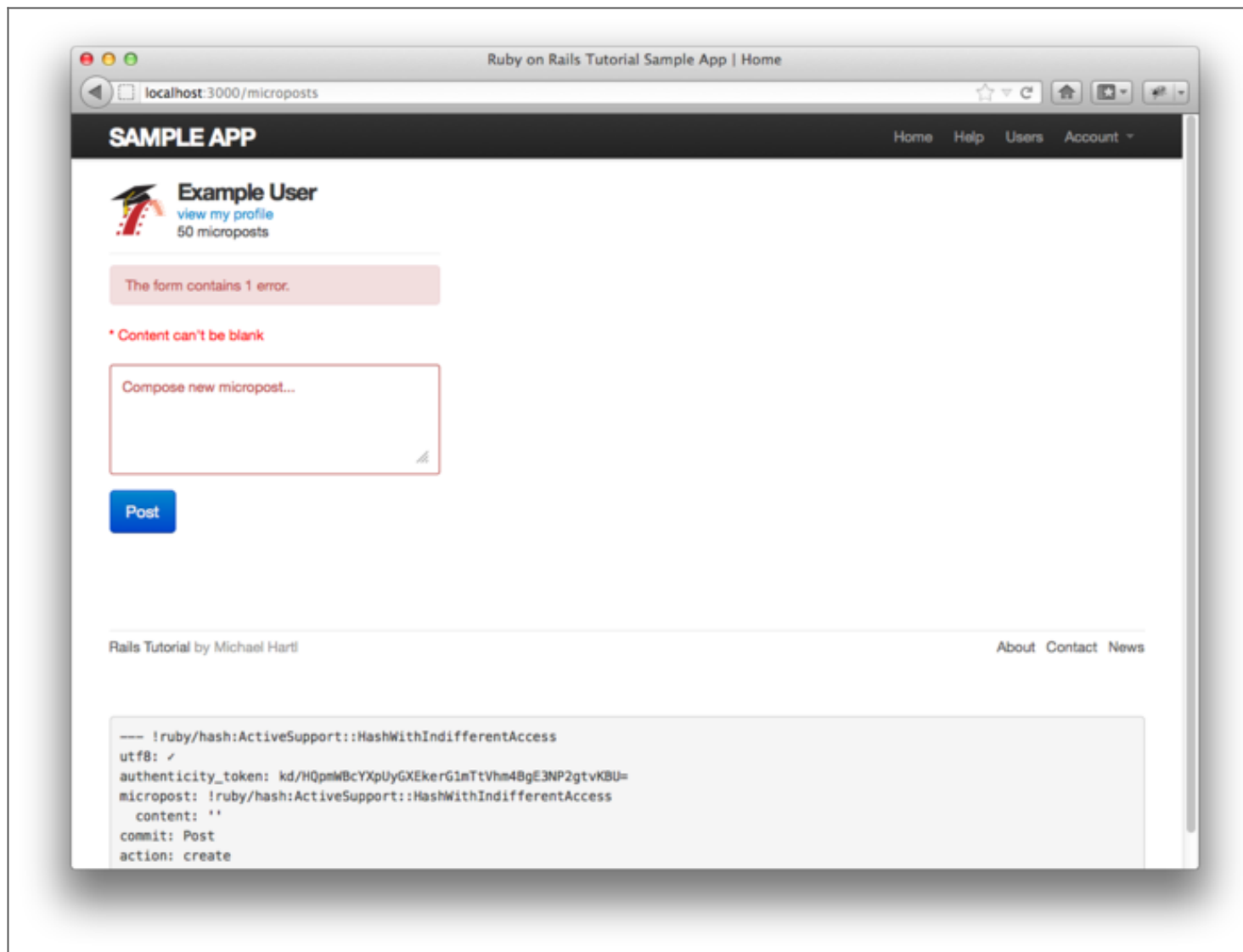


Figure 10.12: The home page with form errors. ([full size](#))

### 10.3.3 A proto-feed

The comment at the end of [Section 10.3.2](#) 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 10.11](#) 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 10.13](#). (In [Chapter 11](#), we'll generalize this feed to include the microposts of users being *followed* by the current user.)



