

(Figure 14.1) and navigates to the Users page (Figure 14.2) to select a user to follow. Calvin navigates to the profile of a second user, Thomas Hobbes (Figure 14.3), 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 14.4). Navigating to his home page, Calvin now sees an incremented “following” count and finds Hobbes’s microposts in his status feed (Figure 14.5). The rest of this chapter is dedicated to making this page flow actually work.

14.1 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 would 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**.

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

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

14.1.1 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 **hobbes.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’ll adopt Twitter’s convention and call them *following* (as in “50 following, 75 followers”), with a corresponding **calvin.following** array.

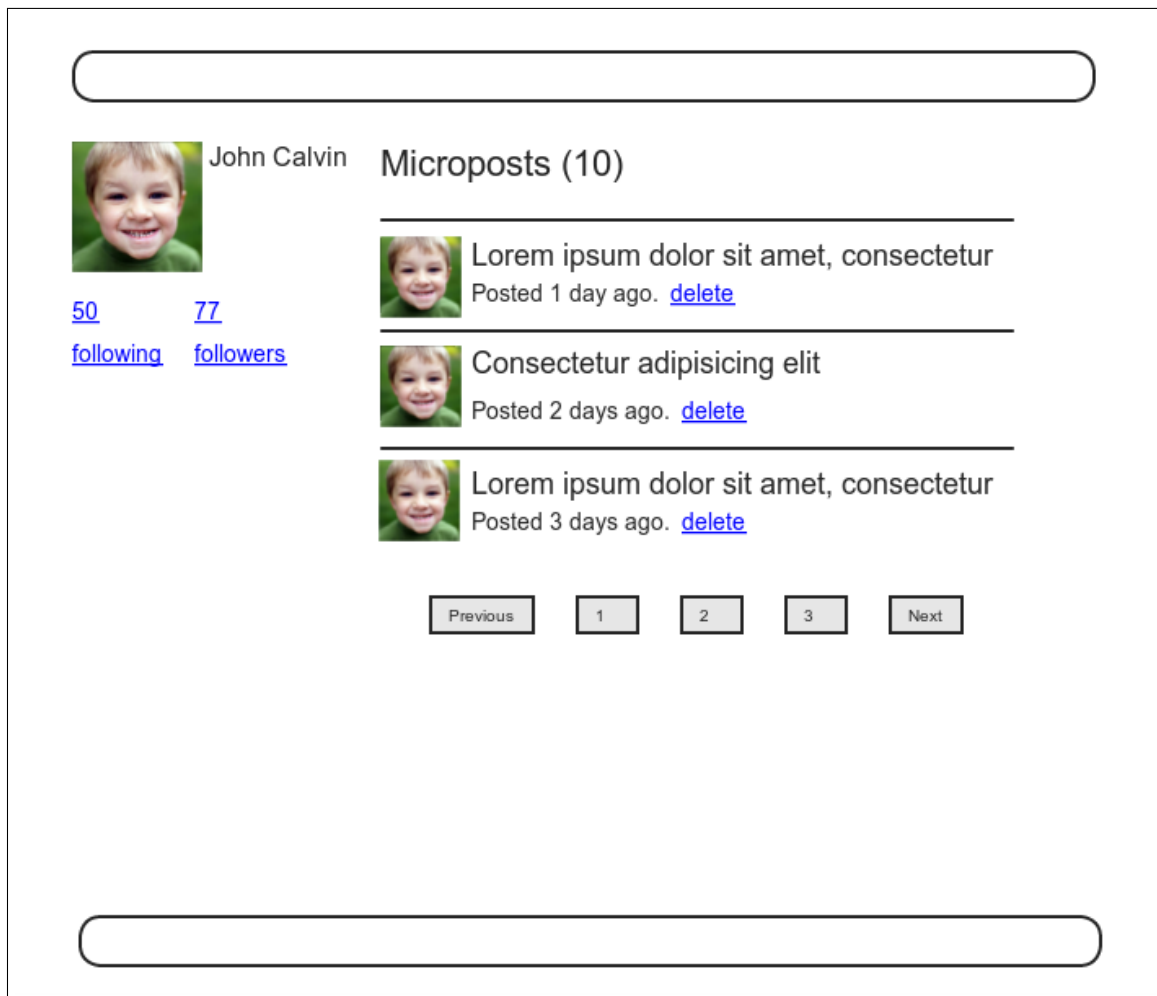


Figure 14.1: A current user's profile.

