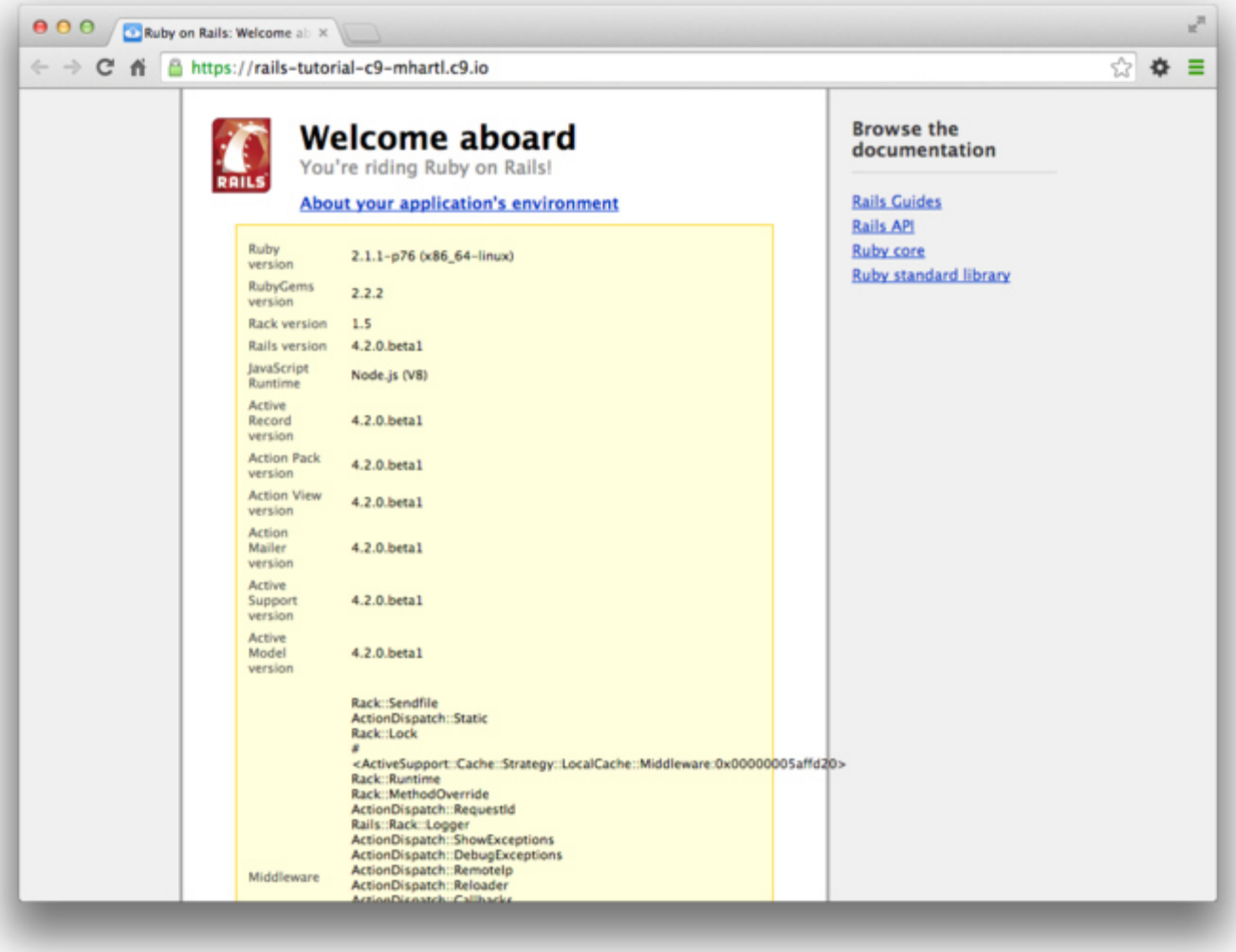

Figure 1.10: The default page with the application's environment.

1.3.3 Model-View-Controller (MVC)

Even at this early stage, it's helpful to get a high-level overview of how Rails applications work (Figure 1.11). You might have noticed that the standard Rails application structure (Figure 1.4) has an application directory called `app/` with three subdirectories: `models`,`views`, and `controllers`. This is a hint that Rails follows the model-view-controller (MVC) architectural pattern, which enforces a separation between "domain logic" (also called "business logic") from the input and presentation logic associated with a graphical user interface (GUI). In the case of web applications, the "domain logic" typically consists of data models for things like users, articles, and products, and the GUI is just a web page in a web browser.

