Please see the post **Do not buy the print version of the Ruby on Rails Tutorial (yet)**

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

Password reminders
Signup confirmation
RSS feed
REST API
Search
11.4.2 Guide to further resources
11.5 Exercises

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)
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

Chapter 11
Following users

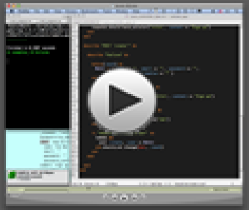
In this chapter, we will complete the core sample application by adding a social layer that allows users to follow (and unfollow) other users, resulting in each user's Home page displaying a status feed of the followed users' microposts. We will also make views to display both a user's followers and the users each user is following. We will learn how to model relationships between users in Section 11.1, and then make the web interface in Section 11.2 (including an introduction to Ajax). Finally, we'll end by developing a fully functional status feed in Section 11.3.

This final chapter contains some of the most challenging material in the tutorial, including some Ruby/SQL trickery to make the status feed. Through these examples, you will see how Rails can handle even rather intricate data models, which should serve you well as you go on to develop your own applications with their own specific requirements. To help with the transition from tutorial to independent development, Section 11.4 contains suggested extensions to the core sample application, along with pointers to more advanced resources.

As usual, Git users should create a new topic branch:

```
$ git checkout -b following-users
```

Because the material in this chapter is particularly challenging, before writing any code we'll pause for a moment and take a tour of the interface. As in previous chapters, at this early stage we'll represent pages using mockups.[1] The full page flow runs as follows: a user (John Calvin) starts at



his profile page (Figure 11.1) and navigates to the Users page (Figure 11.2) to select a user to follow. Calvin navigates to the profile of a second user, Thomas Hobbes (Figure 11.3), clicking on the "Follow" button to follow that user. This changes the "Follow" button to "Unfollow", and increments Hobbes's "followers" count by one (Figure 11.4). Navigating to his home page, Calvin now sees an incremented "following" count and finds Hobbes's microposts in his status feed (Figure 11.5). The rest of this chapter is dedicated to making this page flow actually work.

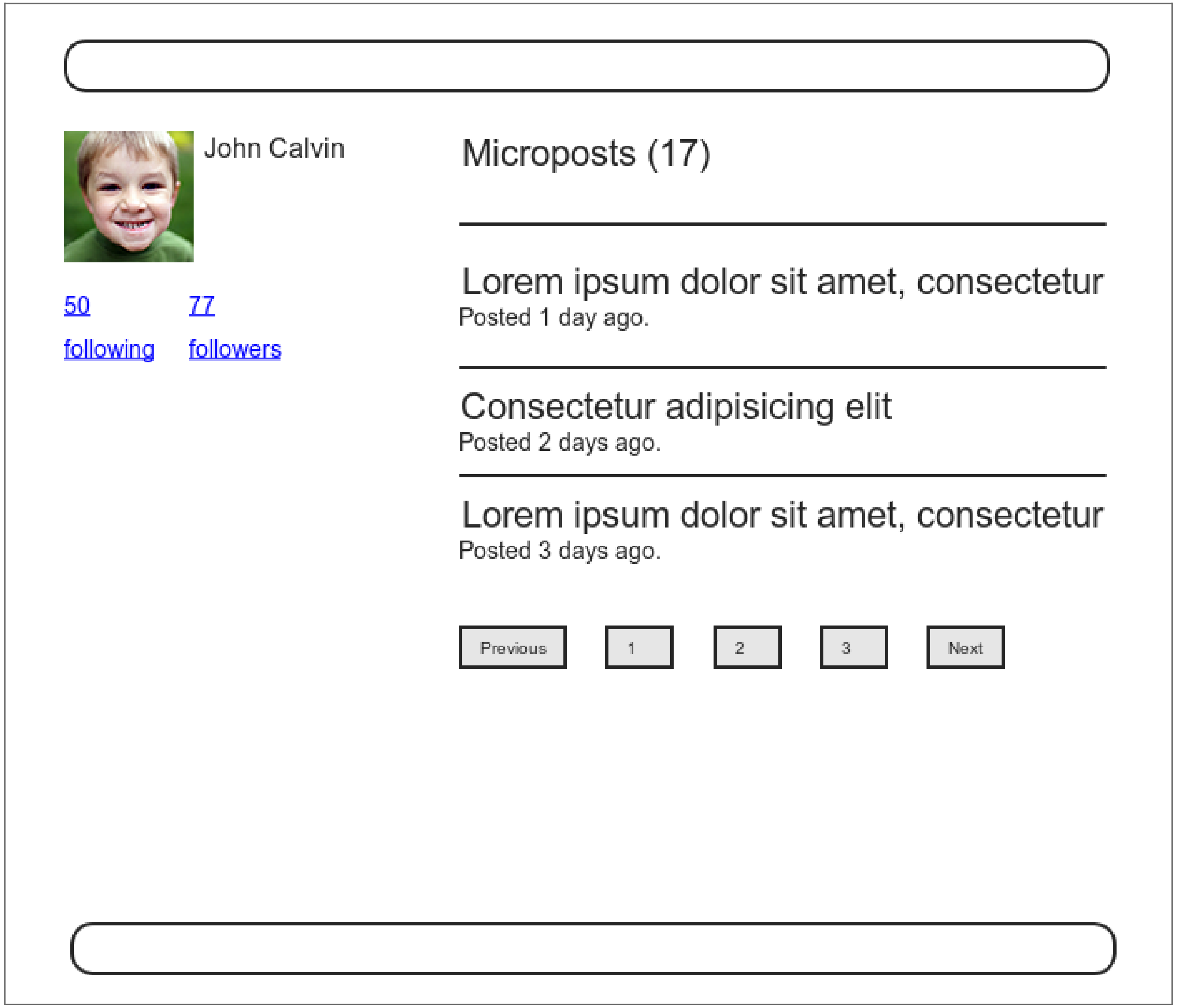


Figure 11.1: The current user’s profile. (full size)

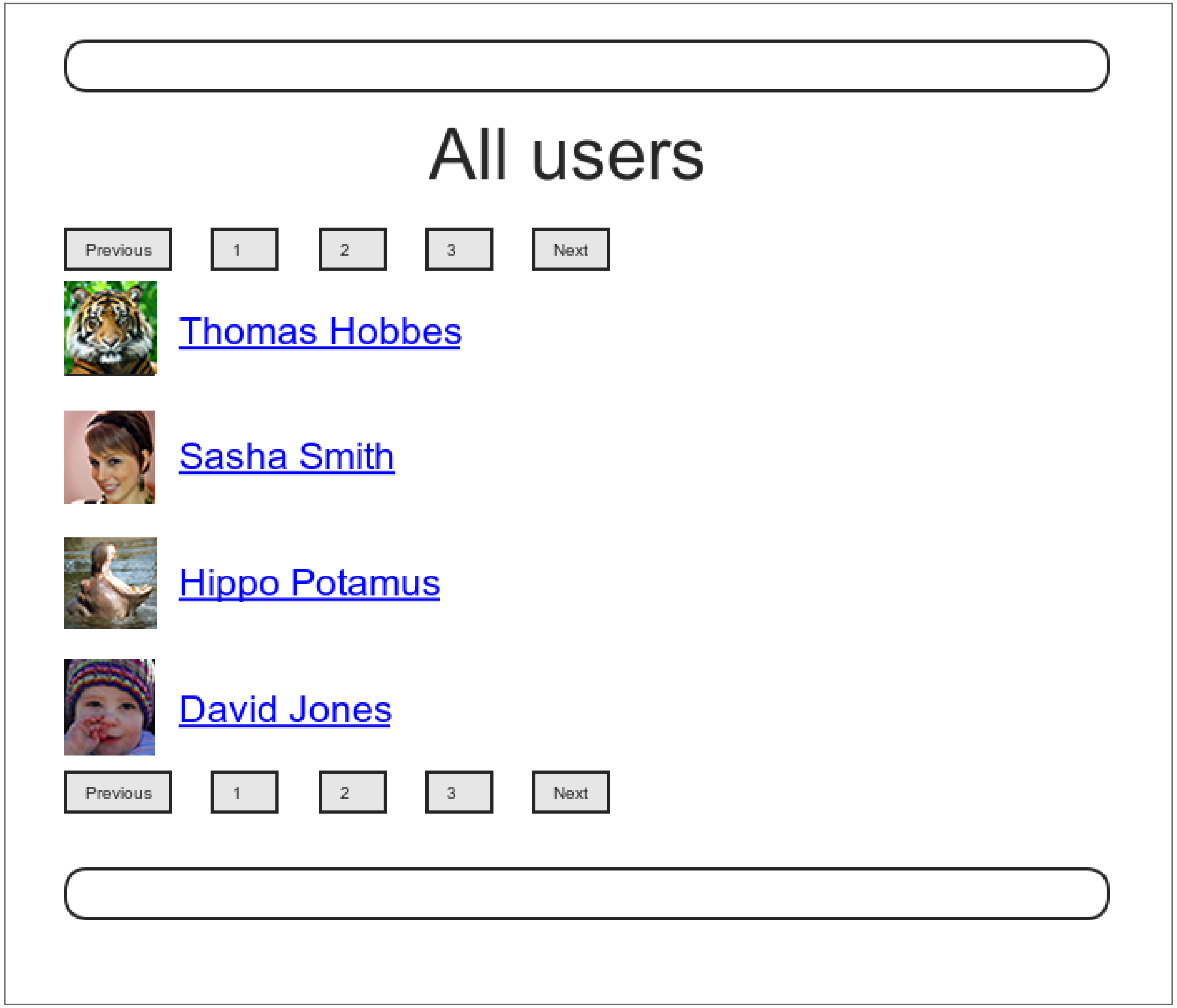


Figure 11.2: Finding a user to follow. (full size)

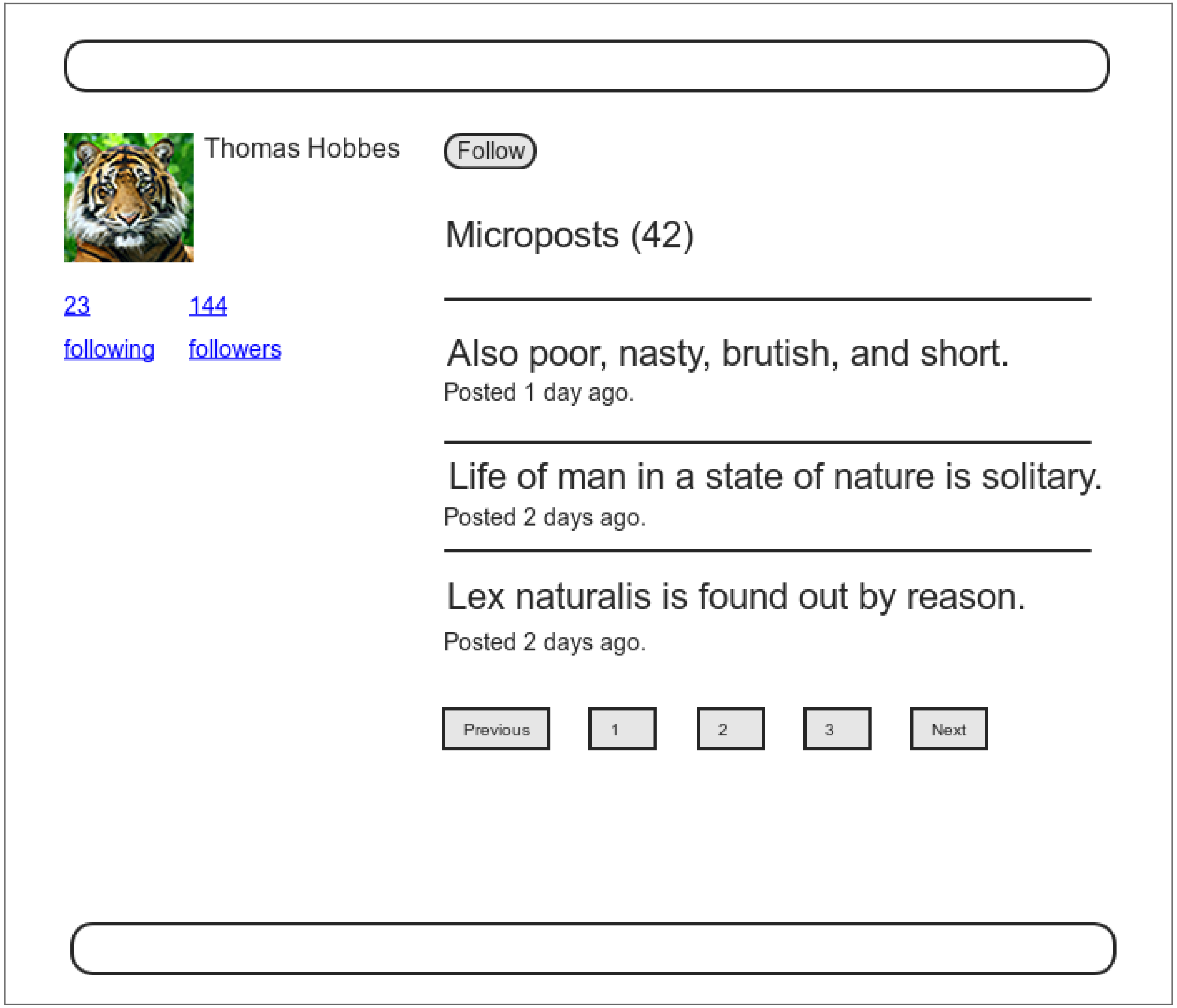


Figure 11.3: The profile of a user to follow, with a follow button. (full size)