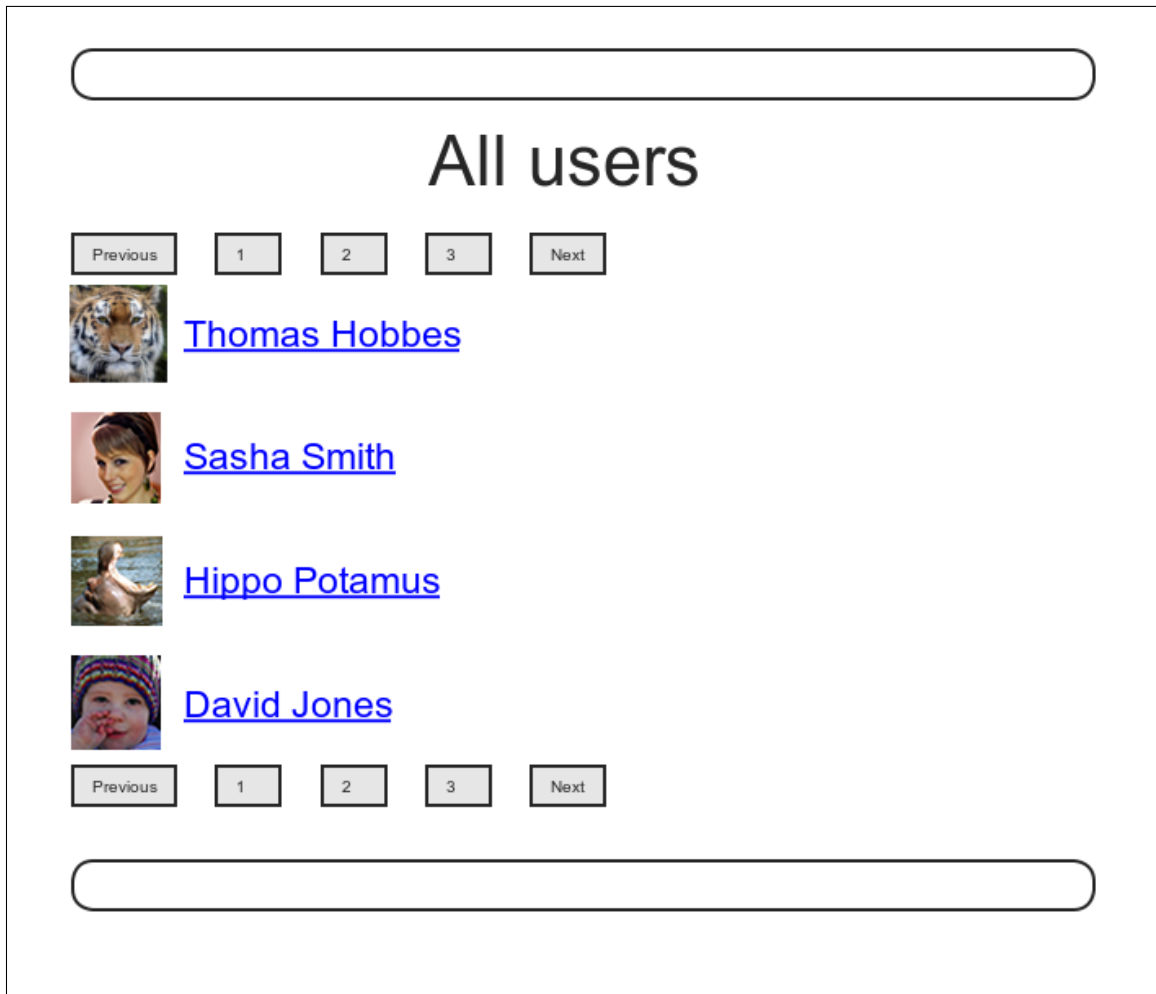


Figure 14.2: Finding a user to follow.