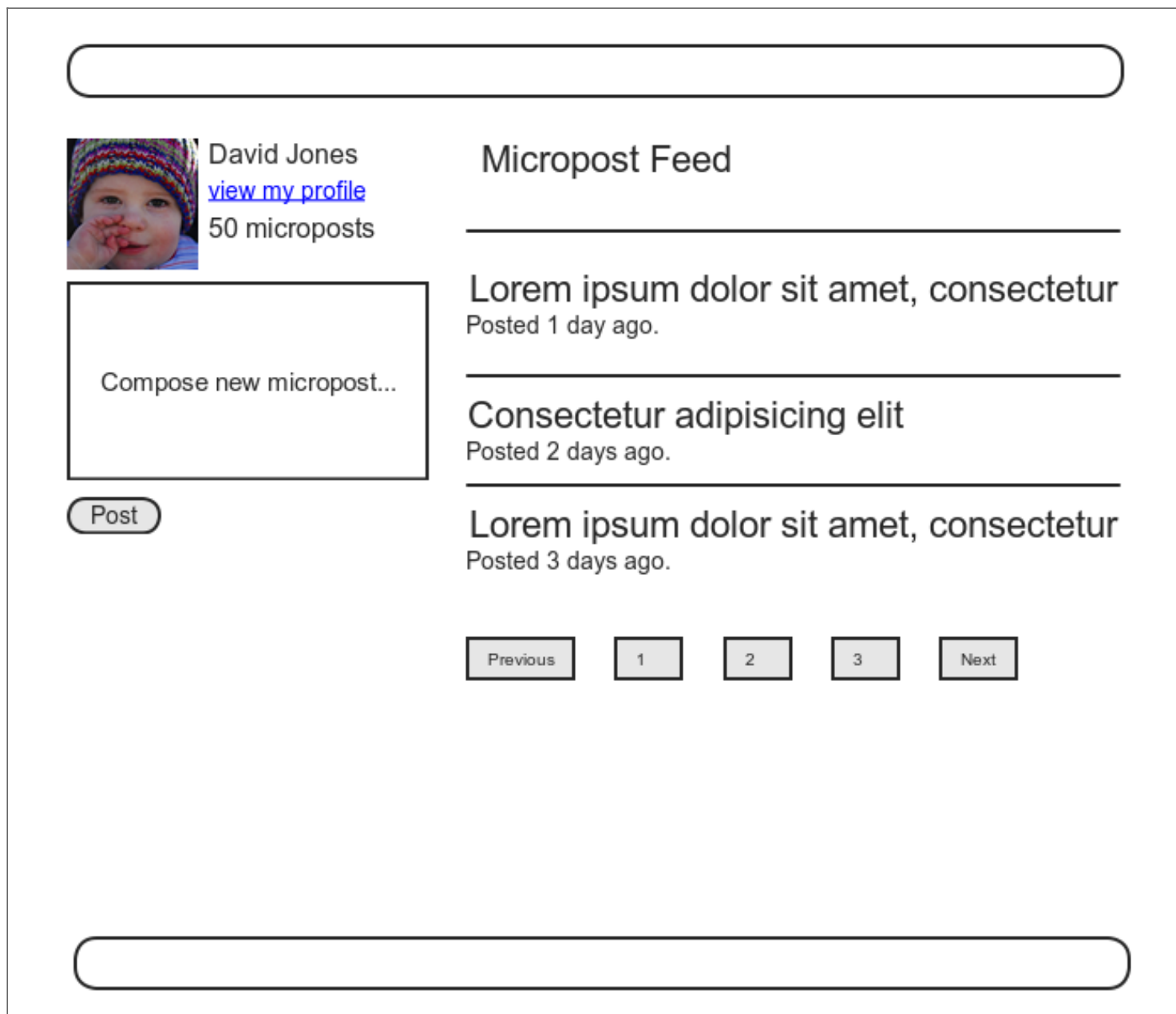


Figure 10.13: A mockup of the Home page with a proto-feed. ([full size](#))

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 10.38](#).

**Listing 10.38.** Tests for the (proto-)status feed.

**spec/models/user\_spec.rb**

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:microposts) }
  it { should respond_to(:feed) }
  .
  .
  .
  describe "micropost associations" do

    before { @user.save }
    let!(:older_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.day.ago)
    end
    let!(:newer_micropost) do
      FactoryGirl.create(:micropost, user: @user, created_at: 1.hour.ago)
    end
    .
    .
    .
    describe "status" do
      let(:unfollowed_post) do
        FactoryGirl.create(:micropost, user: FactoryGirl.create(:user))
      end

      its(:feed) { should include(newer_micropost) }
      its(:feed) { should include(older_micropost) }
      its(:feed) { should_not include(unfollowed_post) }
    end
  end
end
```

These tests introduce (via the RSpec boolean convention) the array `include?` method, which simply checks if an array includes the given element:<sup>9</sup>

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

This example shows just how flexible the RSpec boolean convention is; even though `include` is already a Ruby keyword (used to include a module, as seen in, e.g., [Listing 8.14](#)), in this context RSpec correctly guesses that we want to test array inclusion.

We can arrange for an appropriate micropost `feed` method by selecting all the microposts with `user_id` equal to the current user's id, which we accomplish using the `where` method on the `Micropost` model, as shown in [Listing 10.39](#).<sup>10</sup>

**Listing 10.39.** A preliminary implementation for the micropost status feed.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def feed
    # This is preliminary. See "Following users" for the full implementation.
    Micropost.where("user_id = ?", id)
  end
  .
end
```

```
.  
.  
end
```

The question mark in

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

ensures that `id` is properly *escaped* before being included in the underlying SQL query, thereby avoiding a serious security hole called [SQL injection](#). The `id` attribute here is just an integer, 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 10.39](#) is essentially equivalent to writing

```
def feed  
  microposts  
end
```

We've used the code in [Listing 10.39](#) instead because it generalizes much more naturally to the full status feed needed in [Chapter 11](#).

To test the display of the status feed, we first create a couple of microposts and then verify that a list element (`li`) appears on the page for each one ([Listing 10.40](#)).