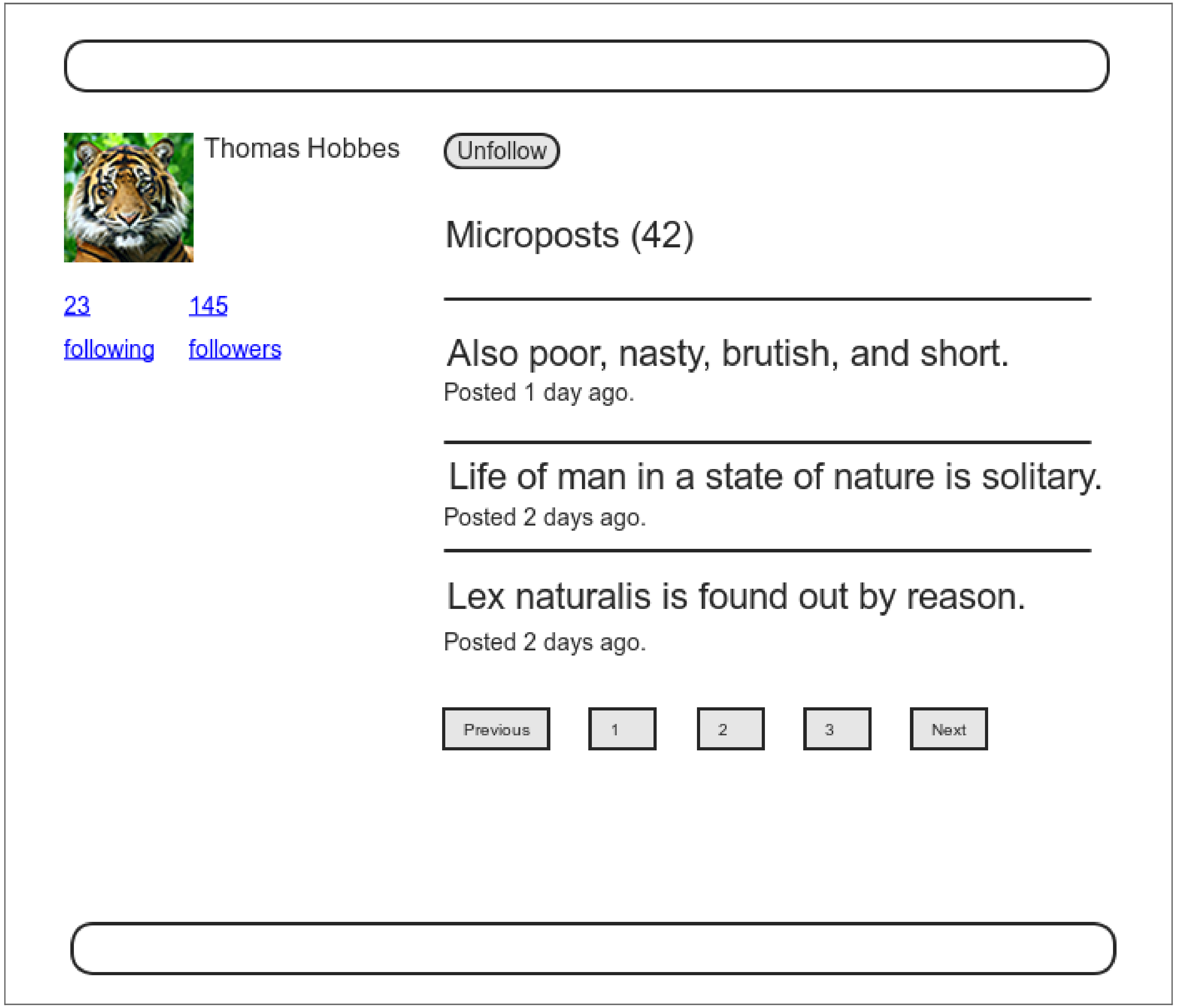
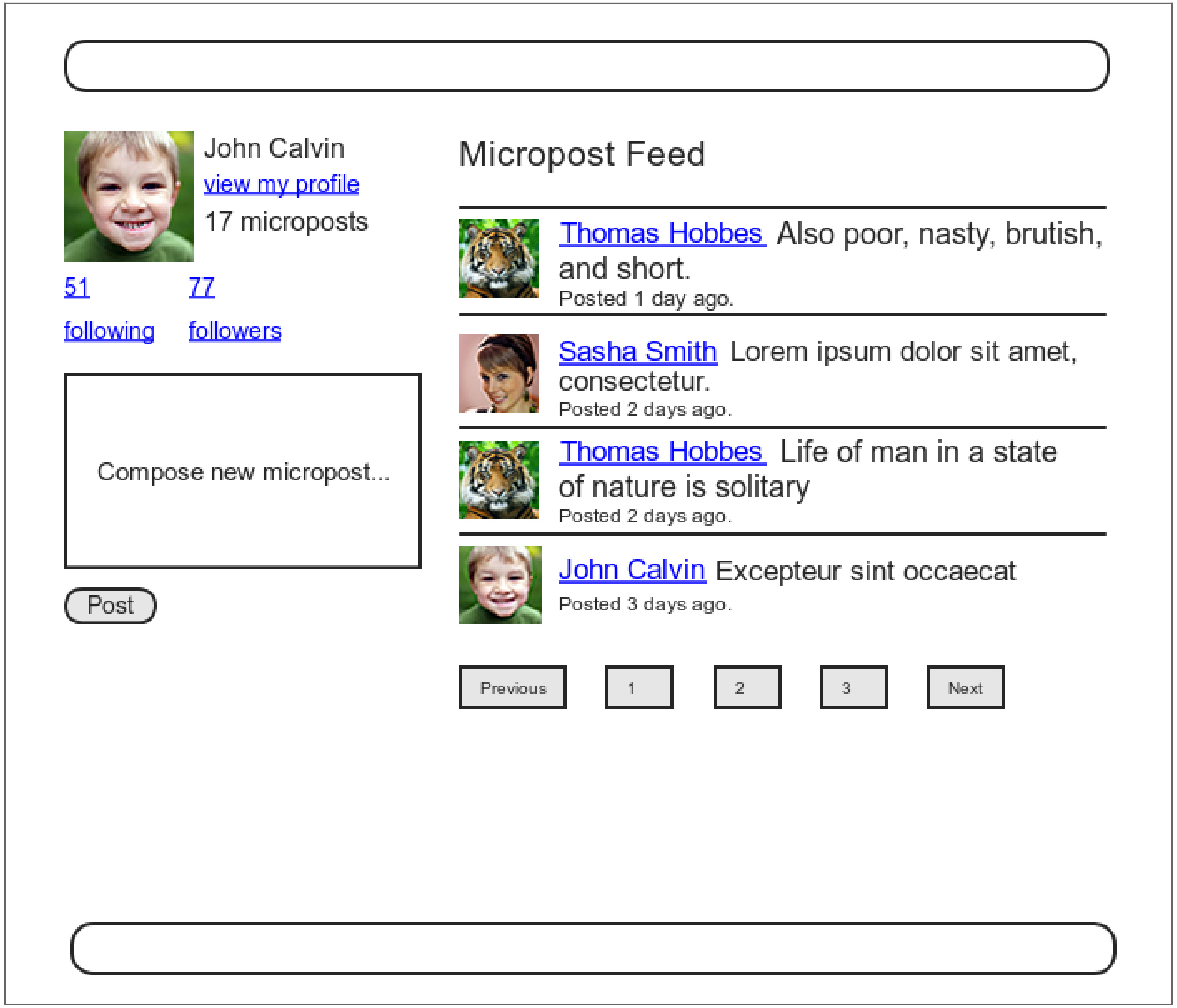


Figure 11.4: A profile with an unfollow button and incremented followers count. (full size)


John Calvin
view my profile
17 microposts
51 following
77 followers
Compose new micropost...
Post

Micropost Feed

Thomas Hobbes Also poor, nasty, brutish, and short.
Posted 1 day ago.

Sasha Smith Lorem ipsum dolor sit amet, consectetur.
Posted 2 days ago.

Thomas Hobbes Life of man in a state of nature is solitary
Posted 2 days ago.

John Calvin Excepteur sint occaecat
Posted 3 days ago.

Figure 11.5: The Home page with status feed and incremented following count. (full size)

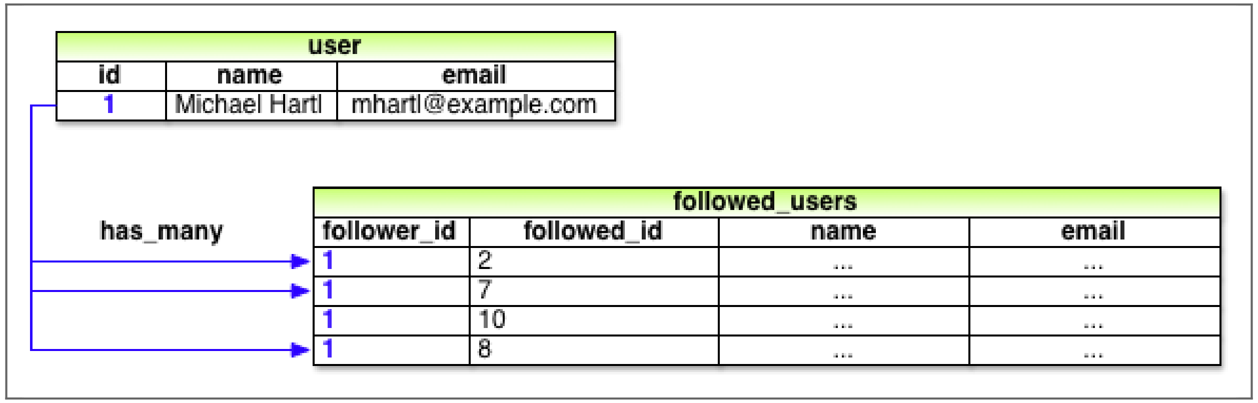
11.1 The Relationship model

Our first step in implementing following users is to construct a data model, which is not as straightforward as it seems. Naïvely, it seems that a **`has_many`** relationship should do: a user **`has_many`** followed users and **`has_many`** followers. As we will see, there is a problem with this approach, and we'll learn how to fix it using **`has_many through`**. It's likely that many of the ideas in this section won't seem obvious at first, and it may take a while for the rather complicated data model to sink in. If you find yourself getting confused, try pushing forward to the end; then, read the section a second time through to see if things are clearer.

11.1.1 A problem with the data model (and a solution)

As a first step toward constructing a data model for following users, let's examine a typical case. For instance, consider a user who follows a second user: we could say that, e.g., Calvin is following Hobbes, and Hobbes is followed by Calvin, so that Calvin is the *follower* and Hobbes is *followed*. Using Rails' default pluralization convention, the set of all users following a given user is that user's *followers*, and **`user.followers`** is an array of those users. Unfortunately, the reverse doesn't work: by default, the set of all followed users would be called the *followeds*, which is ungrammatical and clumsy. We could call them *following*, but that's ambiguous: in normal English, a "following" is the set of people following *you*, i.e., your followers—exactly the opposite of the intended meaning. Although we will use "following" as a label, as in "50 following, 75 followers", we'll use "followed users" for the users themselves, with a corresponding **`user.followed_users`** array.[2]

This discussion suggests modeling the followed users as in Figure 11.6, with a **`followed_users`** table and a **`has_many`** association. Since **`user.followed_users`** should be an array of users, each row of the **`followed_users`** table would need to be a user, as identified by the **`followed_id`**, together with the **`follower_id`** to establish the association.[3] In addition, since each row is a user, we would need to include the user's other attributes, including the name, password, etc.



user		
id	name	email
1	Michael Hartl	mhartl@example.com

followed_users			
follower_id	followed_id	name	email
1	2	...	...
1	7	...	...
1	10	...	...
1	8	...	...

Figure 11.6: A naïve implementation of user following.

The problem with the data model in Figure 11.6 is that it is terribly redundant: each row contains not only each followed user's id, but all their other information as well—all of which are *already* in the **`users`** table. Even worse, to model user *followers* we would need a separate, similarly redundant **`followers`** table. Finally, this data model is a maintainability nightmare: each time a user changed (say) his name, we would need to update not just the user's record in the **`users`** table but also *every row containing that user* in both the **`followed_users`** and **`followers`** tables.

The problem here is that we are missing an underlying abstraction. One way to find the proper abstraction is to consider how we might implement the act of *following* in a web application. Recall from Section 7.1.2 that the REST architecture involves *resources* that are created and destroyed. This leads us to ask two questions: When a user follows another user, what is being created? When a user *un*follows another user, what is being destroyed?

Upon reflection, we see that in these cases the application should either create or destroy a *relationship* between two users. A user then **`has_many :relationships`**, and has many **`followed_users`** (or **`followers`**) *through* these relationships. Indeed, Figure 11.6 already contains most of the implementation: since each followed user is uniquely identified by

followed_id, we could convert **followed_users** to a **relationships** table, omit the user details, and use **followed_id** to retrieve the followed user from the **users** table. Moreover, by considering *reverse* relationships, we could use the **follower_id** column to extract an array of user's followers.

To make a **followed_users** array of users, it would be possible to pull out an array of **followed_id** attributes and then find the user for each one. As you might expect, though, Rails has a way to make this procedure more convenient, and the relevant technique is known as **has_many through**. As we will see in Section 11.1.4, Rails allows us to say that a user is following many users *through* the relationships table, using the succinct code

```
has_many :followed_users, through: :relationships, source: :followed
```

This code automatically populates **user.followed_users** with an array of followed users. A diagram of the data model appears in Figure 11.7.

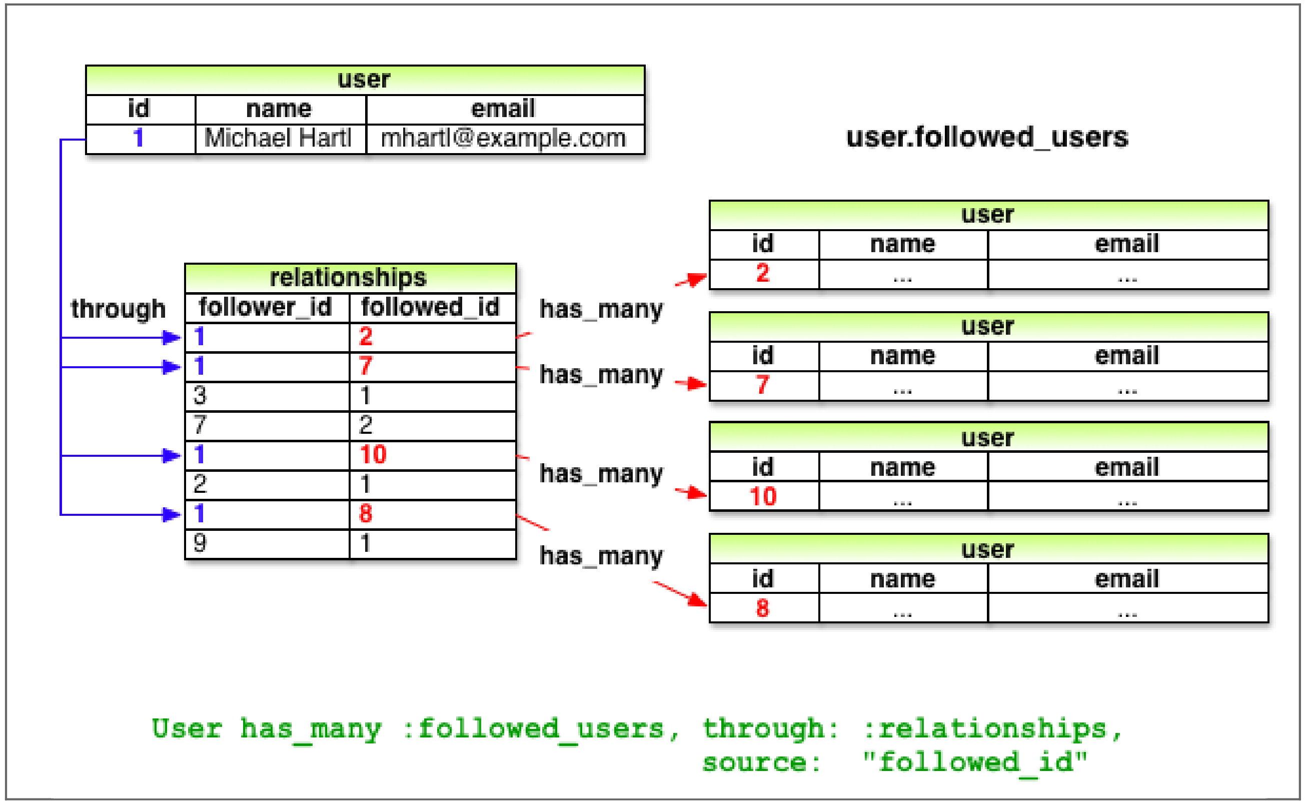


Figure 11.7: A model of followed users through user relationships.

To get started with the implementation, we first generate a Relationship model as follows:

```
$ rails generate model Relationship follower_id:integer followed_id:integer
```

Since we will be finding relationships by `follower_id` and by `followed_id`, we should add an index on each column for efficiency, as shown in Listing 11.1.

Listing 11.1. Adding indices for the `relationships` table.
`db/migrate/[timestamp]_create_relationships.rb`

```
class CreateRelationships < ActiveRecord::Migration
  def change
    create_table :relationships do |t|
      t.integer :follower_id
      t.integer :followed_id

      t.timestamps
    end

    add_index :relationships, :follower_id
    add_index :relationships, :followed_id
    add_index :relationships, [:follower_id, :followed_id], unique: true
  end
end
```

Listing 11.1 also includes a *composite* index that enforces uniqueness of pairs of (**`follower_id`**, **`followed_id`**), so that a user can't follow another user more than once:

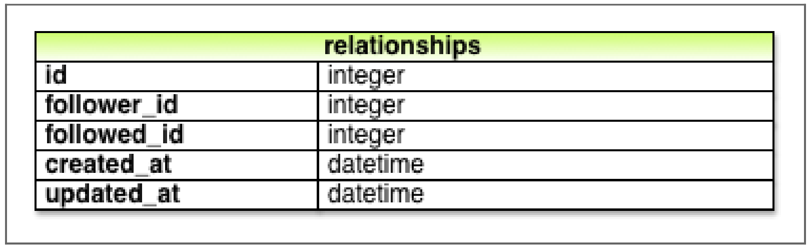
```
add_index :relationships, [:follower_id, :followed_id], unique: true
```

(Compare to the email uniqueness index from Listing 6.22.) As we'll see starting in Section 11.1.4, our user interface won't allow this to happen, but adding a unique index arranges to raise an error if a user tries to create duplicate relationships anyway (using, e.g., a command-line tool such as `curl`). We could also add a uniqueness validation to the Relationship model, but because it is *always* an error to create duplicate relationships, the unique index is sufficient for our purposes.

To create the **`relationships`** table, we migrate the database and prepare the test database as usual:

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

The result is the Relationship data model shown in Figure 11.8.



relationships	
id	integer
follower_id	integer
followed_id	integer
created_at	datetime
updated_at	datetime

Figure 11.8: The Relationship data model.

11.1.2 User/relationship associations

Before implementing followed users and followers, we first need to establish the association between users and relationships. A user **`has_many`** relationships, and—since relationships involve *two* users —a relationship **`belongs_to`** both a follower and a followed user.

As with microposts in Section 10.1.3, we will create new relationships using the user association, with code such as

```
user.relationships.build(followed_id: ...)
```

We start with some tests, shown in Listing 11.2, which make a **`relationship`** variable, checks that it is valid, and ensures that the **`follower_id`** isn't accessible. (If the test for accessible attributes doesn't fail, be sure that your **`application.rb`** has been updated in accordance with Listing 10.6.)