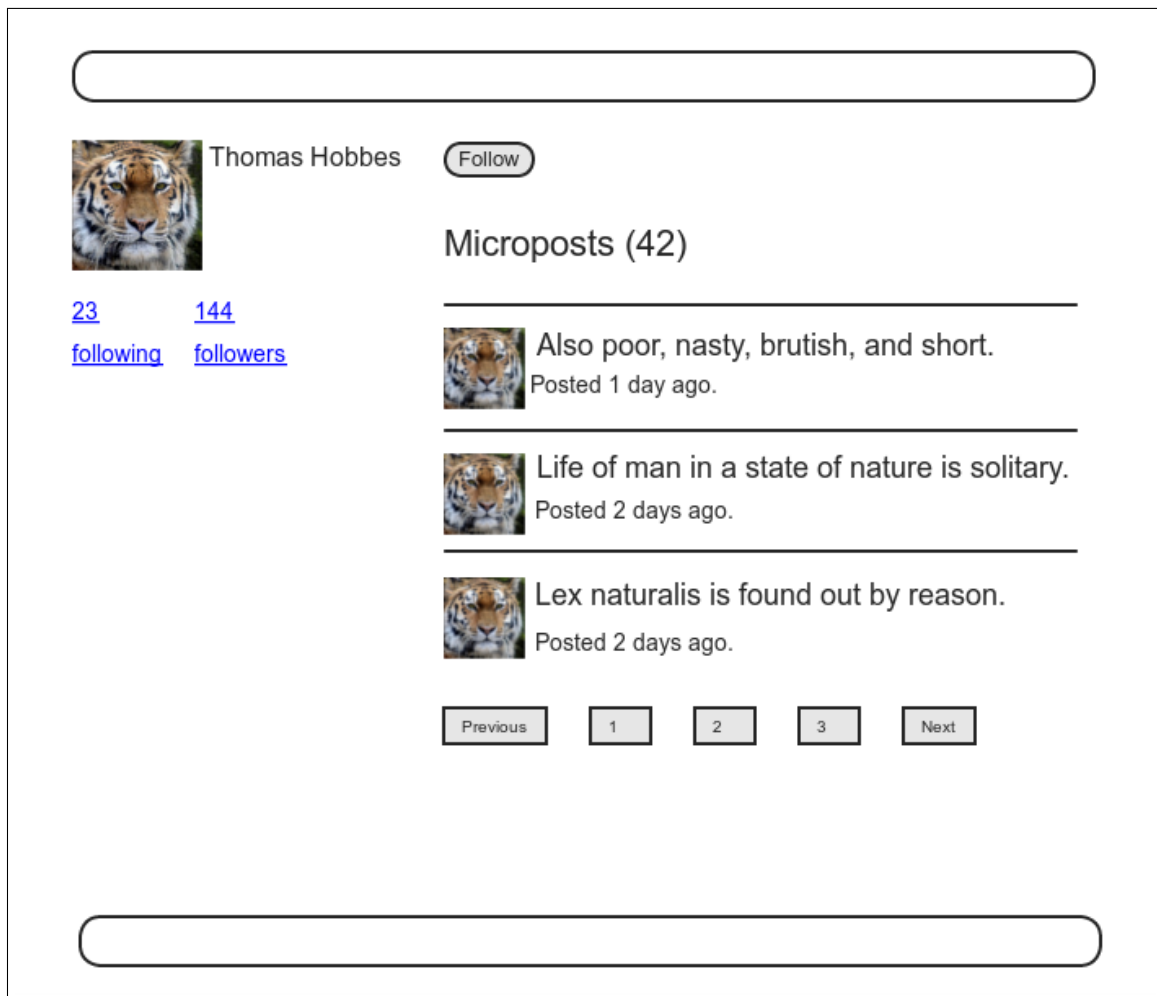




Figure 14.3: The profile of a user to follow, with a follow button.


Thomas Hobbes Unfollow

[23 following](#) [145 followers](#)

Microposts (42)

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

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

 Lex naturalis is found out by reason.
Posted 2 days ago.

Previous 1 2 3 Next

Figure 14.4: A profile with an unfollow button and incremented followers count.