When interacting with a Rails application, a browser sends a *request*, which is received by a web server and passed on to a Rails *controller*, which is in charge of what to do next. In some cases, the controller will immediately render a *view*, which

is a template that gets converted to HTML and sent back to the browser. More commonly for dynamic sites, the controller interacts with a *model*, which is a Ruby object that represents an element of the site (such as a user) and is in charge of communicating with the database. After invoking the model, the controller then renders the view and returns the complete web page to the browser as HTML.

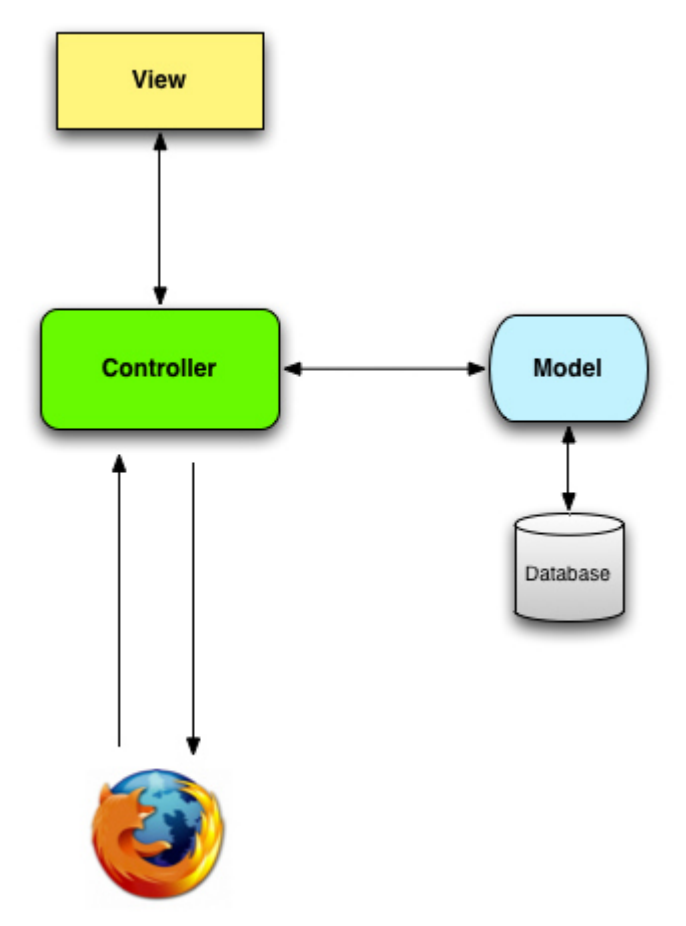

Figure 1.11: A schematic representation of the model-view-controller (MVC) architecture.

If this discussion seems a bit abstract right now, worry not; we'll refer back to this section frequently. Section 1.3.4 shows a first tentative application of MVC, while Section 2.2.2includes a more detailed discussion of MVC in the context of the toy app. Finally, the sample app will use all aspects of MVC; we'll cover controllers and views starting in Section 3.2, models starting in Section 6.1, and we'll see all three working together in Section 7.1.2.

1.3.4 Hello, world!

As a first application of the MVC framework, we'll make a wafer-thin change to the first app by adding a *controller action* to render the string "hello, world!". (We'll

learn more about controller actions starting in Section 2.2.2.) The result will be to replace the default Rails page from Figure 1.9 with the "hello, world" page that is the goal of this section.

As implied by their name, controller actions are defined inside controllers. We'll call our action `hello` and place it in the Application controller. Indeed, at this point the Application controller is the only controller we have, which you can verify by running

```
$ ls app/controllers/*_controller.rb
```

to view the current controllers. (We'll start creating our own controllers in Chapter 2.)Listing 1.8 shows the resulting definition of `hello`, which uses the `render` function to return the text "hello, world!". (Don't worry about the Ruby syntax right now; it will be covered in more depth in Chapter 4.)