Listing 11.2. Testing Relationship creation and attributes.
`spec/models/relationship_spec.rb`

```
require 'spec_helper'

describe Relationship do

  let(:follower) { FactoryGirl.create(:user) }
  let(:followed) { FactoryGirl.create(:user) }
  let(:relationship) { follower.relationships.build(followed_id: followed.id) }

  subject { relationship }

  it { should be_valid }

  describe "accessible attributes" do
    it "should not allow access to follower_id" do
      expect do
        Relationship.new(follower_id: follower.id)
      end.to raise_error(ActiveModel::MassAssignmentSecurity::Error)
    end
  end
end
```

Note that, unlike the tests for the User and Micropost models, which use `@user` and `@micropost`, respectively, Listing 11.2 uses `let` in preference to an instance variable. The differences rarely matter,[4] but I consider `let` to be cleaner than using an instance variable. We originally used instance variables both because instance variables are important to introduce early and because `let` is a little more advanced.

We should also test the User model for a `relationships` attribute, as shown in Listing 11.3.

Listing 11.3. Testing for the `user.relationships` attribute.
`spec/models/user_spec.rb`

```
require 'spec_helper'
```

```ruby
describe User do
  .
  .
  .
  it { should respond_to(:feed) }
  it { should respond_to(:relationships) }
  .
  .
  .
end
```

At this point, you might expect application code as in Section 10.1.3, and it’s similar, but there is one critical difference: in the case of the Micropost model, we could say

```ruby
class Micropost < ActiveRecord::Base
  belongs_to :user
  .
  .
  .
end
```

and

```ruby
class User < ActiveRecord::Base
  has_many :microposts
  .
  .
  .
end
```

because the `microposts` table has a `user_id` attribute to identify the user (Section 10.1.1). An id used in this manner to connect two database tables is known as a *foreign key*, and when the foreign key for a User model object is `user_id`, Rails infers the association automatically: by default, Rails expects a foreign key of the form **`<class>_id`**, where **`<class>`** is the lower-case version of the class name.[5] In the present case, although we are still dealing with users, they are now identified with the foreign key **`follower_id`**, so we have to tell that to Rails, as shown in Listing 11.4.[6]

Listing 11.4. Implementing the user/relationships **`has_many`** association.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :microposts, dependent: :destroy
  has_many :relationships, foreign_key: "follower_id", dependent: :destroy
  .
  .
  .
end
```

(Since destroying a user should also destroy that user's relationships, we've gone ahead and added **`dependent: :destroy`** to the association; writing a test for this is left as an exercise (Section 11.5).)

As with the Micropost model, the Relationship model has a **`belongs_to`** relationship with users; in this case, a relationship object belongs to both a **`follower`** and a **`followed`** user, which we test for in Listing 11.5.

Listing 11.5. Testing the user/relationships **`belongs_to`** association.
`spec/models/relationship_spec.rb`

```
describe Relationship do
  .
  .
  .
  describe "follower methods" do
    it { should respond_to(:follower) }
    it { should respond_to(:followed) }
    its(:follower) { should == follower }
    its(:followed) { should == followed }
  end
end
```

To write the application code, we define the **`belongs_to`** relationship as usual. Rails infers the names of the foreign keys from the corresponding symbols (i.e., **`follower_id`** from **`:follower`**, and **`followed_id`** from **`:followed`**), but since there is neither a Followed nor a Follower model we need to supply the class name **`User`**. The result is shown in Listing 11.6. Note that, unlike the default generate Relationship model, in this case only the **`followed_id`** is accessible.

Listing 11.6. Adding the **`belongs_to`** associations to the Relationship model.
`app/models/relationship.rb`

```
class Relationship < ActiveRecord::Base
  attr_accessible :followed_id

  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end
```

The **`followed`** association isn't actually needed until Section 11.1.5, but the parallel follower/followed structure is clearer if we implement them both at the same time.

At this point, the tests in Listing 11.2 and Listing 11.3 should pass.

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

11.1.3 Validations

Before moving on, we'll add a couple of Relationship model validations for completeness. The tests (Listing 11.7) and application code (Listing 11.8) are straightforward.

Listing 11.7. Testing the Relationship model validations.
spec/models/relationship_spec.rb

```
describe Relationship do
  .
  .
  .
  describe "when followed id is not present" do
    before { relationship.followed_id = nil }
    it { should_not be_valid }
  end

  describe "when follower id is not present" do
    before { relationship.follower_id = nil }
    it { should_not be_valid }
  end
end
```

Listing 11.8. Adding the Relationship model validations.
app/models/relationship.rb

```
class Relationship < ActiveRecord::Base
  attr_accessible :followed_id

  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"

  validates :follower_id, presence: true
  validates :followed_id, presence: true
end
```

11.1.4 Followed users

We come now to the heart of the Relationship associations: **`followed_users`** and **`followers`**. We start with **`followed_users`**, as shown Listing 11.9.

Listing 11.9. A test for the **`user.followed_users`** attribute.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:relationships) }
  it { should respond_to(:followed_users) }
  .
  .
  .
end
```

The implementation uses **`has_many through`** for the first time: a user has many following *through* relationships, as illustrated in Figure 11.7. By default, in a **`has_many through`** association Rails looks for a foreign key corresponding to the singular version of the association; in other words, code like

```
has_many :followeds, through: :relationships
```

would assemble an array using the **`followed_id`** in the **`relationships`** table. But, as noted in Section 11.1.1, **`user.followeds`** is rather awkward; far more natural is to use “followed users” as a plural of “followed”, and write instead **`user.followed_users`** for the array of followed users.

Naturally, Rails allows us to override the default, in this case using the `:source` parameter (Listing 11.10), which explicitly tells Rails that the source of the `followed_users` array is the set of `followed` ids.

Listing 11.10. Adding the User model `followed_users` association.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :microposts, dependent: :destroy
  has_many :relationships, foreign_key: "follower_id", dependent: :destroy
  has_many :followed_users, through: :relationships, source: :followed
  .
  .
  .
end
```

To create a following relationship, we'll introduce a `follow!` utility method so that we can write `user.follow!(other_user)`. (This `follow!` method should always work, so, as with `create!` and `save!`, we indicate with an exclamation point that an exception will be raised on failure.) We'll also add an associated `following?` boolean method to test if one user is following another.[7] The tests in Listing 11.11 show how we expect these methods to be used in practice.

Listing 11.11. Tests for some "following" utility methods.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:followed_users) }
  it { should respond_to(:following?) }
  it { should respond_to(:follow!) }
  .
  .
  .
  describe "following" do
    let(:other_user) { FactoryGirl.create(:user) }
    before do
      @user.save
      @user.follow!(other_user)
    end

    it { should be_following(other_user) }
    its(:followed_users) { should include(other_user) }
  end
end
```

In the application code, the `following?` method takes in a user, called `other_user`, and checks to see if a followed user with that id exists in the database; the `follow!` method calls `create!` through the `relationships` association to create the following relationship. The results appear in Listing 11.12.

Listing 11.12. The `following?` and `follow!` utility methods.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def feed
    .
    .
    .
  end

  def following?(other_user)
    relationships.find_by_followed_id(other_user.id)
  end
  def follow!(other_user)
    relationships.create!(followed_id: other_user.id)
  end
  .
  .
  .
end
```

Note that in Listing 11.12 we have omitted the user itself, writing just

```
relationships.create!(...)
```

instead of the equivalent code

```
self.relationships.create!(...)
```

Whether to include the explicit **`self`** is largely a matter of taste.

Of course, users should be able to unfollow other users as well as follow them, which leads to the somewhat predictable **`unfollow!`** method, as shown in Listing 11.13.[8]

Listing 11.13. A test for unfollowing a user.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:follow!) }
  it { should respond_to(:unfollow!) }
```

```
    .
    .
    .
    describe "following" do
      .
      .
      .
      describe "and unfollowing" do
        before { @user.unfollow!(other_user) }

        it { should_not be_following(other_user) }
        its(:followed_users) { should_not include(other_user) }
      end
    end
  end
```

The code for **`unfollow!`** is straightforward: just find the relationship by followed id and destroy it (Listing 11.14).

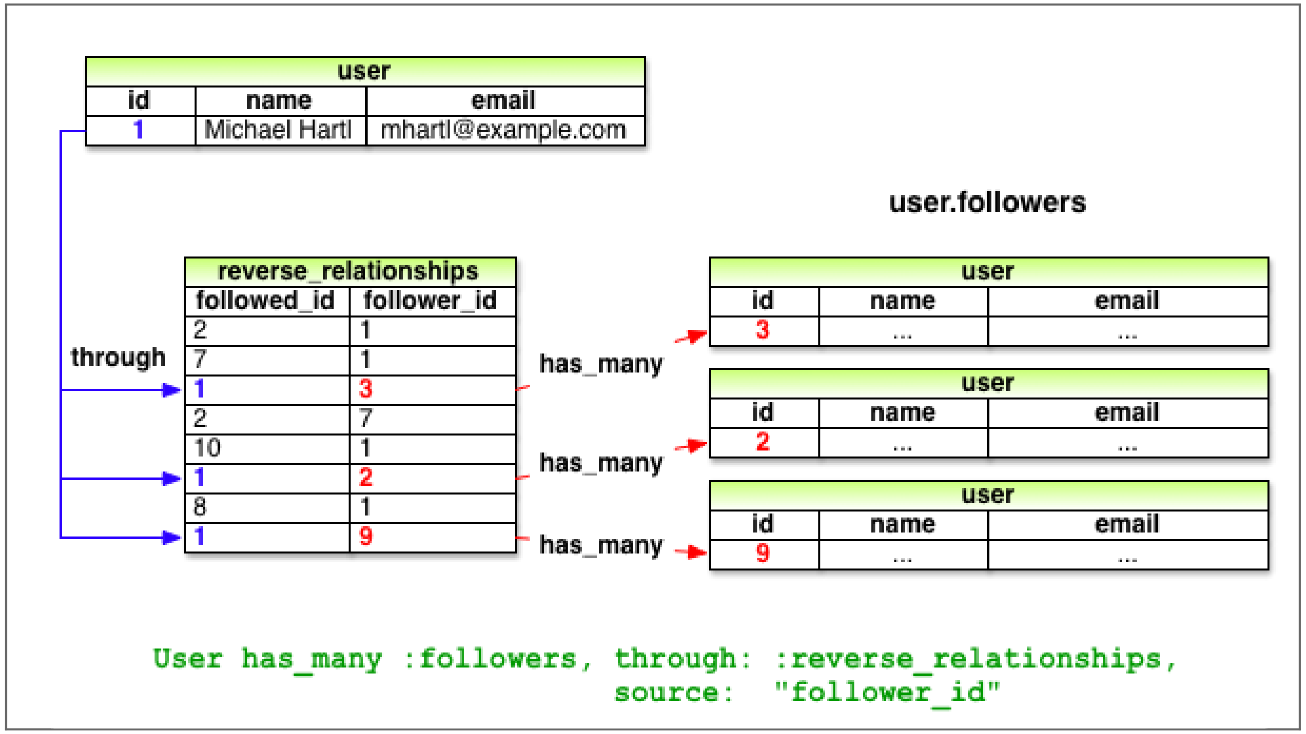
Listing 11.14. Unfollowing a user by destroying a user relationship.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def following?(other_user)
    relationships.find_by_followed_id(other_user.id)
  end

  def follow!(other_user)
    relationships.create!(followed_id: other_user.id)
  end

  def unfollow!(other_user)
    relationships.find_by_followed_id(other_user.id).destroy
  end
  .
  .
  .
```

```
end
```

11.1.5 Followers

The final piece of the relationships puzzle is to add a `user.followers` method to go with `user.followed_users`. You may have noticed from Figure 11.7 that all the information needed to extract an array of followers is already present in the `relationships` table. Indeed, the technique is exactly the same as for user following, with the roles of `follower_id` and `followed_id` reversed. This suggests that, if we could somehow arrange for a `reverse_relationships` table with those two columns reversed (Figure 11.9), we could implement `user.followers` with little effort.



user

id	name	email
1	Michael Hartl	mhartl@example.com

reverse_relationships

followed_id	follower_id
2	1
7	1
1	3
2	7
10	1
1	2
8	1
1	9

user

id	name	email
3	...	...

user

id	name	email
2	...	...

user

id	name	email
9	...	...

```
User has_many :followers, through: :reverse_relationships,
                          source:  "follower_id"
```

Figure 11.9: A model for user followers using a reverse Relationship model.

We begin with the tests, having faith that the magic of Rails will come to the rescue (Listing 11.15).

Listing 11.15. Testing for reverse relationships.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  it { should respond_to(:relationships) }
  it { should respond_to(:followed_users) }
  it { should respond_to(:reverse_relationships) }
  it { should respond_to(:followers) }
  .
  .
  .

  describe "following" do
    .
    .
    .
    it { should be_following(other_user) }
    its(:followed_users) { should include(other_user) }

    describe "followed user" do
      subject { other_user }
      its(:followers) { should include(@user) }
    end
    .
    .
    .
  end
end
```

Notice how we switch subjects using the `subject` method, replacing `@user` with `other_user`, allowing us to test the follower relationship in a natural way:

```
subject { other_user }
its(:followers) { should include(@user) }
```

As you probably suspect, we will not be making a whole database table just to hold reverse relationships. Instead, we will exploit the underlying symmetry between followers and followed users to simulate a **`reverse_relationships`** table by passing **`followed_id`** as the primary key. In other words, where the **`relationships`** association uses the **`follower_id`** foreign key,

```
has_many :relationships, foreign_key: "follower_id"
```

the **`reverse_relationships`** association uses **`followed_id`**:

```
has_many :reverse_relationships, foreign_key: "followed_id"
```

The **`followers`** association then gets built through the reverse relationships, as shown in Listing 11.16.

Listing 11.16. Implementing **`user.followers`** using reverse relationships.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :reverse_relationships, foreign_key: "followed_id",
                                   class_name:  "Relationship",
                                   dependent:   :destroy
  has_many :followers, through: :reverse_relationships, source: :follower
  .
  .
  .
end
```

(As with Listing 11.4, the test for **`dependent :destroy`** is left as an exercise (Section 11.5).) Note that we actually have to include the *class* name for this association, i.e.,

```
has_many :reverse_relationships, foreign_key: "followed_id",
                                 class_name: "Relationship"
```

because otherwise Rails would look for a **`ReverseRelationship`** class, which doesn't exist.

It's also worth noting that we could actually omit the **`:source`** key in this case, using simply

```
has_many :followers, through: :reverse_relationships
```

since, in the case of a **`:followers`** attribute, Rails will singularize "followers" and automatically look for the foreign key **`follower_id`** in this case. I've kept the **`:source`** key to emphasize the parallel structure with the **`has_many :followed_users`** association, but you are free to leave it off.

With the code in Listing 11.16, the following/follower associations are complete, and all the tests should pass:

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

This section has placed rather heavy demands on your data modeling skills, and it's fine if it takes a while to soak in. In fact, one of the best ways to understand the associations is to use them in the web interface, as seen in the next section.

11.2 A web interface for following users

In the introduction to this chapter, we saw a preview of the page flow for user following. In this section, we will implement the basic interface and following/unfollowing functionality shown in those mockups. We will also make separate pages to show the user following and followers arrays. In Section 11.3, we'll complete our sample application by adding the user's status feed.

11.2.1 Sample following data

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

When we last left the sample data populator in Listing 10.23, it was getting rather cluttered, so we begin by defining separate methods to make users and microposts, and then add sample relationship data using a new `make_relationships` method. The results are shown in Listing 11.17.

Listing 11.17. Adding following/follower relationships to the sample data.
`lib/tasks/sample_data.rake`

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    make_users
    make_microposts
    make_relationships
  end
end

def make_users
  admin = User.create!(name:     "Example User",
                       email:    "example@railstutorial.org",
                       password: "foobar",
                       password_confirmation: "foobar")
  admin.toggle!(:admin)
  99.times do |n|
    name  = Faker::Name.name
    email = "example-#{n+1}@railstutorial.org"
    password  = "password"
    User.create!(name:     name,
                 email:    email,
                 password: password,
                 password_confirmation: password)
  end
end

def make_microposts
  users = User.all(limit: 6)
  50.times do
    content = Faker::Lorem.sentence(5)
    users.each { |user| user.microposts.create!(content: content) }
  end
end

def make_relationships
  users = User.all
  user  = users.first
  followed_users = users[2..50]
  followers      = users[3..40]
  followed_users.each { |followed| user.follow!(followed) }
  followers.each      { |follower| follower.follow!(user) }
end
```

Here the sample relationships are created using the code

```
  def make_relationships
    users = User.all
    user  = users.first
    followed_users = users[2..50]
    followers      = users[3..40]
    followed_users.each { |followed| user.follow!(followed) }
    followers.each      { |follower| follower.follow!(user) }
  end
```

We somewhat arbitrarily arrange for the first user to follow users 3 through 51, and then have users 4 through 41 follow that user back. The resulting relationships will be sufficient for developing the application interface.