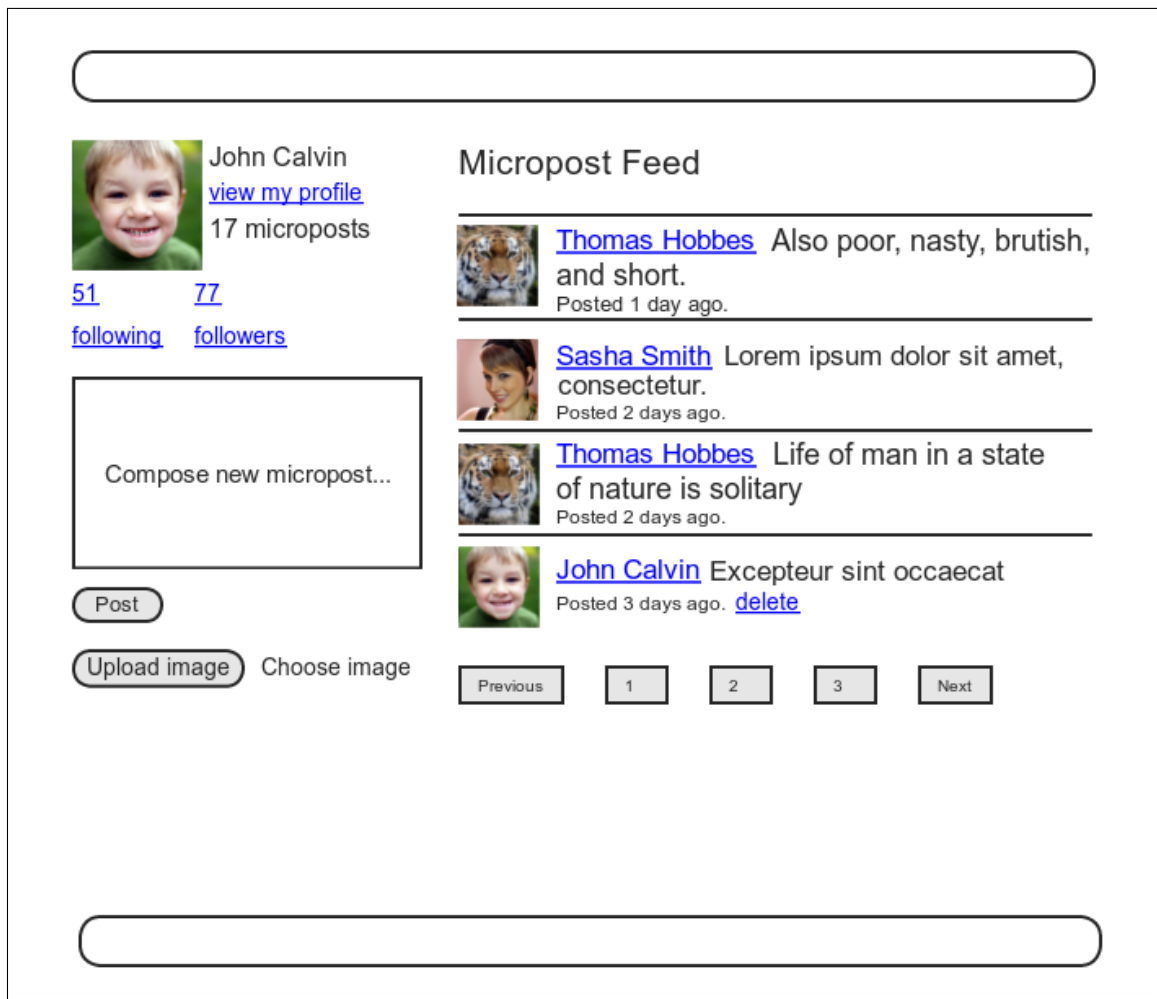


Figure 14.5: The Home page with status feed and incremented following count.

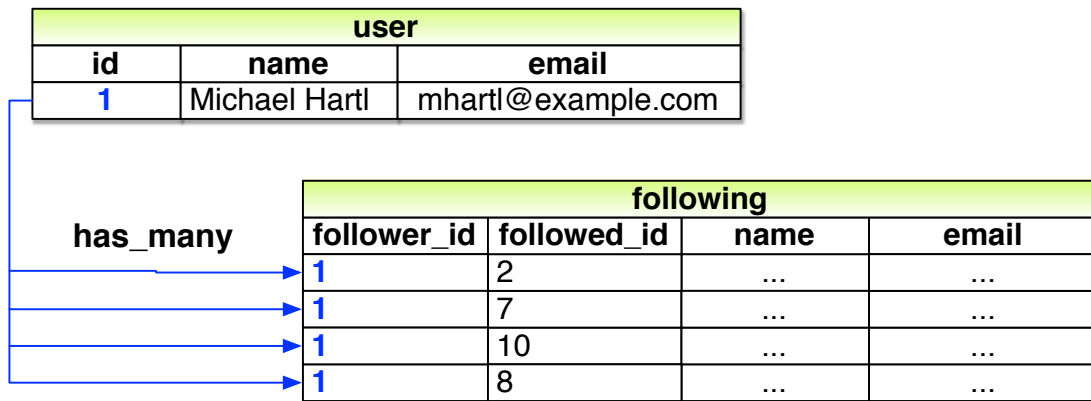


Figure 14.6: A naïve implementation of user following.

This discussion suggests modeling the followed users as in Figure 14.6, with a **following** table and a **has_many** association. Since **user.following** should be a collection 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 user’s other attributes, including the name, email, password, etc.

