∩╗┐ Chapter 12 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 userΓÇÖs Home page displaying a status feed of the followed usersΓÇÖ microposts. We will also make views to display both a userΓÇÖs followers and the users each user is following. We will learn how to model user following in Section 12.1, and then make the web interface in Section 12.2 (including an introduction to Ajax). Finally, weΓÇÖll end by developing a fully functional status feed in Section 12.3.

This final chapter contains some of the most challenging material in the tutorial, including a complicated data model and 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 12.4 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 weΓÇÖll pause for a moment and take a tour of user following. As in previous chapters, at this early stage weΓÇÖll represent pages using mockups.1 The full page flow runs as follows: a user (John Calvin) starts at his profile page (Figure 12.1) and navigates to the Users page (Figure 12.2) to select a user to follow. Calvin navigates to the profile of a second user, Thomas Hobbes (Figure 12.3), clicking on the ΓÇ£FollowΓÇ¥ button to follow that user. This changes the ΓÇ£FollowΓÇ¥ button to ΓÇ£UnfollowΓÇ¥, and increments HobbesΓÇÖs ΓÇ£followersΓÇ¥ count by one (Figure 12.4). Navigating to his home page, Calvin now sees an incremented ΓÇ£followingΓÇ¥ count and finds HobbesΓÇÖs microposts in his status feed (Figure 12.5). The rest of this chapter is dedicated to making this page flow actually work.

page_flow_profile_mockup
Figure 12.1: A mockup of the current userΓÇÖs profile. (full size)
page_flow_user_index_mockup
Figure 12.2: A mockup of finding a user to follow. (full size)
page_flow_other_profile_follow_button_mockup
Figure 12.3: A mockup of the profile of another user, with a follow button. (full size)
page_flow_other_profile_unfollow_button_mockup
Figure 12.4: A profile mockup with an unfollow button and incremented followers count. (full size)
page_flow_home_page_feed_mockup
Figure 12.5: A Home page mockup, with status feed and incremented following count. (full size)

12.1 The Relationship model

Our first step in implementing user following and followers is to construct a data model, which is not as straightforward as it seems. Na├»vely, it seems that a has_many relationship should do: a user has_many following and has_many followers. As we will see, there is a problem with this approach, and weΓÇÖll learn how to fix it using has_many :through. ItΓÇÖs likely that many of the ideas in this section wonΓÇÖt 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.

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

As a first step toward constructing a data model for user following, letΓÇÖs 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 RailsΓÇÖ default pluralization convention, the set of all such followed users would be called the followeds, but that is ungrammatical and clumsy; instead, we will override the default and call them following, so that user.following will contain an array of the users being followed. Similarly, the set of all users following a given user is that userΓÇÖs followers, and user.followers will be an array of those users.

This suggests modeling the following users as in Figure 12.6, with a following table and a has_many association. Since user.following should be an array of users, each row of the following table would need to be a user, as identified by the followed_id, together with the follower_id to establish the association.2 In addition, since each row is a user, we would need to include the userΓÇÖs other attributes, including the name, password, etc.

naive_user_has_many_following

Figure 12.6: A na ├»ve implementation of user following.

The problem with the data model in Figure 12.6 is that it is terribly redundant: each row contains not only each followed userΓÇÖs id, but all their other information as wellΓÇöall of which is already in the users table. Even worse, to model user followers we would need a separate followers table. Finally, this data model is a maintainability nightmare, since each time a user changed (say) his name, we would need to update not just the userΓÇÖs record in the users table but also every row containing that user in both the following 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 following in a web application. Recall from Section 6.3.3 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 unfollows another user, what is being destroyed?

Upon reflection, we see that in these cases the application should either create or destroy a relationship (or connection3) between two users. A user then has_many :relationships, and has many following (or followers) through those relationships. Indeed, Figure 12.6 already contains most of the implementation: since each followed user is uniquely identified by followed_id, we could convert following 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 userΓÇÖs followers.

To make a following 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; the relevant technique is known as has_many :through.4 As we will see in Section 12.1.4, Rails allows us to say that a user is following many users through the relationships table, using the succinct code

```
has_many :following, :through => :relationships, :source => "followed_id"
```

This code automatically populates user.following with an array of followed users. A diagram of the data model appears in Figure 12.7.
user_has_many_following
Figure 12.7: A model of user following through an intermediate Relationship model. (full size)

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

```
$ script/generate rspec_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 12.1.
Listing 12.1. Adding indices on the follower_id and followed_id columns.
db/migrate/<timestamp>_create_relationships.rb

```
class CreateRelationships < ActiveRecord::Migration
  def self.up
    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
  end

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

We then migrate the database and prepare the test database as usual:

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

The result is the Relationship data model shown in Figure 12.8.
relationship_model
Figure 12.8: The Relationship data model.

As with any new model, before moving on, we should define the modelΓÇÖs accessible attributes. In the case of the Relationship model, the followed_id should be accessible, since users will create relationships through the web, but the follower_id attribute should not be accessible; otherwise, malicious users could force other users to follow them. The result appears in Listing 12.2.
Listing 12.2. Making a relationshipΓÇÖs followed_id (but not follower_id) accessible.
app/models/relationship.rb

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

12.1.2 User/relationship associations

Before implementing following and followers, we first need to establish the association between users and relationships. A user has_many relationships, andΓÇösince relationships involve two usersΓÇöa relationship belongs_to both a follower and a followed user.

As with microposts in Section 11.1.2, we will create new relationships using the user association, with code such as

```
user.relationships.create(:followed_id => ...)
```

We start with a test, shown in Listing 12.3, which builds an @relationships instance variable (used below) and makes sure that it can be saved using save!. As with create!, the save! method raises an exception if the save fails; compare this to the use of create! in Listing 11.4.
Listing 12.3. Testing Relationship creation with save!.
spec/models/relationship_spec.rb

```
require 'spec_helper'

describe Relationship do

  before(:each) do
    @follower = Factory(:user)
    @followed = Factory(:user, :email => Factory.next(:email))

    @relationship = @follower.relationships.build(:followed_id => @followed.id)
  end

  it "should create a new instance given valid attributes" do
    @relationship.save!
  end
end
```

We should also test the User model for a relationships attribute, as shown in Listing 12.4.
Listing 12.4. Testing for the user.relationships attribute.
spec/models/user_spec.rb

```
describe User do
```

```
  .
  .
  .
  describe "relationships" do

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

    it "should have a relationships method" do
      @user.should respond_to(:relationships)
    end
  end
end
```

At this point you might expect application code as in Section 11.1.2, and itΓÇÖs similar, but there is one critical difference: in the case of the Micropost model, we could say

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

and

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

because the microposts table has a user_id attribute to identify the user (Section 11.1.1). 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 can infer 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 12.5.6

Listing 12.5. 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 userΓÇÖs relationships, weΓÇÖve gone ahead and added :dependent => :destroy to the association; writing a test for this is left as an exercise (Section 12.5).) At this point, the association tests in Listing 12.3 and Listing 12.4 should pass.