To execute the code in Listing 11.17, populate the database as usual:

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

11.2.2 Stats and a follow form

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

As noted in Section 11.1.1, the word "following" is ambiguous as an attribute (where **`user.following`** could reasonably mean either the followed users or the user's followers), it makes sense as a label, as in "50 following". Indeed, this is the label used by Twitter itself, a usage adopted in the mockups starting in Figure 11.1 and shown in close-up in Figure 11.10.

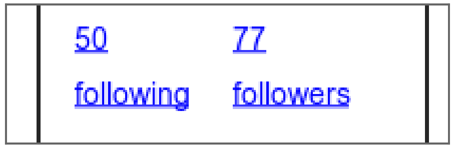


Figure 11.10: A mockup of the stats partial.

The stats in Figure 11.10 consist of the number of users the current user is following and the number of followers, each of which should be a link to its respective dedicated display page. In Chapter 5, we stubbed out such links with the dummy text `'#'`, but that was before we had much experience with routes. This time, although we'll defer the actual pages to Section 11.2.3, we'll make the routes now, as seen in Listing 11.18. This code uses the `:member` method inside a `resources` *block*, which we haven't seen before, but see if you can guess what it does. (*Note*: The code in Listing 11.18 should *replace* the `resources :users`.)

Listing 11.18. Adding `following` and `followers` actions to the Users controller.
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  resources :users do
    member do
      get :following, :followers
    end
  end
  .
  .
  .
end
```

You might suspect that the URIs will look like /users/1/following and /users/1/followers, and that is exactly what the code in Listing 11.18 does. Since both pages will be showing data, we use `get` to arrange for the URIs to respond to `GET` requests (as required by the REST convention for such pages), and the **member** method means that the routes respond to URIs containing the user id. The other possibility, **collection**, works without the id, so that

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

would respond to the URI /users/tigers (presumably to display all the tigers in our application). For more details on such routing options, see the Rails Guides article on "Rails Routing from the Outside In". A table of the routes generated by Listing 11.18 appears in Table 11.1; note the named routes for the followed user and followers pages, which we'll put to use shortly. The unfortunate hybrid usage in the "following" route is forced by our choice to use the unambiguous "followed users" terminology along with the "following" usage from Twitter. Since the former would lead to routes of the form **followed_users_user_path**, which sounds strange, we've opted for the latter in the context of Table 11.1, yielding **following_user_path**.

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

Table 11.1: RESTful routes provided by the custom rules in resource in Listing 11.18.

With the routes defined, we are now in a position to make tests for the stats partial. (We could have written the tests first, but the named routes would have been hard to motivate without the updated routes file.) The stats partial will appear on both the profile page and the Home page; Listing 11.19 opts to test it on the latter.

Listing 11.19. Testing the following/follower statistics on the Home page.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "StaticPages" do
  .
  .
  .
  describe "Home page" do
    .
    .
    .
    describe "for signed-in users" do
      let(:user) { FactoryGirl.create(:user) }
      before do
        FactoryGirl.create(:micropost, user: user, content: "Lorem")
        FactoryGirl.create(:micropost, user: user, content: "Ipsum")
        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

      describe "follower/following counts" do
        let(:other_user) { FactoryGirl.create(:user) }
        before do
          other_user.follow!(user)
          visit root_path
        end

        it { should have_link("0 following", href: following_user_path(user)) }
        it { should have_link("1 followers", href: followers_user_path(user)) }
      end
    end
    end
    .
    .
    .
  end
```

The core of this test is the expectation that the following and follower counts appear on the page, together with the right URIs:

```
it { should have_link("0 following", href: following_user_path(user)) }
it { should have_link("1 followers", href: followers_user_path(user)) }
```

Here we have used the named routes shown in Table 11.1 to verify that the links have the right addresses. Also note that in this case the word "followers" is acting as a *label*, so we keep it plural even when there is only one follower.

The application code for the stats partial is just a couple of links inside a div, as shown in Listing 11.20.

Listing 11.20. A partial for displaying follower stats.
`app/views/shared/_stats.html.erb`

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

Since we will be including the stats on both the user show pages and the home page, the first line of Listing 11.20 picks the right one using

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

As discussed in Box 8.2, this does nothing when `@user` is not `nil` (as on a profile page), but when it is (as on the Home page) it sets `@user` to the current user.

Note also that the following/follower counts are calculated through the associations using

```
@user.followed_users.count
```

and

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

Compare these to the microposts count from Listing 10.20, where we wrote

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

to count the microposts.

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

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

This is for the benefit of the Ajax implementation in Section 11.2.5, which accesses elements on the page using their unique ids.

With the partial in hand, including the stats on the Home page is easy, as shown in Listing 11.21. (This also gets the test in Listing 11.19 to pass.)

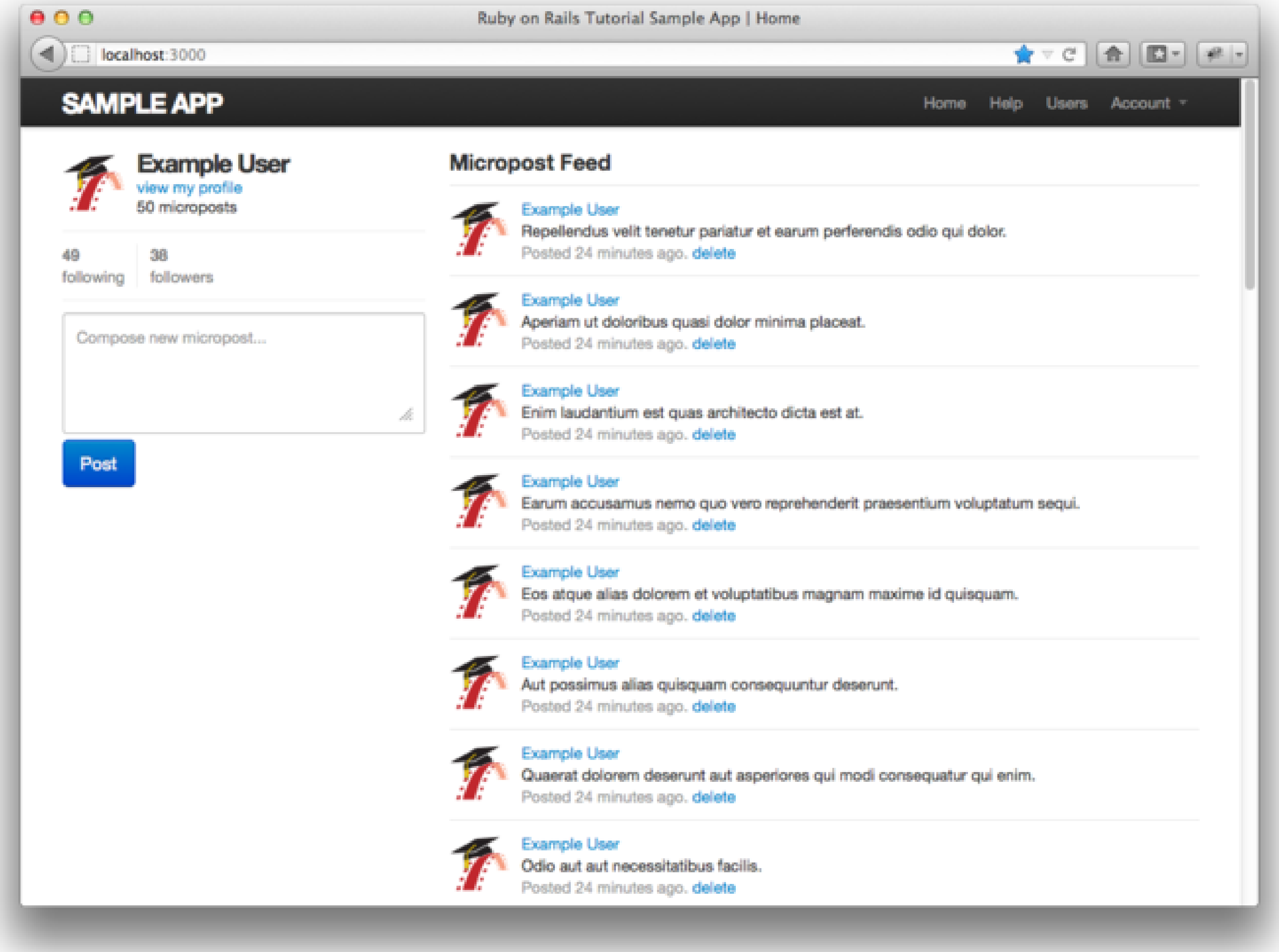
Listing 11.21. Adding follower stats to the Home page.
`app/views/static_pages/home.html.erb`

```
<% if signed_in? %>
      .
      .
      .
      <section>
        <%= render 'shared/user_info' %>
      </section>
      <section>
        <%= render 'shared/stats' %>
      </section>
      <section>
        <%= render 'shared/micropost_form' %>
      </section>
      .
      .
      .
<% else %>
   .
   .
   .
<% end %>
```

To style the stats, we'll add some SCSS, as shown in Listing 11.22 (which contains all the stylesheet code needed in this chapter). The result appears in Figure 11.11.

Listing 11.22. SCSS for the Home page sidebar.
`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* sidebar */
.
.
.
.stats {
  overflow: auto;
  a {
    float: left;
    padding: 0 10px;
    border-left: 1px solid $grayLighter;
    color: gray;
    &:first-child {
      padding-left: 0;
      border: 0;
    }
    &:hover {
      text-decoration: none;
      color: $blue;
    }
  }
  strong {
    display: block;
  }
}

.user_avatars {
  overflow: auto;
  margin-top: 10px;
  .gravatar {
    margin: 1px 1px;
    }
  }
  .
  .
  .
```



Ruby on Rails Tutorial Sample App | Home

localhost:3000

SAMPLE APP — Home Help Users Account

Example User
view my profile
50 microposts

49 following | 38 followers

Compose new micropost...

Post

Micropost Feed

Example User
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted 24 minutes ago. delete

Example User
Aperiam ut doloribus quasi dolor minima placeat.
Posted 24 minutes ago. delete

Example User
Enim laudantium est quas architecto dicta est at.
Posted 24 minutes ago. delete

Example User
Earum accusamus nemo quo vero reprehenderit praesentium voluptatum sequi.
Posted 24 minutes ago. delete

Example User
Eos atque alias dolorem et voluptatibus magnam maxime id quisquam.
Posted 24 minutes ago. delete

Example User
Aut possimus alias quisquam consequuntur deserunt.
Posted 24 minutes ago. delete

Example User
Quaerat dolorem deserunt aut asperiores qui modi consequatur qui enim.
Posted 24 minutes ago. delete

Example User
Odio aut aut necessitatibus facilis.
Posted 24 minutes ago. delete

Figure 11.11: The Home page ([/]()) with follow stats. [(full size)]()

We'll render the stats partial on the profile page in a moment, but first let's make a partial for the follow/unfollow button, as shown in Listing 11.23.

Listing 11.23. A partial for a follow/unfollow form.
app/views/users/_follow_form.html.erb

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

This does nothing but defer the real work to `follow` and `unfollow` partials, which need a new routes file with rules for the Relationships resource, which follows the Microposts resource example (Listing 10.25), as seen in Listing 11.24.

Listing 11.24. Adding the routes for user relationships.
config/routes.rb

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

The follow/unfollow partials themselves are shown in Listing 11.25 and Listing 11.26.

Listing 11.25. A form for following a user.
`app/views/users/_follow.html.erb`

```
<%= form_for(current_user.relationships.build(followed_id: @user.id)) do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <%= f.submit "Follow", class: "btn btn-large btn-primary" %>
<% end %>
```

Listing 11.26. A form for unfollowing a user.
`app/views/users/_unfollow.html.erb`

```
<%= form_for(current_user.relationships.find_by_followed_id(@user),
             html: { method: :delete }) do |f| %>
  <%= f.submit "Unfollow", class: "btn btn-large" %>
<% end %>
```

These two forms both use `form_for` to manipulate a Relationship model object; the main difference between the two is that Listing 11.25 builds a *new* relationship, whereas Listing 11.26 finds the existing relationship. Naturally, the former sends a `POST` request to the Relationships controller to `create` a relationship, while the latter sends a `DELETE` request to `destroy` a relationship. (We'll write these actions in Section 11.2.4.) Finally, you'll note that the follow/unfollow form doesn't have any content other than the button, but it still needs to send the `followed_id`, which we accomplish with the `hidden_field` method, which produces HTML of the form

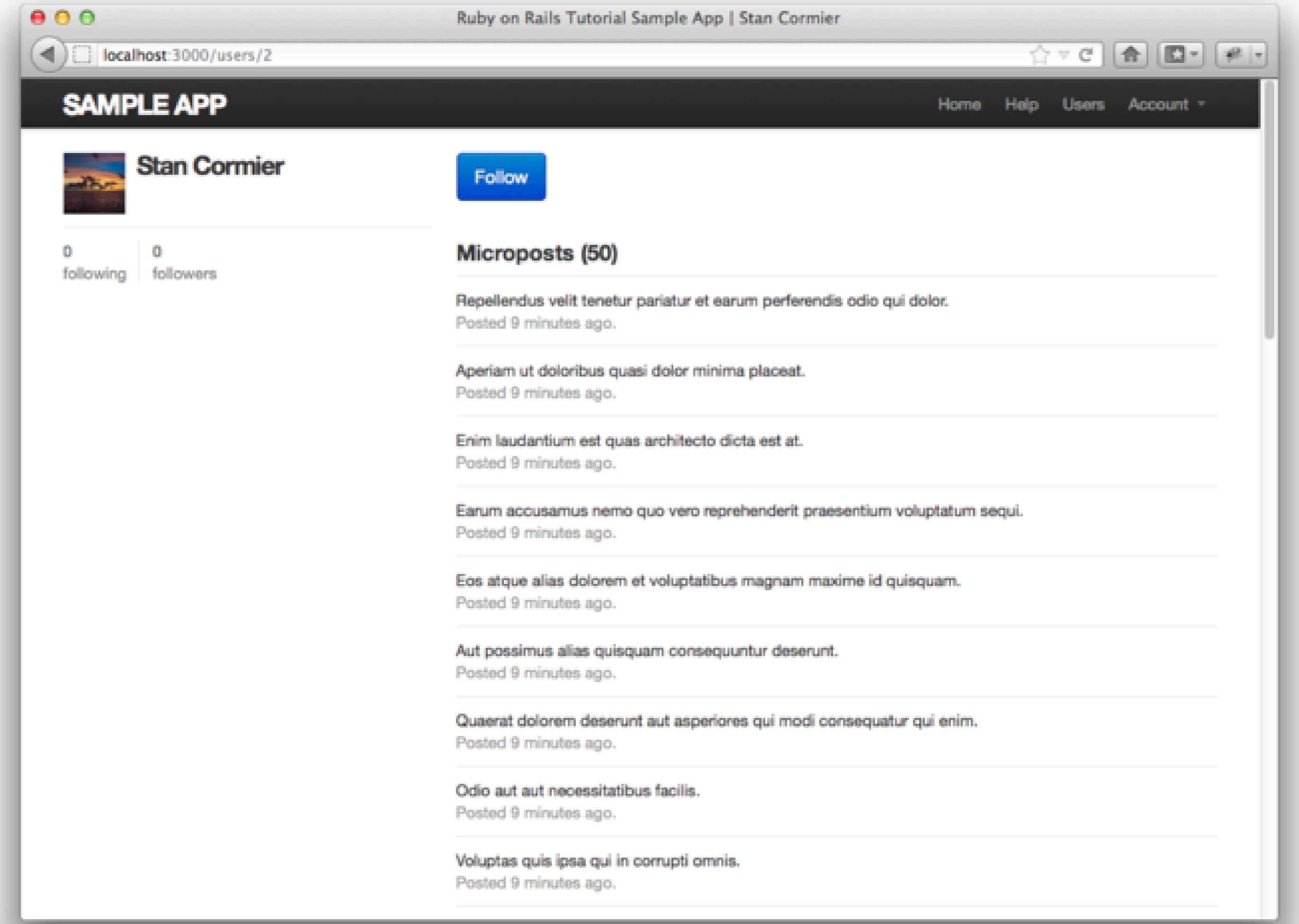
```
<input id="followed_relationship_followed_id"
name="followed_relationship[followed_id]"
type="hidden" value="3" />
```

The “hidden” `input` tag puts the relevant information on the page without displaying it in the browser.

We can now include the follow form and the following statistics on the user profile page simply by rendering the partials, as shown in Listing 11.27. Profiles with follow and unfollow buttons, respectively, appear in Figure 11.12 and Figure 11.13.

Listing 11.27. Adding the follow form and follower stats to the user profile page.
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="span4">
    <section>
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
    <section>
      <%= render 'shared/stats' %>
    </section>
  </aside>
  <div class="span8">
    <%= render 'follow_form' if signed_in? %>
    .
    .
    .
  </div>
</div>
```


Ruby on Rails Tutorial Sample App | Stan Cormier

localhost:3000/users/2

SAMPLE APP

Home Help Users Account

Stan Cormier

Follow

0 following 0 followers

Microposts (50)

Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted 9 minutes ago.