The problem with the data model in Figure 14.6 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 is *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) their 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 **following** and **followers** tables.

The problem here is that we are missing an underlying abstraction. One way to find the proper model 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

²For simplicity, Figure 14.6 omits the **following** table’s **id** column.

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* between two users. A user then has many relationships, and has many **following** (or **followers**) *through* these relationships.

There's an additional detail we need to address regarding our application's data model: unlike symmetric Facebook-style friendships, which are always reciprocal (at least at the data-model level), Twitter-style following relationships are potentially *asymmetric*—Calvin can follow Hobbes without Hobbes following Calvin. To distinguish between these two cases, we'll adopt the terminology of *active* and *passive* relationships: if Calvin is following Hobbes but not vice versa, Calvin has an active relationship with Hobbes and Hobbes has a passive relationship with Calvin.³

We'll focus now on using active relationships to generate a list of followed users, and consider the passive case in [Section 14.1.5](#). [Figure 14.6](#) suggests how to implement it: since each followed user is uniquely identified by **followed_id**, we could convert **following** to an **active_relationships** table, omit the user details, and use **followed_id** to retrieve the followed user from the **users** table. A diagram of the data model appears in [Figure 14.7](#).

Because we'll end up using the same database table for both active and passive relationships, we'll use the generic term *relationship* for the table name, with a corresponding Relationship model. The result is the Relationship data model shown in [Figure 14.8](#). We'll see starting in [Section 14.1.4](#) how to use the Relationship model to simulate both Active Relationship and Passive Relationship models.

To get started with the implementation, we first generate a migration corresponding to [Figure 14.8](#):

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

Because 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

³Thanks to reader Paul Fioravanti for suggesting this terminology.

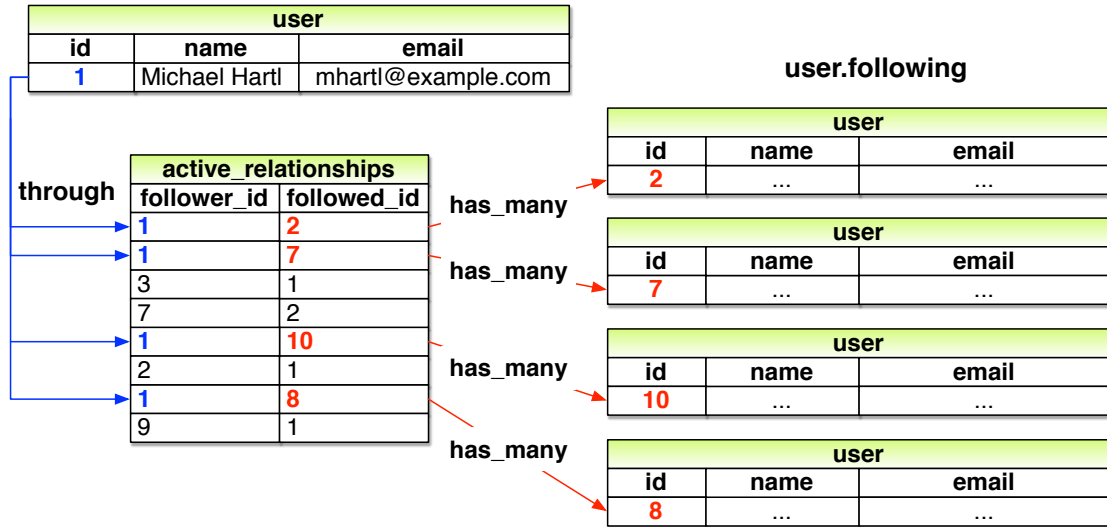


Figure 14.7: A model of followed users through active relationships.

relationships	
<code>id</code>	integer
<code>follower_id</code>	integer
<code>followed_id</code>	integer
<code>created_at</code>	datetime
<code>updated_at</code>	datetime

Figure 14.8: The Relationship data model.

Listing 14.1.

Listing 14.1: Adding indices for the `relationships` table.`db/migrate/[timestamp]_create_relationships.rb`

```
class CreateRelationships < ActiveRecord::Migration[6.0]
  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 14.1 also includes a multiple-key index that enforces uniqueness on (`follower_id`, `followed_id`) pairs, so that a user can't follow another user more than once. (Compare to the email uniqueness index from Listing 6.29 and the multiple-key index in Listing 13.3.) As we'll see starting in Section 14.1.4, 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 (for example, by using a command-line tool such as `curl`).

To create the `relationships` table, we migrate the database as usual:

```
$ rails db:migrate
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers [here](#).

