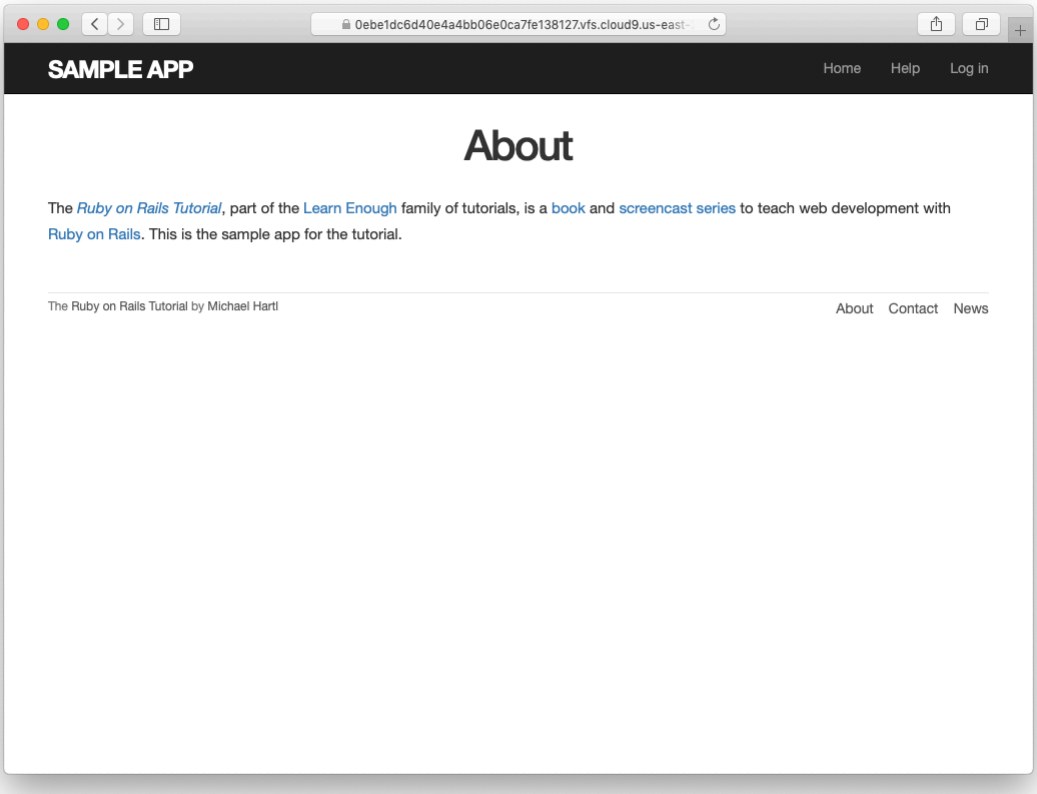


Figure 5.10: The About page at /about.

```
$ rails generate integration_test site_layout
      invoke  test_unit
      create    test/integration/site_layout_test.rb
```

Note that the Rails generator automatically appends **`_test`** to the name of the test file.

Our plan for testing the layout links involves checking the HTML structure of our site:

1. Get the root path (Home page).
2. Verify that the right page template is rendered.
3. Check for the correct links to the Home, Help, About, and Contact pages.

Listing 5.32 shows how we can use Rails integration tests to translate these steps into code, beginning with the **`assert_template`** method to verify that the Home page is rendered using the correct view.[18]

Listing 5.32: A test for the links on the layout. GREEN
test/integration/site_layout_test.rb

```
require 'test_helper'

class SiteLayoutTest < ActionDispatch::IntegrationTest

  test "layout links" do
    get root_path
    assert_template 'static_pages/home'
    assert_select "a[href=?]", root_path, count: 2
    assert_select "a[href=?]", help_path
    assert_select "a[href=?]", about_path
    assert_select "a[href=?]", contact_path
  end
end
```

[18] Some developers insist that a single test shouldn't contain multiple assertions. I find this practice to be unnecessarily complicated, while also incurring an extra overhead if there are common setup tasks needed before each test. In addition, a well-written test tells a coherent story, and breaking it up into individual pieces disrupts the narrative. I thus have a strong preference for including multiple assertions in a test, relying on Ruby (via minitest) to tell me the exact lines of any failed assertions.

Listing 5.32 uses some of the more advanced options of the **assert_select** method, seen before in Listing 3.26 and Listing 5.21. In this case, we use a syntax that allows us to test for the presence of a particular link–URL combination by specifying the tag name **a** and attribute **href**, as in

```
assert_select "a[href=?]", about_path
```

Here Rails automatically inserts the value of **about_path** in place of the question mark (escaping any special characters if necessary), thereby checking for an HTML tag of the form

```
<a href="/about">...</a>
```

Note that the assertion for the root path verifies that there are *two* such links (one each for the logo and navigation menu element):

```
assert_select "a[href=?]", root_path, count: 2
```

This ensures that both links to the Home page defined in Listing 5.30 are present.

Some more uses of **assert_select** appear in Table 5.2. While **assert_select** is flexible and powerful (having many more options than the ones shown here), experience shows that it's wise to take a lightweight approach by testing only HTML elements (such as site layout links) that are unlikely to change much over time.

To check that the new test in Listing 5.32 passes, we can run just the integration tests using the following Rake task:

Listing 5.33: GREEN

```
$ rails test:integration
```

If all went well, you should run the full test suite to verify that all the tests are GREEN:

Code	Matching HTML
`assert_select "div"`	`<div>foobar</div>`
`assert_select "div", "foobar"`	`<div>foobar</div>`
`assert_select "div.nav"`	`<div class="nav">foobar</div>`
`assert_select "div#profile"`	`<div id="profile">foobar</div>`
`assert_select "div[name=yo]"`	`<div name="yo">hey</div>`
`assert_select "a[href=?]", '/', count: 1`	`<a href="/">foo</a>`
`assert_select "a[href=?]", '/', text: "foo"`	`<a href="/">foo</a>`

Table 5.2: Some uses of `assert_select`.

Listing 5.34: GREEN

```
$ rails test
```

With the added integration test for layout links, we are now in a good position to catch regressions quickly using our test suite.

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. In the footer partial, change `about_path` to `contact_path` and verify that the tests catch the error.

2. It's convenient to use the `full_title` helper in the tests by including the Application helper into the test helper, as shown in Listing 5.35. We can then test for the right title using code like Listing 5.36. This is brittle, though, because now any typo in the base title (such as "Ruby on Rails Tutoial") won't be caught by the test suite. Fix this problem by writing a direct test of the `full_title` helper, which involves creating a file to test the application helper and then filling in the code indicated with `FILL_IN` in Listing 5.37. (Listing 5.37 uses `assert_equal`

<expected>, <actual>, which verifies that the expected result matches the actual value when compared with the **==** operator.)

Listing 5.35: Including the Application helper in tests.
test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all
  include ApplicationHelper
  .
  .
  .
end
```

Listing 5.36: Using the **full_title** helper in a test. GREEN
test/integration/site_layout_test.rb

```
require 'test_helper'

class SiteLayoutTest < ActionDispatch::IntegrationTest

  test "layout links" do
    get root_path
    assert_template 'static_pages/home'
    assert_select "a[href=?]", root_path, count: 2
    assert_select "a[href=?]", help_path
    assert_select "a[href=?]", about_path
    assert_select "a[href=?]", contact_path
    get contact_path
    assert_select "title", full_title("Contact")
  end
end
```

Listing 5.37: A direct test of the **full_title** helper.
test/helpers/application_helper_test.rb

```
require 'test_helper'

class ApplicationHelperTest < ActionView::TestCase
```

```
  test "full title helper" do
    assert_equal full_title,         FILL_IN
    assert_equal full_title("Help"), FILL_IN
  end
end
```

5.4 User signup: A first step

As a capstone to our work on the layout and routing, in this section we'll make a route for the signup page, which will mean creating a second controller along the way. This is a first important step toward allowing users to register for our site; we'll take the next step, modeling users, in Chapter 6, and we'll finish the job in Chapter 7.

5.4.1 Users controller

We created our first controller, the Static Pages controller, in Section 3.2. It's time to create a second one, the Users controller. As before, we'll use **generate** to make the simplest controller that meets our present needs, namely, one with a stub signup page for new users. Following the conventional REST architecture favored by Rails, we'll call the action for new users **new**, which we can arrange to create automatically by passing **new** as an argument to **generate**. The result is shown in Listing 5.38.

Listing 5.38: Generating a Users controller (with a **new** action).

```
$ rails generate controller Users new
      create  app/controllers/users_controller.rb
       route  get 'users/new'
      invoke  erb
      create    app/views/users
      create    app/views/users/new.html.erb
      invoke  test_unit
      create    test/controllers/users_controller_test.rb
      invoke  helper
      create    app/helpers/users_helper.rb
      invoke    test_unit
      invoke  assets
```

```
      invoke    scss
      create      app/assets/stylesheets/users.scss
```

As required, Listing 5.38 creates a Users controller with a **new** action (Listing 5.39) and a stub user view (Listing 5.40). It also creates a minimal test for the new user page (Listing 5.41).

Listing 5.39: The initial Users controller, with a **new** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def new
  end
end
```

Listing 5.40: The initial **new** view for Users.
app/views/users/new.html.erb

```
<h1>Users#new</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

Listing 5.41: The generated test for the new user page. GREEN
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  test "should get new" do
    get users_new_url
    assert_response :success
  end
end
```

At this point, the tests should be GREEN:

Listing 5.42: 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. Per Table 5.1, change the route in Listing 5.41 to use **`signup_path`** instead of **`users_new_url`**.

2. The route in the previous exercise doesn't yet exist, so confirm that the tests are now RED. (This is intended to help us get comfortable with the RED/GREEN flow of Test Driven Development (TDD, Box 3.3); we'll get the tests back to GREEN in Section 5.4.2.)

5.4.2 Signup URL

With the code from Section 5.4.1, we already have a working page for new users at /users/new, but recall from Table 5.1 that we want the URL to be /signup instead. We'll follow the examples from Listing 5.27 and add a **`get '/signup'`** rule for the signup URL, as shown in Listing 5.43.

Listing 5.43: A route for the signup page. RED
config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  '/help',    to: 'static_pages#help'
  get  '/about',   to: 'static_pages#about'
  get  '/contact', to: 'static_pages#contact'
  get  '/signup',  to: 'users#new'
end
```

With the routes in Listing 5.43, we also need to update the test generated in Listing 5.38 with the new signup route, as shown in Listing 5.44.

Listing 5.44: Updating the Users controller test to use the signup route. GREEN
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  test "should get new" do
    get signup_path
    assert_response :success
  end
end
```

Next, we'll use the newly defined named route to add the proper link to the button on the Home page. As with the other routes, **`get 'signup'`** automatically gives us the named route **`signup_path`**, which we put to use in Listing 5.45. Adding a test for the signup page is left as an exercise (Section 5.3.2.)

Listing 5.45: Linking the button to the signup page.
app/views/static_pages/home.html.erb

```
<div class="center jumbotron">
  <h1>Welcome to the Sample App</h1>

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

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

<%= link_to image_tag("rails.svg", alt: "Rails logo", width: "200"),
                      "https://rubyonrails.org/" %>
```

Finally, we'll add a custom stub view for the signup page (Listing 5.46).

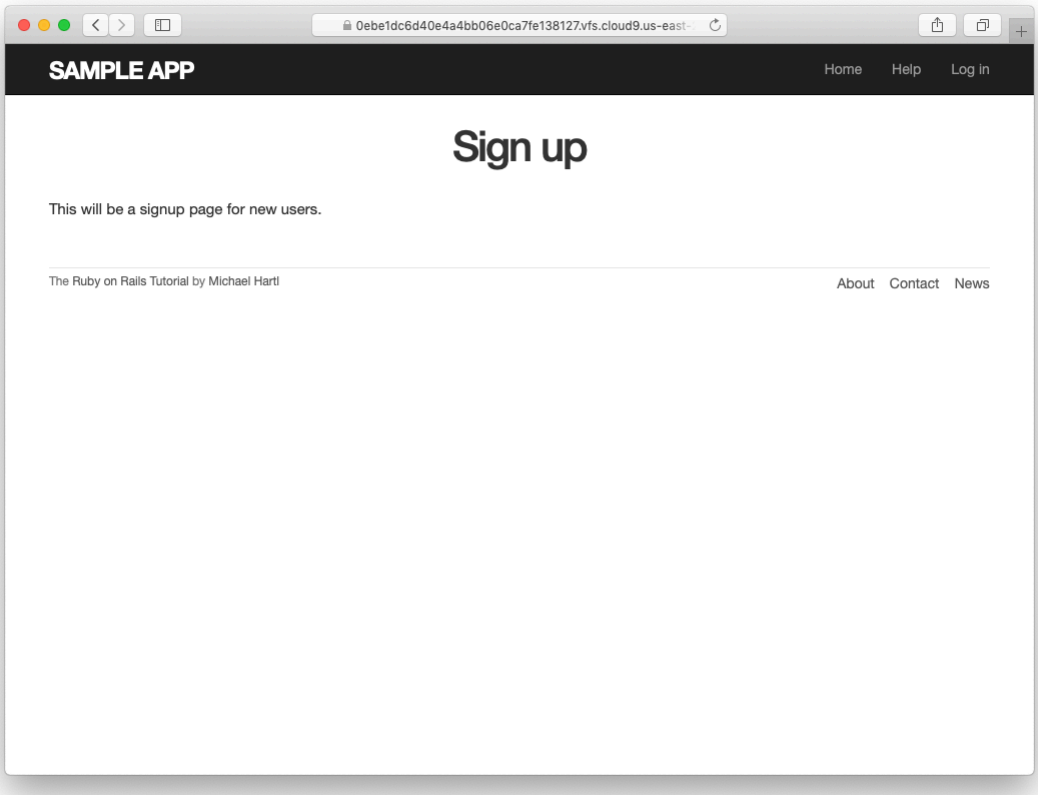


Figure 5.11: The new signup page at /signup.

Listing 5.46: The initial (stub) signup page.
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
<p>This will be a signup page for new users.</p>
```

With that, we're done with the links and named routes, at least until we add a route for logging in (Chapter 8). The resulting new user page (at the URL /signup) appears in Figure 5.11.

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. If you didn't solve the exercise in Section 5.4.1, change the test in Listing 5.41 to use the named route **`signup_path`**. Because of the route defined in Listing 5.43, this test should initially be GREEN.

2. In order to verify the correctness of the test in the previous exercise, comment out the **`signup`** route to get to RED, then uncomment to get to GREEN.

3. In the integration test from Listing 5.32, add code to visit the signup page using the **`get`** method and verify that the resulting page title is correct. *Hint*: Use the **`full_title`** helper as in Listing 5.36.

5.5 Conclusion

In this chapter, we've hammered our application layout into shape and polished up the routes. The rest of the book is dedicated to fleshing out the sample application: first, by adding users who can sign up, log in, and log out; next, by adding user microposts; and, finally, by adding the ability to follow other users.

At this point, if you are using Git, you should merge your changes back into the master branch:

```
$ git add -A
$ git commit -m "Finish layout and routes"
$ git checkout master
$ git merge filling-in-layout
```

Then push up to GitHub (running the test suite first for safety):

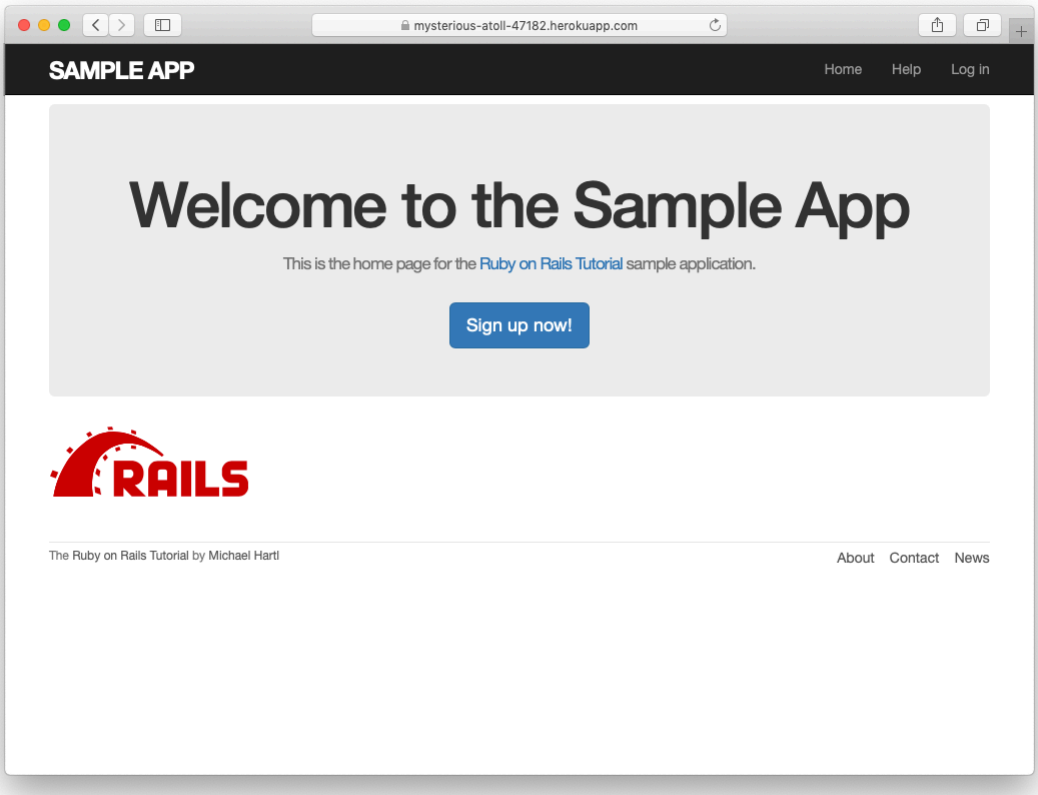


Figure 5.12: The sample application in production.

```
$ rails test
$ git push
```

Finally, deploy to Heroku:

```
$ git push heroku
```

The result of the deployment should be a working sample application on the production server (Figure 5.12).

5.5.1 What we learned in this chapter

- Using HTML5, we can define a site layout with logo, header, footer, and main body content.
- Rails partials are used to place markup in a separate file for convenience.
- CSS allows us to style the site layout based on CSS classes and ids.
- The Bootstrap framework makes it easy to make a nicely designed site quickly.
- Sass and the asset pipeline allow us to eliminate duplication in our CSS while packaging up the results efficiently for production.
- Rails allows us to define custom routing rules, thereby providing named routes.
- Integration tests effectively simulate a browser clicking from page to page.

Chapter 6

Modeling users

In Chapter 5, we ended with a stub page for creating new users (Section 5.4). Over the course of the next six chapters, we'll fulfill the promise implicit in this incipient signup page. In this chapter, we'll take the first critical step by creating a *data model* for users of our site, together with a way to store that data. In Chapter 7, we'll give users the ability to sign up for our site and create a user profile page. Once users can sign up, we'll let them log in and log out as well (Chapter 8 and Chapter 9), and in Chapter 10 (Section 10.2.1) we'll learn how to protect pages from improper access. Finally, in Chapter 11 and Chapter 12 we'll add account activations (thereby confirming a valid email address) and password resets. Taken together, the material in Chapter 6 through Chapter 12 develops a full Rails login and authentication system. As you may know, there are various pre-built authentication solutions for Rails; Box 6.1 explains why, at least at first, it's probably a better idea to roll your own.

Box 6.1. Rolling your own authentication system

Virtually all web applications require a login and authentication system of some sort. As a result, most web frameworks end up with one or more standardized libraries for doing so, and Rails is no exception. In particular, the Devise gem has emerged as a robust solution for a wide variety of uses, and represents a strong choice for professional-grade applications.

Nevertheless, I believe it is a mistake to use a pre-built system like Devise in a tutorial like this one. Off-the-shelf systems can be "black boxes" with potentially mysterious innards, and the complicated data models used by such systems would be utterly overwhelming for beginners (or even for experienced developers not familiar with data modeling). For learning purposes, it's essential to introduce the subject more gradually.

Happily, Rails makes it possible to take such a gradual approach while still developing an industrial-strength login and authentication system suitable for production applications. This way, even if you *do* end up using a third-party system later on, you'll be in a much better position to understand and modify it to meet your particular needs.

6.1 User model

Although the ultimate goal of the next three chapters is to make a signup page for our site (as mocked up in Figure 6.1), it would do little good now to accept information for new users: we don't currently have any place to put it. Thus, the first step in signing up users is to make a data structure to capture and store their information.

In Rails, the default data structure for a data model is called, naturally enough, a *model* (the M in MVC from Section 1.2.3). The default Rails solution to the problem of persistence is to use a *database* for long-term data storage, and the default library for interacting with the database is called *Active Record*.[1] Active Record comes with a host of methods for creating, saving, and finding data objects, all without having to use the Structured Query Language (SQL)[2] used by relational databases. Moreover, Rails has a feature called *migrations*

[1]The name comes from the "active record pattern", identified and named in *Patterns of Enterprise Application Architecture* by Martin Fowler.

[2]Officially pronounced "ess-cue-ell", though the alternate pronunciation "sequel" is also common. You can differentiate an individual author's preference by the choice of indefinite article: those who write "a SQL database" prefer "sequel", whereas those who write "an SQL database" prefer "ess-cue-ell". As you'll soon see, I prefer the latter.

Figure 6.1: A mockup of the user signup page.

to allow data definitions to be written in pure Ruby, without having to learn an SQL data definition language (DDL). The effect is that Rails insulates you almost entirely from the details of the database. In this book, by using SQLite for development and PostgreSQL (via Heroku) for deployment (Section 1.4), we have developed this theme even further, to the point where we barely ever have to think about how Rails stores data, even for production applications.

As usual, if you're following along using Git for version control, now would be a good time to make a topic branch for modeling users:

```
$ git checkout -b modeling-users
```

6.1.1 Database migrations

You may recall from Section 4.4.5 that we have already encountered, via a custom-built **User** class, user objects with **name** and **email** attributes. That class served as a useful example, but it lacked the critical property of *persistence*: when we created a User object at the Rails console, it disappeared as soon as we exited. Our goal in this section is to create a model for users that won't disappear quite so easily.

As with the User class in Section 4.4.5, we'll start by modeling a user with two attributes, a **name** and an **email** address, the latter of which we'll use as a unique username.[3] (We'll add an attribute for passwords in Section 6.3.) In Listing 4.17, we did this with Ruby's **attr_accessor** method:

```
class User
  attr_accessor :name, :email
  .
  .
  .
end
```

In contrast, when using Rails to model users we don't need to identify the attributes explicitly. As noted briefly above, to store data Rails uses a relational

[3] By using an email address as the username, we open the possibility of communicating with our users at a future date (Chapter 11 and Chapter 12).

users		
id	name	email
1	Michael Hartl	mhartl@example.com
2	Sterling Archer	archer@example.gov
3	Lana Kane	lana@example.gov
4	Mallory Archer	boss@example.gov

Figure 6.2: A diagram of sample data in a `users` table.

users	
id	integer
name	string
email	string

Figure 6.3: A sketch of the User data model.

database by default, which consists of *tables* composed of data *rows*, where each row has *columns* of data attributes. For example, to store users with names and email addresses, we'll create a `users` table with `name` and `email` columns (with each row corresponding to one user). An example of such a table appears in Figure 6.2, corresponding to the data model shown in Figure 6.3. (Figure 6.3 is just a sketch; the full data model appears in Figure 6.4.) By naming the columns `name` and `email`, we'll let Active Record figure out the User object attributes for us.

You may recall from Listing 5.38 that we created a Users controller (along with a `new` action) using the command

```
$ rails generate controller Users new
```

The analogous command for making a model is **`generate model`**, which we can use to generate a User model with **`name`** and **`email`** attributes, as shown in Listing 6.1.

Listing 6.1: Generating a User model.