**Listing 10.40.** A test for rendering the feed on the Home page.

```
spec/requests/static_pages_spec.rb
```

```

require 'spec_helper'

describe "Static pages" do

  subject { page }

  describe "Home page" do
    .
    .
    .
    describe "for signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        FactoryGirl.create(:micropost, user: user, content: "Lorem ipsum")
        FactoryGirl.create(:micropost, user: user, content: "Dolor sit amet")
        sign_in user
        visit root_path
      end

      it "should render the user's feed" do
        user.feed.each do |item|
          page.should have_selector("li##{item.id}", text: item.content)
        end
      end
    end
  end
end

```

[Listing 10.40](#) assumes that each feed item has a unique CSS id, so that

```
page.should have_selector("li##{item.id}", text: item.content)
```

will generate a match for each item. (Note that the first `#` in `li##{item.id}` is Capybara syntax for a CSS id, whereas the second `#` is the beginning of a Ruby string interpolation `#{}`.)

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 10.41](#), and then add a feed partial ([Listing 10.42](#)) to the Home page ([Listing 10.44](#)). (Adding tests for pagination is left as an exercise; see [Section 10.5](#).)

**Listing 10.41.** Adding a feed instance variable to the `home` action.

`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

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

**Listing 10.42.** The status feed partial.

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

```
<% if @feed_items.any? %>
  <ol class="microposts">
    <%= render partial: 'shared/feed_item', collection: @feed_items %>
  </ol>
  <%= will_paginate @feed_items %>
<% end %>
```

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

```
<%= render partial: 'shared/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 'shared/micropost'`, but with a `:collection` parameter that syntax doesn't work.) The feed item partial itself appears in [Listing 10.43](#).

**Listing 10.43.** A partial for a single feed item.

`app/views/shared/_feed_item.html.erb`

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

[Listing 10.43](#) also adds a CSS id for each feed item using

```
<li id="<%= feed_item.id %>">
```

as required by the test in [Listing 10.40](#).

We can then add the feed to the Home page by rendering the feed partial as usual ([Listing 10.44](#)). The result is a display of the feed on the Home page, as required ([Figure 10.14](#)).

**Listing 10.44.** Adding a status feed to the Home page.

app/views/static\_pages/home.html.erb