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

1. For the user with id equal to `1` from Figure 14.7, what would the value of `user.following.map(&:id)` be? (Recall the `map(&:method_`

`name`) pattern from Section 4.3.2; `user.following.map(&:id)` just returns the array of ids.)

2. By referring again to Figure 14.7, determine the ids of `user.following` for the user with id equal to 2. What would the value of `user.following.map(&:id)` be for this user?

14.1.2 User/relationship associations

Before implementing user following 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 13.1.3, we will create new relationships using the user association, with code such as

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

At this point, you might expect application code as in Section 13.1.3, and it's similar, but there are two key differences.

First, in the case of the user/micropost association we could write

```
class User < ApplicationRecord
  has_many :microposts
  .
  .
  .
end
```

This works because by convention Rails looks for a Micropost model corresponding to the `:microposts` symbol.⁴ In the present case, though, we want to write

⁴Technically, Rails converts the argument of `has_many` to a class name using the `classify` method, which converts `"foo_bars"` to `"FooBar"`.

```
has_many :active_relationships
```

even though the underlying model is called Relationship. We will thus have to tell Rails the model class name to look for.

Second, before we wrote

```
class Micropost < ApplicationRecord
  belongs_to :user
  .
  .
  .
end
```

in the Micropost model. This works because the **microposts** table has a **user_id** attribute to identify the user (Section 13.1.1). 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, the user following another user is now identified with the foreign key **follower_id**, so we have to tell that to Rails.

The result of the above discussion is the user/relationship association shown in Listing 14.2 and Listing 14.3. As noted in the captions, the tests are currently **RED** (Why?);⁶ we'll fix this issue in Section 14.1.3.

Listing 14.2: Implementing the active relationships **has_many** association.

RED

app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts, dependent: :destroy
```

⁵Technically, 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**.

⁶*Answer:* As in Listing 6.30, the generated fixtures don't satisfy the validations, which causes the tests to fail.

Method	Purpose
<code>active_relationship.follower</code>	Returns the follower
<code>active_relationship.followed</code>	Returns the followed user
<code>user.active_relationships.create(followed_id: other_user.id)</code>	Creates an active relationship associated with user
<code>user.active_relationships.create!(followed_id: other_user.id)</code>	Creates an active relationship associated with user (exception on failure)
<code>user.active_relationships.build(followed_id: other_user.id)</code>	Returns a new Relationship object associated with user

Table 14.1: A summary of user/active relationship association methods.

```

has_many :active_relationships, class_name: "Relationship",
                               foreign_key: "follower_id",
                               dependent: :destroy
.
.
.
end

```

(Since destroying a user should also destroy that user's relationships, we've added **dependent: :destroy** to the association.)

Listing 14.3: Adding the follower **belongs_to** association to the Relationship model. **RED**

```

app/models/relationship.rb

class Relationship < ApplicationRecord
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end

```

The **followed** association isn't actually needed until [Section 14.1.4](#), but the parallel follower/followed structure is clearer if we implement them both at the same time.

The relationships in [Listing 14.2](#) and [Listing 14.3](#) give rise to methods analogous to the ones we saw in [Table 13.1](#), as shown in [Table 14.1](#).

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers [here](#).

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

1. Using the `create` method from [Table 14.1](#) in the console, create an active relationship for the first user in the database where the followed id is the second user.
2. Confirm that the values for `active_relationship.followed` and `active_relationship.follower` are correct.

14.1.3 Relationship validations

Before moving on, we'll add a couple of Relationship model validations for completeness. The tests ([Listing 14.4](#)) and application code ([Listing 14.5](#)) are straightforward. As with the generated user fixture from [Listing 6.30](#), the generated relationship fixture also violates the uniqueness constraint imposed by the corresponding migration ([Listing 14.1](#)). The solution—removing the fixture contents as in [Listing 6.31](#)—is also the same, as seen in [Listing 14.6](#).

Listing 14.4: Testing the Relationship model validations. RED

test/models/relationship_test.rb

```
require 'test_helper'

class RelationshipTest < ActiveSupport::TestCase

  def setup
    @relationship = Relationship.new(follower_id: users(:michael).id,
                                   followed_id: users(:archer).id)
  end

  test "should be valid" do
    assert @relationship.valid?
  end

  test "should require a follower_id" do
    @relationship.follower_id = nil
  end
end
```

```

    assert_not @relationship.valid?
  end

  test "should require a followed_id" do
    @relationship.followed_id = nil
    assert_not @relationship.valid?
  end
end

```

Listing 14.5: Adding the Relationship model validations. **RED**

app/models/relationship.rb