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 12.6.
Listing 12.6. Testing the user/relationships belongs_to association.
spec/models/relationship_spec.rb

```
describe Relationship do
  .
  .
  .
  describe "follow methods" do

    before(:each) do
      @relationship.save
    end

    it "should have a follower attribute" do
      @relationship.should respond_to(:follower)
    end

    it "should have the right follower" do
      @relationship.follower.should == @follower
    end

    it "should have a followed attribute" do
      @relationship.should respond_to(:followed)
    end

    it "should have the right followed user" do
      @relationship.followed.should == @followed
    end
  end
end
```

To write the application code, we face the same problem we had with the has_many association: Rails canΓÇÖt guess the foreign keys from the class name User; moreover, since there is neither a Followed nor a Follower model, we need to supply the class name as well. The result is shown in Listing 12.7.
Listing 12.7. Adding the belongs_to associations to the Relationship model.
app/models/relationship.rb

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

  belongs_to :follower, :foreign_key => "follower_id", :class_name => "User"
  belongs_to :followed, :foreign_key => "followed_id", :class_name => "User"
end
```

The followed association isnΓÇÖt actually needed until Section 12.1.5, but the parallel follower/followed structure is clearer if we implement them both at the same time.
12.1.3 Validations

Before moving on, weΓÇÖll add a couple Relationship model validations for completeness. The tests (Listing 12.8) and application code (Listing 12.9) are straightforward.
Listing 12.8. Testing the Relationship model validations.
spec/models/relationship_spec.rb

```
describe Relationship do
  .
```

```
  .
  .

  describe "validations" do

    it "should reqire a follower_id" do
      @relationship.follower_id = nil
      @relationship.should_not be_valid
    end

    it "should reqire a followed_id" do
      @relationship.followed_id = nil
      @relationship.should_not be_valid
    end
  end
end
```

Listing 12.9. Adding the Relationship model validations.
app/models/relationship.rb

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

  belongs_to :follower, :class_name => "User", :foreign_key => "follower_id"
  belongs_to :followed, :class_name => "User", :foreign_key => "followed_id"

  validates_presence_of :follower_id, :followed_id
end
```

12.1.4 Following

We come now to the heart of the Relationship associations: following and followers. We start with following, as shown Listing 12.10.
Listing 12.10. A test for the user.following attribute.
spec/models/user_spec.rb

```
describe User do
  .
  .
  .
  describe "relationships" do

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

    it "should have a relationships method" do
      @user.should respond_to(:relationships)
    end

    it "should have a following method" do
      @user.should respond_to(:following)
    end
  end
end
```

The implementation uses has_many :through for the first time: a user has many following through relationships, as illustrated in Figure 12.7. 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 12.1.1, user.followeds is rather awkward; far more natural is to treat ΓÇ£followingΓÇ¥ as a plural of ΓÇ£followedΓÇ¥, and write instead user.following for the array of followed users. Naturally, Rails allows us to override the default, in this case using the :source parameter (Listing 12.11), which explicitly tells Rails that the source of the following array is the set of followed ids.

Listing 12.11. Adding the User model following association with has_many :through.
app/models/user.rb

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

To create a following relationship, weΓÇÖll introduce a follow! utility method so that we can write user.follow!(other_user).7 WeΓÇÖll also add an associated following? boolean method to test if one user is following another.8 The tests in Listing 12.12 show how we expect these methods to be used in practice.

Listing 12.12. Tests for some following utility methods.
spec/models/user_spec.rb

```
describe User do
  .
  .
  .
  describe "relationships" do
    .
    .
    .
    it "should have a following? method" do
      @user.should respond_to(:following?)
    end

    it "should have a follow! method" do
      @user.should respond_to(:follow!)
    end

    it "should follow another user" do
      @user.follow!(@followed)
      @user.should be_following(@followed)
    end

    it "should include the followed user in the following array" do
      @user.follow!(@followed)
      @user.following.include?(@followed).should be_true
    end
```

```
  end
end
```

In the application code, the following? method takes in a user, called followed, and checks to see if a follower 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 12.13.9

Listing 12.13. The following? and follow! utility methods.

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  def self.authenticate(email, submitted_password)
    .
    .
    .
  end

  def following?(followed)
    relationships.find_by_followed_id(followed)
  end

  def follow!(followed)
    relationships.create!(:followed_id => followed.id)
  end
  .
  .
  .
end
```

Note that in Listing 12.13 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 12.14.10

Listing 12.14. A test for unfollowing a user.

spec/models/user_spec.rb

```
describe User do
  .
  .
  .
  describe "relationships" do
    .
    .
    .
    it "should have an unfollow! method" do
      @followed.should respond_to(:unfollow!)
    end
```

```
    it "should unfollow a user" do
      @user.follow!(@followed)
      @user.unfollow!(@followed)
      @user.should_not be_following(@followed)
    end
  end
end
```

The code for unfollow! is straightforward: just find the relationship by followed id and destroy it (Listing 12.15).11
Listing 12.15. Unfollowing a user by destroying a user relationship.
app/models/user.rb

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

  def follow!(followed)
    relationships.create!(:followed_id => followed.id)
  end

  def unfollow!(followed)
    relationships.find_by_followed_id(followed).destroy
  end
  .
  .
  .
end
```

12.1.5 Followers

The final piece of the relationships puzzle is to add a user.followers method to go with user.following. You may have noticed from Figure 12.7 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 12.9), we could implement user.followers with little effort.
user_has_many_followers
Figure 12.9: A model for user followers using a reverse Relationship model. (full size)

We begin with the tests, having faith that the magic of Rails will come to the rescue (Listing 12.16).
Listing 12.16. Testing for reverse relationships.
spec/models/user_spec.rb

```
describe User do
  .
  .
  .
  describe "relationships" do
    .
    .
    .
    it "should have a reverse_relationships method" do
      @user.should respond_to(:reverse_relationships)
    end
```

```
    it "should have a followers method" do
      @user.should respond_to(:followers)
    end

    it "should include the follower in the followers array" do
      @user.follow!(@followed)
      @followed.followers.include?(@user).should be_true
    end
  end
end
```

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 following 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 12.17.