Listing 1.8: Adding a `hello` action to the Application controller.*app/controllers/application_controller.rb*

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

  def hello
  render text: "hello, world!"
  end
end
```

Having defined an action that returns the desired string, we need to tell Rails to use that action instead of the default page in Figure 1.10. To do this, we'll edit the Rails *router*, which sits in front of the controller in Figure 1.11 and determines where to send requests that come in from the browser. (I've omitted the router from Figure 1.11 for simplicity, but we'll discuss the router in more detail starting in Section 2.2.2.) In particular, we want to change the default page, the *root route*, which determines the page that is served on the *root URL*.Because it's the URL for an address like http://www.example.com/ (where nothing comes after the final forward slash), the root URL is often referred to as / ("slash") for short.

As seen in Listing 1.9, the Rails routes file (`config/routes.rb`) includes a commented-out line that shows how to structure the root route. Here "welcome" is

the controller name and “index” is the action within that controller. To activate the root route, uncomment this line by removing the hash character and then replace it with the code in Listing 1.10, which tells Rails to send the root route to the `hello` action in the Application controller. (As noted in Section 1.1.2, vertical dots indicate omitted code and should not be copied literally.)

Listing 1.9: The default (commented-out) root route. *config/routes.rb*

```
Rails.application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  .
  .
  .
end
```

Listing 1.10: Setting the root route. *config/routes.rb*