```

class Relationship < ApplicationRecord
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
  validates :follower_id, presence: true
  validates :followed_id, presence: true
end

```

Listing 14.6: Removing the contents of the relationship fixture. **GREEN**

test/fixtures/relationships.yml

```

# empty

```

At this point, the tests should be **GREEN**:

Listing 14.7: **GREEN**

```

$ rails test

```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers [here](#).

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

1. Verify by commenting out the validations in [Listing 14.5](#) that the tests still pass. (This is a change as of Rails 5, and in previous versions of Rails the

validations are required. We'll plan to leave them in for completeness, but it's worth bearing in mind that you may see these validations omitted in other people's code.)

14.1.4 Followed users

We come now to the heart of the Relationship associations: **following** and **followers**. Here we will use **has_many :through** for the first time: a user has many following *through* relationships, as illustrated in [Figure 14.7](#). 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, with code like

```
has_many :followeds, through: :active_relationships
```

Rails would see “followeds” and use the singular “followed”, assembling a collection using the **followed_id** in the **relationships** table. But, as noted in [Section 14.1.1](#), **user.followeds** is rather awkward, so we'll write **user.-following** instead. Naturally, Rails allows us to override the default, in this case using the **source** parameter (as shown in [Listing 14.8](#)), which explicitly tells Rails that the source of the **following** array is the set of **followed** ids.

Listing 14.8: Adding the User model **following** association.

app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts, dependent: :destroy
  has_many :active_relationships, class_name: "Relationship",
                                foreign_key: "follower_id",
                                dependent: :destroy
  has_many :following, through: :active_relationships, source: :followed
  .
  .
  .
end
```


The association defined in [Listing 14.8](#) leads to a powerful combination of Active Record and array-like behavior. For example, we can check if the followed users collection includes another user with the `include?` method ([Section 4.3.1](#)), or find objects through the association:

```
user.following.include?(other_user)
user.following.find(other_user)
```

We can also add and delete elements just as with arrays:

```
user.following << other_user
user.following.delete(other_user)
```

(Recall from [Section 4.3.1](#) that the shovel operator `<<` appends to the end of an array.)

Although in many contexts we can effectively treat `following` as an array, Rails is smart about how it handles things under the hood. For example, code like

```
following.include?(other_user)
```

looks like it might have to pull all the followed users out of the database to apply the `include?` method, but in fact for efficiency Rails arranges for the comparison to happen directly in the database. (Compare to the code in [Section 13.2.1](#), where we saw that

```
user.microposts.count
```

performs the count directly in the database.)

To manipulate following relationships, we'll introduce `follow` and `unfollow` utility methods so that we can write, e.g., `user.follow(other_`

user). We'll also add an associated **following?** boolean method to test if one user is following another.⁷

This is exactly the kind of situation where I like to write some tests first. The reason is that we are quite far from writing a working web interface for following users, but it's hard to proceed without some sort of *client* for the code we're developing. In this case, it's easy to write a short test for the User model, in which we use **following?** to make sure the user isn't following the other user, use **follow** to follow another user, use **following?** to verify that the operation succeeded, and finally **unfollow** and verify that it worked. The result appears in [Listing 14.9](#).

Listing 14.9: Tests for some “following” utility methods. **RED**

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  .
  .
  .
  test "should follow and unfollow a user" do
    michael = users(:michael)
    archer = users(:archer)
    assert_not michael.following?(archer)
    michael.follow(archer)
    assert michael.following?(archer)
    michael.unfollow(archer)
    assert_not michael.following?(archer)
  end
end
```

By treating the **following** association as an array, we can write the **follow**, **unfollow**, and **following?** methods as shown in [Listing 14.10](#). (Note that we have omitted the user **self** variable whenever possible.)

⁷Once 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.

Listing 14.10: Utility methods for following. GREEN*app/models/user.rb*