```
<% if signed_in? %>
  <div class="row">
    .
    .
    .
    <div class="span8">
      <h3>Micropost Feed</h3>
      <%= render 'shared/feed' %>
    </div>
  </div>
<% else %>
  .
  .
  .
<% end %>
```



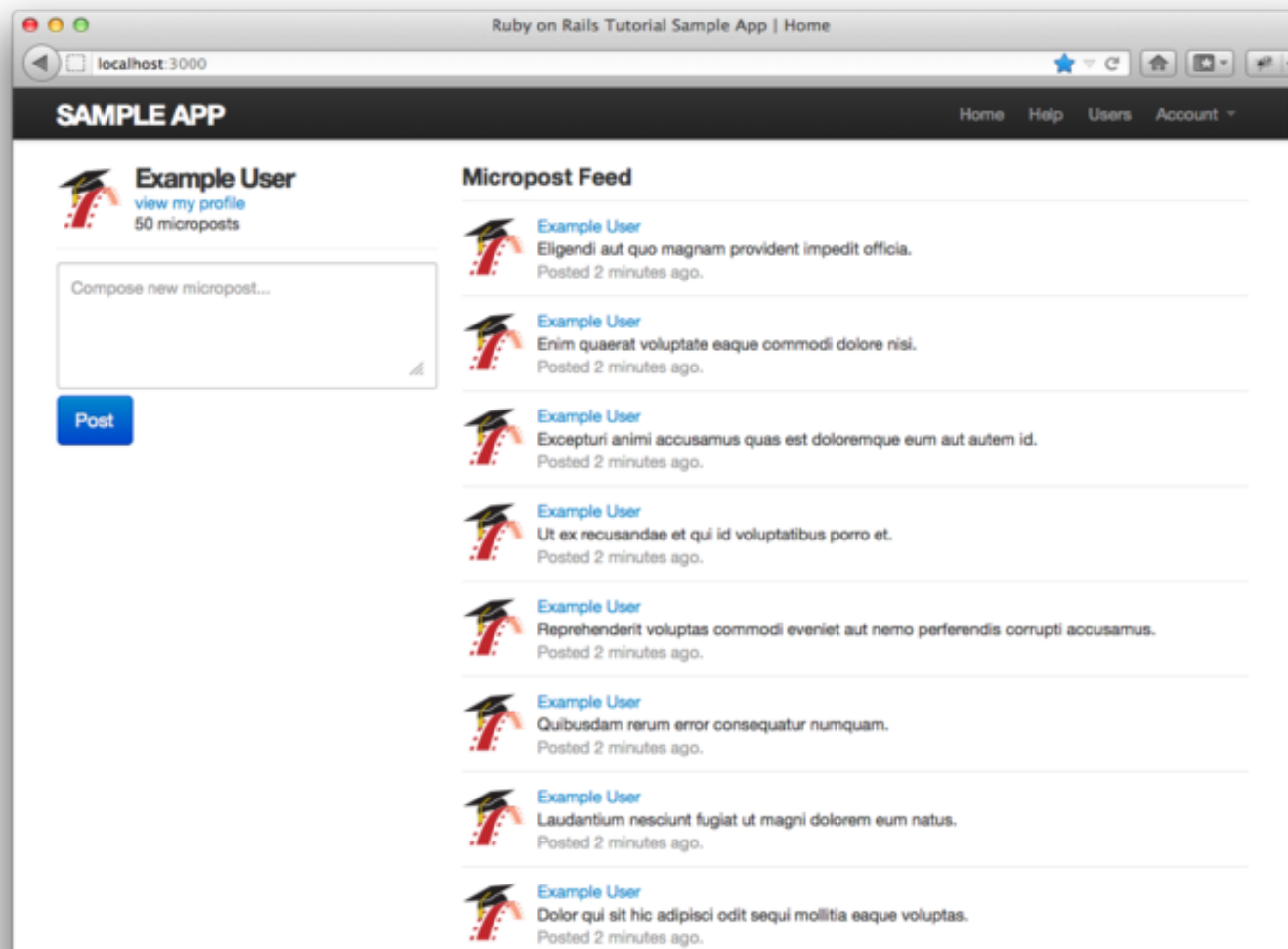


Figure 10.14: The Home page (/) with a proto-feed. [\(full size\)](#)

At this point, creating a new micropost works as expected, as seen in [Figure 10.15](#). 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 10.45](#).<sup>11</sup>

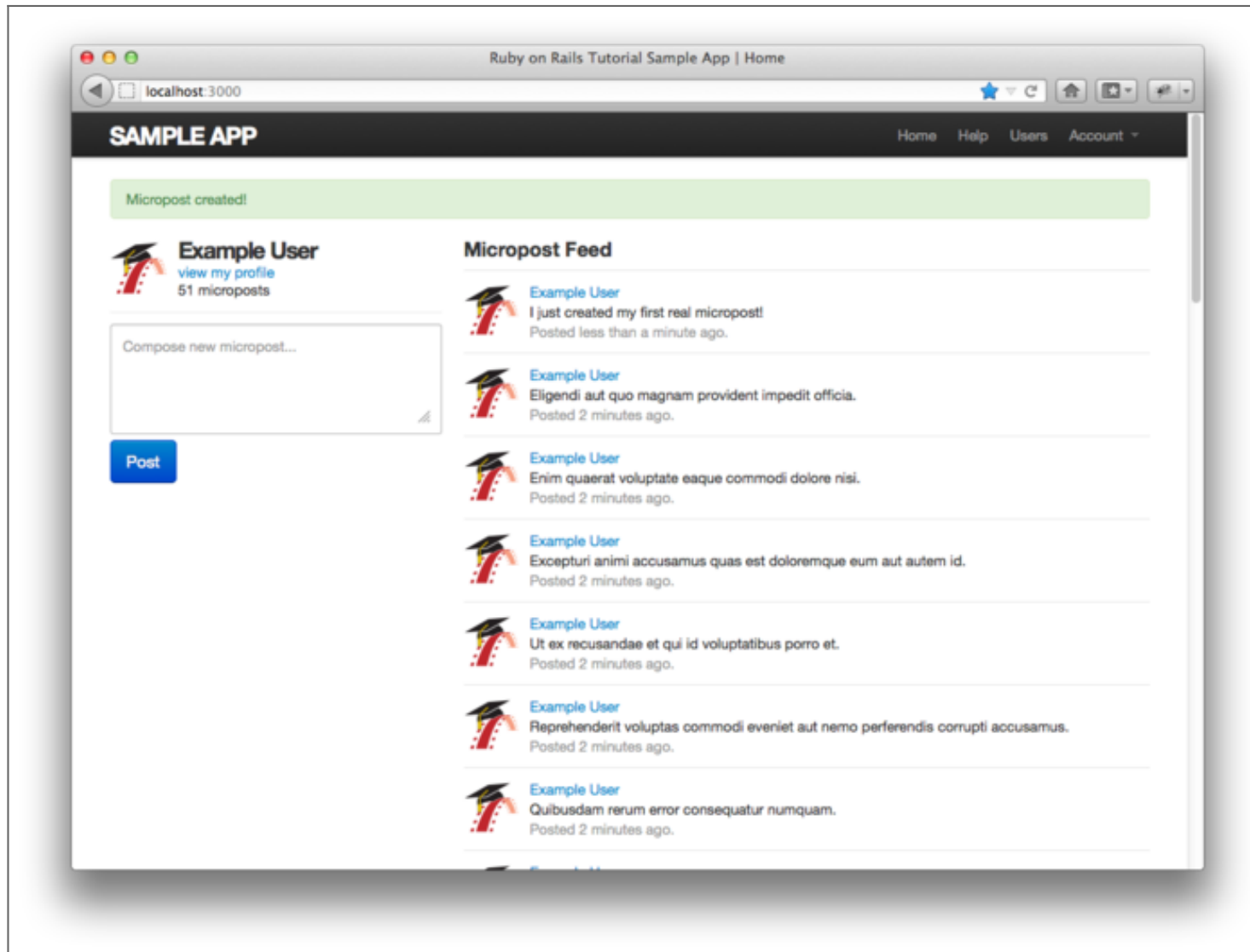


Figure 10.15: The Home page after creating a new micropost. ([full size](#))

**Listing 10.45.** 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_url
    else
      @feed_items = []
      render 'static_pages/home'
    end
  end
  .
  .
  .
end
```

At this point, the proto-feed should be working, and the test suite should pass:

```
$ bundle exec rspec spec/
```

### 10.3.4 Destroying microposts

The last piece of functionality to add to the Microposts resource is the ability to destroy posts. As with user deletion ([Section 9.4.2](#)), we accomplish this with “delete” links, as mocked up in [Figure 10.16](#). Unlike that case, which restricted user destruction to admin users, the delete links will work only for microposts created by the current user.