Listing 12.17. 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 12.5, the test for dependent :destroy is left as an exercise (Section 12.5).) 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 will look for a ReverseRelationship class, which doesnΓÇÖt exist.

ItΓÇÖs also worth noting that we could actually omit the :source key in this case, using simply

```
has_many :followers, :through => :reverse_relationships
```

since Rails will automatically look for the foreign key follower_id in this case. IΓÇÖve kept the :source key to emphasize the parallel structure with the has_many :following association, but you are free to leave it out.

With the code in Listing 12.17, the following/follower associations are complete, and all the tests should pass. This section has placed rather heavy demands on your data modeling skills, and itΓÇÖs 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.

12.2 A web interface for following and followers

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 12.3, weΓÇÖll complete our sample application by adding the userΓÇÖs status feed.

12.2.1 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 11.20, 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 12.18.

Listing 12.18. Adding following/follower relationships to the sample data.

lib/tasks/sample_data.rake

```
require 'faker'

namespace :db do
  desc "Fill database with sample data"
  task :populate => :environment do
    Rake::Task['db:reset'].invoke
    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
  User.all(:limit => 6).each do |user|
    50.times do
      content = Faker::Lorem.sentence(5)
      user.microposts.create!(:content => content)
    end
  end
end

def make_relationships
  users = User.all
  user  = users.first
  following = users[1..50]
```

```
  followers = users[3..40]
  following.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
  following = users[1..50]
  followers = users[3..40]
  following.each { |followed| user.follow!(followed) }
  followers.each { |follower| follower.follow!(user) }
end
```

We somewhat arbitrarily arrange for the first user to follow the next 50 users, and then have users with ids 3 through 40 follow that user back. The resulting relationships will be sufficient for developing the application interface.

To execute the code in Listing 12.18, populate the database as usual:

```
$ rake db:populate
```

12.2.2 Stats and a follow form

Now that our sample users have both following and followers arrays, we need to update the profile pages and home pages to reflect this. WeΓÇÖll start by making a partial to display the following and follower statistics on the profile and home pages, as mocked up in Figure 12.1 and Figure 12.5. The result will be displays of the number following and the number of followers, together with links to their dedicated display pages. WeΓÇÖll next add a follow/unfollow form, and then make dedicated pages for showing user following and followers.
stats_partial_mockup
Figure 12.10: A mockup of the stats partial.

A close-up of the stats area, taken from the mockup in Figure 12.1, appears in Figure 12.10. These stats consist of a count of the number of users the current user is following and that userΓÇÖs 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 ΓÇÖ#ΓÇÖ, but that was before we had much experience with routes. This time, although weΓÇÖll defer the actual pages to Section 12.2.3, weΓÇÖll make the routes now, as seen in Listing 12.19. This code uses the :member option to map.resources, which we havenΓÇÖt seen before, but see if you can guess what it does.
Listing 12.19. Adding following and followers actions to the Users controller. config/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :users, :member => { :following => :get, :followers => :get }
  .
  .
  .
end
```

You might suspect that the URLs for user following and followers will look like /users/1/following and /users/1/followers, and that is exactly what the code in Listing 12.19 does. Since both pages will be showing data, we use :get to arrange for the URLs to respond to GET requests (as required by the REST convention for such pages), and the :member option means that the routes respond to URLs containing the user id. (The opposite option, :collection, works without the id, so that

```
map.resources :users, :collection => { :tigers => :get }
```

would respond to the URL /users/tigersΓÇöpresumably to display all the tigers in our application. For more details on such routing options, see the Rails Guides article on ΓÇ£Rails Routing from the Outside InΓÇ¥.) A table of the routes generated by Listing 12.19 appears in Table 12.1; note the named routes for the following and followers pages, which weΓÇÖll put to

use momentarily.

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

Table 12.1: RESTful routes provided by the custom rules in resource in Listing 12.19.

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.) We could write tests for the user profile page, since the stats partial will appear there, but it will also appear on the Home page, and this is a nice opportunity to refactor the Home page tests to take into account users signing in. The result appears in Listing 12.20.

Listing 12.20. Testing the following/follower statistics on the Home page.

spec/controllers/pages_controller_spec.rb

```
describe PagesController do
  .
  .
  .
  describe "GET 'home'" do

    describe "when not signed in" do

      before(:each) do
        get :home
      end

      it "should be successful" do
        response.should be_success
      end

      it "should have the right title" do
        response.should have_tag("title", "#{@base_title} | Home")
      end
    end

    describe "when signed in" do

      before(:each) do
        @user = test_sign_in(Factory(:user))
        other_user = Factory(:user, :email => Factory.next(:email))
        other_user.follow!(@user)
      end

      it "should have the right follower/following counts" do
        get :home
        response.should have_tag("a[href=?]", following_user_path(@user),
                                 /0 following/)
        response.should have_tag("a[href=?]", followers_user_path(@user),
                                 /1 follower/)
      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 URLs:

```
response.should have_tag("a[href=?]", following_user_path(@user),
                                 /0 following/)
```

```
response.should have_tag("a[href=?]", followers_user_path(@user),
                                 /1 follower/)
```

Here we have used the named routes shown in Table 12.1 to verify that the links have the right URLs.

The application code for the stats partial is just a table inside a div, as shown in Listing 12.21.
Listing 12.21. A partial for displaying follower stats.
app/views/pages/_stats.html.erb

```
<% @user ||= current_user %>
<div class="stats">
  <table>
    <tr>
      <td>
        <a href="<%= following_user_path(@user) %>">
          <span id="following" class="stat">
            <%= @user.following.count %> following
          </span>
        </a>
      </td>
      <td>
        <a href="<%= followers_user_path(@user) %>">
          <span id="followers" class="stat">
            <%= pluralize(@user.followers.count, "follower") %>
          </span>
        </a>
      </td>
    </tr>
  </table>
</div>
```

Here the user following and follower counts are calculated through the associations using

```
@user.following.count
```

and

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

Compare these to the microposts count from Listing 11.16, where we wrote

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

to count the microposts.

Since we will be including the stats on both the user show pages and the home page, the first line of Listing 12.21 picks the right one using

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

As discussed in Box 9.4, 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.