```
$ rails generate model User name:string email:string
      invoke  active_record
      create    db/migrate/<timestamp>_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
```

(Note that, in contrast to the plural convention for controller names, model names are singular: a *Users* controller, but a *User* model.) By passing the optional parameters **`name:string`** and **`email:string`**, we tell Rails about the two attributes we want, along with which types those attributes should be (in this case, **`string`**). Compare this with including the action names in Listing 3.7 and Listing 5.38.

One of the results of the **`generate`** command in Listing 6.1 is a new file called a *migration*. Migrations provide a way to alter the structure of the database incrementally, so that our data model can adapt to changing requirements. In the case of the User model, the migration is created automatically by the model generation script; it creates a **`users`** table with two columns, **`name`** and **`email`**, as shown in Listing 6.2. (We'll see starting in Section 6.2.5 how to make a migration from scratch.)

Listing 6.2: Migration for the User model (to create a **`users`** table).
db/migrate/[timestamp]_create_users.rb

```
class CreateUsers < ActiveRecord::Migration[6.0]
  def change
    create_table :users do |t|
```

```
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

Note that the name of the migration file is prefixed by a *timestamp* based on when the migration was generated. In the early days of migrations, the filenames were prefixed with incrementing integers, which caused conflicts for collaborating teams if multiple programmers had migrations with the same number. Barring the improbable scenario of migrations generated the same second, using timestamps conveniently avoids such collisions.

The migration itself consists of a **change** method that determines the change to be made to the database. In the case of Listing 6.2, **change** uses a Rails method called **create_table** to create a table in the database for storing users. The **create_table** method accepts a block (Section 4.3.2) with one block variable, in this case called **t** (for "table"). Inside the block, the **create_table** method uses the **t** object to create **name** and **email** columns in the database, both of type **string**.[4] Here the table name is plural (**users**) even though the model name is singular (User), which reflects a linguistic convention followed by Rails: a model represents a single user, whereas a database table consists of many users. The final line in the block, **t.timestamps**, is a special command that creates two *magic columns* called **created_at** and **updated_at**, which are timestamps that automatically record when a given user is created and updated. (We'll see concrete examples of the magic columns starting in Section 6.1.3.) The full data model represented by the migration in Listing 6.2 is shown in Figure 6.4. (Note the addition of the magic columns, which weren't present in the sketch shown in Figure 6.3.)

We can run the migration, known as "migrating up", using the **db:migrate** command as follows:

[4] Don't worry about exactly how the **t** object manages to do this; the beauty of *abstraction layers* is that we don't have to know. We can just trust the **t** object to do its job.

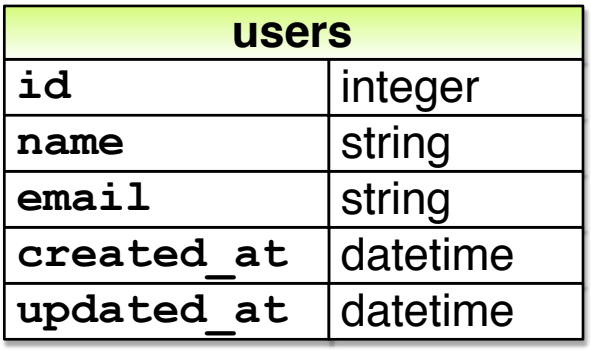


Figure 6.4: The User data model produced by Listing 6.2.

```
$ rails db:migrate
```

(You may recall that we ran this command in a similar context in Section 2.2.) The first time **db:migrate** is run, it creates a file called **db/development.sqlite3**, which is an SQLite[5] database. We can see the structure of the database by opening **development.sqlite3** with DB Browser for SQLite. (If you're using the cloud IDE, you should first download the database file to the local disk, as shown in Figure 6.5.) The result appears in Figure 6.6; compare with the diagram in Figure 6.4. You might note that there's one column in Figure 6.6 not accounted for in the migration: the **id** column. As noted briefly in Section 2.2, this column is created automatically, and is used by Rails to identify each row uniquely.

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.

[5] Officially pronounced "ess-cue-ell-ite", although the (mis)pronunciation "sequel-ite" is also common.

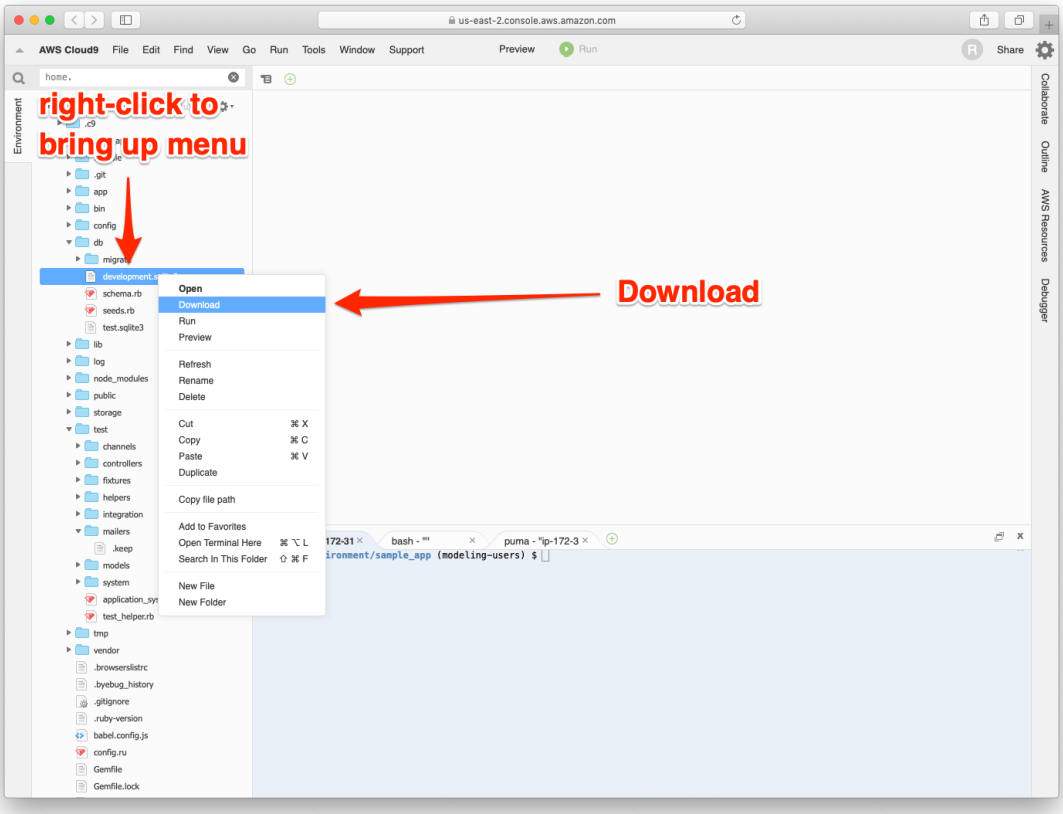


Figure 6.5: Downloading a file from the cloud IDE.

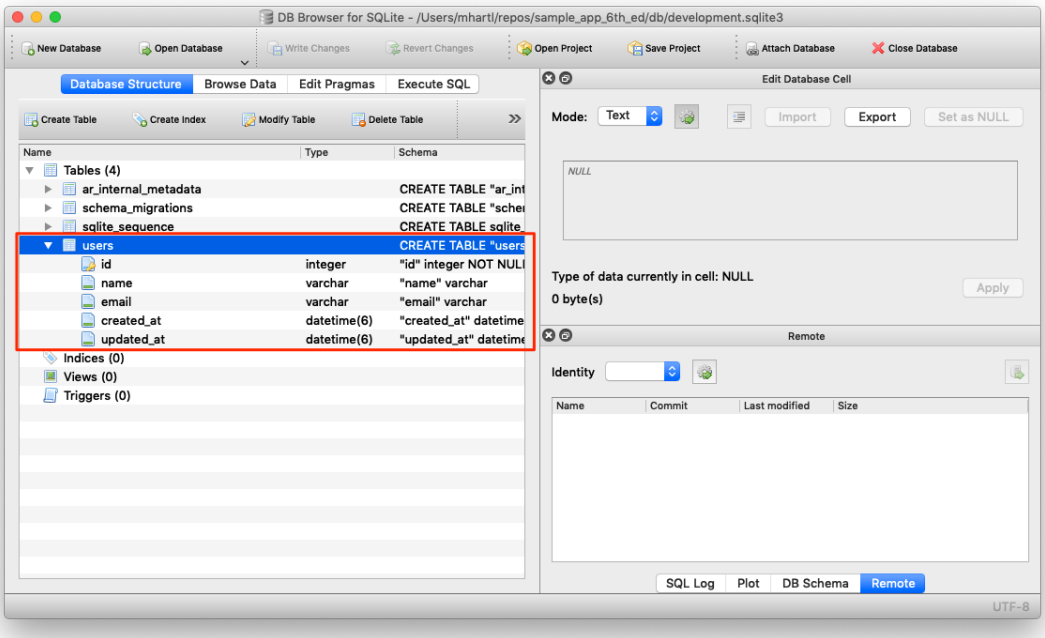


Figure 6.6: DB Browser with our new `users` table.

1. Rails uses a file called **schema.rb** in the **db/** directory to keep track of the structure of the database (called the *schema*, hence the filename). Examine your local copy of **db/schema.rb** and compare its contents to the migration code in Listing 6.2.

2. Most migrations (including all the ones in this tutorial) are *reversible*, which means we can "migrate down" and undo them with a single command, called **db:rollback**:

   ```
   $ rails db:rollback
   ```

   After running this command, examine **db/schema.rb** to confirm that the rollback was successful. (See Box 3.1 for another technique useful for reversing migrations.) Under the hood, this command executes the **drop_table** command to remove the users table from the database. The reason this works is that the **change** method knows that **drop_table** is the inverse of **create_table**, which means that the rollback migration can be easily inferred. In the case of an irreversible migration, such as one to remove a database column, it is necessary to define separate **up** and **down** methods in place of the single **change** method. Read about migrations in the Rails Guides for more information.

3. Re-run the migration by executing **rails db:migrate** again. Confirm that the contents of **db/schema.rb** have been restored.

6.1.2 The model file

We've seen how the User model generation in Listing 6.1 generated a migration file (Listing 6.2), and we saw in Figure 6.6 the results of running this migration: it updated a file called **development.sqlite3** by creating a table **users** with columns **id**, **name**, **email**, **created_at**, and **updated_at**. Listing 6.1 also created the model itself. The rest of this section is dedicated to understanding it.

We begin by looking at the code for the User model, which lives in the file **user.rb** inside the **app/models/** directory. It is, to put it mildly, very compact (Listing 6.3).

Listing 6.3: The brand new User model.
app/models/user.rb

```
class User < ApplicationRecord
end
```

Recall from Section 4.4.2 that the syntax **class User < ApplicationRecord** means that the **User** class *inherits* from the **ApplicationRecord** class, which in turn inherits from **ActiveRecord::Base** (Figure 2.19), so that the User model automatically has all the functionality of the **ActiveRecord::Base** class. Of course, this knowledge doesn't do us any good unless we know what **ActiveRecord::Base** contains, so let's get started with some concrete examples.

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. In a Rails console, use the technique from Section 4.4.4 to confirm that **User.new** is of class **User** and inherits from **ApplicationRecord**.
2. Confirm that **ApplicationRecord** inherits from **ActiveRecord::Base**.

6.1.3 Creating user objects

As in Chapter 4, our tool of choice for exploring data models is the Rails console. Since we don't (yet) want to make any changes to our database, we'll start the console in a *sandbox*:

```
$ rails console --sandbox
Loading development environment in sandbox
Any modifications you make will be rolled back on exit
>>
```

As indicated by the helpful message "Any modifications you make will be rolled back on exit", when started in a sandbox the console will "roll back" (i.e., undo) any database changes introduced during the session.

In the console session in Section 4.4.5, we created a new user object with **`User.new`**, which we had access to only after requiring the example user file in Listing 4.17. With models, the situation is different; as you may recall from Section 4.4.4, the Rails console automatically loads the Rails environment, which includes the models. This means that we can make a new user object without any further work:

```
>> User.new
=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>
```

We see here the default console representation of a user object.

When called with no arguments, **`User.new`** returns an object with all **`nil`** attributes. In Section 4.4.5, we designed the example User class to take an *initialization hash* to set the object attributes; that design choice was motivated by Active Record, which allows objects to be initialized in the same way:

```
>> user = User.new(name: "Michael Hartl", email: "michael@example.com")
=> #<User id: nil, name: "Michael Hartl", email: "michael@example.com",
created_at: nil, updated_at: nil>
```

Here we see that the name and email attributes have been set as expected.

The notion of *validity* is important for understanding Active Record model objects. We'll explore this subject in more depth in Section 6.2, but for now it's worth noting that our initial **`user`** object is valid, which we can verify by calling the boolean **`valid?`** method on it:

```
>> user.valid?
true
```

So far, we haven't touched the database: `User.new` only creates an object *in memory*, while `user.valid?` merely checks to see if the object is valid. In order to save the User object to the database, we need to call the `save` method on the `user` variable:

```
>> user.save
   (0.1ms)  SAVEPOINT active_record_1
  SQL (0.8ms)  INSERT INTO "users" ("name", "email", "created_at",
  "updated_at") VALUES (?, ?, ?, ?)  [["name", "Michael Hartl"],
  ["email", "michael@example.com"], ["created_at", "2019-08-22 01:51:03.453035"],
  ["updated_at", "2019-08-22 01:51:03.453035"]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
=> true
```

The `save` method returns `true` if it succeeds and `false` otherwise. (Currently, all saves should succeed because there are as yet no validations; we'll see cases in Section 6.2 when some will fail.) For reference, the Rails console also shows the SQL command corresponding to `user.save` (namely, `INSERT INTO "users"`...). We'll hardly ever need raw SQL in this book,[6] and I'll omit discussion of the SQL commands from now on, but you can learn a lot by reading the SQL corresponding to Active Record commands.

You may have noticed that the new user object had `nil` values for the `id` and the magic columns `created_at` and `updated_at` attributes. Let's see if our `save` changed anything:

```
>> user
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

We see that the `id` has been assigned a value of `1`, while the magic columns have

[6] The only exception is in Section 14.3.3.

been assigned the current time and date.[7] Currently, the created and updated timestamps are identical; we'll see them differ in Section 6.1.5.

As with the User class in Section 4.4.5, instances of the User model allow access to their attributes using a dot notation:

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "michael@example.com"
>> user.updated_at
=> Thu, 22 Aug 2019 01:51:03 UTC +00:00
```

As we'll see in Chapter 7, it's often convenient to make and save a model in two steps as we have above, but Active Record also lets you combine them into one step with **`User.create`**:

```
>> User.create(name: "A Nother", email: "another@example.org")
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2019-08-22 01:53:22", updated_at: "2019-08-22 01:53:22">
>> foo = User.create(name: "Foo", email: "foo@bar.com")
#<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

Note that **`User.create`**, rather than returning **`true`** or **`false`**, returns the User object itself, which we can optionally assign to a variable (such as **`foo`** in the second command above).

The inverse of **`create`** is **`destroy`**:

```
>> foo.destroy
   (0.1ms)  SAVEPOINT active_record_1
  SQL (0.2ms)  DELETE FROM "users" WHERE "users"."id" = ?  [["id", 3]]
```

[7] The timestamps are recorded in Coordinated Universal Time (UTC), which for most practical purposes is the same as Greenwich Mean Time. But why call it UTC? From the NIST Time and Frequency FAQ: **Q:** Why is UTC used as the acronym for Coordinated Universal Time instead of CUT? **A:** In 1970 the Coordinated Universal Time system was devised by an international advisory group of technical experts within the International Telecommunication Union (ITU). The ITU felt it was best to designate a single abbreviation for use in all languages in order to minimize confusion. Since unanimous agreement could not be achieved on using either the English word order, CUT, or the French word order, TUC, the acronym UTC was chosen as a compromise.

```
   (0.1ms)  RELEASE SAVEPOINT active_record_1
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

Like **`create`**, **`destroy`** returns the object in question, though I can't recall ever having used the return value of **`destroy`**. In addition, the destroyed object still exists in memory:

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

So how do we know if we really destroyed an object? And for saved and non-destroyed objects, how can we retrieve users from the database? To answer these questions, we need to learn how to use Active Record to find user objects.

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. Confirm that **`user.name`** and **`user.email`** are of class **`String`**.
2. Of what class are the **`created_at`** and **`updated_at`** attributes?

6.1.4 Finding user objects

Active Record provides several options for finding objects. Let's use them to find the first user we created while verifying that the third user (**`foo`**) has been destroyed. We'll start with the existing user:

```
>> User.find(1)
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

Here we've passed the id of the user to **`User.find`**; Active Record returns the user with that id.

Let's see if the user with an **`id`** of **`3`** still exists in the database:

```
>> User.find(3)
ActiveRecord::RecordNotFound: Couldn't find User with ID=3
```

Since we destroyed our third user in Section 6.1.3, Active Record can't find it in the database. Instead, **`find`** raises an *exception*, which is a way of indicating an exceptional event in the execution of a program—in this case, a nonexistent Active Record id, leading **`find`** to raise an **`ActiveRecord::RecordNotFound`** exception.[8]

In addition to the generic **`find`**, Active Record also allows us to find users by specific attributes:

```
>> User.find_by(email: "michael@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

Since we will be using email addresses as usernames, this sort of **`find`** will be useful when we learn how to let users log in to our site (Chapter 7). If you're worried that **`find_by`** will be inefficient if there are a large number of users, you're ahead of the game; we'll cover this issue, and its solution via database indices, in Section 6.2.5.

We'll end with a couple of more general ways of finding users. First, there's **`first`**:

```
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

Naturally, **`first`** just returns the first user in the database. There's also **`all`**:

[8] Exceptions and exception handling are somewhat advanced Ruby subjects, and we won't need them much in this book. They are important, though, and I suggest learning about them using one of the Ruby books recommended in Section 14.4.1.