```
Rails.application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  root 'application#hello'
  .
  .
  .
end
```

With the code from Listing 1.8 and Listing 1.10, the root route returns “hello, world!” as required (Figure 1.12).

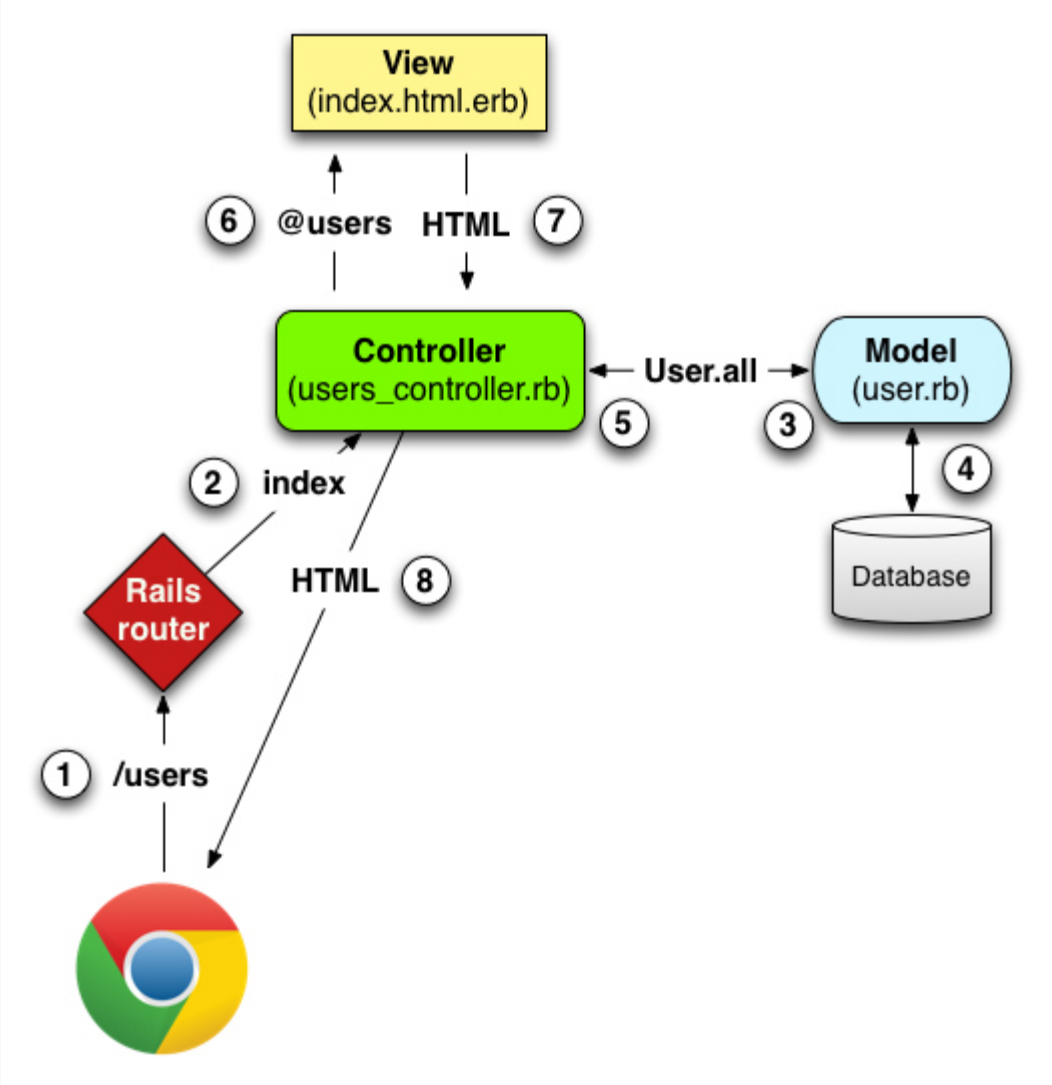

Figure 2.11: A detailed diagram of MVC in Rails.

Here is a summary of the steps shown in Figure 2.11:

1. The browser issues a request for the /users URL.
2. Rails routes /users to the `index` action in the Users controller.
3. The `index` action asks the User model to retrieve all users (`User.all`).
4. The User model pulls all the users from the database.
5. The User model returns the list of users to the controller.
6. The controller captures the users in the `@users` variable, which is passed to the `index`view.
7. The view uses embedded Ruby to render the page as HTML.
8. The controller passes the HTML back to the browser.[5]

Now let's take a look at the above steps in more detail. We start with a request issued from the browser—i.e., the result of typing a URL in the address bar or

12.1.1 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.

This discussion suggests modeling the followed users as in Figure 12.6, 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.