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

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

This is for the benefit of the Ajax implementation in Section 12.2.5, which accesses elements on the page using their unique ids.

With the partial in hand, including the stats on the Home page (thereby getting the test in Listing 12.20 to pass) is easy, as shown in Listing 12.22. The result appears in Figure 12.11.
Listing 12.22. Adding follower stats to the Home page.
app/views/pages/home.html.erb

```
<% if signed_in? %>
        .
        .
        .
        <%= render 'pages/user_info' %>
        <%= render 'pages/stats' %>
      </td>
    </tr>
  </table>
<% else %>
  .
  .
  .
<% end %>
```

home_page_follow_stats
Figure 12.11: The Home page (/) with follow stats. (full size)

WeΓÇÖll render the stats partial on the profile page in a moment, but first letΓÇÖs make a partial for the follow/unfollow button, as shown in Listing 12.23.
Listing 12.23. 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 11.21), as seen in Listing 12.24.
Listing 12.24. Adding the routes for user relationships.
config/routes.rb

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

The follow/unfollow partials themselves are shown in Listing 12.25 and Listing 12.26.
Listing 12.25. 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>
  <div class="actions"><%= f.submit "Follow" %></div>
<% end %>
```

Listing 12.26. 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| %>
  <div><%= f.hidden_field :followed_id %></div>
  <div class="actions"><%= f.submit "Unfollow" %></div>
<% end %>
```

These two forms both use form_for to manipulate a Relationship model object; the main difference between the two is that Listing 12.25 builds a new relationship, whereas Listing 12.26 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. (WeΓÇÖll write these actions in Section 12.2.4.) Finally, youΓÇÖll note that the follow/unfollow form doesnΓÇÖt have any content other than the button, but it still needs to send the followed_id, which we accomplish with hidden_field; this produces HTML of the form

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

which 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 12.27. Profiles with follow and unfollow buttons, respectively, appear in Figure 12.12 and Figure 12.13.
Listing 12.27. Adding the follow form and follower stats to the user profile page.
app/views/users/show.html.erb

```
<table class="profile">
  <tr>
    <td class="main">
      <h2>
        <%= gravatar_for @user %>
        <%= h @user.name %>
      </h2>
      <%= render 'follow_form' if signed_in? %>
      .
      .
      .
    </td>
    <td class="sidebar round">
      <strong>Name</strong> <%= h @user.name %><br />
      <strong>URL</strong> <%= link_to user_path(@user), @user %><br />
      <strong>Microposts</strong> <%= @user.microposts.count %>
      <%= render 'pages/stats' %>
    </td>
  </tr>
</table>
```

profile_follow_button

Figure 12.12: A user profile with a follow button (/users/8). (full size)
profile_unfollow_button
Figure 12.13: A user profile with an unfollow button (/users/6). (full size)

WeΓÇÖll get these buttons working soon enoughΓÇöin fact, weΓÇÖll do it two ways, the standard way (Section 12.2.4) and using Ajax (Section 12.2.5)ΓÇöbut first weΓÇÖll finish the HTML interface by making the following and followers pages.

12.2.3 Following and followers pages

Pages to display user following and followers will resemble a hybrid of the user profile page and the user index page (Section 10.3.1), with a sidebar of user information (including the following stats) and a table of users. In addition, weΓÇÖll include a raster of user profile image links in the sidebar. Mockups matching these requirements appear in Figure 12.14 (following) and Figure 12.15 (followers).
following_mockup
Figure 12.14: A mockup of the user following page. (full size)
followers_mockup
Figure 12.15: A mockup of the user followers page. (full size)

Our first step is to get the following and followers links to work. WeΓÇÖll follow TwitterΓÇÖs lead and have both pages to require user signin. For signed-in users, the pages should have links for following and followers, respectively. Listing 12.28 expresses these expectations in code.12
Listing 12.28. Test for the following and followers actions.
spec/controllers/users_controller_spec.rb

```
describe UsersController do
  .
  .
  .
 describe "follow pages" do

   describe "when not signed in" do

     it "should protect 'following'" do
       get :following
       response.should redirect_to(signin_path)
     end

     it "should protect 'followers'" do
       get :followers
       response.should redirect_to(signin_path)
     end
   end

   describe "when signed in" do

     before(:each) do
       @user = test_sign_in(Factory(:user))
       @other_user = Factory(:user, :email => Factory.next(:email))
       @user.follow!(@other_user)
     end

     it "should show user following" do
       get :following, :id => @user
       response.should have_tag("a[href=?]", user_path(@other_user),
                                @other_user.name)
     end

     it "should show user followers" do
```

```
 get :followers, :id => @other_user
       response.should have_tag("a[href=?]", user_path(@user), @user.name)
     end
   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 12.19, we need to call them following and followers. Each action needs to set a title, find the user, retrieve either @user.following or @user.followers (in paginated form), and then render the page. The result appears in Listing 12.29.

Listing 12.29. The following and followers actions.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_filter :authenticate, :except => [:show, :new, :create]
  .
  .
  .
  def following
    @title = "Following"
    @user = User.find(params[:id])
    @users = @user.following.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 12.30 covers them both.

Listing 12.30. The show_follow view used to render following and followers.

app/views/users/show_follow.html.erb

```
<table>
  <tr>
    <td class="main">
      <h2><%= @title %></h2>

      <% unless @users.empty? %>
        <ul class="users">
          <%= render @users %>
        </ul>
        <%= will_paginate @users %>
      <% end %>
    </td>
    <td class="sidebar round">
      <strong>Name</strong> <%= h @user.name %><br />
```

```
    <strong>URL</strong> <%= link_to user_path(@user), @user %><br />
    <strong>Microposts</strong> <%= @user.microposts.count %>
    <%= render 'pages/stats' %>
    <% unless @users.empty? %>
      <% @users.each do |user| %>
        <%= link_to gravatar_for(user, :size => 30), user %>
      <% end %>
    <% end %>
  </td>
 </tr>
</table>
```

ThereΓÇÖs a second detail in Listing 12.29 worth noting: in order to protect the pages for following and followers from unauthorized access, we have changed the authentication before filter to use :except instead of :only. So far in this tutorial, we have used :only to indicate which actions the filter gets applied to; with the addition of the new protected actions, the balance has shifted, and it is simpler to indicate which actions shouldnΓÇÖt be filtered. We do this with the :except option to the authenticate before filter:

```
before_filter :authenticate, :except => [:show, :new, :create]
```

With that, the tests should now be passing, and the pages should render as shown in Figure 12.16 (following) and Figure 12.17 (followers).
user_following
Figure 12.16: Showing the users being followed by the current user. (full size)
user_followers
Figure 12.17: Showing the current userΓÇÖs followers. (full size)

You might note that, even with the common show_followers partial, the following and followers actions still have a lot of duplication. Moreover, the show_followers partial itself shares common features with the user show page. Section 12.5 includes exercises to eliminate these sources of duplication.

12.2.4 A working follow button the standard way

Now that our views are in order, itΓÇÖs time to get the follow/unfollow buttons working. Since following a user creates a relationship, and unfollowing a user destroys a relationship, this involves writing the create and destroy actions for the Relationships controller. Naturally, both actions should be protected; for signed-in users, we will use the follow! and unfollow! utility methods defined in Section 12.1.4 to create and destroy the relevant relationships. These requirements lead to the tests in Listing 12.31.
Listing 12.31. Tests for the Relationships controller actions.
spec/controllers/relationships_controller_spec.rb

```
require 'spec_helper'

describe RelationshipsController do

  describe "access control" do
    it "should require signin for create" do
      post :create
      response.should redirect_to(signin_path)
    end

    it "should require signin for destroy" do
      delete :destroy
      response.should redirect_to(signin_path)
    end
  end

  describe "POST 'create'" do
```

```
    before(:each) do
      @user = test_sign_in(Factory(:user))
      @followed = Factory(:user, :email => Factory.next(:email))

      @user.should_receive(:follow!).with(@followed)
    end

    it "should create a relationship" do
      post :create, :relationship => { :followed_id => @followed }
      response.should be_redirect
    end
  end

  describe "DELETE 'destroy'" do

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

      @user.should_receive(:unfollow!).with(@followed)
    end

    it "should destroy a relationship" do
      delete :destroy, :relationship => { :followed_id => @followed }
      response.should be_redirect
    end
  end
end
```

Note here how

```
:relationship => { :followed_id => @followed }
```

simulates the submission of the form with hidden field given by

```
<%= f.hidden_field :followed_id %>
```

The controller code needed to get these tests to pass is remarkably concise: we just retrieve the followed user in a before filter, and then follow or unfollow the user using the relevant utility method. The full implementation appears in Listing 12.32. (Since we didnΓÇÖt generate it at the command line, you will have to create the Relationships controller file by hand.)

Listing 12.32. The Relationships controller.

app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_filter :authenticate
  before_filter :get_followed_user

  def create
    current_user.follow!(@user)
    redirect_to @user
  end

  def destroy
    current_user.unfollow!(@user)
    redirect_to @user
  end
```

```
  private

    def get_followed_user
      @user = User.find(params[:relationship][:followed_id])
    end
end
```

With that, the core follow/unfollow functionality is complete, and any user can follow (or unfollow) any other user.

12.2.5 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 12.2.4 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.13 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_remote_for
```

and Rails automagically uses Ajax.14 The updated partials appear in Listing 12.33 and Listing 12.34.

Listing 12.33. A form for following a user using Ajax.
app/views/users/_follow.html.erb

```
<% form_remote_for current_user.relationships.
                       build(:followed_id => @user.id) do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <div class="actions"><%= f.submit "Follow" %></div>
<% end %>
```

Listing 12.34. A form for unfollowing a user using Ajax.
app/views/users/_unfollow.html.erb

```
<% form_remote_for current_user.relationships.find_by_followed_id(@user),
                :html => { :method => :delete } do |f| %>
  <div><%= f.hidden_field :followed_id %></div>
  <div class="actions"><%= f.submit "Unfollow" %></div>
<% end %>
```

The actual HTML generated by this ERb isnΓÇÖt particularly relevant, but you might be curious, so hereΓÇÖs a peek:

```
<form action="/relationships" class="new_relationship"
     id="new_relationship" method="post"
     onsubmit="new Ajax.Request('/relationships',
              {asynchronous:true, evalScripts:true,
               parameters:Form.serialize(this)}); return false;">
  .
  .
  .
</form>
```

This uses the Prototype JavaScript Library to initiate an Ajax request back to the server without reloading the page. Incidentally, including raw JavaScript in the view is now considered bad style; most Rails developers prefer unobtrusive JavaScript, and Rails 3 will use unobtrusive JavaScript by default. See the Railscast on unobtrusive JavaScript for more information.

Having updated the form, we now need to arrange for the Relationships controller to respond to Ajax requests. WeΓÇÖll start with a couple simple tests. 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 12.35. This uses the xhr method (for ΓÇ£XmlHttpRequestΓÇ¥) 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 respond with a success status code when hit with an Ajax request. (To write more thorough test suites for Ajax-heavy applications, take a look at Selenium and Watir.)
Listing 12.35. Tests for the Relationships controller responses to Ajax requests.
spec/controllers/relationships_controller_spec.rb

```
describe RelationshipsController do
  .
  .
  .
  describe "POST 'create'" do
    .
    .
    .
    it "should respond to an Ajax request" do
      xhr :post, :create, :relationship => { :followed_id => @followed }
      response.should be_success
    end
  end

  describe "DELETE 'destroy'" do
    .
    .
    .
    it "should respond to an Ajax request" do
      xhr :delete, :destroy, :relationship => { :followed_id => @followed }
      response.should be_success
    end
  end
end
```

As implied by the tests, the application code uses the same create and delete 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 12.2.4) and respond to an Ajax request with JavaScript. The first step is to include the Prototype library into our application using javascript_include_tag :defaults, as shown in Listing 12.36.
Listing 12.36. Including the Prototype JavaScript library into the site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title><%= title %></title>
    <%= render 'layouts/stylesheets' %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    .
    .
    .
```

```
  </body>
</html>
```

The controller code then appears as in Listing 12.37.
Listing 12.37. Responding to Ajax requests in the Relationships controller.
app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_filter :authenticate
  before_filter :get_followed_user

  def create
    current_user.follow!(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end

  def destroy
    current_user.unfollow!(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end

  private

    def get_followed_user
      @user = User.find(params[:relationship][:followed_id])
    end
end
```

This code uses respond_to to take the appropriate action depending on the kind of request.15 The syntax is potentially confusing, and itΓÇÖs 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 Ruby JavaScript (RJS) file with the same name as the action, i.e., create.rjs or destroy.rjs. It is these files we need to create and edit in order to update the user profile page upon being followed or unfollowed.

Inside an RJS file, Rails automatically provides a variable called page to represent the page issuing the Ajax request. There are many methods available on this page object (do a Google search for ΓÇ£rails rjsΓÇ¥ for up-to-date examples), but we will need only one: replace_html. The replace_html method takes the unique CSS id as its first argument, and then uses JavaScript to replace the HTML inside that id with the result of its second argument. For example, Listing 12.38 shows how the create RJS file replaces the follow form with an unfollow form, and then replaces the followers statistic with an incremented count. The destroy RJS in Listing 12.39 simply reverses this process.
Listing 12.38. The Ruby JavaScript (RJS) to create a following relationship.
app/views/relationships/create.rjs