```
>> User.all
=> #<ActiveRecord::Relation [#<User id: 1, name: "Michael Hartl", email:
"michael@example.com", created_at: "2019-08-22 01:51:03", updated_at:
"2019-08-22 01:51:03">, #<User id: 2, name: "A Nother", email:
"another@example.org", created_at: "2019-08-22 01:53:22", updated_at:
"2019-08-22 01:53:22">]>
```

As you can see from the console output, **`User.all`** returns all the users in the database as an object of class **`ActiveRecord::Relation`**, which is effectively an array (Section 4.3.1).

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. Find the user by **`name`**. Confirm that **`find_by_name`** works as well. (You will often encounter this older style of **`find_by`** in legacy Rails applications.)

2. For most practical purposes, **`User.all`** acts like an array, but confirm that in fact it's of class **`User::ActiveRecord_Relation`**.

3. Confirm that you can find the length of **`User.all`** by passing it the **`length`** method (Section 4.2.2). Ruby's ability to manipulate objects based on how they act rather than on their formal class type is called *duck typing*, based on the aphorism that "If it looks like a duck, and it quacks like a duck, it's probably a duck."

6.1.5 Updating user objects

Once we've created objects, we often want to update them. There are two basic ways to do this. First, we can assign attributes individually, as we did in Section 4.4.5:

```
>> user           # Just a reminder about our user's attributes
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
>> user.email = "mhartl@example.net"
=> "mhartl@example.net"
>> user.save
=> true
```

Note that the final step is necessary to write the changes to the database. We can see what happens without a save by using **`reload`**, which reloads the object based on the database information:

```
>> user.email
=> "mhartl@example.net"
>> user.email = "foo@bar.com"
=> "foo@bar.com"
>> user.reload.email
=> "mhartl@example.net"
```

Now that we've updated the user by running **`user.save`**, the magic columns differ, as promised in Section 6.1.3:

```
>> user.created_at
=> Thu, 22 Aug 2019 01:51:03 UTC +00:00
>> user.updated_at
=> Thu, 22 Aug 2019 01:58:08 UTC +00:00
```

The second main way to update multiple attributes is to use **`update`**:[9]

```
>> user.update(name: "The Dude", email: "dude@abides.org")
=> true
>> user.name
=> "The Dude"
>> user.email
=> "dude@abides.org"
```

[9] Formerly **`update_attributes`**.

The **update** method accepts a hash of attributes, and on success performs both the update and the save in one step (returning **true** to indicate that the save went through). Note that if any of the validations fail, such as when a password is required to save a record (as implemented in Section 6.3), the call to **update** will fail. If we need to update only a single attribute, using the singular **update_attribute** bypasses this restriction by skipping the validations:

```
>> user.update_attribute(:name, "El Duderino")
=> true
>> user.name
=> "El Duderino"
```

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. Update the user's name using assignment and a call to **save**.
2. Update the user's email address using a call to **update**.
3. Confirm that you can change the magic columns directly by updating the **created_at** column using assignment and a save. Use the value **1.year.ago**, which is a Rails way to create a timestamp one year before the present time.

6.2 User validations

The User model we created in Section 6.1 now has working **name** and **email** attributes, but they are completely generic: any string (including an empty one) is currently valid in either case. And yet, names and email addresses are more specific than this. For example, **name** should be non-blank, and **email** should match the specific format characteristic of email addresses. Moreover, since

we'll be using email addresses as unique usernames when users log in, we shouldn't allow email duplicates in the database.

In short, we shouldn't allow **`name`** and **`email`** to be just any strings; we should enforce certain constraints on their values. Active Record allows us to impose such constraints using *validations* (seen briefly before in Section 2.3.2). In this section, we'll cover several of the most common cases, validating *presence*, *length*, *format* and *uniqueness*. In Section 6.3.2 we'll add a final common validation, *confirmation*. And we'll see in Section 7.3 how validations give us convenient error messages when users make submissions that violate them.

6.2.1 A validity test

As noted in Box 3.3, test-driven development isn't always the right tool for the job, but model validations are exactly the kind of features for which TDD is a perfect fit. It's difficult to be confident that a given validation is doing exactly what we expect it to without writing a failing test and then getting it to pass.

Our method will be to start with a *valid* model object, set one of its attributes to something we want to be invalid, and then test that it in fact is invalid. As a safety net, we'll first write a test to make sure the initial model object is valid. This way, when the validation tests fail we'll know it's for the right reason (and not because the initial object was invalid in the first place).

In what follows, and when doing TDD generally, it's convenenient to work with your editor split into two *panes*, with test code on the left and application code on the right. My preferred setup with the cloud IDE is shown in Figure 6.7.

To get us started, the command in Listing 6.1 produced an initial test for testing users, though in this case it's practically blank (Listing 6.4).

Listing 6.4: The practically blank default User test.
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

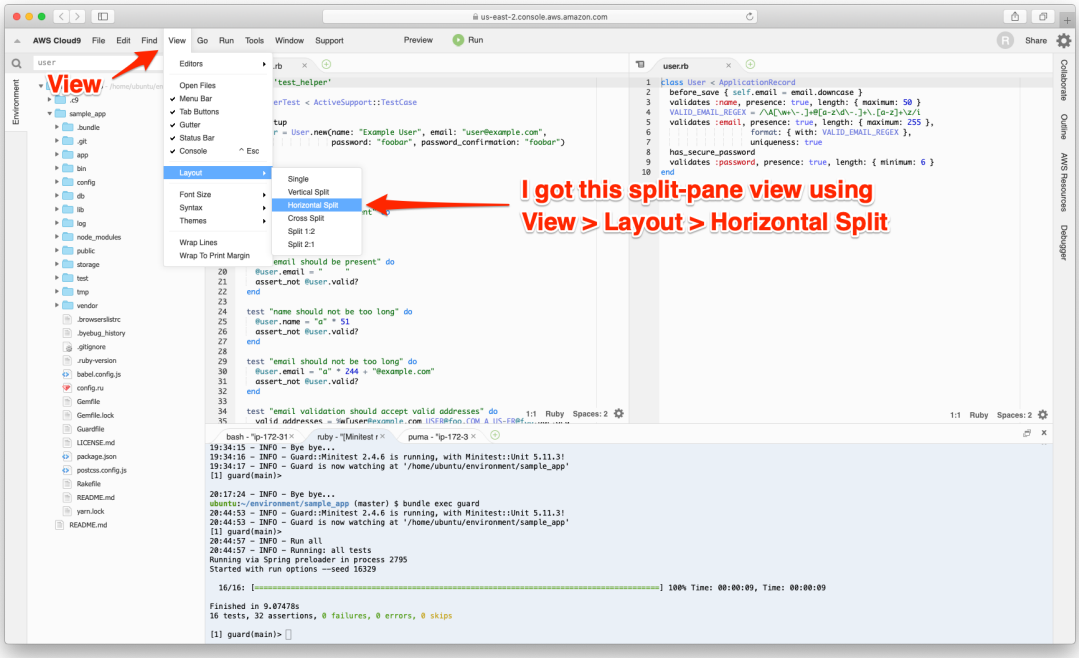


Figure 6.7: TDD with a split pane.

To write a test for a valid object, we'll create an initially valid User model object `@user` using the special `setup` method (discussed briefly in the Chapter 3 exercises), which automatically gets run before each test. Because `@user` is an instance variable, it's automatically available in all the tests, and we can test its validity using the `valid?` method (Section 6.1.3). The result appears in Listing 6.5.

Listing 6.5: A test for an initially valid user. GREEN
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end

  test "should be valid" do
    assert @user.valid?
  end
end
```

Listing 6.5 uses the plain `assert` method, which in this case succeeds if `@user.valid?` returns `true` and fails if it returns `false`.

Because our User model doesn't currently have any validations, the initial test should pass:

Listing 6.6: GREEN

```
$ rails test:models
```

Here we've used `rails test:models` to run just the model tests (compare to `rails test:integration` from Section 5.3.4).

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. In the console, confirm that a new user is currently valid.
2. Confirm that the user created in Section 6.1.3 is also valid.

6.2.2 Validating presence

Perhaps the most elementary validation is *presence*, which simply verifies that a given attribute is present. For example, in this section we'll ensure that both the name and email fields are present before a user gets saved to the database. In Section 7.3.3, we'll see how to propagate this requirement up to the signup form for creating new users.

We'll start with a test for the presence of a **`name`** attribute by building on the test in Listing 6.5. As seen in Listing 6.7, all we need to do is set the **`@user`** variable's **`name`** attribute to a blank string (in this case, a string of spaces) and then check (using the **`assert_not`** method) that the resulting User object is not valid.

Listing 6.7: A test for validation of the **`name`** attribute. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end

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

  test "name should be present" do
    @user.name = "     "
    assert_not @user.valid?
  end
end
```

At this point, the model tests should be RED:

Listing 6.8: RED

```
$ rails test:models
```

As we saw briefly before in the Chapter 2 exercises, the way to validate the presence of the name attribute is to use the `validates` method with argument `presence: true`, as shown in Listing 6.9. The `presence: true` argument is a one-element *options hash*; recall from Section 4.3.4 that curly braces are optional when passing hashes as the final argument in a method. (As noted in Section 5.1.1, the use of options hashes is a recurring theme in Rails.)

Listing 6.9: Validating the presence of a `name` attribute. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true
end
```

Listing 6.9 may look like magic, but `validates` is just a method. An equivalent formulation of Listing 6.9 using parentheses is as follows:

```
class User < ApplicationRecord
  validates(:name, presence: true)
end
```

Let's drop into the console to see the effects of adding a validation to our User model:[10]

```
$ rails console --sandbox
>> user = User.new(name: "", email: "michael@example.com")
>> user.valid?
=> false
```

[10] I'll omit the output of console commands when they are not particularly instructive—for example, the results of `User.new`.

Here we check the validity of the **user** variable using the **valid?** method, which returns **false** when the object fails one or more validations, and **true** when all validations pass. In this case, we only have one validation, so we know which one failed, but it can still be helpful to check using the **errors** object generated on failure:

```
>> user.errors.full_messages
=> ["Name can't be blank"]
```

(The error message is a hint that Rails validates the presence of an attribute using the **blank?** method, which we saw at the end of Section 4.4.3.)

Because the user isn't valid, an attempt to save the user to the database automatically fails:

```
>> user.save
=> false
```

As a result, the test in Listing 6.7 should now be GREEN:

Listing 6.10: GREEN

```
$ rails test:models
```

Following the model in Listing 6.7, writing a test for **email** attribute presence is easy (Listing 6.11), as is the application code to get it to pass (Listing 6.12).

Listing 6.11: A test for validation of the **email** attribute. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
```

```
  end

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

  test "name should be present" do
    @user.name = ""
    assert_not @user.valid?
  end

  test "email should be present" do
    @user.email = "     "
    assert_not @user.valid?
  end
end
```

Listing 6.12: Validating the presence of an `email` attribute. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name,  presence: true
  validates :email, presence: true
end
```

At this point, the presence validations are complete, and the test suite should be GREEN:

Listing 6.13: 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. Make a new user called `u` and confirm that it's initially invalid. What are the full error messages?

2. Confirm that `u.errors.messages` is a hash of errors. How would you access just the email errors?

6.2.3 Length validation

We've constrained our User model to require a name for each user, but we should go further: the user's names will be displayed on the sample site, so we should enforce some limit on their length. With all the work we did in Section 6.2.2, this step is easy.

There's no science to picking a maximum length; we'll just pull `50` out of thin air as a reasonable upper bound, which means verifying that names of `51` characters are too long. In addition, although it's unlikely ever to be a problem, there's a chance that a user's email address could overrun the maximum length of strings, which for many databases is 255. Because the format validation in Section 6.2.4 won't enforce such a constraint, we'll add one in this section for completeness. Listing 6.14 shows the resulting tests.

Listing 6.14: Tests for `name` and `email` length validations. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "name should not be too long" do
    @user.name = "a" * 51
    assert_not @user.valid?
  end

  test "email should not be too long" do
    @user.email = "a" * 244 + "@example.com"
    assert_not @user.valid?
  end
end
```

For convenience, we've used "string multiplication" in Listing 6.14 to make a string 51 characters long. We can see how this works using the console:

```
>> "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> ("a" * 51).length
=> 51
```

The email length validation arranges to make a valid email address that's one character too long:

```
>> "a" * 244 + "@example.com"
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaa@example.com"
>> ("a" * 244 + "@example.com").length
=> 256
```

At this point, the tests in Listing 6.14 should be RED:

Listing 6.15: RED

```
$ rails test
```

To get them to pass, we need to use the validation argument to constrain length, which is just **`length`**, along with the **`maximum`** parameter to enforce the upper bound (Listing 6.16).

Listing 6.16: Adding a length validation for the **`name`** attribute. GREEN
`app/models/user.rb`

```
class User < ApplicationRecord
  validates :name,  presence: true, length: { maximum: 50 }
  validates :email, presence: true, length: { maximum: 255 }
end
```

Now the tests should be GREEN:

Listing 6.17: GREEN

```
$ rails test
```

With our test suite passing again, we can move on to a more challenging validation: email format.

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. Make a new user with too-long name and email and confirm that it's not valid.
2. What are the error messages generated by the length validation?

6.2.4 Format validation

Our validations for the **`name`** attribute enforce only minimal constraints—any non-blank name under 51 characters will do—but of course the **`email`** attribute must satisfy the more stringent requirement of being a valid email address. So far we've only rejected blank email addresses; in this section, we'll require email addresses to conform to the familiar pattern **`user@example.com`**.

Neither the tests nor the validation will be exhaustive, just good enough to accept most valid email addresses and reject most invalid ones. We'll start with a couple of tests involving collections of valid and invalid addresses. To make these collections, it's worth knowing about the useful **`%w[]`** technique for making arrays of strings, as seen in this console session:

```
>> %w[foo bar baz]
=> ["foo", "bar", "baz"]
>> addresses = %w[USER@foo.COM THE_US-ER@foo.bar.org first.last@foo.jp]
=> ["USER@foo.COM", "THE_US-ER@foo.bar.org", "first.last@foo.jp"]
```

```
>> addresses.each do |address|
?>   puts address
>> end
USER@foo.COM
THE_US-ER@foo.bar.org
first.last@foo.jp
```

Here we've iterated over the elements of the **addresses** array using the **each** method (Section 4.3.2). With this technique in hand, we're ready to write some basic email format validation tests.

Because email format validation is tricky and error-prone, we'll start with some passing tests for *valid* email addresses to catch any errors in the validation. In other words, we want to make sure not just that invalid email addresses like *user@example,com* are rejected, but also that valid addresses like *user@example.com* are accepted, even after we impose the validation constraint. (Right now they'll be accepted because all non-blank email addresses are currently valid.) The result for a representative sample of valid email addresses appears in Listing 6.18.

Listing 6.18: Tests for valid email formats. GREEN
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email validation should accept valid addresses" do
    valid_addresses = %w[user@example.com USER@foo.COM A_US-ER@foo.bar.org
                         first.last@foo.jp alice+bob@baz.cn]
    valid_addresses.each do |valid_address|
      @user.email = valid_address
      assert @user.valid?, "#{valid_address.inspect} should be valid"
    end
  end
end
```

Note that we've included an optional second argument to the assertion with a custom error message, which in this case identifies the address causing the test to fail:

```
assert @user.valid?, "#{valid_address.inspect} should be valid"
```

(This uses the interpolated **inspect** method mentioned in Section 4.3.3.) Including the specific address that causes any failure is especially useful in a test with an **each** loop like Listing 6.18; otherwise, any failure would merely identify the line number, which is the same for all the email addresses, and which wouldn't be sufficient to identify the source of the problem.

Next we'll add tests for the *invalidity* of a variety of invalid email addresses, such as *user@example,com* (comma in place of dot) and *user_at_foo.org* (missing the '@' sign). As in Listing 6.18, Listing 6.19 includes a custom error message to identify the exact address causing any failure.

Listing 6.19: Tests for email format validation. RED

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email validation should reject invalid addresses" do
    invalid_addresses = %w[user@example,com user_at_foo.org user.name@example.
                           foo@bar_baz.com foo@bar+baz.com]
    invalid_addresses.each do |invalid_address|
      @user.email = invalid_address
      assert_not @user.valid?, "#{invalid_address.inspect} should be invalid"
    end
  end
end
```

At this point, the tests should be RED:

Listing 6.20: RED

```
$ rails test
```

The application code for email format validation uses the **`format`** validation, which works like this:

```
validates :email, format: { with: /<regular expression>/ }
```

This validates the attribute with the given *regular expression* (or *regex*), which is a powerful (and often cryptic) language for matching patterns in strings. This means we need to construct a regular expression to match valid email addresses while *not* matching invalid ones.

There actually exists a full regex for matching email addresses according to the official email standard, but it's enormous, obscure, and quite possibly counter-productive.[11] In this tutorial, we'll adopt a more pragmatic regex that has proven to be robust in practice. Here's what it looks like:

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
```

To help understand where this comes from, Table 6.1 breaks it into bite-sized pieces.[12]

Although you can learn a lot by studying Table 6.1, to really understand regular expressions I consider using an interactive regular expression matcher like Rubular to be essential (Figure 6.8).[13] The Rubular website has a beautiful interactive interface for making regular expressions, along with a handy regex quick reference. I encourage you to study Table 6.1 with a browser window

[11] For example, did you know that **`"Michael Hartl"@example.com`**, with quotation marks and a space in the middle, is a valid email address according to the standard? Incredibly, it is—but it's absurd.

[12] Note that, in Table 6.1, "letter" really means "lower-case letter", but the **`i`** at the end of the regex enforces case-insensitive matching.

[13] If you find it as useful as I do, I encourage you to donate to Rubular to reward developer Michael Lovitt for his wonderful work.

Expression	Meaning
`/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i`	full regex
`/`	start of regex
`\A`	match start of a string
`[\w+\-.]+`	at least one word character, plus, hyphen, or dot
`@`	literal "at sign"
`[a-z\d\-.]+`	at least one letter, digit, hyphen, or dot
`\.`	literal dot
`[a-z]+`	at least one letter
`\z`	match end of a string
`/`	end of regex
`i`	case-insensitive

Table 6.1: Breaking down the valid email regex.

open to Rubular—no amount of reading about regular expressions can replace playing with them interactively. (*Note*: If you use the regex from Table 6.1 in Rubular, I recommend leaving off the `\A` and `\z` characters so that you can match more than one email address at a time in the given test string. Also note that the regex consists of the characters *inside* the slashes **`/.../`**, so you should omit those when using Rubular.)

Applying the regular expression from Table 6.1 to the **`email`** format validation yields the code in Listing 6.21.

Listing 6.21: Validating the email format with a regular expression. GREEN

app/models/user.rb

```
class User < ApplicationRecord
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX }
end
```

Here the regex **`VALID_EMAIL_REGEX`** is a *constant*, indicated in Ruby by a name starting with a capital letter. The code

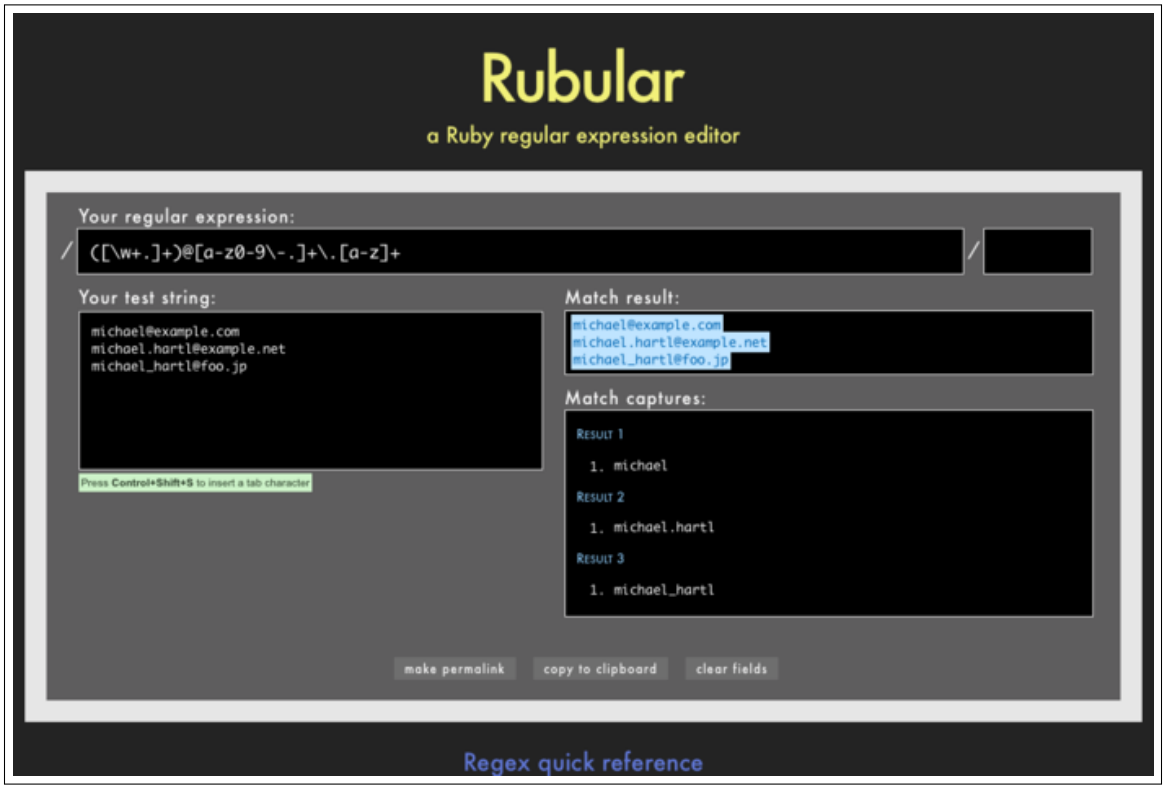


Figure 6.8: The awesome Rubular regular expression editor.