Aperiam ut doloribus quasi dolor minima placeat.
Posted 9 minutes ago.

Enim laudantium est quas architecto dicta est at.
Posted 9 minutes ago.

Earum accusamus nemo quo vero reprehenderit praesentium voluptatum sequi.
Posted 9 minutes ago.

Eos atque alias dolorem et voluptatibus magnam maxime id quisquam.
Posted 9 minutes ago.

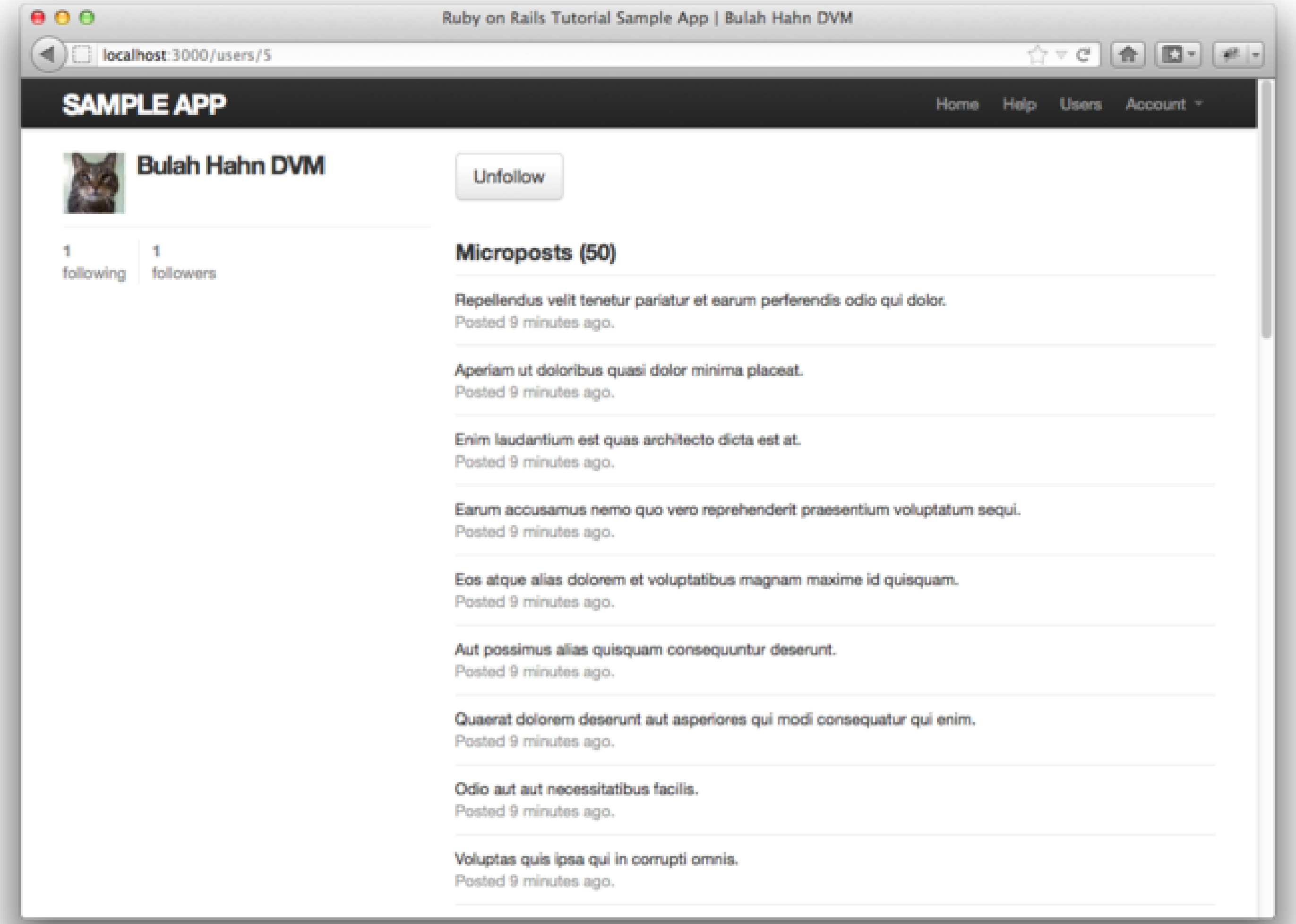
Aut possimus alias quisquam consequuntur deserunt.
Posted 9 minutes ago.

Quaerat dolorem deserunt aut asperiores qui modi consequatur qui enim.
Posted 9 minutes ago.

Odio aut aut necessitatibus facilis.
Posted 9 minutes ago.

Voluptas quis ipsa qui in corrupti omnis.
Posted 9 minutes ago.

Figure 11.12: A user profile with a follow button (/users/2). (full size)


Ruby on Rails Tutorial Sample App | Bulah Hahn DVM

localhost:3000/users/5

SAMPLE APP

Home Help Users Account

Bulah Hahn DVM

Unfollow

1 following | 1 followers

Microposts (50)

Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted 9 minutes ago.

Aperiam ut doloribus quasi dolor minima placeat.
Posted 9 minutes ago.

Enim laudantium est quas architecto dicta est at.
Posted 9 minutes ago.

Earum accusamus nemo quo vero reprehenderit praesentium voluptatum sequi.
Posted 9 minutes ago.

Eos atque alias dolorem et voluptatibus magnam maxime id quisquam.
Posted 9 minutes ago.

Aut possimus alias quisquam consequuntur deserunt.
Posted 9 minutes ago.

Quaerat dolorem deserunt aut asperiores qui modi consequatur qui enim.
Posted 9 minutes ago.

Odio aut aut necessitatibus facilis.
Posted 9 minutes ago.

Voluptas quis ipsa qui in corrupti omnis.
Posted 9 minutes ago.

Figure 11.13: A user profile with an unfollow button (/users/6). (full size)

We'll get these buttons working soon enough—in fact, we'll do it two ways, the standard way (Section 11.2.4) and using Ajax (Section 11.2.5)—but first we'll finish the HTML interface by making the following and followers pages.

11.2.3 Following and followers pages

Pages to display followed users and followers will resemble a hybrid of the user profile page and the user index page (Section 9.3.1), with a sidebar of user information (including the following stats) and a list of users. In addition, we’ll include a raster of user profile image links in the sidebar. Mockups matching these requirements appear in Figure 11.14 (following) and Figure 11.15 (followers).

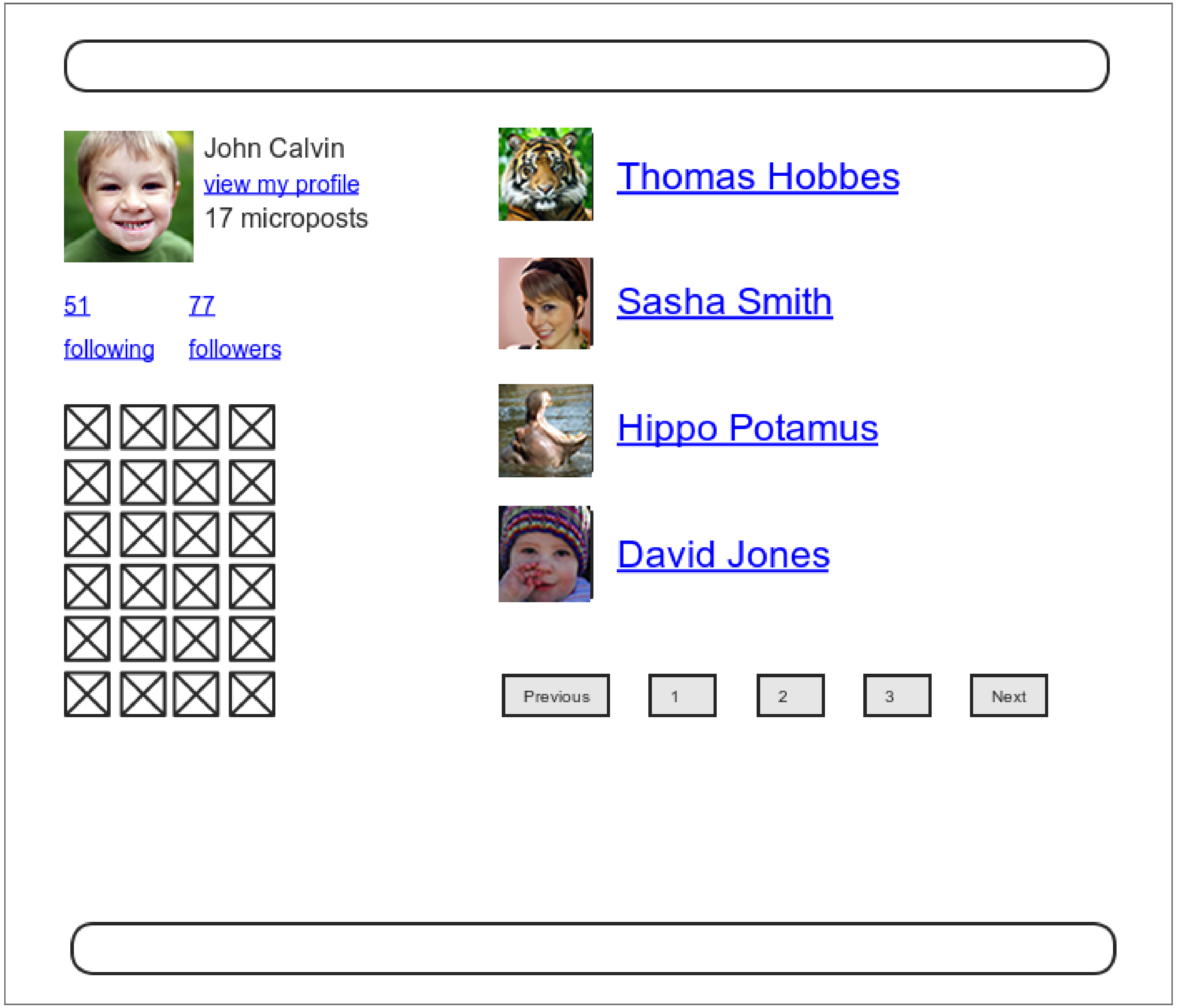


Figure 11.14: A mockup of the user following page. (full size)

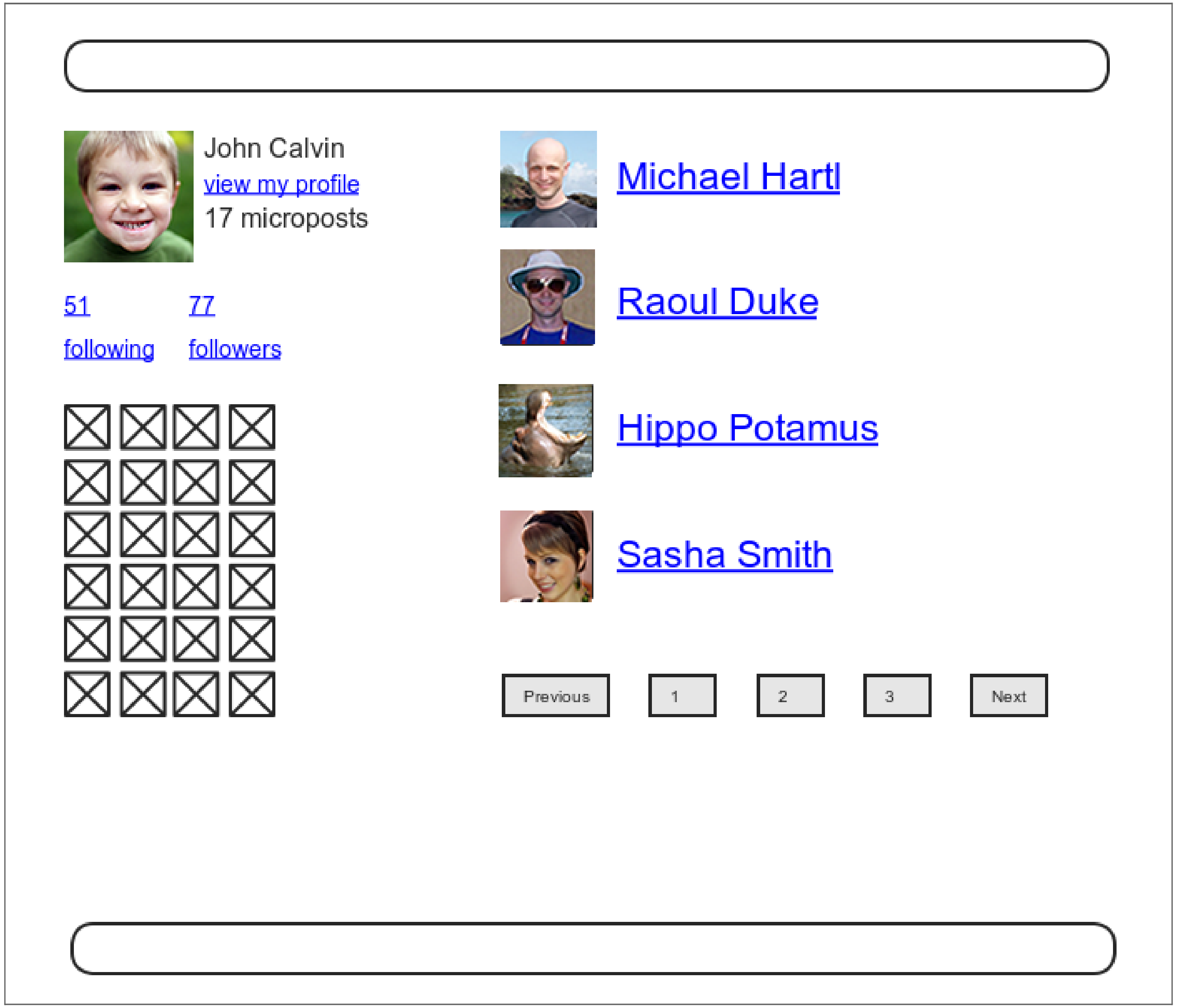


Figure 11.15: A mockup of the user followers page. (full size)

Our first step is to get the following and followers links to work. We'll follow Twitter's lead and have both pages to require user signin, as tested in Listing 11.28. For signed-in users, the pages should have links for following and followers, respectively, as tested in Listing 11.29.

Listing 11.28. Tests for the authorization of the following and followers pages.
`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 Users controller" do
        .
        .
        .
        describe "visiting the following page" do
          before { visit following_user_path(user) }
          it { should have_selector('title', text: 'Sign in') }
        end

        describe "visiting the followers page" do
          before { visit followers_user_path(user) }
          it { should have_selector('title', text: 'Sign in') }
        end
      end
      .
      .
      .
    end
    .
    .
    .
  end
  .
  .
  .
```

```
end
```

Listing 11.29. Test for the `followed_users` and `followers` pages.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "following/followers" do
    let(:user) { FactoryGirl.create(:user) }
    let(:other_user) { FactoryGirl.create(:user) }
    before { user.follow!(other_user) }

    describe "followed users" do
      before do
        sign_in user
        visit following_user_path(user)
      end

      it { should have_selector('title', text: full_title('Following')) }
      it { should have_selector('h3', text: 'Following') }
      it { should have_link(other_user.name, href: user_path(other_user)) }
    end

    describe "followers" do
      before do
        sign_in other_user
        visit followers_user_path(other_user)
      end

      it { should have_selector('title', text: full_title('Followers')) }
      it { should have_selector('h3', text: 'Followers') }
      it { should have_link(user.name, href: user_path(user)) }
    end
  end
end
```

The only tricky part of the implementation is realizing that we need to add two new actions to the Users controller; based on the routes defined in Listing 11.18, we need to call them `following` and `followers`. Each action needs to set a title, find the user, retrieve either `@user.followed_users` or `@user.followers` (in paginated form), and then render the page. The result appears in Listing 11.30.