```
page.replace_html :follow_form, :partial => 'users/unfollow'
```

```
page.replace_html :followers, "#{@user.followers.count} followers"
```

Listing 12.39. The Ruby JavaScript (RJS) to destroy a following relationship.
app/views/relationships/destroy.rjs

```
page.replace_html :follow_form, :partial => 'users/follow'
page.replace_html :followers, "#{@user.followers.count} followers"
```

With that, you should navigate to a user profile page and verify that you can follow and unfollow without a page refresh.

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

12.3 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. Making the status feed involves assembling an array of the microposts from the users being followed by the current user, along with the current userΓÇÖs 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, itΓÇÖs especially important to have a sense of where weΓÇÖre going. A mockup of the final user status feed, which builds on the proto-feed from Section 11.3.3, appears in Figure 12.18.
home_page_feed_mockup
Figure 12.18: A mockup of a userΓÇÖs Home page with a status feed. (full size)

12.3.1 Motivation and strategy

The basic idea behind the feed is simple. Figure 12.19 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.
user_feed
Figure 12.19: 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, weΓÇÖll plan on implementing a method called from_users_followed_by, which we will use as follows:

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

Although we donΓÇÖt yet know how to implement it, we can already write tests for from_users_followed_by, as seen in Listing 12.40.

Listing 12.40. Tests for Micropost.from_users_followed_by.
spec/models/micropost_spec.rb

```
describe Micropost do
  .
  .
  .
  describe "from_users_followed_by" do

    before(:each) do
      @other_user = Factory(:user, :email => Factory.next(:email))
      @third_user = Factory(:user, :email => Factory.next(:email))

      @user_post  = @user.microposts.create!(:content => "foo")
      @other_post = @other_user.microposts.create!(:content => "bar")
      @third_post = @third_user.microposts.create!(:content => "baz")

      @user.follow!(@other_user)
    end
```

```
    it "should have a from_users_followed_by class method" do
      Micropost.should respond_to(:from_users_followed_by)
    end

    it "should include the followed user's microposts" do
      Micropost.from_users_followed_by(@user).
              include?(@other_post).should be_true
    end

    it "should include the user's own microposts" do
      Micropost.from_users_followed_by(@user).
              include?(@user_post).should be_true
    end

    it "should not include an unfollowed user's microposts" do
      Micropost.from_users_followed_by(@user).
              include?(@third_post).should be_false
    end
  end
end
```

The key here is building the associations in the before(:each) block and then checking all three requirements: microposts for followed users and the user itself are included, but a post from an unfollowed user is not.

The feed itself lives in the User model (Section 11.3.3), so we should add an additional test to the User model specs from Listing 11.31, as shown in Listing 12.41.

Listing 12.41. The final tests for the user feed.

spec/models/user_spec.rb

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

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

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

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

      it "should include the microposts of followed users" do
        followed = Factory(:user, :email => Factory.next(:email))
        mp3 = Factory(:micropost, :user => followed)
        @user.follow!(followed)
```

```
      @user.feed.include?(mp3).should be_true
    end
  end
  .
  .
  .
 end
end
```

Implementing the feed will be easy; we will simply defer to Micropost.from_users_followed_by, as shown in Listing 12.42.
Listing 12.42. 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
```

12.3.2 A first feed implementation

Now itΓÇÖs time to implement Micropost.from_users_followed_by, which for simplicity weΓÇÖll just refer to as ΓÇ£the feedΓÇ¥. Since the final result is rather intricate, weΓÇÖll 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 weΓÇÖll 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, weΓÇÖve 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 11.3.3 that Active Record accepts a :conditions parameter to accomplish the kind of select shown above, as illustrated in Listing 11.32. There, our select was very simple; we just picked out all the microposts with user id corresponding to the current user:

```
Micropost.all(conditions => ["user_id = ?", id])
```

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

```
all(:conditions => ["user_id in (#{followed_ids}) OR user_id = ?", user])
```

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

We see from these conditions that weΓÇÖll need an array of ids that a given user is following (or something equivalent). One way to do this is to use RubyΓÇÖs map method, available on any ΓÇ£enumerableΓÇ¥ object, i.e., any object (such as an Array or a Hash) that consists of a collection of elements.16 We saw an example of this method in Section 4.3.2; it works like this

```
$ script/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 thereΓÇÖs a shorthand notation using an ampersand & and a symbol corresponding to the method:17

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

At this point, you might be able to guess a way to get all the ids for the users being followed; what we need to do is call id on each element in user.following. For example, for the first user in the database this array appears as follows:

```
>> User.first.following.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]
```

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 last seen in the User class in Section 7.12). A proposed implementation along these lines appears in Listing 12.43.

Listing 12.43. 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_ids = user.following.map(&:id)
    all(:conditions => ["user_id IN (#{followed_ids}) OR user_id = ?", user])
  end
  .
  .
  .
end
```

Although the discussion leading up to Listing 12.43 was couched in hypothetical terms, it actually works! In fact, it might be good enough for most practical purposes. But itΓÇÖs 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?)

12.3.3 Scopes, subselects, and a lambda

As hinted at in the last section, the feed implementation in Section 12.3.2 doesnΓÇÖt 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, weΓÇÖll reimplement the status feed in a way that scales better with the number of followed users.

There are a couple of problems with the code in Section 12.3.2. First, the expression

```
followed_ids = user.following.map(&:id)
```

pulls all the followed users into memory, and creates an array the full length of the following list. Since the condition in Listing 12.43 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. Second, the method in Listing 12.43 always pulls out all the microposts and sticks them into a Ruby array. Although these microposts are paginated in the view (Listing 11.33), the array is still full-sized.18

What we really want is honest pagination than only pulls out 30 elements at a time.

The solution to both problems involves converting the feed from a class method to a scope, which is a Rails method for restricting database selects based on certain conditions. For example, to arrange for a method to select all the administrative users in our application, we could add a scope to the User model as follows:

```
class User < ActiveRecord::Base
  .
  .
  .
  named_scope :admin, :conditions => { :admin => true }
  .
  .
  .
end
```

As a result of this scope, the code

```
User.admin
```

would return an array of all the site admins.

The main reason scopes are better than plain class methods is that they can be chained with other methods, so that, for example,

```
User.admin.paginate(:page => 1)
```

actually paginates the admins in the database; if (for some odd reason) the site has 100 administrators, the code above will still only pull out the first 30.

The scope for the feed is a bit more complex than the one illustrated above: it needs an argument, namely, the user whose feed we need to generate. We can do this with an anonymous function, or lambda (discussed in Section 8.4.2), as shown in Listing 12.44.19
Listing 12.44. Improving from_users_followed_by.
app/models/micropost.rb

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

  # Return microposts from the users being followed by the given user.
  named_scope :from_users_followed_by, lambda { |user| followed_by(user) }

  private

    # Return an SQL condition for users followed by the given user.
    # We include the user's own id as well.
    def self.followed_by(user)
      followed_ids = user.following.map(&:id)
      { :conditions => ["user_id IN (#{followed_ids}) OR user_id = :user_id",
                        { :user_id => user }] }
    end
end
```

Since the conditions on the from_users_followed_by scope are rather long, we have defined an auxiliary function to handle it:

```
def self.followed_by(user)
  followed_ids = user.following.map(&:id)
  { :conditions => ["user_id IN (#{followed_ids}) OR user_id = :user_id",
                    { :user_id => user }] }
end
```

As preparation for the next step, we have replaced

```
:conditions => ["... OR user_id = ?", user]
```

with the equivalent

```
:conditions => ["... OR user_id = :user_id", { :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, using a hash, is more convenient.
Box 12.1.Percent paren

The code in this section uses the Ruby percent-parentheses construction, as in

```
  %(SELECT followed_id FROM relationships
    WHERE follower_id = :user_id)
```

You can think of %() as equivalent to double quotes, but capable of making multiline strings. (If you need a way to produce a multiline string without leading whitespace, do a Google search for ΓÇ£ruby here documentΓÇ¥.) Since %() supports string interpolation, it is particularly useful when you need to put double quotes in a string and interpolate at the same time. For example, the code

```
  >> foo = "bar"
  >> puts %(The variable "foo" is equal to "#{foo}".)
```

produces

```
  The variable "foo" is equal to "bar".
```

To get the same output with double-quoted strings, you would need to escape the internal double quotes with backslashes, as in

```
  >> "The variable \"foo\" is equal to \"#{foo}\"."
```

In this case, the %() syntax is more convenient since is gets you the same result without the explicit escaping.

The above discussion implies that we will be adding a second occurrence of user_id in the SQL query, and indeed this is the case. We can replace the Ruby code

```
followed_ids = user.following.map(&:id)
```

with the SQL snippet

```
followed_ids = %(SELECT followed_id FROM relationships
                 WHERE follower_id = :user_id)
```

(See Box 12.1 for an explanation of the %() syntax.) 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.20

With this foundation, we are ready for the final feed implementation, as seen in Listing 12.45.
Listing 12.45. The final implementation of from_users_followed_by.
app/models/micropost.rb

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

  # Return microposts from the users being followed by the given user.
  named_scope :from_users_followed_by, lambda { |user| followed_by(user) }

  private

    # Return an SQL condition for users followed by the given user.
    # We include the user's own id as well.
    def self.followed_by(user)
      followed_ids = %(SELECT followed_id FROM relationships
                       WHERE follower_id = :user_id)
      { :conditions => ["user_id IN (#{followed_ids}) OR user_id = :user_id",
                        { :user_id => user }] }
    end
end
```

This code has a formidable combination of Rails, Ruby, and SQL, but it does the job, and does it well.21
12.3.4 The new status feed

With the code in Listing 12.45, our status feed is complete. As a reminder, the code for the Home page appears in Listing 12.46; this code creates a paginated feed of the relevant microposts for use in the view, as seen in Figure 12.20.22 Note that the paginate method actually reaches all the way into the Micropost model method in Listing 12.45, arranging to pull out only 30 microposts at a time from the database.23
Listing 12.46. The home action with a paginated feed.
app/controllers/pages_controller.rb

```
class PagesController < ApplicationController

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

home_page_with_feed
Figure 12.20: The Home page with a working status feed. (full size)
12.4 Conclusion

With the addition of the status feed, weΓÇÖve finished the core sample application for 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, itΓÇÖs a good idea to merge in your changes and deploy the application:

```
$ git add .
$ git commit -am "Added user following"
$ git checkout master
$ git merge following-users
$ git push heroku
$ heroku rake db:migrate
```

12.4.1 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.

DonΓÇÖt be surprised if itΓÇÖs 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.24 If he has, it will often save you a ton of time to watch the relevant Railscast first. 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 bookΓÇÖs end-of-chapter exercises, so donΓÇÖt be discouraged if you canΓÇÖt 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://www.railstutorial.org/ and subscribe to the news feed to get the latest updates.

Replies

Twitter allows users to make ΓÇ£@repliesΓÇ¥, which are microposts whose first characters are the userΓÇÖs 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 ΓÇ£dΓÇ¥. 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. There is a Railscast on sending email to get you started. Beware that the main Rails library for sending email, Action Mailer, has gotten a major overhaul in Rails

3, as seen in the Railscast on Action Mailer in Rails 3. You might want to skip implementing email until you have learned Rails 3.

Password reminders

Currently, if our applicationΓÇÖs users forget their passwords, they have no way to retrieve them. Due to the one-way secure password hashing in Chapter 7, our application canΓÇÖt email the userΓÇÖs password, but it can send a link to a reset form. Introduce a PasswordReminders resource to implement this feature. For each reset, you should create a unique token and email it to the user. Visiting a URL with the token should then allow them to reset their password to a value of their choice.

Signup confirmation

Apart from an email regular expression, the sample application currently has no way to verify the validity of a userΓÇÖs email address. Add an email address verification step to confirm a userΓÇÖs signup. The new feature should create users in an inactive state, email the user an activation URL, and then change the user to an active state when the URL 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 web sites expose an Application Programmer Interface (API) so that third-party applications can get, post, put, and delete the applicationΓÇÖs resources. Implement such a REST API for the sample application. The solution will involve adding respond_to blocks (Section 12.2.5) to many of the applicationΓÇÖs 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 than 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.

12.4.2 Guide to further resources

There are a wealth of Rails resources in stores and on the webΓÇöindeed, the supply is so rich that it can be overwhelming, and it can be hard to know where to start. By now you know where to startΓÇöwith this book, of course. And if youΓÇÖve gotten this far, youΓÇÖre ready for almost anything else out there. Here are some suggestions:

* Ruby on Rails Tutorial screencasts: I will be preparing a full-length screencast course based on this book (covering both Rails 2.3 and Rails 3). In addition to covering all the material in the book, the screencasts will be filled with tips, tricks, and the kind of see-how-itΓÇÖs-done demos that are hard to capture in print. They will be available on the Ruby on Rails Tutorial website, through Safari Books Online, and through InformIT. Visit the Rails Tutorial website at http://www.railstutorial.org/ and sign up for the news feed to find out when the screencasts will be released.25
* Railscasts: ItΓÇÖs hard to overemphasize what a great resource the Railscasts are. I suggest starting by visiting the Railscasts episode archive and clicking on subjects that catch your eye.
* Scaling Rails: One topic weΓÇÖve 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 great place to start. I also recommend investigating the site monitoring applications Scout and New Relic.26 And, as you might suspect by now, there are Railscasts on many scaling subjects, including profiling, caching, and background jobs.
* Ruby and Rails books: As mentioned in Chapter 1, 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 Way by Obie Fernandez for more about Rails.
* PeepCode: I mentioned several commercial screencasters in Chapter 1, but the only one I have extensive experience with is PeepCode. The screencasts at PeepCode are consistently high-quality, and I warmly recommend them.

12.5 Exercises

1. Add tests for dependent :destroy in the Relationship model (Listing 12.5 and Listing 12.17) by following the example in Listing 11.11.
2. The following and followers actions in Listing 12.29 still have considerable duplication. Verify that the show_follow method in Listing 12.47 eliminates this duplication. (See if you can infer what the send method does, as in, e.g., @user.send(:following).)
3. Refactor Listing 12.30 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 12.20, write tests for the stats on the profile page.
5. Write an integration test for following and unfollowing a user.

Listing 12.47. Refactored following and followers actions.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def following
    show_follow(:following)
  end

  def followers
    show_follow(:followers)
  end

  def show_follow(action)
    @title = action.to_s.capitalize
    @user = User.find(params[:id])
    @users = @user.send(action).paginate(:page => params[:page])
    render 'show_follow'
  end
  .
  .
  .
end
```

┬½ Chapter 11 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/. Γåæ
2. For simplicity, Figure 12.6 suppresses the following tableΓÇÖs id column. Γåæ
3. Unfortunately, Rails uses connection for a database connection, so introducing a Connection model leads to some rather subtle bugs. (I learned this the hard way when developing Insoshi.) Γåæ
4. Indeed, this construction is so characteristic of Rails that well-knows Rails programmer Josh Susser used it as the name of his geek blog. Γåæ
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.) Γåæ
6. If youΓÇÖve noticed that followed_id also identifies a user, and are concerned about the asymmetric treatment of followed and follower, youΓÇÖre ahead of the game. WeΓÇÖll deal with this issue in Section 12.1.5. Γåæ
7. This follow! method should always work, so (following the model of create! and save!) we indicate with an exclamation point that an exception will be raised on failure. Γåæ
8. Once you have a lot of experience modeling a particular domain, you can often guess such utility methods in advance, and even when you canΓÇÖt youΓÇÖll often find yourself writing them to make the tests cleaner. In this case, though, itΓÇÖs OK if you wouldnΓÇÖt have guessed them. Software development is usually an iterative processΓÇöyou write code until it starts getting ugly, and then you refactor itΓÇöbut for brevity the tutorial presentation is streamlined a bit. Γåæ
9. The authenticate method is included simply to orient you within the User model file. Γåæ

10. The unfollow! method doesnΓÇÖt raise an exception on failureΓÇöin fact, I donΓÇÖt even know how Rails indicates a failed destroyΓÇöbut we use an exclamation point to maintain the follow!/unfollow! symmetry. Γåæ

11. You might notice that sometimes we access id explicitly, as in followed.id, and sometimes we just use followed. I shame to admit that my usual algorithm for telling when to leave it off is to see if it works without .id, and then add .id if it breaks. Γåæ

12. Everything in Listing 12.28 has been covered elsewhere in this tutorial, so this is a good exercise in reading code. Γåæ

13. Because it is nominally an acronym for asynchronous JavaScript and XML, Ajax is sometimes misspelled ΓÇ£AJAXΓÇ¥, even though the original Ajax article spells it as ΓÇ£AjaxΓÇ¥ throughout. Γåæ

14. This only works if JavaScript is enabled in the browser, but it degrades gracefully, working exactly as in Section 12.2.4 if JavaScript is disabled. Γåæ

15. There is no relationship between this respond_to and the respond_to used in the RSpec examples. Γåæ

16. The main requirement is that enumerable objects must implement an each method to iterate through the collection. Γåæ

17. 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? Γåæ

18. Calling paginate on an Array object converts it into a WillPaginate::Collection object, but that doesnΓÇÖt help us much since the entire array has already been created in memory. Γåæ

19. A function bundled with a piece of data (a user, in this case) is known as a closure, which we encountered briefly in the discussion of blocks in Section 4.3.2. Γåæ

20. For a more advanced way to create the necessary subselect, see the blog post ΓÇ£Hacking a subselect in ActiveRecordΓÇ¥. Γåæ

21. Of course, even the subselect wonΓÇÖt 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. Γåæ

22. In order make a prettier feed for Figure 12.20, IΓÇÖve added a few extra microposts by hand using the Rails console. Γåæ

23. You can verify this by examining the SQL statements in the development server log file. (The Rails Tutorial screencasts will cover such subtleties in more depth.) Γåæ

24. My only reservation about Railscasts is that they often omit the tests. This is probably necessary to keep the episodes nice and short, but you could get the wrong idea about the importance of tests. Once youΓÇÖve watched the relevant Railscast to get a basic idea of how to proceed, I suggest writing the new feature using test-driven development. Γåæ

25. Of course, by the time you read this, they might already be out! In that case, you should definitely buy them. Γåæ

26. In addition to being a clever phraseΓÇönew relic being a contradiction in termsΓÇöNew Relic is also an anagram for the name of the companyΓÇÖs founder, Lew Cirne. Γåæ