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
validates :email, presence: true, length: { maximum: 255 },
                  format: { with: VALID_EMAIL_REGEX }
```

ensures that only email addresses that match the pattern will be considered valid. (The expression above has one minor weakness: it allows invalid addresses that contain consecutive dots, such as **foo@bar..com**. Updating the regex in Listing 6.21 to fix this blemish is left as an exercise (Section 6.2.4).)

At this point, the tests should be GREEN:

Listing 6.22: GREEN

```
$ rails test:models
```

This means that there's only one constraint left: enforcing email uniqueness.

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. By pasting in the valid addresses from Listing 6.18 and invalid addresses from Listing 6.19 into the test string area at Rubular, confirm that the regex from Listing 6.21 matches all of the valid addresses and none of the invalid ones.

2. As noted above, the email regex in Listing 6.21 allows invalid email addresses with consecutive dots in the domain name, i.e., addresses of the form *foo@bar..com*. Add this address to the list of invalid addresses in Listing 6.19 to get a failing test, and then use the more complicated regex shown in Listing 6.23 to get the test to pass.

3. Add *foo@bar..com* to the list of addresses at Rubular, and confirm that the regex shown in Listing 6.23 matches all the valid addresses and none of the invalid ones.

Listing 6.23: Disallowing double dots in email domain names. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-]+(\.[a-z\d\-]+)*\.[a-z]+\z/i
  validates :email, presence:   true, length: { maximum: 255 },
                    format:     { with: VALID_EMAIL_REGEX }
end
```

6.2.5 Uniqueness validation

To enforce uniqueness of email addresses (so that we can use them as usernames), we'll be using the **`:uniqueness`** option to the **`validates`** method. But be warned: there's a *major* caveat, so don't just skim this section—read it carefully.

We'll start with some short tests. In our previous model tests, we've mainly used **`User.new`**, which just creates a Ruby object in memory, but for uniqueness tests we actually need to put a record into the database.[14] The initial duplicate email test appears in Listing 6.24.

Listing 6.24: A test for the rejection of duplicate email addresses. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
```

[14] As noted briefly in the introduction to this section, there is a dedicated test database, **`db/test.sqlite3`**, for this purpose.

```
  end
end
```

The method here is to make a user with the same email address as **`@user`** using **`@user.dup`**, which creates a duplicate user with the same attributes. Since we then save **`@user`**, the duplicate user has an email address that already exists in the database, and hence should not be valid.

We can get the new test in Listing 6.24 to pass by adding **`uniqueness: true`** to the **`email`** validation, as shown in Listing 6.25.

Listing 6.25: Validating the uniqueness of email addresses. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
end
```

We're not quite done, though. Email addresses are typically processed as if they were case-insensitive—i.e., **`foo@bar.com`** is treated the same as **`FOO@BAR.COM`** or **`FoO@BAr.coM`**—so our validation should incorporate this as well.[15] It's thus important to test for case-insensitivity, which we do with the code in Listing 6.26.

Listing 6.26: Testing case-insensitive email uniqueness. RED
test/models/user_test.rb

```
require 'test_helper'
```

[15]Technically, only the domain part of the email address is case-insensitive: *foo@bar.com* is actually different from *Foo@bar.com*. In practice, though, it is a bad idea to rely on this fact; as noted at about.com, "Since the case sensitivity of email addresses can create a lot of confusion, interoperability problems and widespread headaches, it would be foolish to require email addresses to be typed with the correct case. Hardly any email service or ISP does enforce case sensitive email addresses, returning messages whose recipient's email address was not typed correctly (in all upper case, for example)." Thanks to reader Riley Moses for pointing this out.

```
class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    duplicate_user.email = @user.email.upcase
    @user.save
    assert_not duplicate_user.valid?
  end
end
```

Here we are using the **upcase** method on strings (seen briefly in Section 4.3.2). This test does the same thing as the initial duplicate email test, but with an uppercase email address instead. If this test feels a little abstract, go ahead and fire up the console:

```
$ rails console --sandbox
>> user = User.create(name: "Example User", email: "user@example.com")
>> user.email.upcase
=> "USER@EXAMPLE.COM"
>> duplicate_user = user.dup
>> duplicate_user.email = user.email.upcase
>> duplicate_user.valid?
=> true
```

Of course, **duplicate_user.valid?** is currently **true** because the uniqueness validation is case-sensitive, but we want it to be **false**. Fortunately, **:uniqueness** accepts an option, **:case_sensitive**, for just this purpose (Listing 6.27).

Listing 6.27: Validating the uniqueness of email addresses, ignoring case.
GREEN

app/models/user.rb

```
class User < ApplicationRecord
  validates :name,  presence: true, length: { maximum: 50 }
```

```
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
end
```

Note that we have simply replaced **`true`** in Listing 6.25 with **`case_sensitive: false`** in Listing 6.27. (Rails infers that **`uniqueness`** should be **`true`** as well.)

At this point, our application—with an important caveat—enforces email uniqueness, and our test suite should pass:

Listing 6.28: GREEN

```
$ rails test
```

There's just one small problem, which is that *the Active Record uniqueness validation does not guarantee uniqueness at the database level*. Here's a scenario that explains why:

1. Alice signs up for the sample app, with address alice@wonderland.com.
2. Alice accidentally clicks on "Submit" *twice*, sending two requests in quick succession.
3. The following sequence occurs: request 1 creates a user in memory that passes validation, request 2 does the same, request 1's user gets saved, request 2's user gets saved.
4. Result: two user records with the exact same email address, despite the uniqueness validation

If the above sequence seems implausible, believe me, it isn't: it can happen on any Rails website with significant traffic (which I once learned the hard way). Luckily, the solution is straightforward to implement: we just need to enforce uniqueness at the database level as well as at the model level. Our method is

to create a database *index* on the email column (Box 6.2), and then require that the index be unique.

Box 6.2. Database indices

When creating a column in a database, it is important to consider whether we will need to *find* records by that column. Consider, for example, the `email` attribute created by the migration in Listing 6.2. When we allow users to log in to the sample app starting in Chapter 7, we will need to find the user record corresponding to the submitted email address. Unfortunately, based on the naïve data model, the only way to find a user by email address is to look through *each* user row in the database and compare its email attribute to the given email—which means we might have to examine *every* row (since the user could be the last one in the database). This is known in the database business as a *full-table scan*, and for a real site with thousands of users it is a Bad Thing.

Putting an index on the email column fixes the problem. To understand a database index, it's helpful to consider the analogy of a book index. In a book, to find all the occurrences of a given string, say "foobar", you would have to scan each page for "foobar"—the paper version of a full-table scan. With a book index, on the other hand, you can just look up "foobar" in the index to see all the pages containing "foobar". A database index works essentially the same way.

The email index represents an update to our data modeling requirements, which (as discussed in Section 6.1.1) is handled in Rails using migrations. We saw in Section 6.1.1 that generating the User model automatically created a new migration (Listing 6.2); in the present case, we are adding structure to an existing model, so we need to create a migration directly using the **`migration`** generator:

```
$ rails generate migration add_index_to_users_email
```

Unlike the migration for users, the email uniqueness migration is not predefined, so we need to fill in its contents with Listing 6.29.[16]

Listing 6.29: The migration for enforcing email uniqueness.
db/migrate/[timestamp]_add_index_to_users_email.rb

```
class AddIndexToUsersEmail < ActiveRecord::Migration[6.0]
  def change
    add_index :users, :email, unique: true
  end
end
```

This uses a Rails method called **add_index** to add an index on the **email** column of the **users** table. The index by itself doesn't enforce uniqueness, but the option **unique: true** does.

The final step is to migrate the database:

```
$ rails db:migrate
```

(If the migration fails, make sure to exit any running sandbox console sessions, which can lock the database and prevent migrations.)

At this point, the test suite should be RED due to a violation of the uniqueness constraint in the *fixtures*, which contain sample data for the test database. User fixtures were generated automatically in Listing 6.1, and as shown in Listing 6.30 the email addresses are not unique. (They're not *valid* either, but fixture data doesn't get run through the validations.)

Listing 6.30: The default user fixtures. RED
test/fixtures/users.yml

```
# Read about fixtures at https://api.rubyonrails.org/classes/ActiveRecord/
# FixtureSet.html
```

[16] Of course, we could just edit the migration file for the **users** table in Listing 6.2, but that would require rolling back and then migrating back up. The Rails Way™ is to use migrations every time we discover that our data model needs to change.

```
one:
  name: MyString
  email: MyString

two:
  name: MyString
  email: MyString
```

Because we won't need fixtures until Chapter 8, for now we'll just remove them, leaving an empty fixtures file (Listing 6.31).

Listing 6.31: An empty fixtures file. GREEN
test/fixtures/users.yml

```
# empty
```

Having addressed the uniqueness caveat, there's one more change we need to make to be assured of email uniqueness. Some database adapters use case-sensitive indices, considering the strings "Foo@ExAMPle.CoM" and "foo@example.com" to be distinct, but our application treats those addresses as the same. To avoid this incompatibility, we'll standardize on all lower-case addresses, converting "Foo@ExAMPle.CoM" to "foo@example.com" before saving it to the database. The way to do this is with a *callback*, which is a method that gets invoked at a particular point in the lifecycle of an Active Record object.

In the present case, that point is before the object is saved, so we'll use a **before_save** callback to downcase the email attribute before saving the user.[17] The result appears in Listing 6.32. (This is just a first implementation; we'll discuss this subject again in Section 11.1, where we'll use the preferred *method reference* convention for defining callbacks.)

Listing 6.32: Ensuring email uniqueness by downcasing the email attribute. RED
app/models/user.rb

[17] See the Rails API entry on callbacks for more information on which callbacks Rails supports.

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
end
```

The code in Listing 6.32 passes a block to the **before_save** callback and sets the user's email address to a lower-case version of its current value using the **downcase** string method. Note also that Listing 6.32 reverts the **uniqueness** constraint back to **true**, since case-sensitive matching works fine if all of the emails are lower-case. Indeed, this practice prevents problems applying the database index from Listing 6.29, since many databases have difficulty using an index when combined with a case-insensitive match.[18]

Restoring the original constraint does break the test in Listing 6.26, but that's easy to fix by reverting the test to its previous form from Listing 6.24, as shown again in Listing 6.33.

Listing 6.33: Restoring the original email uniqueness test. GREEN
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
  end
end
```

[18] Thanks to reader Alex Friedman for pointing this out.

By the way, in Listing 6.32 we could have written the assignment as

```
self.email = self.email.downcase
```

(where **self** refers to the current user), but inside the User model the **self** keyword is optional on the right-hand side:

```
self.email = email.downcase
```

We encountered this idea briefly in the context of **reverse** in the **palindrome** method (Section 4.4.2), which also noted that **self** is *not* optional in an assignment, so

```
email = email.downcase
```

wouldn't work. (We'll discuss this subject in more depth in Section 9.1.)

At this point, the Alice scenario above will work fine: the database will save a user record based on the first request, and it will reject the second save because the duplicate email address violates the uniqueness constraint. (An error will appear in the Rails log, but that doesn't do any harm.) Moreover, adding this index on the email attribute accomplishes a second goal, alluded to briefly in Section 6.1.4: as noted in Box 6.2, the index on the **email** attribute fixes a potential efficiency problem by preventing a full-table scan when finding users by email address.

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. Add a test for the email downcasing from Listing 6.32, as shown in Listing 6.34. This test uses the **reload** method for reloading a value from the

database and the **assert_equal** method for testing equality. To verify that Listing 6.34 tests the right thing, comment out the **before_save** line to get to RED, then uncomment it to get to GREEN.

2. By running the test suite, verify that the **before_save** callback can be written using the "bang" method **email.downcase!** to modify the **email** attribute directly, as shown in Listing 6.35.

Listing 6.34: A test for the email downcasing from Listing 6.32.
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
  end

  test "email addresses should be saved as lower-case" do
    mixed_case_email = "Foo@ExAMPle.CoM"
    @user.email = mixed_case_email
    @user.save
    assert_equal mixed_case_email.downcase, @user.reload.email
  end
end
```

Listing 6.35: An alternate callback implementation. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  before_save { email.downcase! }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
```

```
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
end
```

6.3 Adding a secure password

Now that we've defined validations for the name and email fields, we're ready to add the last of the basic User attributes: a secure password. The method is to require each user to have a password (with a password confirmation), and then store a *hashed* version of the password in the database. (There is some potential for confusion here. In the present context, a *hash* refers not to the Ruby data structure from Section 4.3.3 but rather to the result of applying an irreversible hash function to input data.) We'll also add a way to *authenticate* a user based on a given password, a method we'll use in Chapter 8 to allow users to log in to the site.

The method for authenticating users will be to take a submitted password, hash it, and compare the result to the hashed value stored in the database. If the two match, then the submitted password is correct and the user is authenticated. By comparing hashed values instead of raw passwords, we will be able to authenticate users without storing the passwords themselves. This means that, even if our database is compromised, our users' passwords will still be secure.

6.3.1 A hashed password

Most of the secure password machinery will be implemented using a single Rails method called **`has_secure_password`**, which we'll include in the User model as follows:

```
class User < ApplicationRecord
  .
  .
  .
  has_secure_password
end
```

When included in a model as above, this one method adds the following functionality:

- The ability to save a securely hashed **password_digest** attribute to the database
- A pair of virtual attributes[19] (**password** and **password_confirmation**), including presence validations upon object creation and a validation requiring that they match
- An **authenticate** method that returns the user when the password is correct (and **false** otherwise)

The only requirement for **has_secure_password** to work its magic is for the corresponding model to have an attribute called **password_digest**. (The name *digest* comes from the terminology of cryptographic hash functions. In this context, *hashed password* and *password digest* are synonyms.)[20] In the case of the User model, this leads to the data model shown in Figure 6.9.

To implement the data model in Figure 6.9 , we first generate an appropriate migration for the **password_digest** column. We can choose any migration name we want, but it's convenient to end the name with **to_users**, since in this case Rails automatically constructs a migration to add columns to the **users** table. The result, with migration name **add_password_digest_to_users**, appears as follows:

```
$ rails generate migration add_password_digest_to_users password_digest:string
```

[19] In this context, *virtual* means that the attributes exist on the model object but do not correspond to columns in the database.

[20] Hashed password digests are often erroneously referred to as *encrypted passwords*. For example, the source code of **has_secure_password** makes this mistake, as did the first two editions of this tutorial. This terminology is wrong because by design encryption is *reversible*—the ability to encrypt implies the ability to *decrypt* as well. In contrast, the whole point of calculating a password's hash digest is to be *irreversible*, so that it is computationally intractable to infer the original password from the digest. (Thanks to reader Andy Philips for pointing out this issue and for encouraging me to fix the broken terminology.)

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string

Figure 6.9: The User data model with an added `password_digest` attribute.

Here we've also supplied the argument `password_digest:string` with the name and type of attribute we want to create. (Compare this to the original generation of the `users` table in Listing 6.1, which included the arguments `name:string` and `email:string`.) By including `password_digest:string`, we've given Rails enough information to construct the entire migration for us, as seen in Listing 6.36.

Listing 6.36: The migration to add a `password_digest` column.
db/migrate/[timestamp]_add_password_digest_to_users.rb

```
class AddPasswordDigestToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :password_digest, :string
  end
end
```

Listing 6.36 uses the `add_column` method to add a `password_digest` column to the `users` table. To apply it, we just migrate the database:

```
$ rails db:migrate
```

To make the password digest, **has_secure_password** uses a state-of-the-art hash function called bcrypt. By hashing the password with bcrypt, we ensure that an attacker won't be able to log in to the site even if they manage to obtain a copy of the database. To use bcrypt in the sample application, we need to add the `bcrypt` gem to our **Gemfile** (Listing 6.37).[21]

Listing 6.37: Adding **bcrypt** to the **Gemfile**.

```
source 'https://rubygems.org'

gem 'rails',          '6.0.2.1'
gem 'bcrypt',         '3.1.13'
gem 'bootstrap-sass', '3.4.1'
.
.
.
```

Then run **bundle install** as usual:

```
$ bundle install
```

6.3.2 User has secure password

Now that we've supplied the User model with the required **password_digest** attribute and installed bcrypt, we're ready to add **has_secure_password** to the User model, as shown in Listing 6.38.

Listing 6.38: Adding **has_secure_password** to the User model. RED
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
```

[21] As always, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed here.

```
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
end
```

As indicated by the RED indicator in Listing 6.38, the tests are now failing, as you can confirm at the command line:

Listing 6.39: RED

```
$ rails test
```

The reason is that, as noted in Section 6.3.1, **has_secure_password** enforces validations on the virtual **password** and **password_confirmation** attributes, but the tests in Listing 6.26 create an **@user** variable without these attributes:

```
def setup
  @user = User.new(name: "Example User", email: "user@example.com")
end
```

So, to get the test suite passing again, we just need to add a password and its confirmation, as shown in Listing 6.40.

Listing 6.40: Adding a password and its confirmation. GREEN
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

Note that the first line inside the `setup` method includes an additional comma at the end, as required by Ruby's hash syntax (Section 4.3.3). Leaving this comma off will produce a syntax error, and you should use your technical sophistication (Box 1.2) to identify and resolve such errors if (or, more realistically, when) they occur.

At this point the tests should be GREEN:

Listing 6.41: GREEN

```
$ rails test
```

We'll see in just a moment the benefits of adding `has_secure_password` to the User model (Section 6.3.4), but first we'll add a minimal requirement on password security.

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. Confirm that a user with valid name and email still isn't valid overall.

2. What are the error messages for a user with no password?

6.3.3 Minimum password standards

It's good practice in general to enforce some minimum standards on passwords to make them harder to guess. There are many options for enforcing password strength in Rails, but for simplicity we'll just enforce a minimum length and the requirement that the password not be blank. Picking a length of 6 as a reasonable minimum leads to the validation test shown in Listing 6.42.

Listing 6.42: Testing for a minimum password length. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "password should be present (nonblank)" do
    @user.password = @user.password_confirmation = " " * 6
    assert_not @user.valid?
  end

  test "password should have a minimum length" do
    @user.password = @user.password_confirmation = "a" * 5
    assert_not @user.valid?
  end
end
```

Note the use of the compact multiple assignment

```
@user.password = @user.password_confirmation = "a" * 5
```

in Listing 6.42. This arranges to assign a particular value to the password and its confirmation at the same time (in this case, a string of length 5, constructed using string multiplication as in Listing 6.14).

You may be able to guess the code for enforcing a **minimum** length constraint by referring to the corresponding **maximum** validation for the user's name (Listing 6.16):

```
validates :password, length: { minimum: 6 }
```

Combining this with a **presence** validation (Section 6.2.2) to ensure nonblank passwords, this leads to the User model shown in Listing 6.43. (It turns out the

has_secure_password method includes a presence validation, but unfortunately it applies only to records with *empty* passwords, which allows users to create invalid passwords like ' ' (six spaces).)

Listing 6.43: The complete implementation for secure passwords. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }
end
```

At this point, the tests should be GREEN:

Listing 6.44: GREEN

```
$ rails test:models
```

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. Confirm that a user with valid name and email but a too-short password isn't valid.

2. What are the associated error messages?

6.3.4 Creating and authenticating a user

Now that the basic User model is complete, we'll create a user in the database as preparation for making a page to show the user's information in Section 7.1. We'll also take a more concrete look at the effects of adding **has_secure_password** to the User model, including an examination of the important **authenticate** method.

Since users can't yet sign up for the sample application through the web—that's the goal of Chapter 7—we'll use the Rails console to create a new user by hand. For convenience, we'll use the **create** method discussed in Section 6.1.3, but in the present case we'll take care *not* to start in a sandbox so that the resulting user will be saved to the database. This means starting an ordinary **rails console** session and then creating a user with a valid name and email address together with a valid password and matching confirmation:

```
$ rails console
>> User.create(name: "Michael Hartl", email: "michael@example.com",
?>             password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 03:15:38", updated_at: "2019-08-22 03:15:38",
password_digest: [FILTERED]>
```

To check that this worked, let's look at the resulting **users** table in the development database using DB Browser for SQLite, as shown in Figure 6.10.[22] (If you're using the cloud IDE, you should download the database file as in Figure 6.5.) Note that the columns correspond to the attributes of the data model defined in Figure 6.9.

Returning to the console, we can see the effect of **has_secure_password** from Listing 6.43 by looking at the **password_digest** attribute:

[22] If for any reason something went wrong, you can always reset the database as follows:

1. Quit the console.
2. Run **$ rm -f development.sqlite3** at the command line to remove the database. (We'll learn a more elegant method for doing this in Chapter 7.)
3. Re-run the migrations using **$ rails db:migrate**.
4. Restart the console.

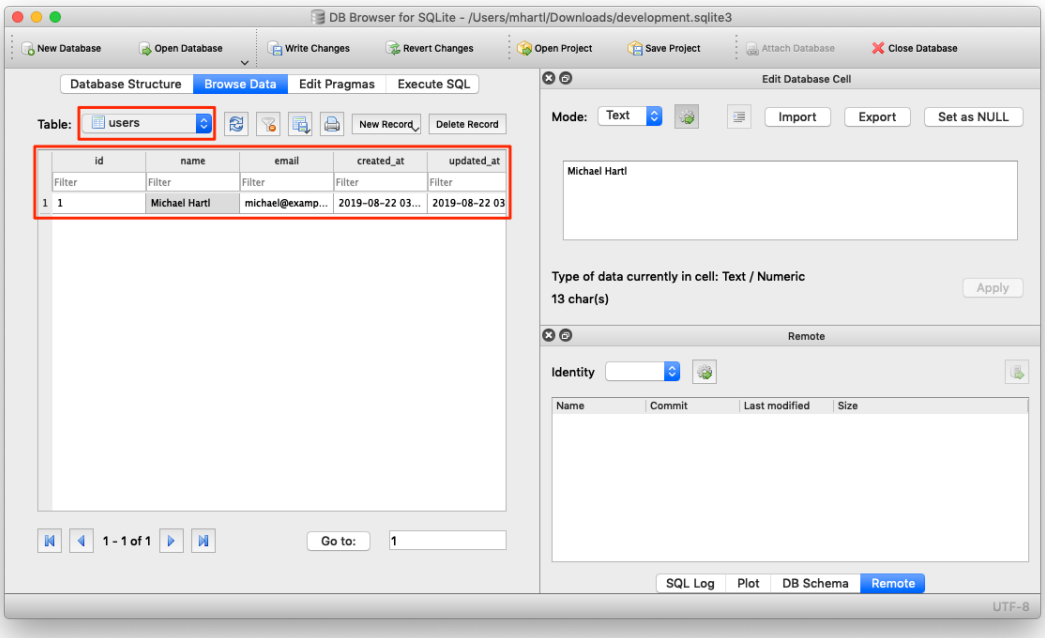


Figure 6.10: A user row in the SQLite database `db/development.sqlite3`.