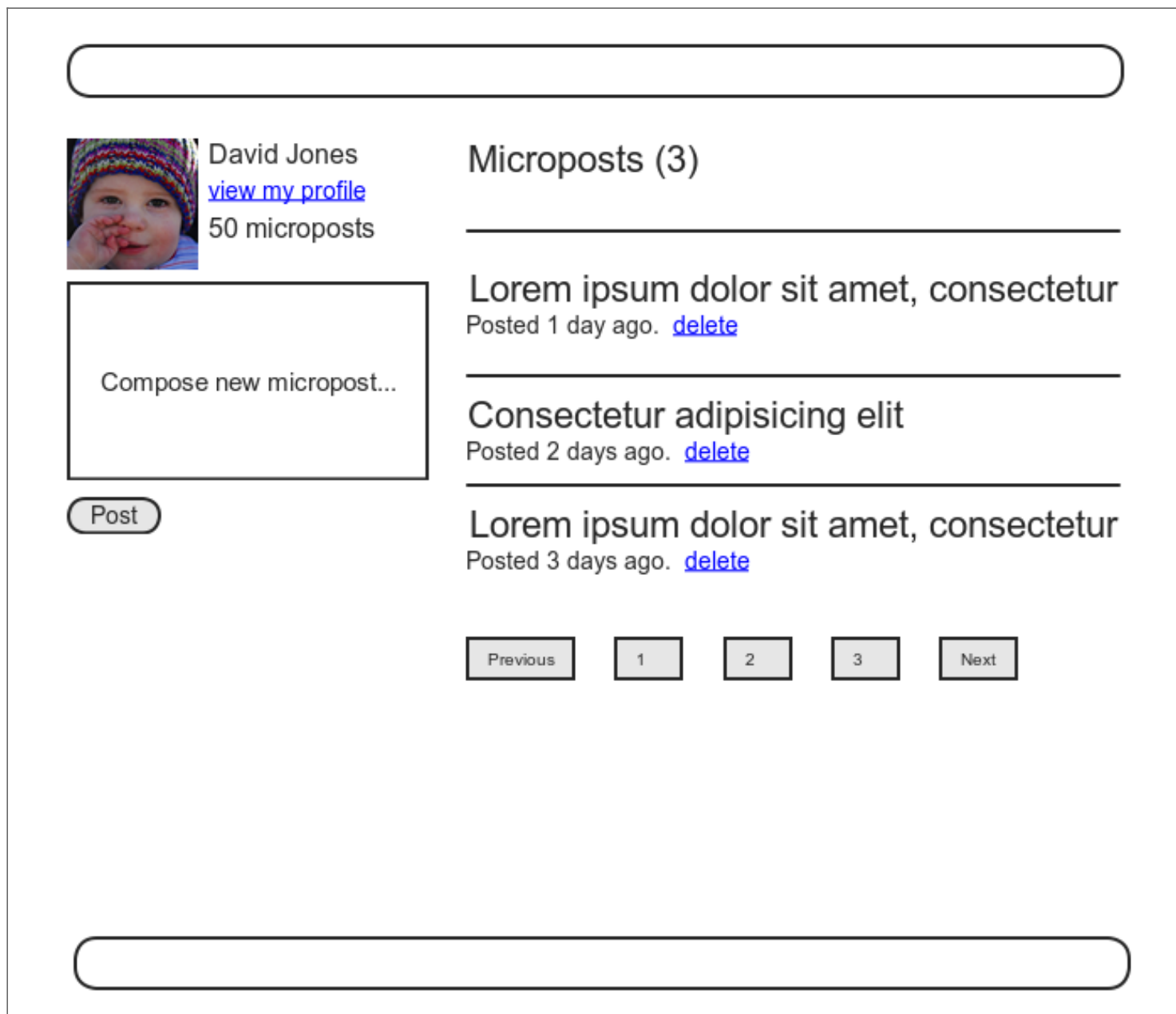


Figure 10.16: A mockup of the proto-feed with micropost delete links. ([full size](#))

Our first step is to add a delete link to the micropost partial as in [Listing 10.43](#), and while we're at it

we'll add a similar link to the feed item partial from [Listing 10.43](#). The results appear in [Listing 10.46](#) and [Listing 10.47](#). (The two cases are almost identical, and eliminating this duplication is left as an exercise ([Section 10.5](#)).)