```
class User < ApplicationRecord
  .
  .
  .
  def feed
    .
    .
  end

  # Follows a user.
  def follow(other_user)
    following << other_user
  end

  # Unfollows a user.
  def unfollow(other_user)
    following.delete(other_user)
  end

  # Returns true if the current user is following the other user.
  def following?(other_user)
    following.include?(other_user)
  end

  private
  .
  .
  .
end
```

With the code in Listing 14.10, the tests should be GREEN:

Listing 14.11: GREEN

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers [here](#).

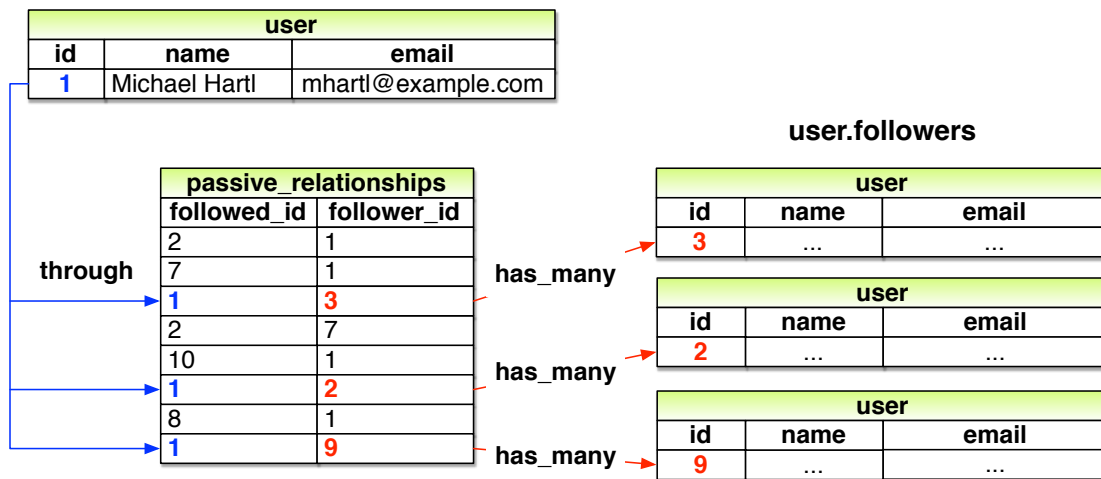


Figure 14.9: A model for user followers through passive relationships.

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

1. At the console, replicate the steps shown in [Listing 14.9](#).
2. What is the SQL for each of the commands in the previous exercise?

14.1.5 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 14.7](#) that all the information needed to extract an array of followers is already present in the `relationships` table (which we are treating as the `active_relationships` table via the code in [Listing 14.2](#)). Indeed, the technique is exactly the same as for followed users, with the roles of `follower_id` and `followed_id` reversed, and with `passive_relationships` in place of `active_relationships`. The data model then appears as in [Figure 14.9](#).

The implementation of the data model in [Figure 14.9](#) parallels [Listing 14.8](#) exactly, as seen in [Listing 14.12](#).

Listing 14.12: Implementing `user.followers` using passive relationships.
app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts, dependent: :destroy
  has_many :active_relationships, class_name: "Relationship",
                                foreign_key: "follower_id",
                                dependent: :destroy
  has_many :passive_relationships, class_name: "Relationship",
                                  foreign_key: "followed_id",
                                  dependent: :destroy
  has_many :following, through: :active_relationships, source: :followed
  has_many :followers, through: :passive_relationships, source: :follower
  .
  .
  .
end
```

It's worth noting that we could actually omit the `:source` key for `followers` in Listing 14.12, using simply

```
has_many :followers, through: :passive_relationships
```

This is because, in the case of a `:followers` attribute, Rails will singularize “followers” and automatically look for the foreign key `follower_id` in this case. Listing 14.12 keeps the `:source` key to emphasize the parallel structure with the `has_many :following` association.

We can conveniently test the data model above using the `followers.include?` method, as shown in Listing 14.13. (Listing 14.13 might have used a `followed_by?` method to complement the `following?` method, but it turns out we won't need it in our application.)

Listing 14.13: A test for `followers`. GREEN

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  .
```

```
.  
.br/>test "should follow and unfollow a user" do  
  michael = users(:michael)  
  archer = users(:archer)  
  assert_not michael.following?(archer)  
  michael.follow(archer)  
  assert michael.following?(archer)  
  assert archer.followers.include?(michael)  
  michael.unfollow(archer)  
  assert_not michael.following?(archer)  
end  
end
```

Listing 14.13 adds only one line to the test from Listing 14.9, but so many things have to go right to get it to pass that it's a very sensitive test of the code in Listing 14.12.

At this point, the full test suite should be **GREEN**:

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers [here](#).

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

1. At the console, create several followers for the first user in the database (which you should call **user**). What is the value of **user.followers.-map(&:id)**?
2. Confirm that **user.followers.count** matches the number of followers you created in the previous exercise.
3. What is the SQL used by **user.followers.count**? How is this different from **user.followers.to_a.count**? *Hint*: Suppose that the user had a million followers.