```
>> user = User.find_by(email: "michael@example.com")
>> user.password_digest
=> "$2a$12$WgjER5ovLFjC2hmCItmbTe6nAXzT3bO66GiAQ83Ev03eVp32zyNYG"
```

This is the hashed version of the password (**"foobar"**) used to initialize the user object. Because it's constructed using bcrypt, it is computationally impractical to use the digest to discover the original password.[23]

As noted in Section 6.3.1, **has_secure_password** automatically adds an **authenticate** method to the corresponding model objects. This method determines if a given password is valid for a particular user by computing its digest and comparing the result to **password_digest** in the database. In the case of the user we just created, we can try a couple of invalid passwords as follows:

```
>> user.authenticate("not_the_right_password")
false
>> user.authenticate("foobaz")
false
```

Here **user.authenticate** returns **false** for invalid password. If we instead authenticate with the correct password, **authenticate** returns the user itself:

```
>> user.authenticate("foobar")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 03:15:38", updated_at: "2019-08-22 03:15:38",
password_digest: [FILTERED]>
```

In Chapter 8, we'll use the **authenticate** method to sign registered users into our site. In fact, it will turn out not to be important to us that **authenticate** returns the user itself; all that will matter is that it returns a value that is **true** in a boolean context. Recalling from Section 4.2.2 that **!!** converts an object to its corresponding boolean value, we can see that **user.authenticate** does the job nicely:

[23] By design, the bcrypt algorithm produces a *salted hash*, which protects against two important classes of attacks (dictionary attacks and rainbow table attacks).

```
>> !!user.authenticate("foobar")
=> true
```

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. Quit and restart the console, and then find the user created in this section.
2. Try changing the name by assigning a new name and calling **`save`**. Why didn't it work?
3. Update **`user`**'s name to use your name. *Hint*: The necessary technique is covered in Section 6.1.5.

6.4 Conclusion

Starting from scratch, in this chapter we created a working User model with name, email, and password attributes, together with validations enforcing several important constraints on their values. In addition, we have the ability to securely authenticate users using a given password. This is a remarkable amount of functionality for only twelve lines of code.

In Chapter 7, we'll make a working signup form to create new users, together with a page to display each user's information. In Chapter 8, we'll then use the authentication machinery from Section 6.3 to let users log into the site.

If you're using Git, now would be a good time to commit if you haven't done so in a while:

```
$ rails test
$ git add -A
$ git commit -m "Make a basic User model (including secure passwords)"
```

Then merge back into the master branch and push to the remote repository:

```
$ git checkout master
$ git merge modeling-users
$ git push
```

To get the User model working in production, we need to run the migrations at Heroku, which we can do with **heroku run**:

```
$ rails test
$ git push heroku
$ heroku run rails db:migrate
```

We can verify that this worked by running a console in production:

```
$ heroku run rails console --sandbox
>> User.create(name: "Michael Hartl", email: "michael@example.com",
?>             password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 03:20:06", updated_at: "2019-08-22 03:20:06",
password_digest: [FILTERED]>
```

6.4.1 What we learned in this chapter

- Migrations allow us to modify our application's data model.
- Active Record comes with a large number of methods for creating and manipulating data models.
- Active Record validations allow us to place constraints on the data in our models.
- Common validations include presence, length, and format.
- Regular expressions are cryptic but powerful.

- Defining a database index improves lookup efficiency while allowing enforcement of uniqueness at the database level.
- We can add a secure password to a model using the built-in **`has_secure_password`** method.

Chapter 7

Sign up

Now that we have a working User model, it's time to add an ability few websites can live without: letting users sign up. We'll use an HTML *form* to submit user signup information to our application (Section 7.2), which will then be used to create a new user and save its attributes to the database (Section 7.4). At the end of the signup process, it's important to render a profile page with the newly created user's information, so we'll begin by making a page for *showing* users, which will serve as the first step toward implementing the REST architecture for users (Section 2.2.2). Along the way, we'll build on our work in Section 5.3.4 to write succinct and expressive integration tests.

In this chapter, we'll rely on the User model validations from Chapter 6 to increase the odds of new users having valid email addresses. In Chapter 11, we'll make *sure* of email validity by adding a separate *account activation* step to user signup.

Although this tutorial is designed to be as simple as possible while still being professional-grade, web development is a complicated subject, and Chapter 7 necessarily marks a significant increase in the difficulty of the exposition. I recommend taking your time with the material and reviewing it as necessary. (Some readers have reported simply doing the chapter twice is a helpful exercise.) You might also consider subscribing to the courses at Learn Enough to gain additional assistance, both with this tutorial and with its relevant prerequisites (especially *Learn Enough Ruby to Be Dangerous*).

7.1 Showing users

In this section, we'll take the first steps toward the final profile by making a page to display a user's name and profile photo, as indicated by the mockup in Figure 7.1.[1] Our eventual goal for the user profile pages is to show the user's profile image, basic user data, and a list of microposts, as mocked up in Figure 7.2.[2] (Figure 7.2 includes an example of *lorem ipsum* text, which has a fascinating story that you should definitely read about some time.) We'll complete this task, and with it the sample application, in Chapter 14.

If you're following along with version control, make a topic branch as usual:

```
$ git checkout -b sign-up
```

7.1.1 Debug and Rails environments

The profiles in this section will be the first truly dynamic pages in our application. Although the view will exist as a single page of code, each profile will be customized using information retrieved from the application's database. As preparation for adding dynamic pages to our sample application, now is a good time to add some debug information to our site layout (Listing 7.1). This displays some useful information about each page using the built-in **`debug`** method and **`params`** variable (which we'll learn more about in Section 7.1.2).

Listing 7.1: Adding some debug information to the site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  .
```

[1] Mockingbird doesn't support custom images like the profile photo in Figure 7.1; I put that in by hand using GIMP.

[2] Image retrieved from https://www.flickr.com/photos/43803060@N00/24308857/ on 2014-06-16. Copyright © 2002 by Shaun Wallin and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

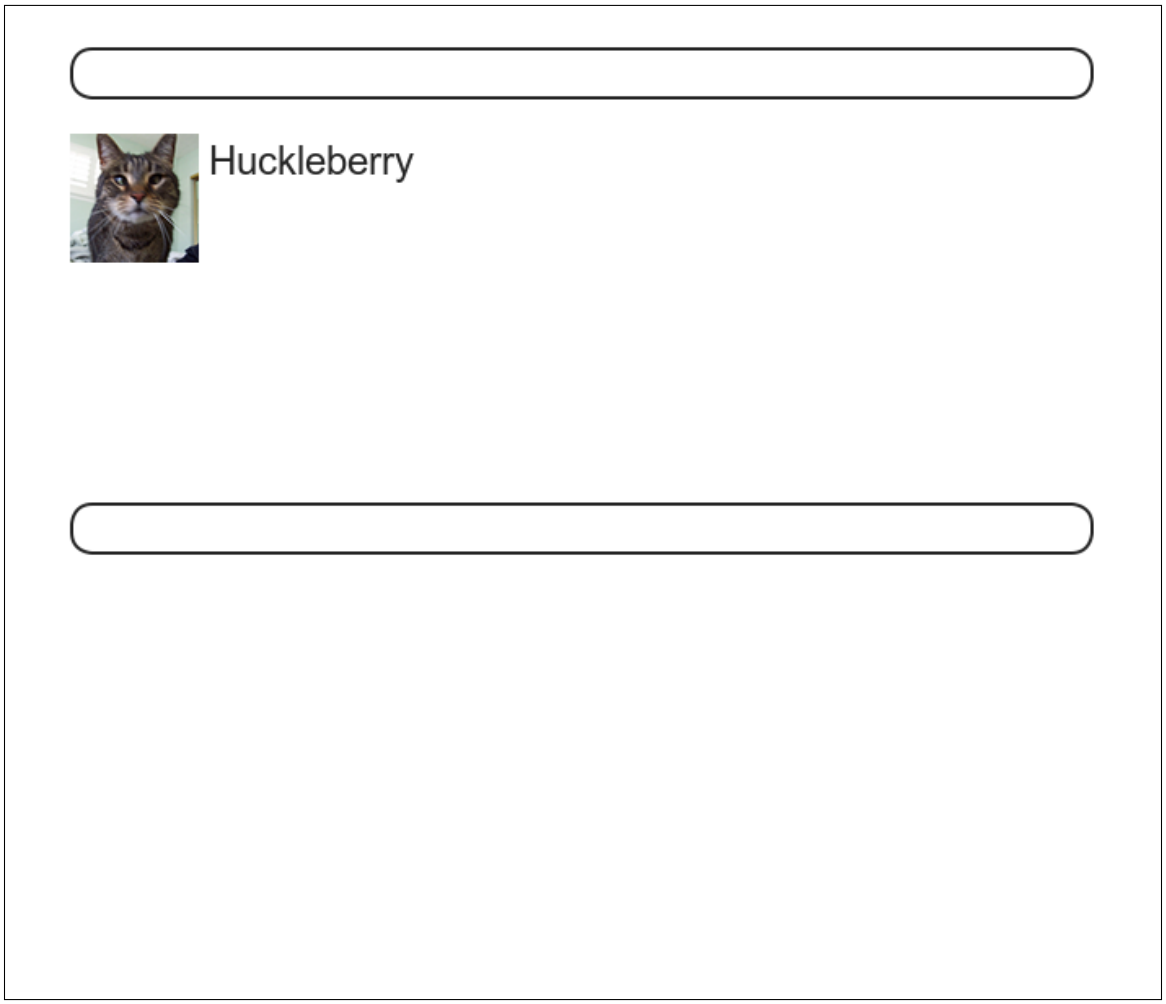


Figure 7.1: A mockup of the user profile made in this section.

Figure 7.2: A mockup of our best guess at the final profile page.

```
  .
  .
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
  </body>
</html>
```

Since we don't want to display debug information to users of a deployed application, Listing 7.1 uses

```
if Rails.env.development?
```

to restrict the debug information to the *development environment*, which is one of three environments defined by default in Rails (Box 7.1).[3] In particular, **`Rails.env.development?`** is **`true`** only in a development environment, so the embedded Ruby

```
<%= debug(params) if Rails.env.development? %>
```

won't be inserted into production applications or tests. (Inserting the debug information into tests probably wouldn't do any harm, but it probably wouldn't do any good, either, so it's best to restrict the debug display to development only.)

Box 7.1. Rails environments

Rails comes equipped with three environments: `test`, `development`, and `production`. The default environment for the Rails console is `development`:

[3] You can define your own custom environments as well; see the RailsCast on adding an environment for details.

```
$ rails console
Loading development environment
>> Rails.env
=> "development"
>> Rails.env.development?
=> true
>> Rails.env.test?
=> false
```

As you can see, Rails provides a `Rails` object with an `env` attribute and associated environment boolean methods, so that, for example, `Rails.env.test?` returns `true` in a test environment and `false` otherwise.

If you ever need to run a console in a different environment (to debug a test, for example), you can pass the environment as a parameter to the `console` script:

```
$ rails console test
Loading test environment
>> Rails.env
=> "test"
>> Rails.env.test?
=> true
```

As with the console, `development` is the default environment for the Rails server, but you can also run it in a different environment:

```
$ rails server --environment production
```

If you view your app running in production, it won't work without a production database, which we can create by running `rails db:migrate` in production:

```
$ rails db:migrate RAILS_ENV=production
```

(I find it confusing that the idiomatic commands to run the console, server, and migrate commands in non-default environments use different syntax, which is why I bothered showing all three. It's worth noting, though, that preceding any of them with `RAILS_ENV=<env>` will also work, as in `RAILS_ENV=production rails server`).

By the way, if you have deployed your sample app to Heroku, you can see its environment using `heroku run rails console`:

```
$ heroku run rails console
>> Rails.env
=> "production"
>> Rails.env.production?
=> true
```

Naturally, since Heroku is a platform for production sites, it runs each application in a production environment.

To make the debug output look nice, we'll add some rules to the custom stylesheet created in Chapter 5, as shown in Listing 7.2.

Listing 7.2: Adding code for a pretty debug box, including a Sass mixin.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* mixins, variables, etc. */

$gray-medium-light: #eaeaea;

@mixin box_sizing {
  -moz-box-sizing:    border-box;
  -webkit-box-sizing: border-box;
  box-sizing:         border-box;
}
.
.
```

```
.
/* miscellaneous */

.debug_dump {
  clear: both;
  float: left;
  width: 100%;
  margin-top: 45px;
  @include box_sizing;
}
```

This introduces the Sass *mixin* facility, in this case called **box_sizing**. A mixin allows a group of CSS rules to be packaged up and used for multiple elements, converting

```
.debug_dump {
  .
  .
  .
  @include box_sizing;
}
```

to

```
.debug_dump {
  .
  .
  .
  -moz-box-sizing:    border-box;
  -webkit-box-sizing: border-box;
  box-sizing:         border-box;
}
```

We'll put this mixin to use again in Section 7.2.1. The result in the case of the debug box is shown in Figure 7.3.[4]

The debug output in Figure 7.3 gives potentially useful information about the page being rendered:

[4]The exact appearance of the Rails debug information is slightly version-dependent. For example, as of Rails 5 the debug information shows the **permitted** status of the information, a subject we'll cover in Section 7.3.2. Use your technical sophistication (Box 1.2) to resolve such minor discrepancies.

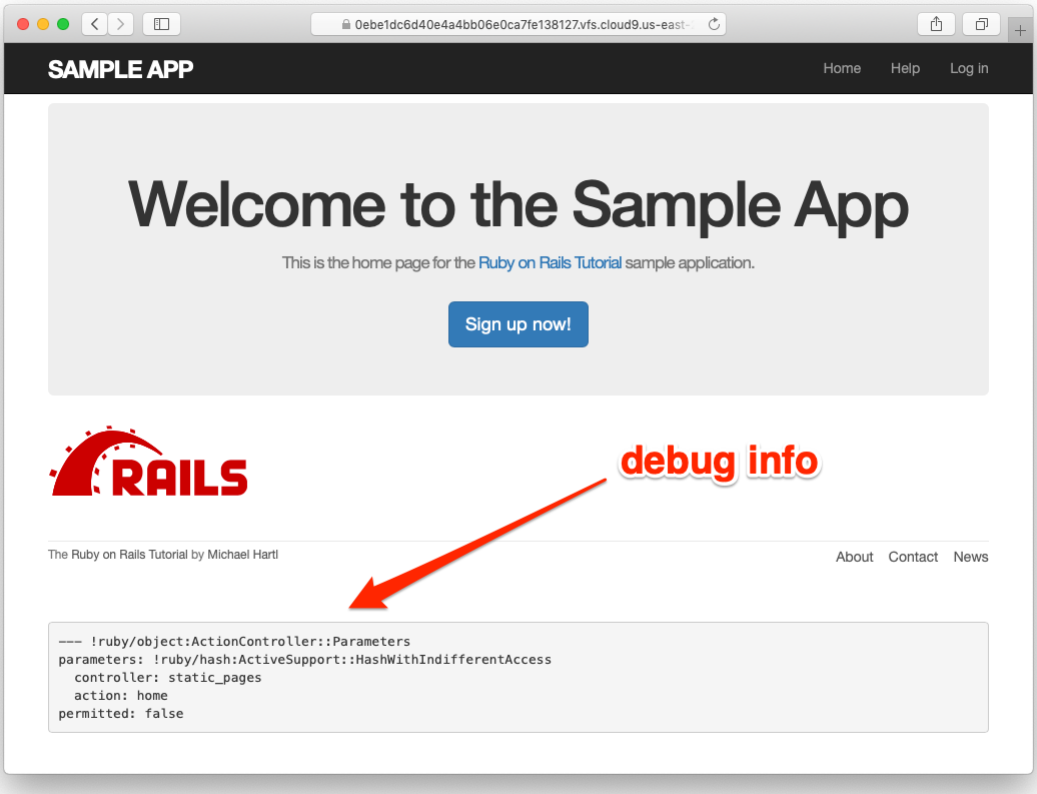


Figure 7.3: The sample application Home page with debug information.

```
---
controller: static_pages
action: home
```

This is a YAML[5] representation of **params**, which is basically a hash, and in this case identifies the controller and action for the page. We'll see another example in Section 7.1.2.

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. Visit /about in your browser and use the debug information to determine the controller and action of the **params** hash.

2. In the Rails console, pull the first user out of the database and assign it to the variable **user**. What is the output of **puts user.attributes.-to_yaml**? Compare this to using the **y** method via **y user.attributes**.

7.1.2 A Users resource

In order to make a user profile page, we need to have a user in the database, which introduces a chicken-and-egg problem: how can the site have a user before there is a working signup page? Happily, this problem has already been solved: in Section 6.3.4, we created a User record by hand using the Rails console, so there should be one user in the database:

[5] The Rails **debug** information is shown as YAML (a recursive acronym standing for "YAML Ain't Markup Language"), which is a friendly data format designed to be both machine- *and* human-readable.

```
$ rails console
>> User.count
=> 1
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2019-08-22 03:15:38", updated_at: "2019-08-22 03:15:38",
password_digest: [FILTERED]>
```

(If you don't currently have a user in your database, you should visit Section 6.3.4 now and complete it before proceeding.) We see from the console output above that the user has id `1`, and our goal now is to make a page to display this user's information. We'll follow the conventions of the REST architecture favored in Rails applications (Box 2.2), which means representing data as *resources* that can be created, shown, updated, or destroyed—four actions corresponding to the four fundamental operations `POST`, `GET`, `PATCH`, and `DELETE` defined by the HTTP standard (Box 3.2).

When following REST principles, resources are typically referenced using the resource name and a unique identifier. What this means in the context of users—which we're now thinking of as a Users *resource*—is that we should view the user with id `1` by issuing a `GET` request to the URL /users/1. Here the **`show`** action is *implicit* in the type of request—when Rails' REST features are activated, `GET` requests are automatically handled by the **`show`** action.

We saw in Section 2.2.1 that the page for a user with id `1` has URL /users/1. Unfortunately, visiting that URL right now just gives an error (Figure 7.4).

We can get the routing for /users/1 to work by adding a single line to our routes file (**`config/routes.rb`**):

```
resources :users
```

The result appears in Listing 7.3.

Listing 7.3: Adding a Users resource to the routes file.
`config/routes.rb`

```
Rails.application.routes.draw do
  root 'static_pages#home'
```

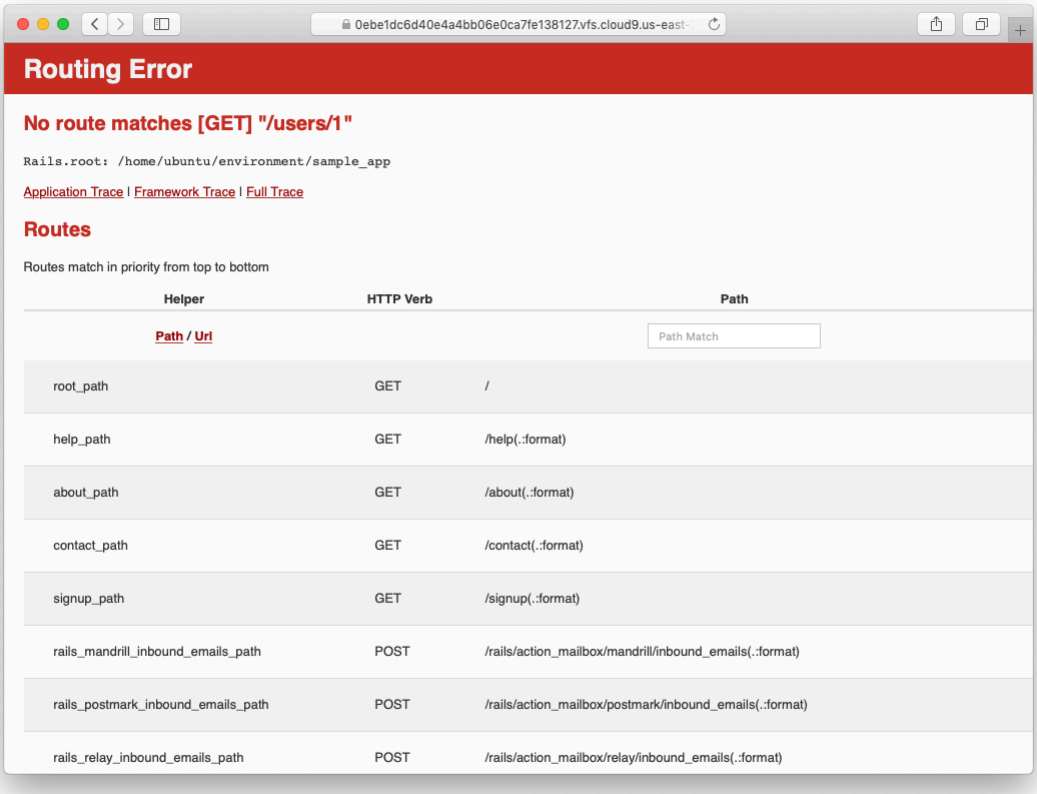


Figure 7.4: The current state of /users/1.

HTTP request	URL	Action	Named route	Purpose
GET	/users	index	users_path	page to list all users
GET	/users/1	show	user_path(user)	page to show user
GET	/users/new	new	new_user_path	page to make a new user (signup)
POST	/users	create	users_path	create a new user
GET	/users/1/edit	edit	edit_user_path(user)	page to edit user with id 1
PATCH	/users/1	update	user_path(user)	update user
DELETE	/users/1	destroy	user_path(user)	delete user

Table 7.1: RESTful routes provided by the Users resource in Listing 7.3.