Listing 11.30. The `following` and `followers` actions.
`app/controllers/users_controller.rb`

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

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

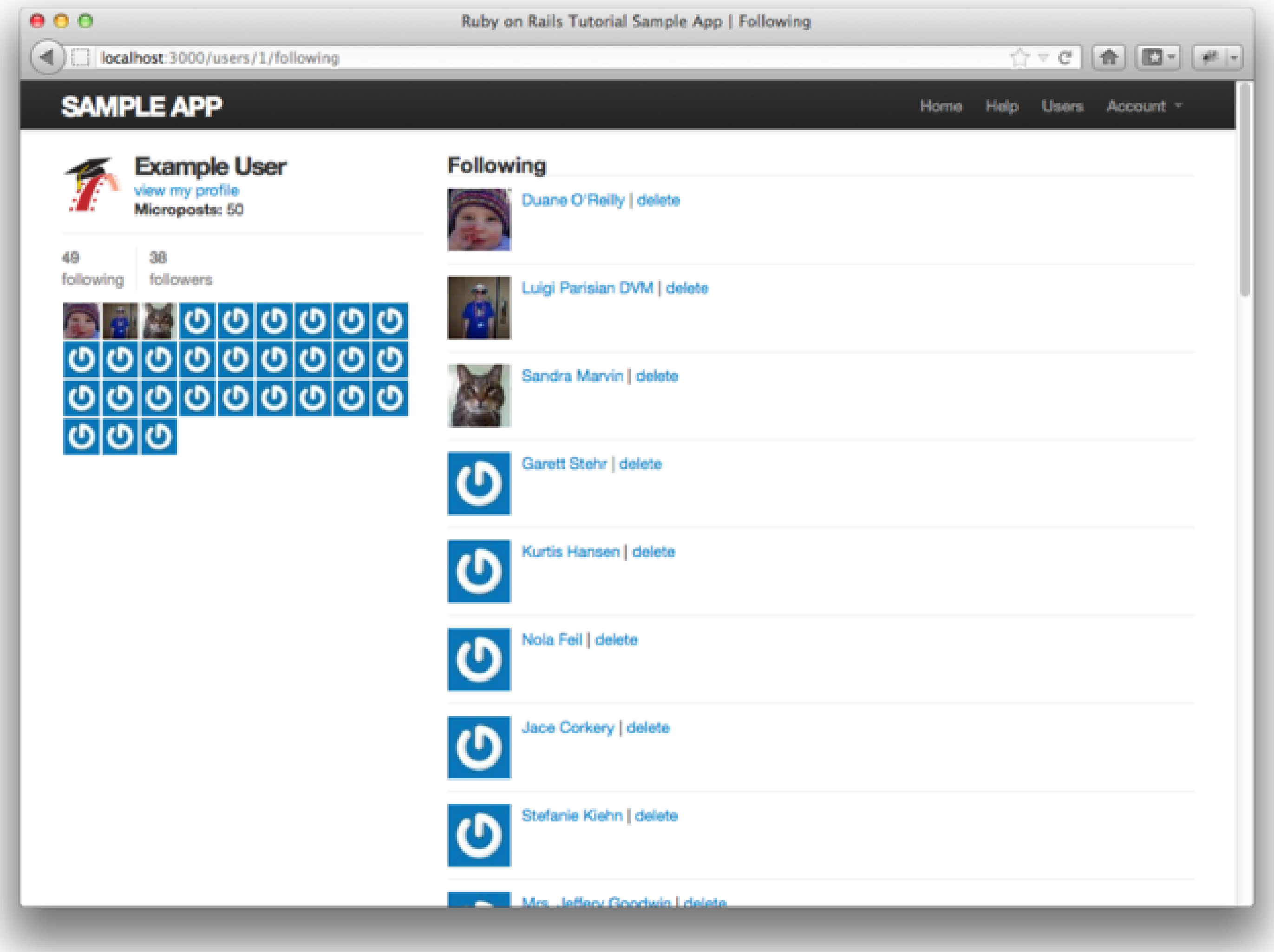
Note here that both actions make an *explicit* call to `render`, in this case rendering a view called `show_follow`, which we must create. The reason for the common view is that the ERb is nearly identical for the two cases, and Listing 11.31 covers them both.

Listing 11.31. The `show_follow` view used to render following and followers.

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

```
<% provide(:title, @title) %>
<div class="row">
  <aside class="span4">
    <section>
      <%= gravatar_for @user %>
      <h1><%= @user.name %></h1>
      <span><%= link_to "view my profile", @user %></span>
      <span><b>Microposts:</b> <%= @user.microposts.count %></span>
    </section>
    <section>
      <%= render 'shared/stats' %>
      <% if @users.any? %>
        <div class="user_avatars">
          <% @users.each do |user| %>
            <%= link_to gravatar_for(user, size: 30), user %>
          <% end %>
        </div>
      <% end %>
    </section>
  </aside>
  <div class="span8">
    <h3><%= @title %></h3>
    <% if @users.any? %>
      <ul class="users">
        <%= render @users %>
      </ul>
      <%= will_paginate %>
    <% end %>
  </div>
</div>
```

With that, the tests should now be passing, and the pages should render as shown in Figure 11.16 (following) and Figure 11.17 (followers).


Ruby on Rails Tutorial Sample App | Following

localhost:3000/users/1/following

SAMPLE APP

Home Help Users Account

Example User
view my profile
Microposts: 50

49 following 38 followers

Following

Duane O'Reilly | delete

Luigi Parisian DVM | delete

Sandra Marvin | delete

Garett Stehr | delete

Kurtis Hansen | delete

Nola Feil | delete

Jace Corkery | delete

Stefanie Kiehn | delete

Mrs. Jeffery Goodwin | delete

Figure 11.16: Showing the users being followed by the current user. (full size)

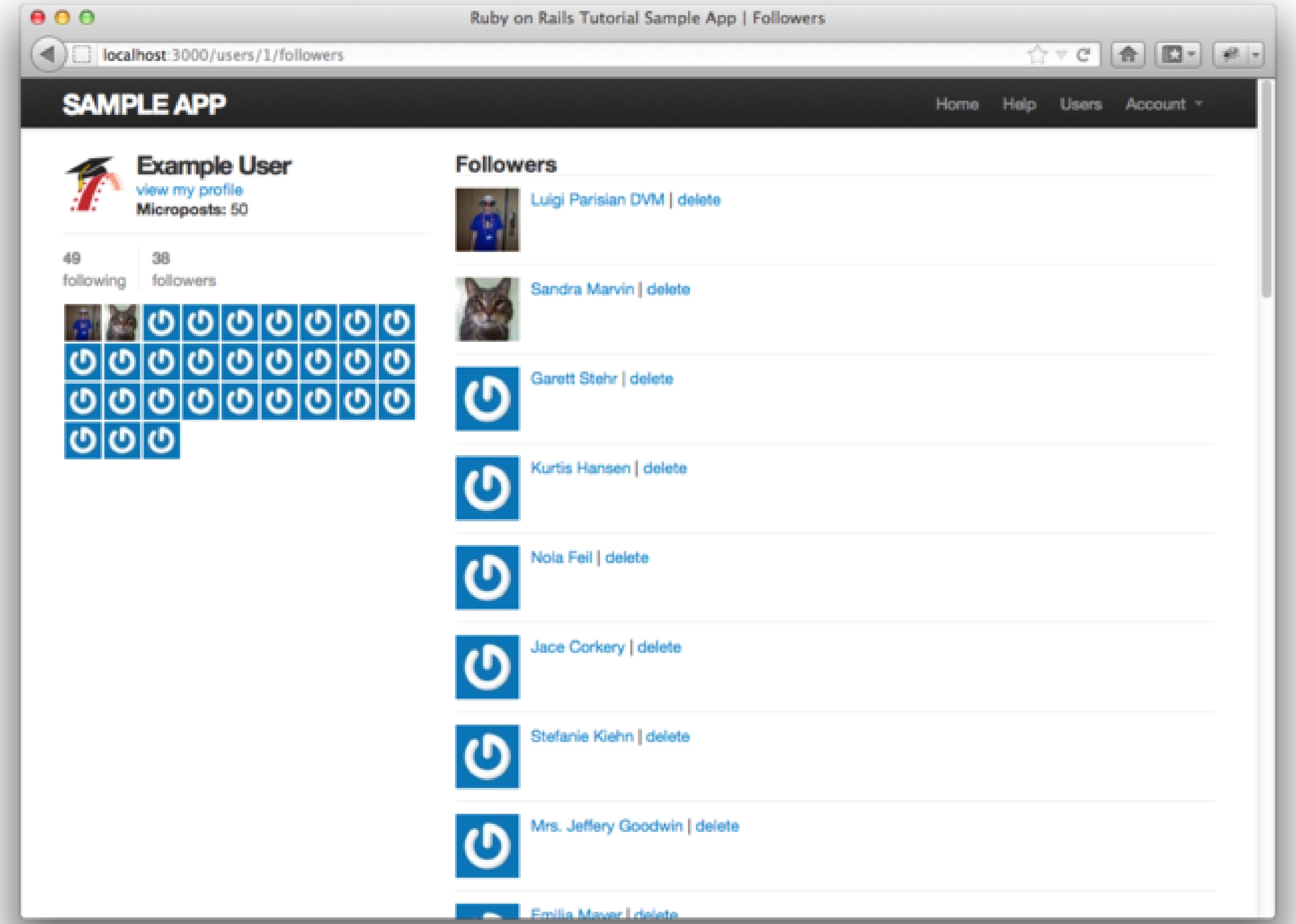


Figure 11.17: Showing the current user's followers. (full size)

11.2.4 A working follow button the standard way

Now that our views are in order, it's time to get the follow/unfollow buttons working. The tests for these button combine many of the testing techniques covered throughout this tutorial and make for a good exercise in reading code. Study Listing 11.32 until you are convinced that you understand what it's testing and why. (There's one minor security ommission as well; see if you can spot it. We'll cover it momentarily.)

Listing 11.32. Tests for the Follow/Unfollow button.
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do
  .
  .
  .
  describe "profile page" do
    let(:user) { FactoryGirl.create(:user) }
    .
    .
    .
    describe "follow/unfollow buttons" do
      let(:other_user) { FactoryGirl.create(:user) }
      before { sign_in user }

      describe "following a user" do
        before { visit user_path(other_user) }

        it "should increment the followed user count" do
          expect do
            click_button "Follow"
          end.to change(user.followed_users, :count).by(1)
        end

        it "should increment the other user's followers count" do
          expect do
            click_button "Follow"
          end.to change(other_user.followers, :count).by(1)
        end

        describe "toggling the button" do
          before { click_button "Follow" }
          it { should have_selector('input', value: 'Unfollow') }
        end
      end

      describe "unfollowing a user" do
        before do
          user.follow!(other_user)
          visit user_path(other_user)
        end

        it "should decrement the followed user count" do
          expect do
            click_button "Unfollow"
          end.to change(user.followed_users, :count).by(-1)
        end

        it "should decrement the other user's followers count" do
          expect do
            click_button "Unfollow"
          end.to change(other_user.followers, :count).by(-1)
        end

        describe "toggling the button" do
          before { click_button "Unfollow" }
          it { should have_selector('input', value: 'Follow') }
        end
      end
    end
  end
  .
  .
  .
end
```

Listing 11.32 tests the following buttons by clicking on them and specifying the proper behavior. Writing the implementation involves digging a little deeper: following and unfollowing involve *creating* and *destroying* relationships, which means defining `create` and `destroy` actions in a Relationships controller (which we must create). Although the following buttons only appear for signed-in users, giving us a superficial layer of security, the tests in Listing 11.32 miss a lower-level issue, namely, the `create` and `destroy` actions themselves should only be accessible to signed-in users. (This is the security hole alluded to above.) Listing 11.33 expresses this requirement using the **post** and **delete** methods to hit those actions directly.

Listing 11.33. Tests for the Relationships controller authorization.
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 Relationships controller" do
        describe "submitting to the create action" do
          before { post relationships_path }
          specify { response.should redirect_to(signin_path) }
        end

        describe "submitting to the destroy action" do
          before { delete relationship_path(1) }
          specify { response.should redirect_to(signin_path) }
        end
      end
      .
      .
      .
    end
  end
end
```

Note that, in order to avoid the overhead of creating a virtually useless Relationship object, the **delete** test hard-codes the id **1** in the named route:

```
before { delete relationship_path(1) }
```

This works because the user should be redirected before the application ever tries to retrieve the relationship with this id.

The controller code needed to get these tests to pass is remarkably concise: we just retrieve the user followed or to be followed, and then follow or unfollow the user using the relevant utility method. The full implementation appears in Listing 11.34.

Listing 11.34. The Relationships controller.
`app/controllers/relationships_controller.rb`

```
class RelationshipsController < ApplicationController
  before_filter :signed_in_user

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

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

We can see from Listing 11.34 why the security issue mentioned above is minor: if an unsigned-in user were to hit either action directly (e.g., using a command-line tool), **`current_user`** would be **`nil`**, and in both cases the action's second line would raise an exception, resulting in an error but no harm to the application or its data. It's best not to rely on that, though, so we've taken the extra step and added an extra layer of security.

With that, the core follow/unfollow functionality is complete, and any user can follow (or unfollow) any other user, which you should verify both by clicking around in the sample application and by running the test suite:

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

11.2.5 A working follow button with Ajax

Although our user following implementation is complete as it stands, we have one bit of polish left to add before starting work on the status feed. You may have noticed in Section 11.2.4 that both the **`create`** and **`destroy`** actions in the Relationships controller simply redirect *back* to the original profile. In other words, a user starts on a profile page, follows the user, and is immediately redirected back to the original page. It is reasonable to ask why the user needs to leave that page at all.

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

```
form_for
```

to

```
form_for ..., remote: true
```

and Rails automagically uses Ajax.[10] The updated partials appear in Listing 11.35 and Listing 11.36.

Listing 11.35. A form for following a user using Ajax.
`app/views/users/_follow.html.erb`

```
<%= form_for(current_user.relationships.build(followed_id: @user.id),
             remote: true) do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <%= f.submit "Follow", class: "btn btn-large btn-primary" %>
<% end %>
```

Listing 11.36. A form for unfollowing a user using Ajax.
`app/views/users/_unfollow.html.erb`

```
<%= form_for(current_user.relationships.find_by_followed_id(@user),
             html: { method: :delete },
             remote: true) do |f| %>
  <%= f.submit "Unfollow", class: "btn btn-large" %>
<% end %>
```

The actual HTML generated by this ERb isn't particularly relevant, but you might be curious, so here's a peek:

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

This sets the variable `data-remote="true"` inside the form tag, which tells Rails to allow the form to be handled by JavaScript. By using a simple HTML property instead of inserting the full

JavaScript code (as in previous versions of Rails), Rails 3 follows the philosophy of *unobtrusive JavaScript*.

Having updated the form, we now need to arrange for the Relationships controller to respond to Ajax requests. Testing Ajax is quite tricky, and doing it thoroughly is a large subject in its own right, but we can get started with the code in Listing 11.37. This uses the `xhr` method (for "XmlHttpRequest") to issue an Ajax request; compare to the `get`, `post`, `put`, and `delete` methods used in previous tests. We then verify that the `create` and `destroy` actions do the correct things when hit with an Ajax request. (To write more thorough test suites for Ajax-heavy applications, take a look at Selenium and Watir.)

Listing 11.37. Tests for the Relationships controller responses to Ajax requests.
`spec/controllers/relationships_controller_spec.rb`

```ruby
require 'spec_helper'

describe RelationshipsController do

  let(:user) { FactoryGirl.create(:user) }
  let(:other_user) { FactoryGirl.create(:user) }

  before { sign_in user }

  describe "creating a relationship with Ajax" do

    it "should increment the Relationship count" do
      expect do
        xhr :post, :create, relationship: { followed_id: other_user.id }
      end.to change(Relationship, :count).by(1)
    end

    it "should respond with success" do
      xhr :post, :create, relationship: { followed_id: other_user.id }
      response.should be_success
    end
  end

  describe "destroying a relationship with Ajax" do
```

```
    before { user.follow!(other_user) }
    let(:relationship) { user.relationships.find_by_followed_id(other_user) }

    it "should decrement the Relationship count" do
      expect do
        xhr :delete, :destroy, id: relationship.id
      end.to change(Relationship, :count).by(-1)
    end

    it "should respond with success" do
      xhr :delete, :destroy, id: relationship.id
      response.should be_success
    end
  end
end
```

The code in Listing 11.37 is our first example of a *controller* test, which I used to use extensively (as in the previous edition of this book) but now mainly eschew in favor of integration tests. In this case, though, the **`xhr`** method is (somewhat inexplicably) not available in integration tests. Although our use of **`xhr`** is new, at this point in the tutorial you should be able to infer from context what the code does:

```
xhr :post, :create, relationship: { followed_id: other_user.id }
```

We see that **`xhr`** takes as arguments a symbol for the relevant HTTP method, a symbol for the action, and a hash representing the contents of **`params`** in the controller itself. As in previous examples, we use **`expect`** to wrap the operation in a block and test for an increment or decrement in the relevant count.

As implied by the tests, the application code uses the same **`create`** and **`destroy`** actions to respond to the Ajax requests that it uses to respond to ordinary `POST` and `DELETE` HTTP requests. All we need to do is respond to a normal HTTP request with a redirect (as in Section 11.2.4) and respond to an Ajax request with JavaScript. The controller code appears as in Listing 11.38. (See Section 11.5 for an exercise showing an even more compact way to accomplish the same thing.)

Listing 11.38. Responding to Ajax requests in the Relationships controller.
`app/controllers/relationships_controller.rb`

```
class RelationshipsController < ApplicationController
  before_filter :signed_in_user

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

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

This code uses `respond_to` to take the appropriate action depending on the kind of request. (There is no relationship between this `respond_to` and the `respond_to` used in the RSpec examples.) The syntax is potentially confusing, and it's important to understand that in

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

only *one* of the lines gets executed (based on the nature of the request).