**Listing 10.46.** Adding a delete link to the micropost partial.

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

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

  <% end %>
</li>
```

**Listing 10.47.** The feed item partial with added delete link.

`app/views/shared/_feed_item.html.erb`

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

  <% end %>
</li>
```

The test for destroying microposts uses Capybara to click the “delete” link and expects the Micropost count to decrease by 1 ([Listing 10.48](#)).

**Listing 10.48.** Tests for the Microposts controller `destroy` action.

`spec/requests/micropost_pages_spec.rb`

```
require 'spec_helper'

describe "Micropost pages" do
  .
  .
  .
  describe "micropost destruction" do
    before { FactoryGirl.create(:micropost, user: user) }

    describe "as correct user" do
      before { visit root_path }

      it "should delete a micropost" do
        expect { click_link "delete" }.to change(Micropost, :count).by(-1)
      end
    end
  end
end
```

The application code is also analogous to the user case in [Listing 9.48](#); the main difference is that, rather than using an `admin_user` before filter, in the case of microposts we have a `correct_user` before filter to check that the current user actually has a micropost with the given id. The code appears in [Listing 10.49](#), and the result of destroying the second-most-recent post appears in [Figure 10.17](#).

**Listing 10.49.** The Microposts controller `destroy` action.

`app/controllers/microposts_controller.rb`

```

class MicropostsController < ApplicationController
  before_filter :signed_in_user, only: [:create, :destroy]
  before_filter :correct_user,    only: :destroy
  .
  .
  .
  def destroy
    @micropost.destroy
    redirect_to root_url
  end

  private

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

```

In the `correct_user` before filter, note that we find microposts *through* the association:

```
current_user.microposts.find_by_id(params[:id])
```

This automatically ensures that we find only microposts belonging to the current user. In this case, we use `find_by_id` instead of `find` because the latter raises an exception when the micropost doesn't exist instead of returning `nil`. By the way, if you're comfortable with exceptions in Ruby, you could also write the `correct_user` filter like this:

```

def correct_user
  @micropost = current_user.microposts.find(params[:id])
rescue
  redirect_to root_url
end

```

It might occur to you that we could implement the `correct_user` filter using the `Micropost` model directly, like this:

```
@micropost = Micropost.find_by_id(params[:id])  
redirect_to root_url unless current_user?(@micropost.user)
```

This would be equivalent to the code in [Listing 10.49](#), but, as explained by [Wolfram Arnold](#) in the blog post [Access Control 101 in Rails and the Citibank Hack](#), for security purposes it is a good practice always to run lookups through the association.