```
  get  '/help',    to: 'static_pages#help'
  get  '/about',   to: 'static_pages#about'
  get  '/contact', to: 'static_pages#contact'
  get  '/signup',  to: 'users#new'
  resources :users
end
```

Although our immediate motivation is making a page to show users, the single line **`resources :users`** doesn't just add a working /users/1 URL; it endows our sample application with *all* the actions needed for a RESTful Users resource,[6] along with a large number of named routes (Section 5.3.3) for generating user URLs. The resulting correspondence of URLs, actions, and named routes is shown in Table 7.1. (Compare to Table 2.2.) Over the course of the next three chapters, we'll cover all of the other entries in Table 7.1 as we fill in all the actions necessary to make Users a fully RESTful resource.

With the code in Listing 7.3, the routing works, but there's still no page there (Figure 7.5). To fix this, we'll begin with a minimalist version of the profile page, which we'll flesh out in Section 7.1.4.

We'll use the standard Rails location for showing a user, which is **`app/views/users/show.html.erb`**. Unlike the **`new.html.erb`** view, which we created with the generator in Listing 5.38, the **`show.html.erb`** file doesn't currently exist, so you'll have to create it by hand,[7] and then fill it with the

[6] This means that the *routing* works, but the corresponding pages don't necessarily work at this point. For example, /users/1/edit gets routed properly to the **`edit`** action of the Users controller, but since the **`edit`** action doesn't exist yet actually hitting that URL will return an error.

[7] Using, e.g., **`touch app/views/users/show.html.erb`**.

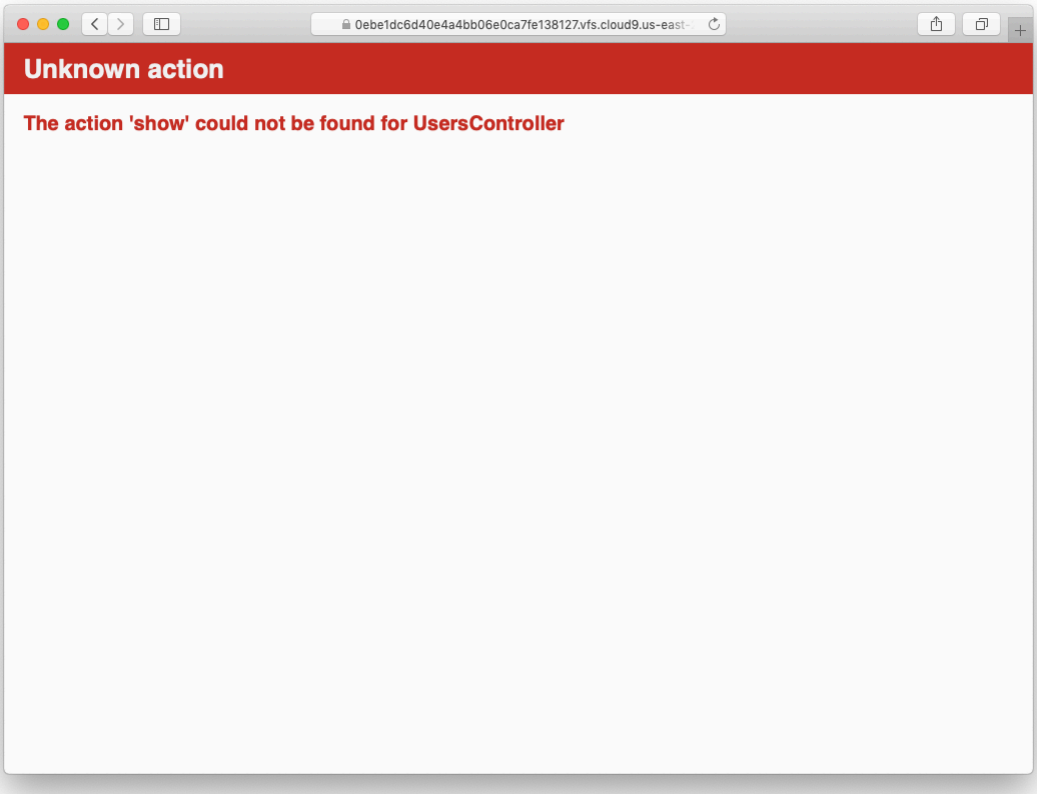


Figure 7.5: The URL /users/1 with routing but no page.

content shown in Listing 7.4.

Listing 7.4: A stub view for showing user information.
app/views/users/show.html.erb

```
<%= @user.name %>, <%= @user.email %>
```

This view uses embedded Ruby to display the user's name and email address, assuming the existence of an instance variable called **@user**. Of course, eventually the real user show page will look very different (and won't display the email address publicly).

In order to get the user show view to work, we need to define an **@user** variable in the corresponding **show** action in the Users controller. As you might expect, we use the **find** method on the User model (Section 6.1.4) to retrieve the user from the database, as shown in Listing 7.5.

Listing 7.5: The Users controller with a **show** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
  end
end
```

Here we've used **params** to retrieve the user id. When we make the appropriate request to the Users controller, **params[:id]** will be the user id 1, so the effect is the same as the **find** method **User.find(1)** we saw in Section 6.1.4. (Technically, **params[:id]** is the string **"1"**, but **find** is smart enough to convert this to an integer.)

With the user view and action defined, the URL /users/1 works perfectly, as seen in Figure 7.6. (If you haven't restarted the Rails server since adding

bcrypt, you may have to do so at this time. This sort of thing is a good application of technical sophistication (Box 1.2).) Note that the debug information in Figure 7.6 confirms the value of **`params[:id]`**:

```
---
action: show
controller: users
id: '1'
```

This is why the code

```
User.find(params[:id])
```

in Listing 7.5 finds the user with id `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 embedded Ruby, add the **`created_at`** and **`updated_at`** "magic column" attributes to the user show page from Listing 7.4.
2. Using embedded Ruby, add **`Time.now`** to the user show page. What happens when you refresh the browser?

7.1.3 Debugger

We saw in Section 7.1.2 how the information in the **`debug`** could help us understand what's going on in our application, but there's also a more direct way to get debugging information using the `byebug` gem (Listing 3.2). To see how it works, we just need to add a line consisting of **`debugger`** to our application, as shown in Listing 7.6.

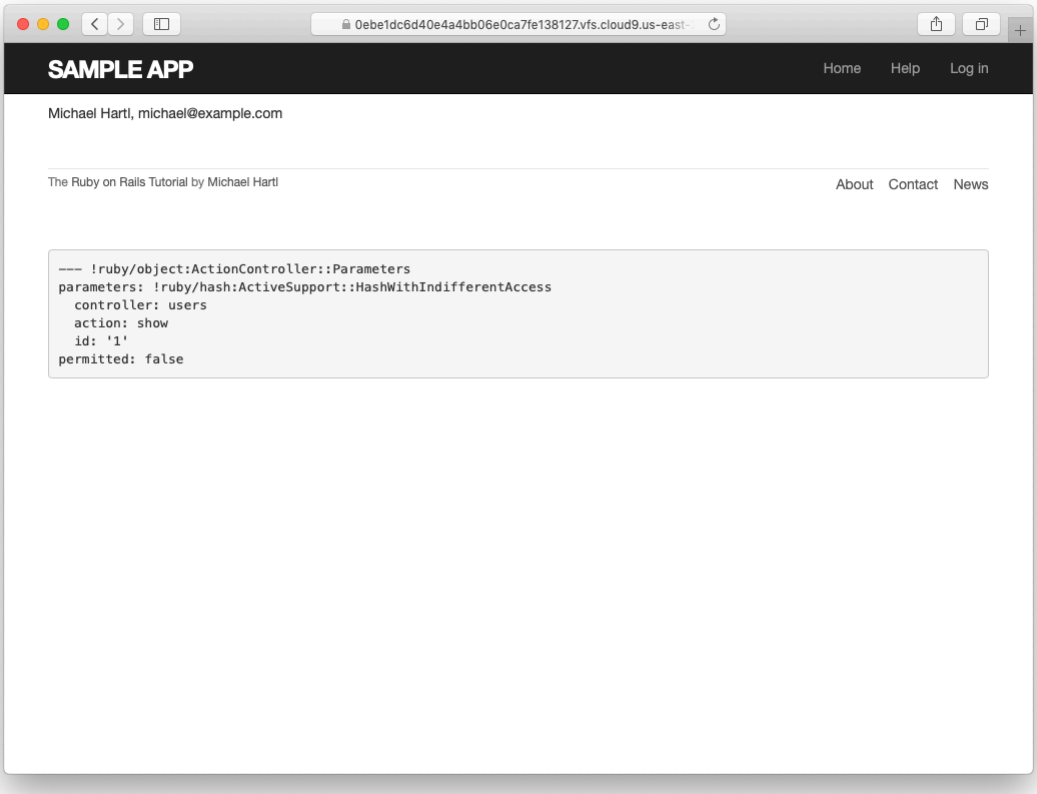


Figure 7.6: The user show page after adding a Users resource.

```
Processing by UsersController#show as HTML
  Parameters: {"id"=>"1"}
  User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
  ↳ app/controllers/users_controller.rb:4:in `show'
Return value is: nil

[1, 10] in /home/ubuntu/environment/sample_app/app/controllers/users_controller.rb
    1: class UsersController < ApplicationController
    2:
    3:   def show
    4:     @user = User.find(params[:id])
    5:     debugger
=>  6:   end
    7:
    8:   def new
    9:   end
   10: end
(byebug)
```

Figure 7.7: The **byebug** prompt in the Rails server.

Listing 7.6: The Users controller with a debugger.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
    debugger
  end

  def new
  end
end
```

Now, when we visit /users/1, the Rails server shows a **byebug** prompt (Figure 7.7):

```
(byebug)
```

We can treat **byebug** like a Rails console, issuing commands to figure out the state of the application:

```
(byebug) @user.name
"Michael Hartl"
(byebug) @user.email
"michael@example.com"
(byebug) params[:id]
"1"
```

To release the prompt and continue execution of the application, press Ctrl-D, then remove the **debugger** line from the **show** action (Listing 7.7).

Listing 7.7: The Users controller with the debugger line removed.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
  end
end
```

Whenever you're confused about something in a Rails application, it's a good practice to put **debugger** close to the code you think might be causing the trouble. Inspecting the state of the system using `byebug` is a powerful method for tracking down application errors and interactively debugging your application.

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. With the **debugger** in the **show** action as in Listing 7.6, hit /users/1. Use **puts** to display the value of the YAML form of the **params** hash. *Hint*: Refer to the relevant exercise in Section 7.1.1. How does it compare to the debug information shown by the **debug** method in the site template?

2. Put the **debugger** in the User **new** action and hit /users/new. What is the value of **@user**?

7.1.4 A Gravatar image and a sidebar

Having defined a basic user page in the previous section, we'll now flesh it out a little with a profile image for each user and the first cut of the user sidebar. We'll start by adding a "globally recognized avatar", or Gravatar, to the user profile.[8] Gravatar is a free service that allows users to upload images and associate them with email addresses they control. As a result, Gravatars are a convenient way to include user profile images without going through the trouble of managing image upload, cropping, and storage; all we need to do is construct the proper Gravatar image URL using the user's email address and the corresponding Gravatar image will automatically appear. (We'll learn how to handle custom image upload in Section 13.4.)

Our plan is to define a **gravatar_for** helper function to return a Gravatar image for a given user, as shown in Listing 7.8.

Listing 7.8: The user show view with name and Gravatar.
app/views/users/show.html.erb

```
<% provide(:title, @user.name) %>
<h1>
  <%= gravatar_for @user %>
  <%= @user.name %>
</h1>
```

By default, methods defined in any helper file are automatically available in any view, but for convenience we'll put the **gravatar_for** method in the file for helpers associated with the Users controller. As noted in the Gravatar documentation, Gravatar URLs are based on an MD5 hash of the user's email

[8] In Hinduism, an avatar is the manifestation of a deity in human or animal form. By extension, the term *avatar* is commonly used to mean some kind of personal representation, especially in a virtual environment. (In the context of Twitter and other social media, the term *avi* has gained currency, which is likely a mutated form of *avatar*.)

address. In Ruby, the MD5 hashing algorithm is implemented using the **hexdigest** method, which is part of the **Digest** library:

```
>> email = "MHARTL@example.COM"
>> Digest::MD5::hexdigest(email.downcase)
=> "1fda4469bcbec3badf5418269ffc5968"
```

Since email addresses are case-insensitive (Section 6.2.4) but MD5 hashes are not, we've used the **downcase** method to ensure that the argument to **hexdigest** is all lower-case. (Because of the email downcasing callback in Listing 6.32, this will never make a difference in this tutorial, but it's a good practice in case the **gravatar_for** ever gets used on email addresses from other sources.) The resulting **gravatar_for** helper appears in Listing 7.9.

Listing 7.9: Defining a **gravatar_for** helper method.
app/helpers/users_helper.rb

```
module UsersHelper

  # Returns the Gravatar for the given user.
  def gravatar_for(user)
    gravatar_id  = Digest::MD5::hexdigest(user.email.downcase)
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

The code in Listing 7.9 returns an image tag for the Gravatar with a **gravatar** CSS class and alt text equal to the user's name (which is especially convenient for visually impaired users using a screen reader).

The profile page appears as in Figure 7.8, which shows the default Gravatar image, which appears because **michael@example.com** isn't a real email address. (In fact, as you can see by visiting it, the example.com domain is reserved for examples like this one.)

To get our application to display a custom Gravatar, we'll use **update_attributes** (Section 6.1.5) to change the user's email to something I control:[9]

[9] The password confirmation isn't technically necessary here because **has_secure_password** (Section 6.3.1)

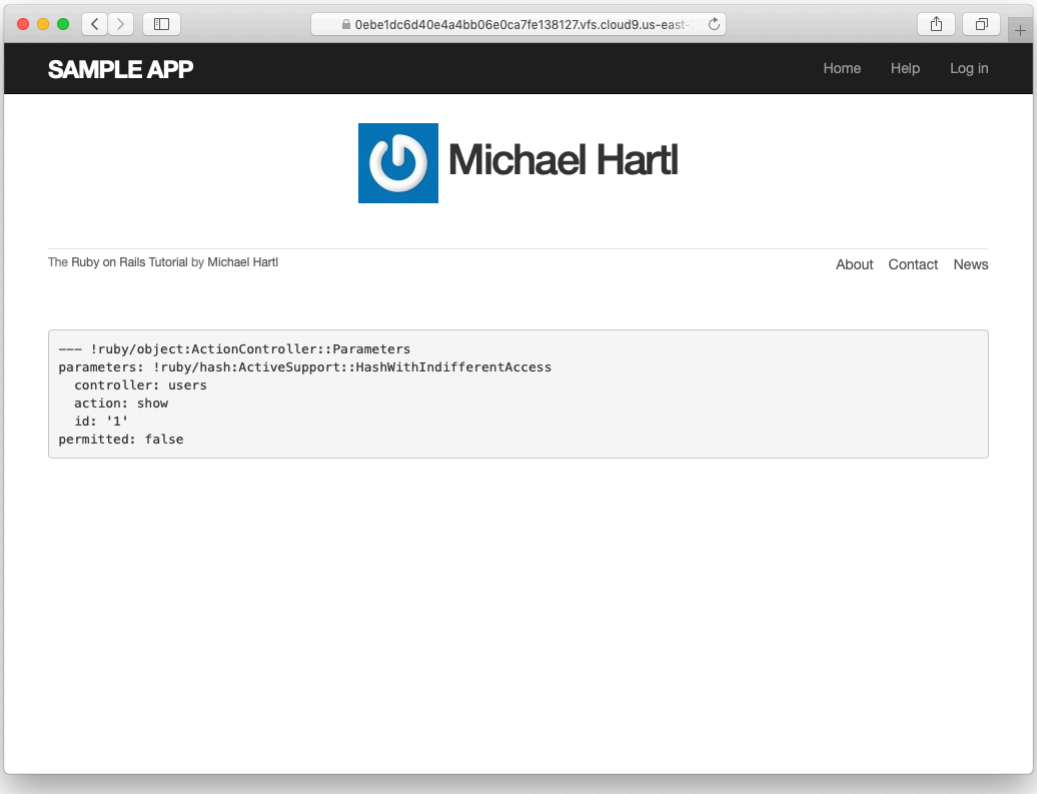


Figure 7.8: The user profile page with the default Gravatar.

```
$ rails console
>> user = User.first
>> user.update(name: "Example User",
?>             email: "example@railstutorial.org",
?>             password: "foobar",
?>             password_confirmation: "foobar")
=> true
```

Here we've assigned the user the email address **example@railstutorial.org**, which I've associated with the Rails Tutorial logo, as seen in Figure 7.9.

The last element needed to complete the mockup from Figure 7.1 is the initial version of the user sidebar. We'll implement it using the **aside** tag, which is used for content (such as sidebars) that complements the rest of the page but can also stand alone. We include **row** and **col-md-4** classes, which are both part of Bootstrap. The code for the modified user show page appears in Listing 7.10.

Listing 7.10: Adding a sidebar to the user **show** view.
app/views/users/show.html.erb

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="col-md-4">
    <section class="user_info">
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
  </aside>
</div>
```

With the HTML elements and CSS classes in place, we can style the profile page (including the sidebar and the Gravatar) with the SCSS shown in Listing 7.11.[10] (Note the nesting of the table CSS rules, which works only because of the Sass engine used by the asset pipeline.) The resulting page is shown in Figure 7.10.

actually allows the confirmation to be **nil**. The reason is so that apps that don't need password confirmation can simply omit the confirmation field. We do want a confirmation, though, so we'll include such a field in Listing 7.15.

[10] Listing 7.11 includes the **.gravatar_edit** class, which we'll put to work in Chapter 10.