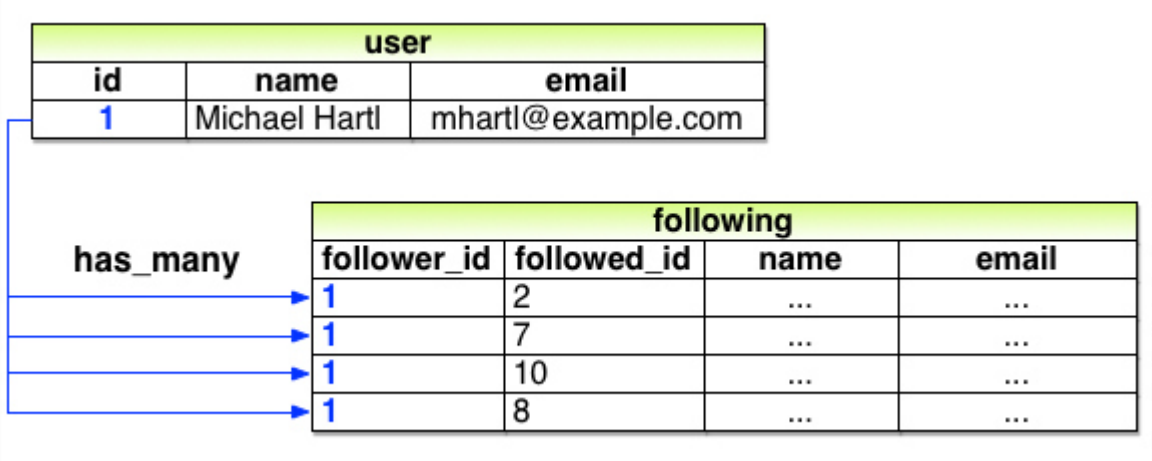

Figure 12.6: A naïve implementation of user following.

The problem with the data model in Figure 12.6 is that it is terribly redundant: each row contains not only each followed 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 ask two questions: When a user follows another user, what is being created? When a user *un*follows another user, what is being destroyed? Upon reflection, we see that in these cases the application should either create or destroy a *relationship* between two users. A user then has many relationships, and has many `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 12.1.5. Figure 12.6 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 12.7.

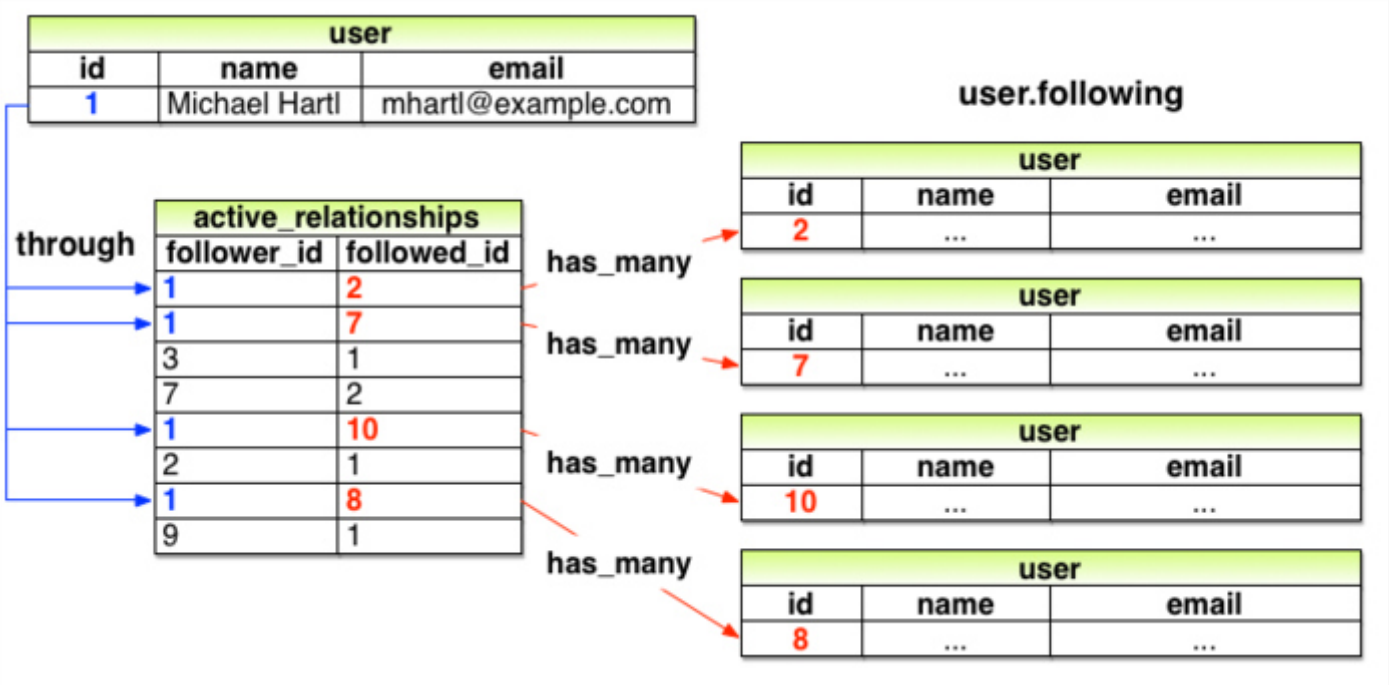

Figure 12.7: A model of followed users through active relationships.

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 12.8. We'll see starting inSection 12.1.4 how to use the Relationship model to simulate both Active Relationship and Passive Relationship models.

relationships	
id	integer
follower_id	integer
followed_id	integer
created_at	datetime
updated_at	datetime

Figure 12.8: The Relationship data model.

To get started with the implementation, we first generate a migration corresponding toFigure 12.8:

```
$ 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 Listing 12.1.

Listing 12.1: Adding indices for the `relationships` table. *db/migrate/[timestamp]_create_relationships.rb*

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

      t.timestamps null: false
    end
  add_index :relationships, :follower_id
  add_index :relationships, :followed_id
  add_index :relationships, [:follower_id, :followed_id], unique: true
  end
end
```

Listing 12.1 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 6.28 and the multiple-key index in Listing 11.1.) As we'll see starting in Section 12.1.4, 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:

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

12.1.2 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 11.1.3, 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 11.1.3, and it's similar, but there are two key differences.

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

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

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

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

in the Micropost model. This works 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 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 12.2and Listing 12.3.

Listing 12.2: Implementing the active relationships `has_many` association.*app/models/user.rb*

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
 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 12.3: Adding the follower `belongs_to` association to the Relationship model.*app/models/relationship.rb*

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end
```

The `followed` association isn't actually needed until Section 12.1.5, but the parallel follower/followed structure is clearer if we implement them both at the same time.

The relationships in Listing 12.2 and Listing 12.3 give rise to methods analogous to the ones we saw in Table 11.1, as shown in Table 12.1.

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

Table 12.1: A summary of user/active relationship association methods.

By referring to the methods in Table 12.1, we can write the `follow`, `unfollow`, and `following?` methods using the association with `following`, as shown in Listing 12.10. (Note that we have omitted the user `self` variable whenever possible.)

Listing 12.10: Utility methods for following. GREEN *app/models/user.rb*

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

  # Follows a user.
  def follow(other_user)
    active_relationships.create(followed_id: other_user.id)
  end

  # Unfollows a user.
  def unfollow(other_user)
    active_relationships.find_by(followed_id: other_user.id).destroy
  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 12.10, the tests should be GREEN:

Listing 12.11: GREEN

```
$ bundle exec rake test
```

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 (which we are treating as the `active_relationships` table via the code in Listing 12.2). 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 12.9.

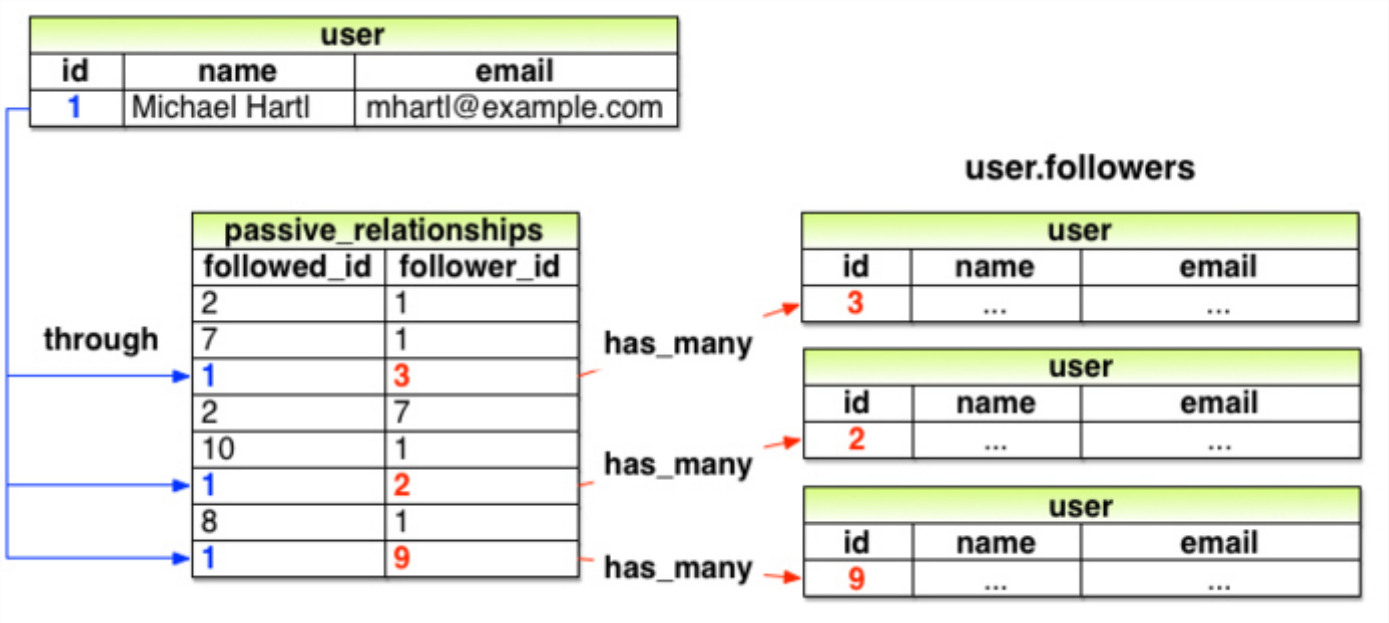

Figure 12.9: A model for user followers through passive relationships.

The implementation of the data model in Figure 12.9 parallels Listing 12.8 exactly, as seen in Listing 12.12.

Listing 12.12: Implementing `user.followers` using passive relationships. *app/models/user.rb*

```
class User < ActiveRecord::Base
  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 12.12, 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 12.8 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 12.13. (Listing 12.13 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 12.13: A test for `followers`. GREEN *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)
    assert archer.followers.include?(michael)
    michael.unfollow(archer)
    assert_not michael.following?(archer)
  end
end
```

Listing 12.13 adds only one line to the test from Listing 12.9, 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 12.12.

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

```
$ bundle exec rake test
```