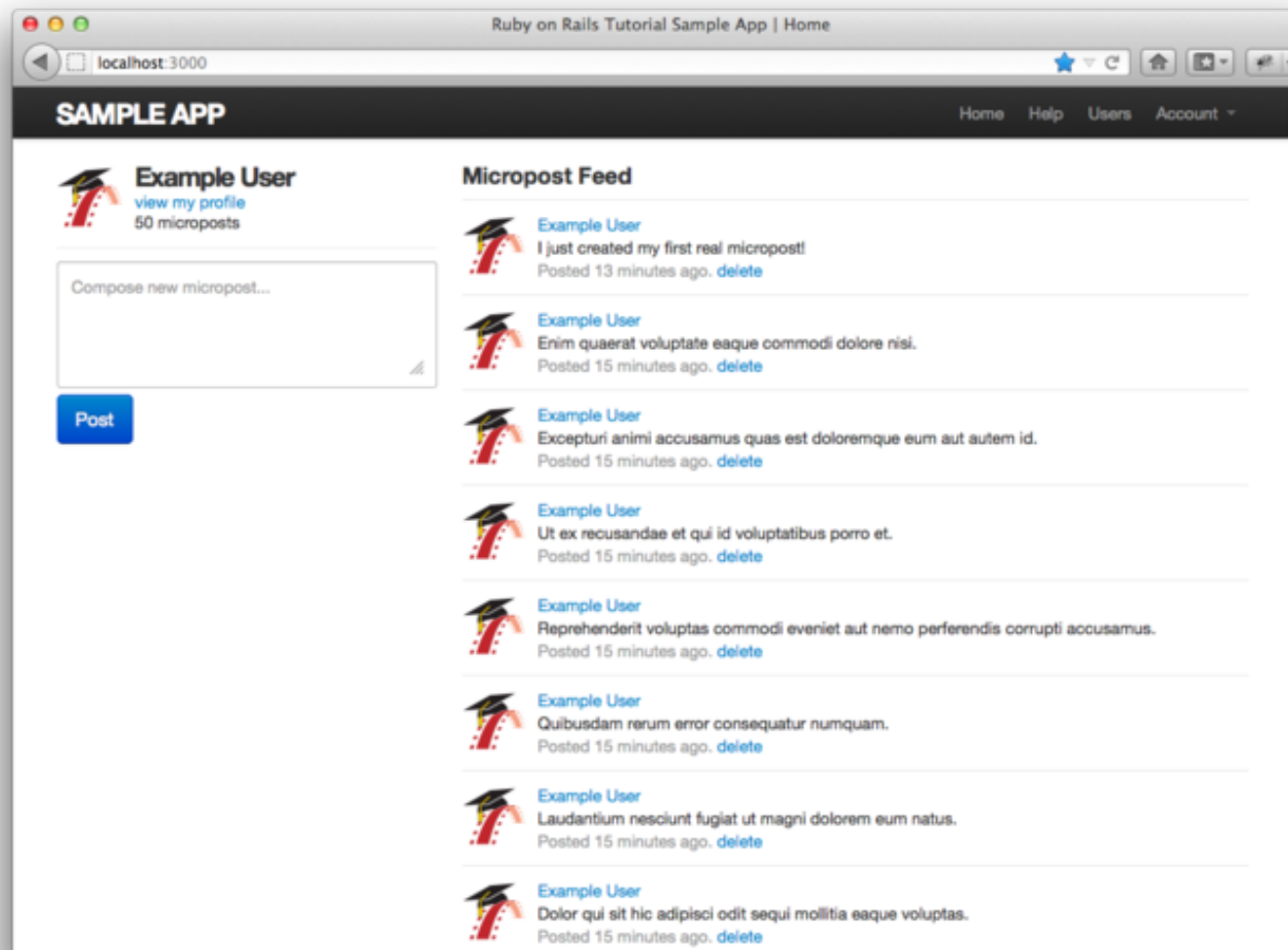


Figure 10.17: The user home page after deleting the second-most-recent micropost. ([full size](#))

With the code in this section, our Micropost model and interface are complete, and the test suite should pass:

```
$ bundle exec rspec spec/
```

## 10.4 Conclusion

With the addition of the Microposts resource, we are nearly finished with our sample application. All that remains is to add a social layer by letting users follow each other. We'll learn how to model such user relationships, and see the implications for the status feed, in [Chapter 11](#).

Before proceeding, be sure to commit and merge your changes if you're using Git for version control:

```
$ git add .  
$ git commit -m "Add user microposts"  
$ git checkout master  
$ git merge user-microposts  
$ git push
```

You can also push the app up to Heroku at this point. Because the data model has changed through the addition of the **microposts** table, you will also need to migrate the production database:

```
$ git push heroku  
$ heroku pg:reset <DATABASE>  
$ heroku run rake db:migrate  
$ heroku run rake db:populate
```

Follow the instructions in [Section 9.5](#) to find the right replacement for the **DATABASE** argument in the second command above.

## 10.5 Exercises

We've covered enough material now that there is a combinatorial explosion of possible extensions to the application. Below are just a few of the many possibilities.

1. Add tests for the sidebar micropost counts (including proper pluralization).
2. Add tests for micropost pagination.
3. Refactor the Home page to use separate partials for the two branches of the `if-else` statement.
4. Write a test to make sure delete links do not appear for microposts not created by the current user.
5. Using partials, eliminate the duplication in the delete links from [Listing 10.46](#) and [Listing 10.47](#).
6. Very long words currently break our layout, as shown in [Figure 10.18](#). Fix this problem using the `wrap` helper defined in [Listing 10.50](#). Note the use of the `raw` method to prevent Rails from escaping the resulting HTML, together with the `sanitize` method needed to prevent cross-site scripting. This code also uses the strange-looking but useful *ternary operator* ([Box 10.1](#)).
7. **(challenging)** Add a JavaScript display to the Home page to count down from 140 characters.

#### Box 10.1. 10 types of people

There are 10 kinds of people in the world: Those who like the ternary operator, those who don't, and those who don't know about it. (If you happen to be in the third category, soon you won't be any longer.)

When you do a lot of programming, you quickly learn that one of the most common bits of control flow goes something like this:

```
if boolean?  
  do_one_thing  
else  
  do_something_else  
end
```

Ruby, like many other languages (including C/C++, Perl, PHP, and Java), allows you to replace this with a much more compact expression using the *ternary operator* (so called because it consists of three parts):

```
boolean? ? do_one_thing : do_something_else
```

You can also use the ternary operator to replace assignment:

```
if boolean?  
  var = foo  
else  
  var = bar  
end
```

becomes

```
var = boolean? ? foo : bar
```

Another common use is in a function's return value:

```
def foo  
  do_stuff  
  boolean? ? "bar" : "baz"
```

end

Since Ruby implicitly returns the value of the last expression in a function, here the `foo` method returns `"bar"` or `"baz"` depending on the value of `boolean?`. It is this final construction that appears in [Listing 10.50](#).