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

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

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

(Recall from Listing 11.23 that this is a **`div`** that wraps the form, not the form itself.) This syntax, inspired by CSS, uses the **`#`** symbol to indicate a CSS id. As you might guess, jQuery, like CSS, uses a dot **`.`** to manipulate CSS classes.

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

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

Unlike plain JavaScript files, JS-ERb files also allow the use of Embedded Ruby, which we apply in the `create.js.erb` file to update the follow form with the `unfollow` partial (which is what should show after a successful following) and update the follower count. The result is shown in Listing 11.39. This uses the `escape_javascript` function, which is needed to escape out the result when inserting HTML in a JavaScript file.

Listing 11.39. The JavaScript Embedded Ruby to create a following relationship.
`app/views/relationships/create.js.erb`

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

The `destroy.js.erb` file is analogous (Listing 11.40).

Listing 11.40. The Ruby JavaScript (RJS) to destroy a following relationship.
`app/views/relationships/destroy.js.erb`

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

With that, you should navigate to a user profile page and verify that you can follow and unfollow without a page refresh, and the test suite should also pass:

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

Using Ajax in Rails is a large and fast-moving subject, so we've only been able to scratch the surface here, but (as with the rest of the material in this tutorial) our treatment should give you a good foundation for more advanced resources.

11.3 The status feed

We come now to the pinnacle of our sample application: the status feed. Appropriately, this section contains some of the most advanced material in the entire tutorial. The full status feed builds on the proto-feed from Section 10.3.3 by assembling an array of the microposts from the users being followed by the current user, along with the current user's own microposts. To accomplish this feat, we will need some fairly advanced Rails, Ruby, and even SQL programming techniques.

Because of the heavy lifting ahead, it's especially important to review where we're going. A recap of the final user status feed, shown in Figure 11.5, appears again in Figure 11.18.

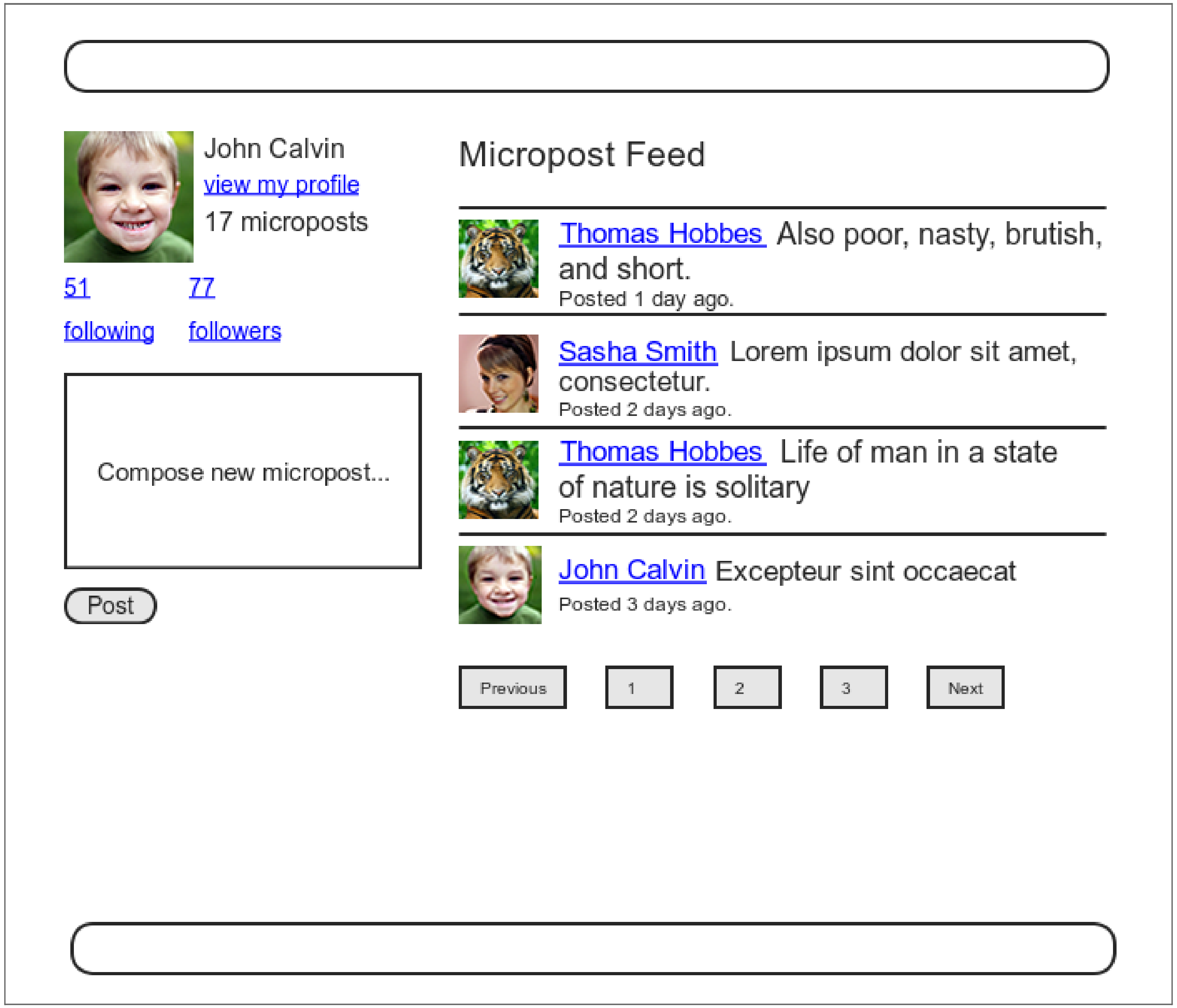
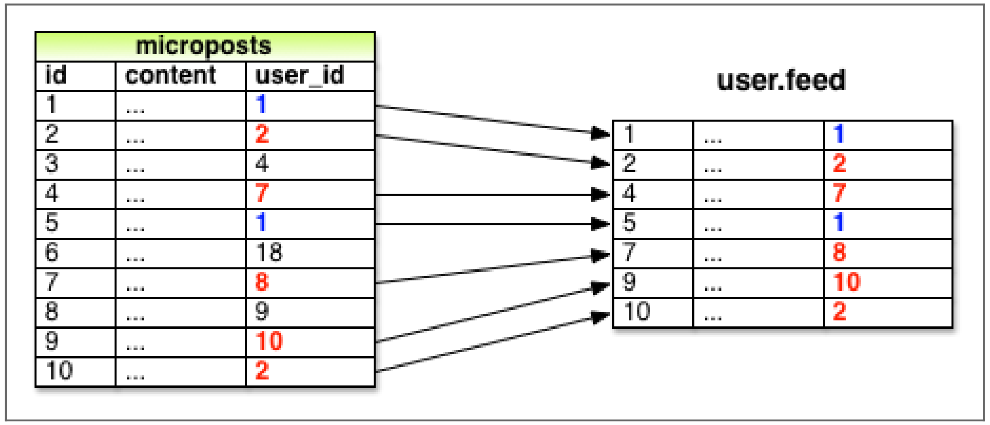

John Calvin
view my profile
17 microposts
51 following
77 followers
Compose new micropost...
Post
Micropost Feed
Thomas Hobbes Also poor, nasty, brutish, and short.
Posted 1 day ago.
Sasha Smith Lorem ipsum dolor sit amet, consectetur.
Posted 2 days ago.
Thomas Hobbes Life of man in a state of nature is solitary
Posted 2 days ago.
John Calvin Excepteur sint occaecat
Posted 3 days ago.
Previous
1
2
3
Next

Figure 11.18: A mockup of a user's Home page with a status feed. (full size)

11.3.1 Motivation and strategy

The basic idea behind the feed is simple. Figure 11.19 shows a sample **microposts** database table and the resulting feed. The purpose of a feed is to pull out the microposts whose user ids correspond to the users being followed by the current user (and the current user itself), as indicated by the arrows in the diagram.



id	content	user_id
1	...	1
2	...	2
3	...	4
4	...	7
5	...	1
6	...	18
7	...	8
8	...	9
9	...	10
10	...	2

1	...	1
2	...	2
4	...	7
5	...	1
7	...	8
9	...	10
10	...	2

Figure 11.19: The feed for a user (id 1) following users 2, 7, 8, and 10.

Since we need a way to find all the microposts from users followed by a given user, we'll plan on implementing a method called **from_users_followed_by**, which we will use as follows:

```
Micropost.from_users_followed_by(user)
```

Although we don't yet know how to implement it, we can already write tests for its functionality. The key is to check all three requirements for the feed: microposts for followed users and the user itself should be included in the feed, but a post from an *unfollowed* user should not be included. Two of these requirements already appear in our tests: Listing 10.38 verifies that a user's own microposts appear in the feed, while the micropost from an unfollowed user doesn't appear. Now that we know how to follow users, we can add a third type of test, this time checking that the microposts of a followed user appear in the feed, as shown in Listing 11.41.

Listing 11.41. The final tests for the status feed.
`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
    .
    .
    .
    describe "status" do
      let(:unfollowed_post) do
        FactoryGirl.create(:micropost, user: FactoryGirl.create(:user))
      end
      let(:followed_user) { FactoryGirl.create(:user) }

      before do
        @user.follow!(followed_user)
        3.times { followed_user.microposts.create!(content: "Lorem ipsum") }
      end

      its(:feed) { should include(newer_micropost) }
      its(:feed) { should include(older_micropost) }
      its(:feed) { should_not include(unfollowed_post) }
      its(:feed) do
        followed_user.microposts.each do |micropost|
          should include(micropost)
          end
        end
      end
    end
  end
```

The feed itself simply defers the hard work to `Micropost.from_users_followed_by`, as shown in Listing 11.42.

Listing 11.42. Adding the completed feed to the User model.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def feed
    Micropost.from_users_followed_by(self)
  end
  .
  .
  .
end
```

11.3.2 A first feed implementation

Now it's time to implement `Micropost.from_users_followed_by`, which for simplicity we'll just refer to as "the feed". Since the final result is rather intricate, we'll build up to the final feed implementation by introducing one piece at a time.

The first step is to think of the kind of query we'll need. What we want to do is select from the `microposts` table all the microposts with ids corresponding to the users being followed by a given user (or the user itself). We might write this schematically as follows:

```
SELECT * FROM microposts
WHERE user_id IN (<list of ids>) OR user_id = <user id>
```

In writing this code, we've guessed that SQL supports an `IN` keyword that allows us to test for set inclusion. (Happily, it does.)

Recall from the proto-feed in Section 10.3.3 that Active Record uses the `where` method to accomplish the kind of select shown above, as illustrated in Listing 10.39. There, our select was very simple; we just picked out all the microposts with user id corresponding to the current user:

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

Here, we expect it to be more complicated, something like

```
where("user_id in (?) OR user_id = ?", following_ids, user)
```

(Here we've used the Rails convention of `user` instead of `user.id` in the condition; Rails automatically uses the `id`. We've also omitted the leading `Micropost.` since we expect this method to live in the Micropost model itself.)

We see from these conditions that we'll need an array of ids corresponding to the users being followed. One way to do this is to use Ruby's `map` method, available on any "enumerable" object, i.e., any object (such as an Array or a Hash) that consists of a collection of elements.[11] We saw an example of this method in Section 4.3.2; it works like this:

```
$ rails console
```

```
>> [1, 2, 3, 4].map { |i| i.to_s }
=> ["1", "2", "3", "4"]
```

Situations like the one illustrated above, where the same method (e.g., `to_s`) gets called on each element, are common enough that there's a shorthand notation using an *ampersand* `&` and a symbol corresponding to the method:[12]

```
>> [1, 2, 3, 4].map(&:to_s)
=> ["1", "2", "3", "4"]
```

Using the `join` method(Section 4.3.1), we can create a string composed of the ids by joining them on comma-space :

```
>> [1, 2, 3, 4].map(&:to_s).join(', ')
=> "1, 2, 3, 4"
```

We can use the above method to construct the necessary array of followed user ids by calling `id` on each element in `user.followed_users`. For example, for the first user in the database this array appears as follows:

```
>> User.first.followed_users.map(&:id)
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

In fact, because this sort of construction is so useful, Active Record provides it by default:

```
>> User.first.followed_user_ids
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

Here the **`followed_user_ids`** method is synthesized by Active Record based on the **`has_many :followed_users`** association (Listing 11.10); the result is that we need only append **`_ids`** to the association name to get the ids corresponding to the **`user.followed_users`** collection. A string of followed user ids then appears as follows:

```
>> User.first.followed_user_ids.join(', ')
=> "4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51"
```

When inserting into an SQL string, though, you don't need to do this; the **`?`** interpolation takes care of it for you (and in fact eliminates some database-dependent incompatibilities). This means we can use

```
user.followed_user_ids
```

by itself.

At this point, you might guess that code like

```
Micropost.from_users_followed_by(user)
```

will involve a class method in the **`Micropost`** class (a construction mentioned briefly in

Section 4.4.1). A proposed implementation along these lines appears in Listing 11.43.

Listing 11.43. A first cut at the `from_users_followed_by` method.
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
  def self.from_users_followed_by(user)
    followed_user_ids = user.followed_user_ids
    where("user_id IN (?) OR user_id = ?", followed_user_ids, user)
  end
end
```

Although the discussion leading up to Listing 11.43 was couched in hypothetical terms, it actually works! You can verify this yourself by running the test suite, which should pass:

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

In some applications, this initial implementation might be good enough for most practical purposes. But it's not the final implementation; see if you can make a guess about why not before moving on to the next section. (*Hint:* What if a user is following 5000 other users?)

11.3.3 Subselects

As hinted at in the last section, the feed implementation in Section 11.3.2 doesn't scale well when the number of microposts in the feed is large, as would likely happen if a user were following, say, 5000 other users. In this section, we'll reimplement the status feed in a way that scales better with the number of followed users.

The problem with the code in Section 11.3.2 is that

```
followed_user_ids = user.followed_user_ids
```

pulls *all* the followed users' ids into memory, and creates an array the full length of the followed users array. Since the condition in Listing 11.43 actually just checks inclusion in a set, there must be a more efficient way to do this, and indeed SQL is optimized for just such set operations.The solution involves pushing the finding of followed user ids into the database using a *subselect*.

We'll start by refactoring the feed with the slightly modified code in Listing 11.44.

Listing 11.44. Improving **`from_users_followed_by`**.
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
  # Returns microposts from the users being followed by the given user.
  def self.from_users_followed_by(user)
    followed_user_ids = user.followed_user_ids
    where("user_id IN (:followed_user_ids) OR user_id = :user_id",
          followed_user_ids: followed_user_ids, user_id: user)
  end
end
```

As preparation for the next step, we have replaced

```
where("user_id IN (?) OR user_id = ?", followed_user_ids, user)
```

with the equivalent

```
where("user_id IN (:followed_user_ids) OR user_id = :user_id",
      followed_user_ids: followed_user_ids, user_id: user))
```

The question mark syntax is fine, but when we want the *same* variable inserted in more than one place, the second syntax is more convenient.