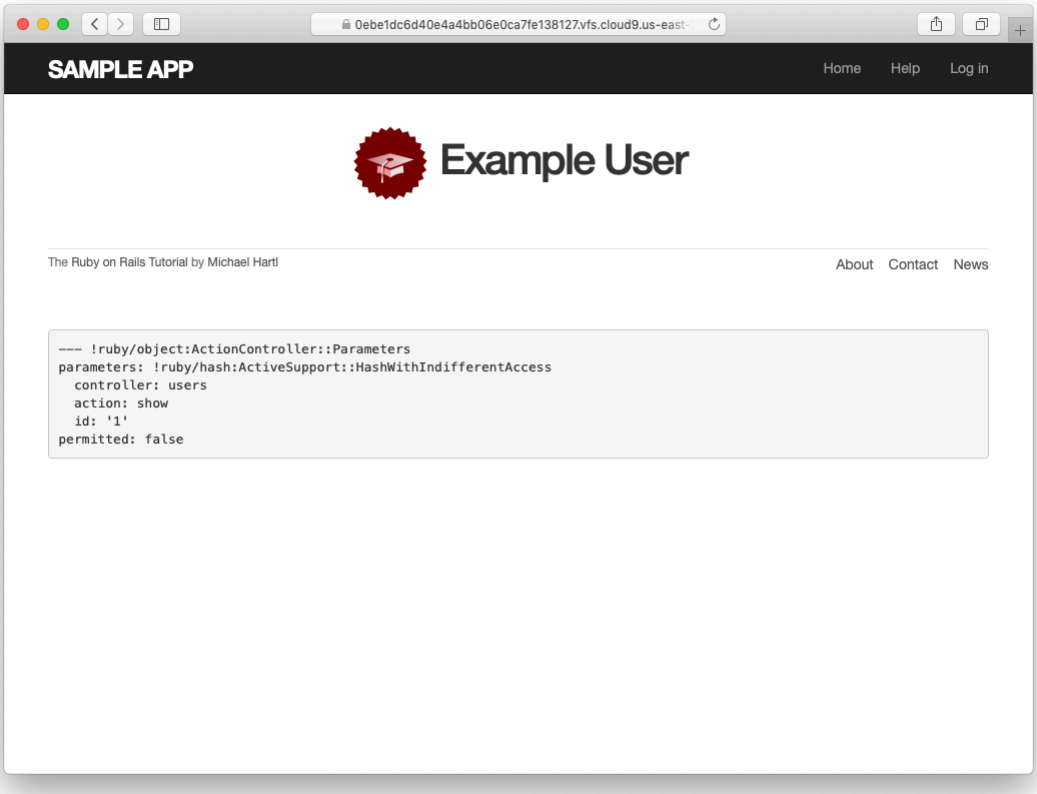


Figure 7.9: The user show page with a custom Gravatar.

Listing 7.11: SCSS for styling the user show page, including the sidebar.
app/assets/stylesheets/custom.scss

```
.
.
.
/* sidebar */

aside {
  section.user_info {
    margin-top: 20px;
  }
  section {
    padding: 10px 0;
    margin-top: 20px;
    &:first-child {
      border: 0;
      padding-top: 0;
    }
    span {
      display: block;
      margin-bottom: 3px;
      line-height: 1;
    }
    h1 {
      font-size: 1.4em;
      text-align: left;
      letter-spacing: -1px;
      margin-bottom: 3px;
      margin-top: 0px;
    }
  }
}

.gravatar {
  float: left;
  margin-right: 10px;
}

.gravatar_edit {
  margin-top: 15px;
}
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

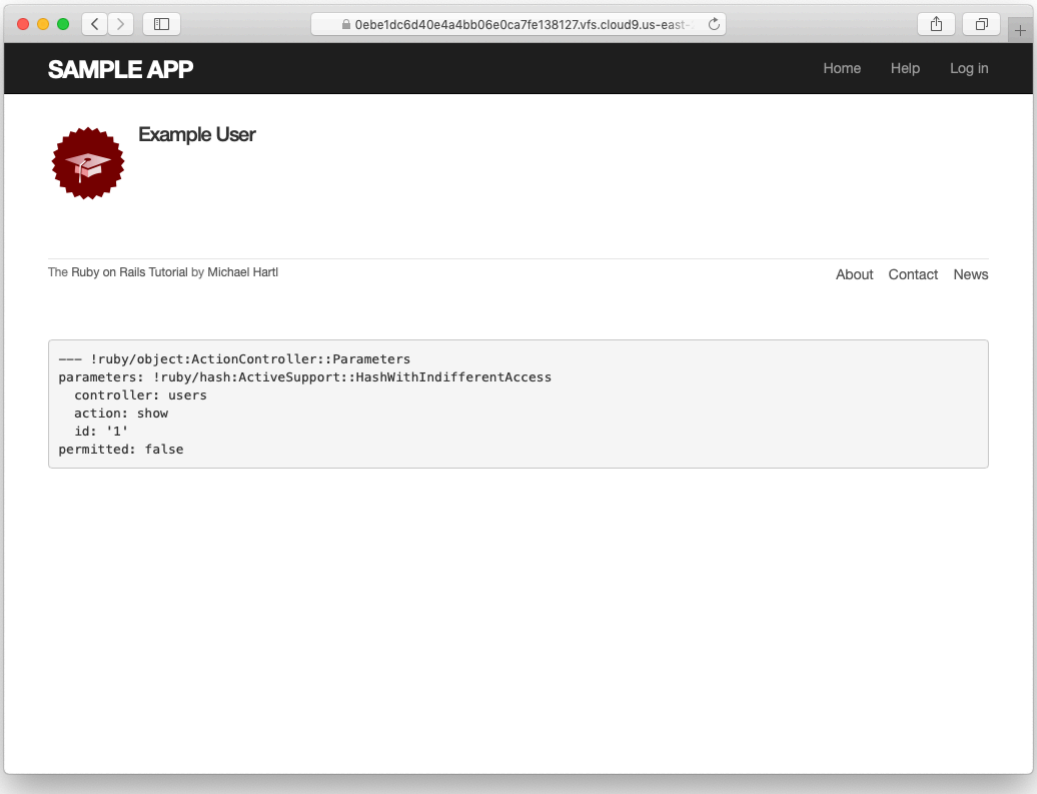


Figure 7.10: The user show page with a sidebar and CSS.

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. Associate a Gravatar with your primary email address if you haven't already. What is the MD5 hash associated with the image?
2. Verify that the code in Listing 7.12 allows the **`gravatar_for`** helper defined in Section 7.1.4 to take an optional **`size`** parameter, allowing code like **`gravatar_for user, size: 50`** in the view. (We'll put this improved helper to use in Section 10.3.1.)
3. The options hash used in the previous exercise is still commonly used, but as of Ruby 2.0 we can use *keyword arguments* instead. Confirm that the code in Listing 7.13 can be used in place of Listing 7.12. What are the diffs between the two?

Listing 7.12: Adding an options hash in the **`gravatar_for`** helper.
app/helpers/users_helper.rb

```
module UsersHelper

  # Returns the Gravatar for the given user.
  def gravatar_for(user, options = { size: 80 })
    size         = options[:size]
    gravatar_id  = Digest::MD5::hexdigest(user.email.downcase)
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}?s=#{size}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

Listing 7.13: Using keyword arguments in the **`gravatar_for`** helper.
app/helpers/users_helper.rb

```
module UsersHelper

  # Returns the Gravatar for the given user.
  def gravatar_for(user, size: 80)
    gravatar_id  = Digest::MD5::hexdigest(user.email.downcase)
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}?s=#{size}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

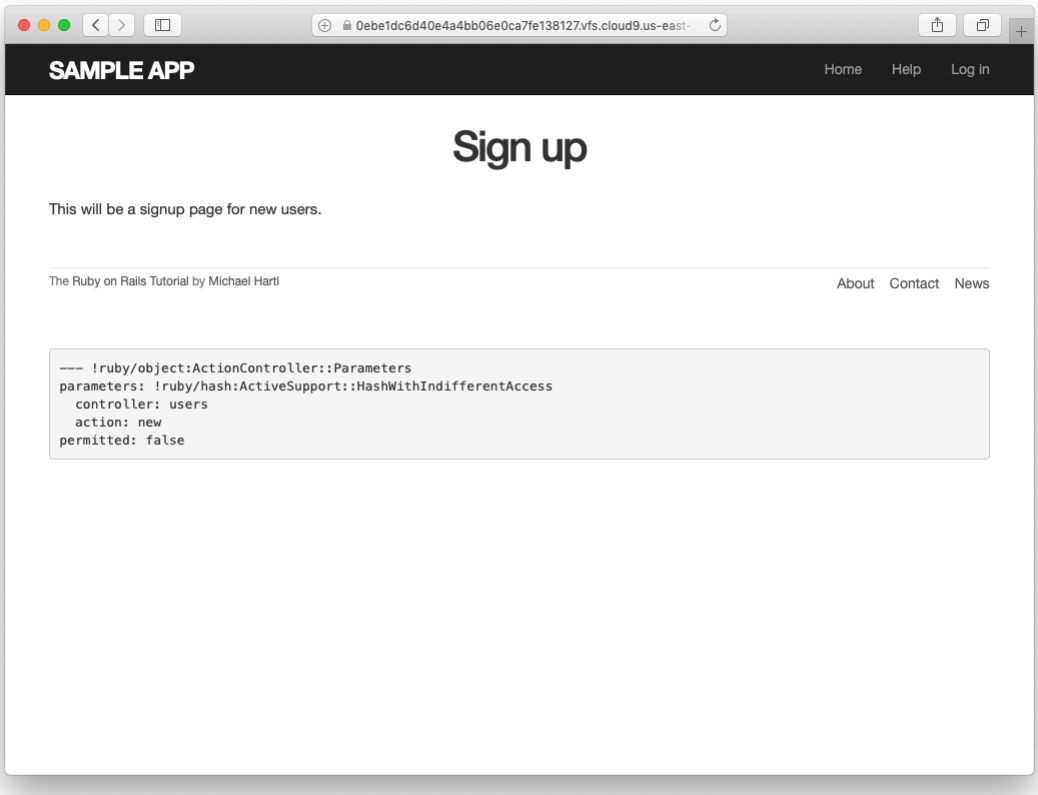


Figure 7.11: The current state of the signup page /signup.

7.2 Signup form

Now that we have a working (though not yet complete) user profile page, we're ready to make a signup form for our site. We saw in Figure 5.11 (shown again in Figure 7.11) that the signup page is currently blank: useless for signing up new users. The goal of this section is to start changing this sad state of affairs by producing the signup form mocked up in Figure 7.12.

Figure 7.12: A mockup of the user signup page.

7.2.1 Using `form_with`

The heart of the signup page is a *form* for submitting the relevant signup information (name, email, password, confirmation). We can accomplish this in Rails with the **`form_with`** helper method, which uses an Active Record object to build a form using the object's attributes.

Recalling that the signup page /signup is routed to the **`new`** action in the Users controller (Listing 5.43), our first step is to create the User object required as an argument to **`form_with`**. The resulting **`@user`** variable definition appears in Listing 7.14.

Listing 7.14: Adding an **`@user`** variable to the **`new`** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end
end
```

The form itself appears as in Listing 7.15. We'll discuss it in detail in Section 7.2.2, but first let's style it a little with the SCSS in Listing 7.16. (Note the reuse of the **`box_sizing`** mixin from Listing 7.2.) Once these CSS rules have been applied, the signup page appears as in Figure 7.13.

Listing 7.15: A form to sign up new users.
app/views/users/new.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, local: true) do |f| %>
      <%= f.label :name %>
```

```
      <%= f.text_field :name %>

      <%= f.label :email %>
      <%= f.email_field :email %>

      <%= f.label :password %>
      <%= f.password_field :password %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation %>

      <%= f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Listing 7.16: CSS for the signup form.
app/assets/stylesheets/custom.scss

```
.
.
.
/* forms */

input, textarea, select, .uneditable-input {
  border: 1px solid #bbb;
  width: 100%;
  margin-bottom: 15px;
  @include box_sizing;
}

input {
  height: auto !important;
}
```

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. Confirm by replacing all occurrences of **`f`** with **`foobar`** that the name of the block variable is irrelevant as far as the result is concerned. Why might **`foobar`** nevertheless be a bad choice?

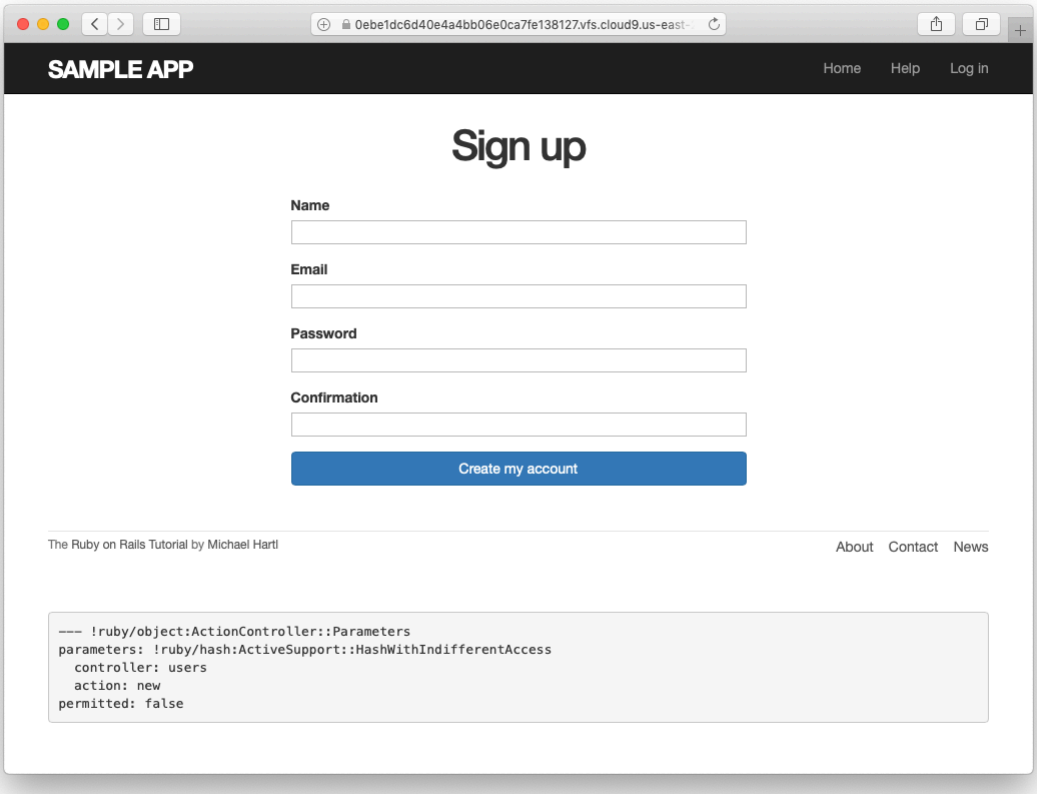


Figure 7.13: The user signup form.

7.2.2 Signup form HTML

To understand the form defined in Listing 7.15, it's helpful to break it into smaller pieces. We'll first look at the outer structure, which consists of embedded Ruby opening with a call to `form_with` and closing with `end`:

```
<%= form_with(model: @user, local: true) do |f| %>
  .
  .
  .
<% end %>
```

The presence of the `do` keyword indicates that `form_with` takes a block with one variable, which we've called `f` (for "form"). Note the presence of the hash argument `local: true`; by default, `form_with` sends a "remote" XHR request, whereas we want a regular "local" form request, mostly so that our error messages will render properly (Section 7.3.3).

As is usually the case with Rails helpers, we don't need to know any details about the implementation, but what we *do* need to know is what the `f` object does: when called with a method corresponding to an HTML form element—such as a text field, radio button, or password field—`f` returns code for that element specifically designed to set an attribute of the `@user` object. In other words,

```
<%= f.label :name %>
<%= f.text_field :name %>
```

creates the HTML needed to make a labeled text field element appropriate for setting the `name` attribute of a User model.

If you look at the HTML for the generated form by Ctrl-clicking and using the "inspect element" function of your browser, the page's source should look something like Listing 7.17. Let's take a moment to discuss its structure.

Listing 7.17: The HTML for the form in Figure 7.13.

```
<form accept-charset="UTF-8" action="/users" class="new_user"
      id="new_user" method="post">
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />
  <label for="user_name">Name</label>
  <input id="user_name" name="user[name]" type="text" />

  <label for="user_email">Email</label>
  <input id="user_email" name="user[email]" type="email" />

  <label for="user_password">Password</label>
  <input id="user_password" name="user[password]"
         type="password" />

  <label for="user_password_confirmation">Confirmation</label>
  <input id="user_password_confirmation"
         name="user[password_confirmation]" type="password" />

  <input class="btn btn-primary" name="commit" type="submit"
         value="Create my account" />
</form>
```

We'll start with the internal structure of the document. Comparing Listing 7.15 with Listing 7.17, we see that the embedded Ruby

```
<%= f.label :name %>
<%= f.text_field :name %>
```

produces the HTML

```
<label for="user_name">Name</label>
<input id="user_name" name="user[name]" type="text" />
```

while

```
<%= f.label :email %>
<%= f.email_field :email %>
```

produces the HTML

```
<label for="user_email">Email</label>
<input id="user_email" name="user[email]" type="email" />
```

and

```
<%= f.label :password %>
<%= f.password_field :password %>
```

produces the HTML

```
<label for="user_password">Password</label>
<input id="user_password" name="user[password]" type="password" />
```

As seen in Figure 7.14, text and email fields (**type="text"** and **type="email"**) simply display their contents, whereas password fields (**type="password"**) obscure the input for security purposes, as seen in Figure 7.14. (The benefit of using an email field is that some systems treat it differently from a text field; for example, the code **type="email"** will cause some mobile devices to display a special keyboard optimized for entering email addresses.)

As we'll see in Section 7.4, the key to creating a user is the special **name** attribute in each **input**:

```
<input id="user_name" name="user[name]" - - - />
.
.
.
<input id="user_password" name="user[password]" - - - />
```

These **name** values allow Rails to construct an initialization hash (via the **params** variable) for creating users using the values entered by the user, as we'll see in Section 7.3.

The second important element is the **form** tag itself. Rails creates the **form** tag using the **@user** object: because every Ruby object knows its own class

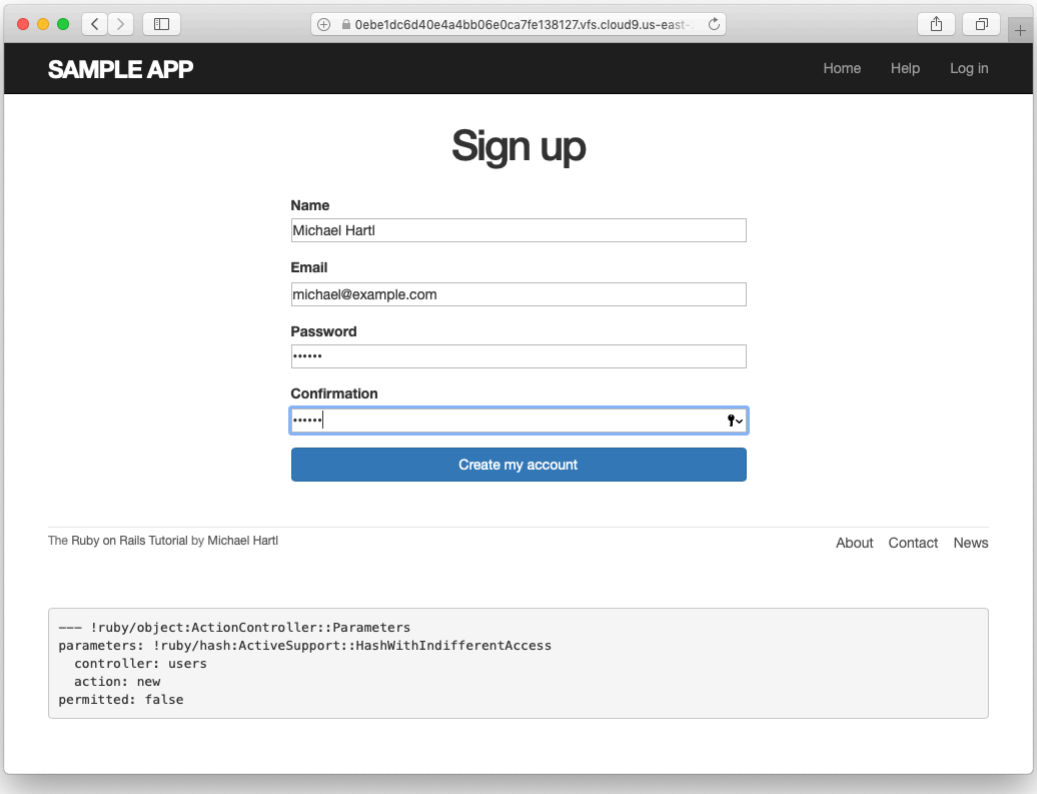


Figure 7.14: A filled-in form with **`text`** and **`password`** fields.

(Section 4.4.1), Rails figures out that `@user` is of class `User`; moreover, since `@user` is a *new* user, Rails knows to construct a form with the `post` method, which is the proper verb for creating a new object (Box 3.2):

```
<form action="/users" class="new_user" id="new_user" method="post">
```

Here the `class` and `id` attributes are largely irrelevant; what's important is `action="/users"` and `method="post"`. Together, these constitute instructions to issue an HTTP `POST` request to the /users URL. We'll see in the next two sections what effects this has.

(You may also have noticed the code that appears just inside the `form` tag:

```
<input name="authenticity_token" type="hidden"
       value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />
```

This code, which isn't displayed in the browser, is used internally by Rails, so it's not important for us to understand what it does. Briefly, it includes an *authenticity token*, which Rails uses to thwart an attack called a *cross-site request forgery* (CSRF). Knowing when it's OK to ignore details like this is a good mark of technical sophistication (Box 1.2).)[11]

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. *Learn Enough HTML to Be Dangerous*, in which all HTML is written by hand, doesn't cover the `form` tag. Why not?

[11] See the Stack Overflow entry on the Rails authenticity token if you're interested in the details of how this works.

7.3 Unsuccessful signups

Although we've briefly examined the HTML for the form in Figure 7.13 (shown in Listing 7.17), we haven't yet covered any details, and the form is best understood in the context of *signup failure*. In this section, we'll create a signup form that accepts an invalid submission and re-renders the signup page with a list of errors, as mocked up in Figure 7.15.