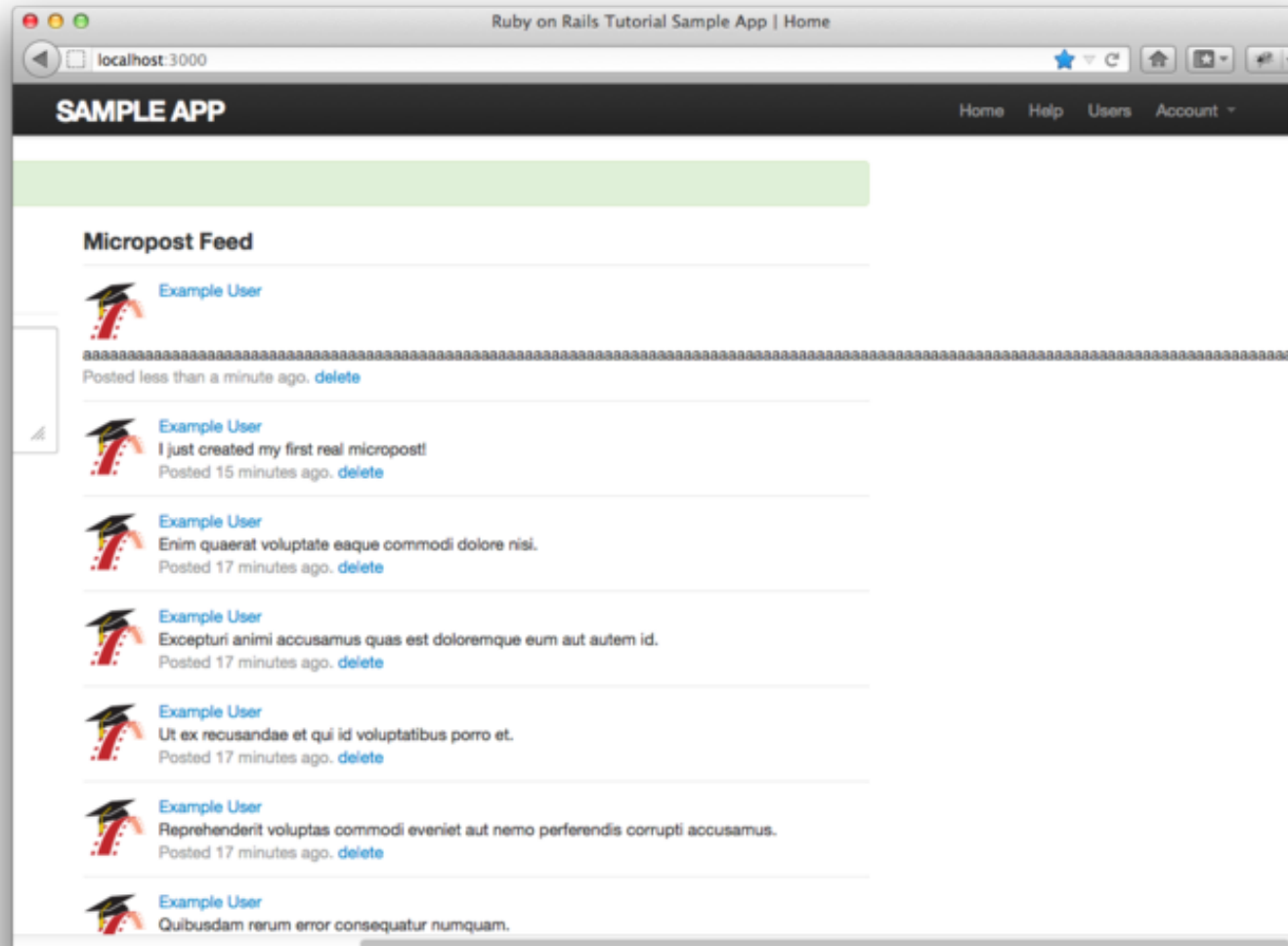


Figure 10.18: The (broken) site layout with a particularly long word. [\(full size\)](#)

**Listing 10.50.** A helper to wrap long words.

`app/helpers/microposts_helper.rb`

```
module MicropostsHelper

  def wrap(content)
    sanitize(raw(content.split.map{ |s| wrap_long_string(s) }.join(' ')))
  end

  private

  def wrap_long_string(text, max_width = 30)
    zero_width_space = "&#8203;"
    regex = /.{1,#{max_width}}/
    (text.length < max_width) ? text :
      text.scan(regex).join(zero_width_space)
  end
end
```

[« Chapter 9 Updating, showing, and deleting users](#)

[Chapter 11 Following users »](#)

- 
1. Technically, we treated sessions as a resource in [Chapter 8](#), but sessions are not saved to the database the way users and microposts are. ↑
  2. The `content` attribute will be a `string`, but, as noted briefly in [Section 2.1.2](#), for longer text fields you should use the `text` data type. ↑
  3. (i.e., the five users with custom Gravatars, and one with the default Gravatar) ↑
  4. Tail your `log/development.log` file if you're curious about the SQL this method generates. ↑

5. By design, the Faker gem's *lorem ipsum* text is randomized, so the contents of your sample microposts will differ. ↑
6. For convenience, [Listing 10.24](#) actually has *all* the CSS needed for this chapter. ↑
7. The other two resources are Users in [Section 7.2](#) and Sessions in [Section 8.1](#). ↑
8. We noted in [Section 8.2.1](#) that helper methods are available only in *views* by default, but we arranged for the Sessions helper methods to be available in the controllers as well by adding `include SessionsHelper` to the Application controller ([Listing 8.14](#)). ↑
9. Learning about methods such as `include?` is one reason why, as noted in [Section 1.1.1](#), I recommend reading a pure Ruby book after finishing this one. ↑
10. See the Rails Guide on the [Active Record Query Interface](#) for more on `where` and the like. ↑
11. Unfortunately, returning a paginated feed doesn't work in this case. Implement it and click on a pagination link to see why. ↑

Michael Hartl is a participant in the Amazon Services LLC Associates Program, an affiliate advertising program designed to provide a means for sites to earn advertising fees by advertising and linking to Amazon.com.