The above discussion implies that we will be adding a *second* occurrence of `user_id` in the SQL query. In particular, we can replace the Ruby code

```
followed_user_ids = user.followed_user_ids
```

with the SQL snippet

```
followed_user_ids = "SELECT followed_id FROM relationships
                     WHERE follower_id = :user_id"
```

This code contains an SQL subselect, and internally the entire select for user 1 would look something like this:

```
SELECT * FROM microposts
WHERE user_id IN (SELECT followed_id FROM relationships
                  WHERE follower_id = 1)
      OR user_id = 1
```

This subselect arranges for all the set logic to be pushed into the database, which is more efficient.[13]

With this foundation, we are ready for an efficient feed implementation, as seen in Listing 11.45. Note that, because it is now raw SQL, `followed_user_ids` is *interpolated*, not escaped. (It actually works either way, but logically it makes more sense to interpolate in this context.)

Listing 11.45. The final implementation of `from_users_followed_by`.
`app/models/micropost.rb`

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

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

  default_scope order: 'microposts.created_at DESC'

  def self.from_users_followed_by(user)
    followed_user_ids = "SELECT followed_id FROM relationships
                         WHERE follower_id = :user_id"
    where("user_id IN (#{followed_user_ids}) OR user_id = :user_id",
          user_id: user.id)
  end
end
```

This code has a formidable combination of Rails, Ruby, and SQL, but it does the job, and does it well. (Of course, even the subselect won't scale forever. For bigger sites, you would probably need to generate the feed asynchronously using a background job. Such scaling subtleties are beyond the scope of this tutorial, but the Scaling Rails screencasts are a good place to start.)

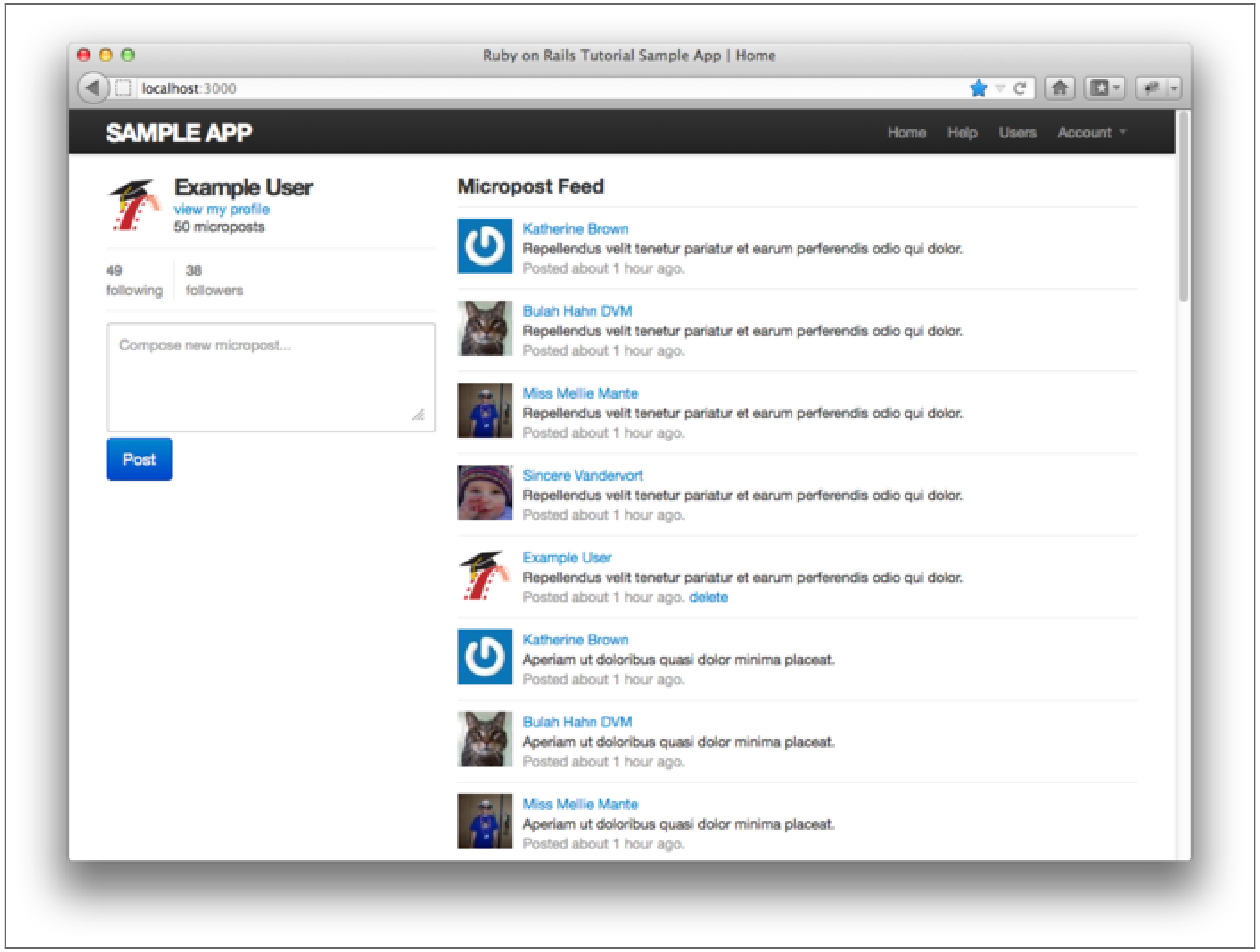
11.3.4 The new status feed

With the code in Listing 11.45, our status feed is complete. As a reminder, the code for the Home page appears in Listing 11.46; this code creates a paginated feed of the relevant microposts for use in the view, as seen in Figure 11.20.[14] Note that the `paginate` method actually reaches all the way into the Micropost model method in Listing 11.45, arranging to pull out only 30 microposts at a time from the database. (You can verify this by examining the SQL statements in the development server log file.)

Listing 11.46. The `home` action with a paginated feed.

`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
```



Example User
view my profile
50 microposts

49 following | 38 followers

Compose new micropost...

Micropost Feed

Katherine Brown
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted about 1 hour ago.

Bulah Hahn DVM
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted about 1 hour ago.

Miss Mellie Mante
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted about 1 hour ago.

Sincere Vandervort
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted about 1 hour ago.

Example User
Repellendus velit tenetur pariatur et earum perferendis odio qui dolor.
Posted about 1 hour ago. delete

Katherine Brown
Aperiam ut doloribus quasi dolor minima placeat.
Posted about 1 hour ago.

Bulah Hahn DVM
Aperiam ut doloribus quasi dolor minima placeat.
Posted about 1 hour ago.

Miss Mellie Mante
Aperiam ut doloribus quasi dolor minima placeat.
Posted about 1 hour ago.

Figure 11.20: The Home page with a working status feed. (full size)

11.4 Conclusion

With the addition of the status feed, we've finished the core sample application for the *Ruby on Rails Tutorial*. This application includes examples of all the major features of Rails, including models, views, controllers, templates, partials, filters, validations, callbacks, **`has_many`**/**`belongs_to`** and **`has_many through`** associations, security, testing, and deployment. Despite this impressive list, there is still much to learn about Rails. As a first step in this process, this section contains some suggested extensions to the core application, as well as suggestions for further learning.

Before moving on to tackle any of the application extensions, it's a good idea to merge in your changes:

```
$ git add .
$ git commit -m "Add user following"
$ git checkout master
$ git merge following-users
```

As usual, you can also push the code and deploy the application if you want:

```
$ git push
$ 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.

11.4.1 Extensions to the sample application

The proposed extensions in this section are mostly inspired either by general features common to web applications, such as password reminders and email confirmation, or features specific to our type of sample application, such as search, replies, and messaging. Implementing one or more of these application extensions will help you make the transition from following a tutorial to writing original applications of your own.

Don't be surprised if it's tough going at first; the blank slate of a new feature can be quite intimidating. To help get you started, I can give two pieces of general advice. First, before adding any feature to a Rails application, take a look at the RailsCasts archive to see if Ryan Bates has already covered the subject.[15] If he has, watching the relevant RailsCast first will often save you a ton of time. Second, always do extensive Google searches on your proposed feature to find relevant blog posts and tutorials. Web application development is hard, and it helps to learn from the experience (and mistakes) of others.

Many of the following features are quite challenging, and I have given some hints about the tools you might need to implement them. Even with hints, they are *much* more difficult than the book's end-of-chapter exercises, so don't be discouraged if you can't solve them without considerable effort. Due to time constraints, I am not available for one-on-one assistance, but if there is sufficient interest I might release standalone article/screencast bundles on some of these extensions in the future; go to the main Rails Tutorial website at http://railstutorial.org/ and subscribe to the news feed to get the latest updates.

Replies

Twitter allows users to make "@replies", which are microposts whose first characters are the user's login preceded by the @ sign. These posts only appear in the feed of the user in question or users following that user. Implement a simplified version of this, restricting @replies to appear only in the feeds of the recipient and the sender. This might involve adding an `in_reply_to` column in the `microposts` table and an extra `including_replies` scope to the Micropost model.

Since our application lacks unique user logins, you will also have to decide on a way to represent users. One option is to use a combination of the id and the name, such as `@1-michael-hartl`. Another is to *add* a unique username to the signup process and then use it in @replies.

Messaging

Twitter supports direct (private) messaging by prefixing a micropost with the letter “d”. Implement this feature for the sample application. The solution will probably involve a Message model and a regular expression match on new microposts.

Follower notifications

Implement a feature to send each user an email notification when they gain a new follower. Then make the notification optional, so that users can opt out if desired. Among other things, adding this feature requires learning how to send mail with Rails. To get started, I suggest viewing the RailsCast on Action Mailer in Rails 3.

Password reminders

Currently, if our application’s users forget their passwords, they have no way to retrieve them. Because of the one-way secure password hashing in Chapter 6, our application can’t email the user’s password, but it can send a link to a reset form. Follow the steps in the RailsCast on Remember Me & Reset Password to fix this omission.

Signup confirmation

Apart from an email regular expression, the sample application currently has no way to verify the validity of a user’s email address. Add an email address verification step to confirm a user’s signup. The new feature should create users in an inactive state, email the user an activation URI, and then change the user to an active state when the URI gets hit. You might want to read up on state machines in Rails to help you with the inactive/active transition.

RSS feed

For each user, implement an RSS feed for their microposts. Then implement an RSS feed for their status feed, optionally restricting access to that feed using an authentication scheme. The RailsCast on generating RSS feeds will help get you started.

REST API

Many websites expose an Application Programmer Interface (API) so that third-party applications can get, post, put, and delete the application's resources. Implement such a REST API for the sample application. The solution will involve adding `respond_to` blocks (Section 11.2.5) to many of the application's controller actions; these should respond to requests for XML. Be careful about security; the API should only be accessible to authorized users.

Search

Currently, there is no way for users to find each other, apart from paging through the user index or viewing the feeds of other users. Implement a search feature to remedy this. Then add another search feature for microposts. The RailsCast on simple search forms will help get you started. If you deploy using a shared host or a dedicated server, I suggest using Thinking Sphinx (following the RailsCast on Thinking Sphinx). If you deploy on Heroku, you should follow the Heroku full text search instructions.

11.4.2 Guide to further resources

There are a wealth of Rails resources in stores and on the web—indeed, the supply is so rich that it can be overwhelming. The good news is that, having gotten this far, you're ready for almost anything else out there. Here are some suggestions for further learning:

- The *Ruby on Rails Tutorial* screencasts: I have prepared a full-length screencast course based on this book. In addition to covering all the material in the book, the screencasts are

filled with tips, tricks, and the kind of see-how-it's-done demos that are hard to capture in print. They are available for purchase through the Ruby on Rails Tutorial website.
- RailsCasts: It's hard to overemphasize what a great resource RailsCasts is. I suggest starting by visiting the RailsCasts episode archive and clicking on subjects that catch your eye.
- Scaling Rails: One topic we've hardly covered in the *Ruby on Rails Tutorial* book is performance, optimization, and scaling. Luckily, most sites will never run into serious scaling issues, and using anything beyond plain Rails is probably premature optimization. If you do run into performance issues, the Scaling Rails series from Gregg Pollack of Envy Labs is a good place to start. I also recommend investigating the site monitoring applications Scout and New Relic.[16] And, as you might suspect by now, there are RailsCasts on many scaling subjects, including profiling, caching, and background jobs.
- Ruby and Rails books: I recommend *Beginning Ruby* by Peter Cooper, *The Well-Grounded Rubyist* by David A. Black, and *The Ruby Way* by Hal Fulton for further Ruby learning, and *The Rails 3 Way* by Obie Fernandez and *Rails 3 in Action* (wait for the second edition) by Ryan Bigg and Yehuda Katz for more about Rails.
- PeepCode and Code School: The screencasts at PeepCode and interactive courses at Code School are consistently high-quality, and I warmly recommend them.

11.5 Exercises

1. Add tests for destroying relationships associated with a given user (i.e., as implemented by `dependent :destroy` in Listing 11.4 and Listing 11.16). *Hint*: Follow the example in Listing 10.15.

2. The `respond_to` method seen in Listing 11.38 can actually be hoisted out of the actions into the Relationships controller itself, and the `respond_to` blocks can be replaced with a Rails method called `respond_with`. Prove that the resulting code, shown in Listing 11.47, is correct by verifying that the test suite still passes. (For details on this method, do a Google search on "rails respond_with".)

3. Refactor Listing 11.31 by adding partials for the code common to the following/followers pages, the Home page, and the user show page.

4. Following the model in Listing 11.19, write tests for the stats on the profile page.

Listing 11.47. A compact refactoring of Listing 11.38.

```
class RelationshipsController < ApplicationController
  before_filter :signed_in_user

  respond_to :html, :js

  def create
    @user = User.find(params[:relationship][:followed_id])
    current_user.follow!(@user)
    respond_with @user
  end

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

« Chapter 10 User microposts

---

1. The photographs in the mockup tour are from http://www.flickr.com/photos/john_lustig/2518452221/ and http://www.flickr.com/photos/30775272@N05/2884963755/. ↑

2. The first edition of this book used the `user.following` terminology, which even I found confusing at times. Thanks to reader Cosmo Lee for convincing me to change the terminology and for offering suggestions on how to make it clearer. (I didn't follow his exact advice, though, so if it's

still confusing he bears none of the blame.) ↑

3. For simplicity, Figure 11.6 suppresses the **`followed_users`** table's **`id`** column. ↑

4. See the discussion on when to use let at Stack Overflow for more information. ↑

5. Technically, Rails uses the **`underscore`** method to convert the class name to an id. For example, **`"FooBar".underscore`** is **`"foo_bar"`**, so the foreign key for a **`FooBar`** object would be **`foo_bar_id`**. (Incidentally, the inverse of **`underscore`** is **`camelize`**, which converts **`"camel_case"`** to **`"CamelCase"`**.) ↑

6. If you've noticed that **`followed_id`** also identifies a user, and are concerned about the asymmetric treatment of followed and follower, you're ahead of the game. We'll deal with this issue in Section 11.1.5. ↑

7. Once you have a lot of experience modeling a particular domain, you can often guess such utility methods in advance, and even when you can't you'll often find yourself writing them to make the tests cleaner. In this case, though, it's OK if you wouldn't have guessed them. Software development is usually an iterative process—you write code until it starts getting ugly, and then you refactor it—but for brevity the tutorial presentation is streamlined a bit. ↑

8. The **`unfollow!`** method *doesn't* raise an exception on failure—in fact, I don't even know how Rails indicates a failed destroy—but we use an exclamation point to maintain the symmetry with **`follow!`**. ↑

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

10. This only works if JavaScript is enabled in the browser, but it degrades gracefully, working exactly as in Section 11.2.4 if JavaScript is disabled. ↑

11. The main requirement is that enumerable objects must implement an **`each`** method to iterate through the collection. ↑

12. This notation actually started as an extension Rails made to the core Ruby language; it was so useful that it has now been incorporated into Ruby itself. How cool is that? ↑

13. For a more advanced way to create the necessary subselect, see the blog post “Hacking a subselect in ActiveRecord”. ↑

14. In order to make a prettier feed for Figure 11.20, I’ve added a few extra microposts by hand using the Rails console. ↑

15. Note that RailsCasts usually omit the tests, which is probably necessary to keep the episodes nice and short, but you could get the wrong idea about the importance of testing. Once you’ve watched the relevant RailsCast to get a basic idea of how to proceed, I suggest writing the new feature using test-driven development. (In this context, I recommend taking a look at the RailsCast on “How I test”. You’ll see that Ryan Bates himself often uses TDD for real-life development, and in fact his testing style is similar to style used in this tutorial.) ↑

16. In addition to being a clever phrase—*new relic* being a contradiction in terms—New Relic is also an anagram for the name of the company’s founder, Lew Cirne. ↑