7.3.1 A working form

Recall from Section 7.1.2 that adding `resources :users` to the `routes.rb` file (Listing 7.3) automatically ensures that our Rails application responds to the RESTful URLs from Table 7.1. In particular, it ensures that a `POST` request to /users is handled by the `create` action. Our strategy for the `create` action is to use the form submission to make a new user object using `User.new`, try (and fail) to save that user, and then render the signup page for possible resubmission. Let's get started by reviewing the code for the signup form:

```
<form action="/users" class="new_user" id="new_user" method="post">
```

As noted in Section 7.2.2, this HTML issues a `POST` request to the /users URL.

Our first step toward a working signup form is adding the code in Listing 7.18. This listing includes a second use of the `render` method, which we first saw in the context of partials (Section 5.1.3); as you can see, `render` works in controller actions as well. Note that we've taken this opportunity to introduce an `if-else` branching structure, which allows us to handle the cases of failure and success separately based on the value of `@user.save`, which (as we saw in Section 6.1.3) is either `true` or `false` depending on whether or not the save succeeds.

Listing 7.18: A `create` action that can handle signup failure.
app/controllers/users_controller.rb

Figure 7.15: A mockup of the signup failure page.

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])    # Not the final implementation!
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end
end
```

Note the comment: this is not the final implementation. But it's enough to get us started, and we'll finish the implementation in Section 7.3.2.

The best way to understand how the code in Listing 7.18 works is to *submit* the form with some invalid signup data. The result appears in Figure 7.16, and the full debug information appears in Figure 7.17.

To get a better picture of how Rails handles the submission, let's take a closer look at the **user** part of the parameters hash from the debug information (Figure 7.17):

```
"user" => { "name" => "Foo Bar",
            "email" => "foo@invalid",
            "password" => "[FILTERED]",
            "password_confirmation" => "[FILTERED]"
          }
```

This hash gets passed to the Users controller as part of **params**, and we saw starting in Section 7.1.2 that the **params** hash contains information about each request. In the case of a URL like /users/1, the value of **params[:id]** is the **id** of the corresponding user (**1** in this example). In the case of posting to the signup form, **params** instead contains a hash of hashes, a construction we first

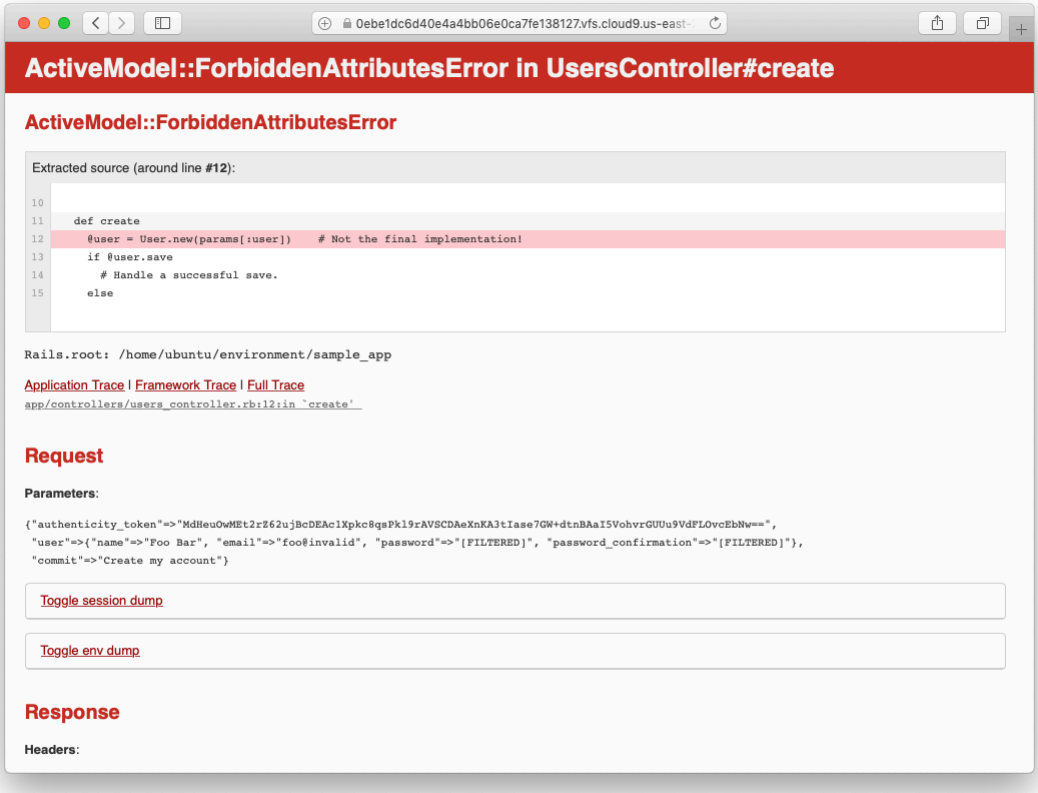


Figure 7.16: Signup failure upon submitting invalid data.

Figure 7.17: Signup failure debug information.

saw in Section 4.3.3, which introduced the strategically named **params** variable in a console session. The debug information above shows that submitting the form results in a **user** hash with attributes corresponding to the submitted values, where the keys come from the **name** attributes of the **input** tags seen in Listing 7.17. For example, the value of

```
<input id="user_email" name="user[email]" type="email" />
```

with name **"user[email]"** is precisely the **email** attribute of the **user** hash.

Although the hash keys appear as strings in the debug output, we can access them in the Users controller as symbols, so that **params[:user]** is the hash of user attributes—in fact, exactly the attributes needed as an argument to **User.new**, as first seen in Section 4.4.5 and appearing in Listing 7.18. This means that the line

```
@user = User.new(params[:user])
```

is mostly equivalent to

```
@user = User.new(name: "Foo Bar", email: "foo@invalid",
                 password: "foo", password_confirmation: "bar")
```

In previous versions of Rails, using

```
@user = User.new(params[:user])
```

actually worked, but it was insecure by default and required a careful and error-prone procedure to prevent malicious users from potentially modifying the application database. In Rails version later than 4.0, this code raises an error (as seen in Figure 7.16 and Figure 7.17 above), which means it is secure by default.

7.3.2 Strong parameters

We mentioned briefly in Section 4.4.5 the idea of *mass assignment*, which involves initializing a Ruby variable using a hash of values, as in

```
@user = User.new(params[:user])    # Not the final implementation!
```

The comment included in Listing 7.18 and reproduced above indicates that this is not the final implementation. The reason is that initializing the entire **params** hash is *extremely* dangerous—it arranges to pass to **User.new** *all* data submitted by a user. In particular, suppose that, in addition to the current attributes, the User model included an **admin** attribute used to identify administrative users of the site. (We will implement just such an attribute in Section 10.4.1.) The way to set such an attribute to **true** is to pass the value **admin='1'** as part of **params[:user]**, a task that is easy to accomplish using a command-line HTTP client such as `curl`. The result would be that, by passing in the entire **params** hash to **User.new**, we would allow any user of the site to gain administrative access by including **admin='1'** in the web request.

Previous versions of Rails used a method called **attr_accessible** in the *model* layer to solve this problem, and you may still see that method in legacy Rails applications, but as of Rails 4.0 the preferred technique is to use so-called *strong parameters* in the controller layer. This allows us to specify which parameters are *required* and which ones are *permitted*. In addition, passing in a raw **params** hash as above will cause an error to be raised, so that Rails applications are now immune to mass assignment vulnerabilities by default.

In the present instance, we want to require the **params** hash to have a **:user** attribute, and we want to permit the name, email, password, and password confirmation attributes (but no others). We can accomplish this as follows:

```
params.require(:user).permit(:name, :email, :password, :password_confirmation)
```

This code returns a version of the **params** hash with only the permitted attributes (while raising an error if the **:user** attribute is missing).

To facilitate the use of these parameters, it's conventional to introduce an auxiliary method called **user_params** (which returns an appropriate initialization hash) and use it in place of **params[:user]**:

```
@user = User.new(user_params)
```

Since **user_params** will only be used internally by the Users controller and need not be exposed to external users via the web, we'll make it *private* using Ruby's **private** keyword, as shown in Listing 7.19. (We'll discuss **private** in more detail in Section 9.1.)

Listing 7.19: Using strong parameters in the **create** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

By the way, the extra level of indentation on the **user_params** method is designed to make it visually apparent which methods are defined after **private**. (Experience shows that this is a wise practice; in classes with a large number of methods, it is easy to define a private method accidentally, which leads to considerable confusion when it isn't available to call on the corresponding object.)

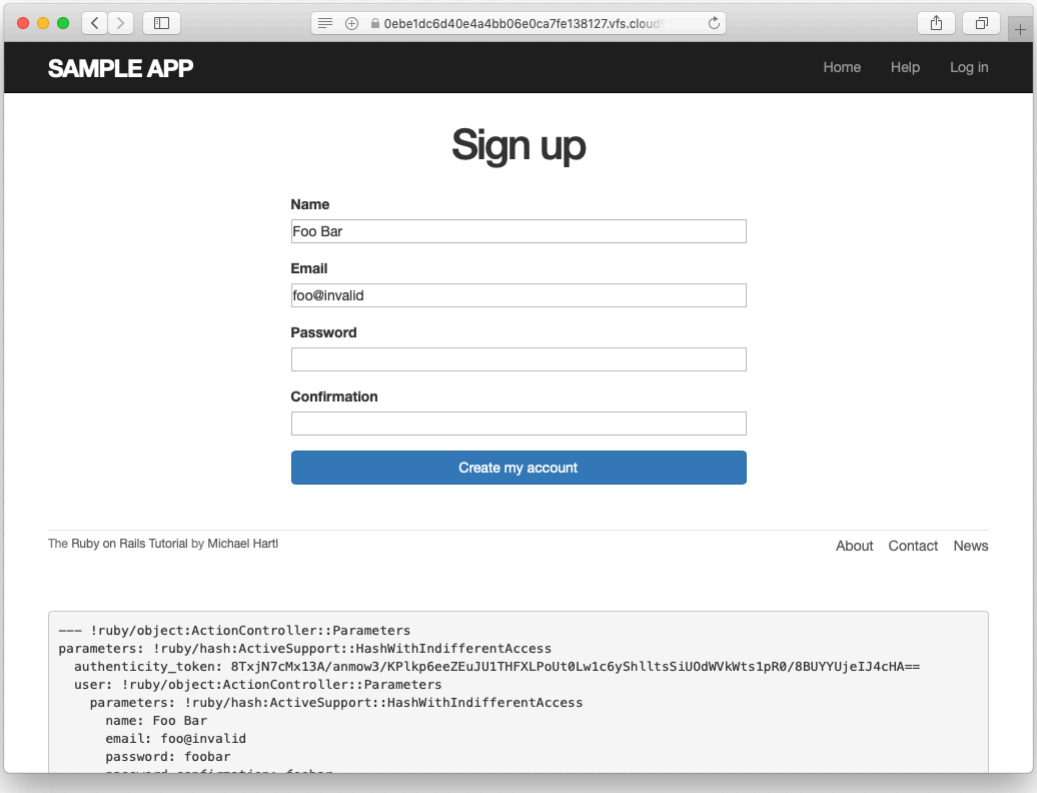


Figure 7.18: The signup form submitted with invalid information.

At this point, the signup form is working, at least in the sense that it no longer produces an error upon submission. On the other hand, as seen in Figure 7.18, it doesn't display any feedback on invalid submissions (apart from the development-only debug area), which is potentially confusing. It also doesn't actually create a new user. We'll fix the first issue in Section 7.3.3 and the second in Section 7.4.

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. By hitting the URL /signup?admin=1, confirm that the **admin** attribute appears in the **params** debug information.

7.3.3 Signup error messages

As a final step in handling failed user creation, we'll add helpful error messages to indicate the problems that prevented successful signup. Conveniently, Rails automatically provides such messages based on the User model validations. For example, consider trying to save a user with an invalid email address and with a password that's too short:

```
$ rails console
>> user = User.new(name: "Foo Bar", email: "foo@invalid",
?>                  password: "dude", password_confirmation: "dude")
>> user.save
=> false
>> user.errors.full_messages
=> ["Email is invalid", "Password is too short (minimum is 6 characters)"]
```

Here the **errors.full_messages** object (which we saw briefly before in Section 6.2.2) contains an array of error messages.

As in the console session above, the failed save in Listing 7.18 generates a list of error messages associated with the **@user** object. To display the messages in the browser, we'll render an error-messages partial on the user **new** page while adding the CSS class **form-control** (which has special meaning to Bootstrap) to each entry field, as shown in Listing 7.20. It's worth noting that this error-messages partial is only a first attempt; the final version appears in Section 13.3.2.

Listing 7.20: Code to display error messages on the signup form.
app/views/users/new.html.erb

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

```
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, local: true) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%= f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Notice here that we **render** a partial called **'shared/error_messages'**; this reflects the common Rails convention of using a dedicated **shared/** directory for partials expected to be used in views across multiple controllers. (We'll see this expectation fulfilled in Section 10.1.1.)

This means that we have to create a new **app/views/shared** directory using **mkdir** and an error messages partial using (Table 1.1):

```
$ mkdir app/views/shared
```

We then need to create the **_error_messages.html.erb** partial file using **touch** or the text editor as usual. The contents of the partial appear in Listing 7.21.

Listing 7.21: A partial for displaying form submission error messages.
app/views/shared/_error_messages.html.erb

```
<% if @user.errors.any? %>
  <div id="error_explanation">
```

```
    <div class="alert alert-danger">
      The form contains <%= pluralize(@user.errors.count, "error") %>.
    </div>
    <ul>
    <% @user.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

This partial introduces several new Rails and Ruby constructs, including two methods for Rails error objects. The first method is **`count`**, which simply returns the number of errors:

```
>> user.errors.count
=> 2
```

The other new method is **`any?`**, which (together with **`empty?`**) is one of a pair of complementary methods:

```
>> user.errors.empty?
=> false
>> user.errors.any?
=> true
```

We see here that the **`empty?`** method, which we first saw in Section 4.2.2 in the context of strings, also works on Rails error objects, returning **`true`** for an empty object and **`false`** otherwise. The **`any?`** method is just the opposite of **`empty?`**, returning **`true`** if there are any elements present and **`false`** otherwise. (By the way, all of these methods—**`count`**, **`empty?`**, and **`any?`**—work on Ruby arrays as well. We'll put this fact to good use starting in Section 13.2.)

The other new idea is the **`pluralize`** text helper, which is available in the console via the **`helper`** object:

```
>> helper.pluralize(1, "error")
=> "1 error"
>> helper.pluralize(5, "error")
=> "5 errors"
```

We see here that **pluralize** takes an integer argument and then returns the number with a properly pluralized version of its second argument. Underlying this method is a powerful *inflector* that knows how to pluralize a large number of words, including many with irregular plurals:

```
>> helper.pluralize(2, "woman")
=> "2 women"
>> helper.pluralize(3, "erratum")
=> "3 errata"
```

As a result of its use of **pluralize**, the code

```
<%= pluralize(@user.errors.count, "error") %>
```

returns **"0 errors"**, **"1 error"**, **"2 errors"**, and so on, depending on how many errors there are, thereby avoiding ungrammatical phrases such as **"1 errors"** (a distressingly common mistake in both web and desktop applications).

Note that Listing 7.21 includes the CSS id **error_explanation** for use in styling the error messages. (Recall from Section 5.1.2 that CSS uses the pound sign **#** to style ids.) In addition, after an invalid submission Rails automatically wraps the fields with errors in **div**s with the CSS class **field_with_errors**. These labels then allow us to style the error messages with the SCSS shown in Listing 7.22, which makes use of Sass's **@extend** function to include the functionality of the Bootstrap class **has-error**.

Listing 7.22: CSS for styling error messages.
app/assets/stylesheets/custom.scss

```
.
.
.
/* forms */
.
.
.
#error_explanation {
  color: red;
  ul {
    color: red;
    margin: 0 0 30px 0;
  }
}

.field_with_errors {
  @extend .has-error;
  .form-control {
    color: $state-danger-text;
  }
}
```

With the code in Listing 7.20 and Listing 7.21 and the SCSS from Listing 7.22, helpful error messages now appear when submitting invalid signup information, as seen in Figure 7.19. Because the messages are generated by the model validations, they will automatically change if you ever change your mind about, say, the format of email addresses, or the minimum length of passwords. (*Note*: Because both the presence validation and the **`has_secure_password`** validation catch the case of *empty* (**`nil`**) passwords, the signup form currently produces duplicate error messages when the user submits empty passwords. We could manipulate the error messages directly to eliminate duplicates, but luckily this issue will be fixed automatically by the addition of **`allow_nil: true`** in Section 10.1.4.)

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. Confirm by changing the minimum length of passwords to 5 that the error message updates automatically as well.

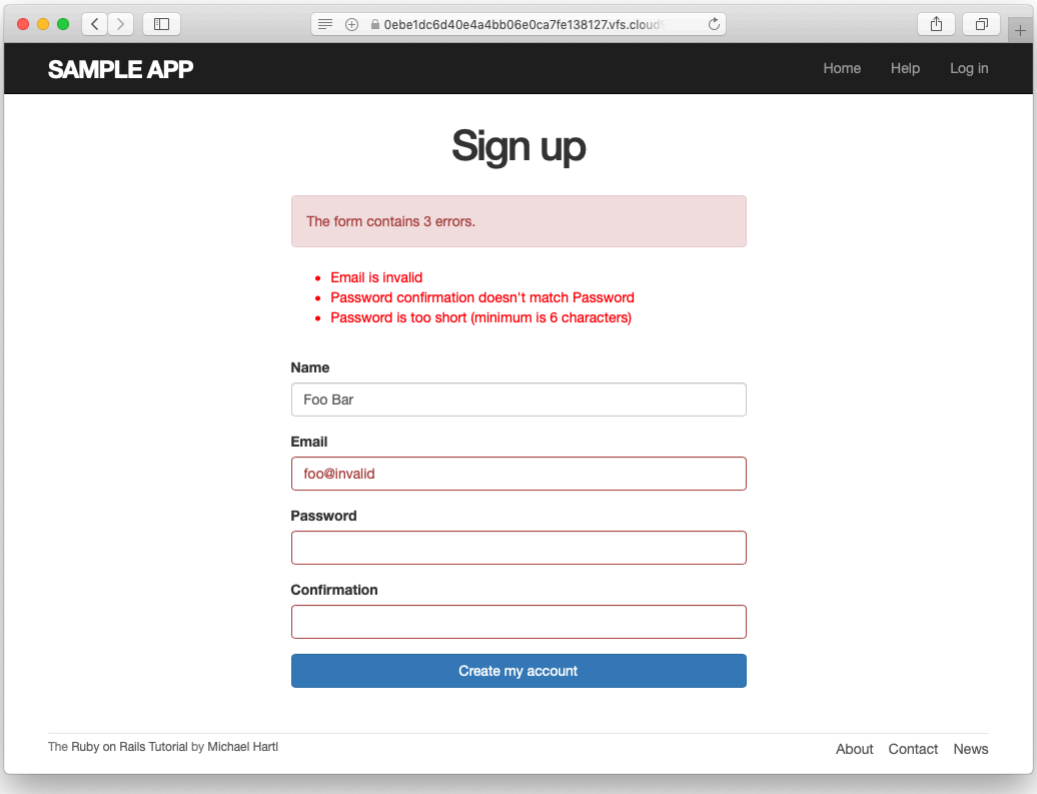


Figure 7.19: Failed signup with error messages.

2. How does the URL on the unsubmitted signup form (Figure 7.13) compare to the URL for a submitted signup form (Figure 7.19)? Why don't they match?

7.3.4 A test for invalid submission

In the days before powerful web frameworks with automated testing capabilities, developers had to test forms by hand. For example, to test a signup page manually, we would have to visit the page in a browser and then submit alternately invalid and valid data, verifying in each case that the application's behavior was correct. Moreover, we would have to remember to repeat the process any time the application changed. This process was painful and error-prone.

Happily, with Rails we can write tests to automate the testing of forms. In this section, we'll write one such test to verify the correct behavior upon invalid form submission; in Section 7.4.4, we'll write a corresponding test for valid submission.

To get started, we first generate an integration test file for signing up users, which we'll call **`users_signup`** (adopting the controller convention of a plural resource name):

```
$ rails generate integration_test users_signup
      invoke  test_unit
      create    test/integration/users_signup_test.rb
```

(We'll use this same file in Section 7.4.4 to test a valid signup.)

The main purpose of our test is to verify that clicking the signup button results in *not* creating a new user when the submitted information is invalid. (Writing a test for the error messages is left as an exercise (Section 7.3.4).) The way to do this is to check the *count* of users, and under the hood our tests will use the **`count`** method available on every Active Record class, including **`User`**:

```
$ rails console
>> User.count
=> 1
```

(Here `User.count` is `1` because of the user created in Section 6.3.4, though it may differ if you've added or deleted any users in the interim.) As in Section 5.3.4, we'll use `assert_select` to test HTML elements of the relevant pages, taking care to check only elements unlikely to change in the future.

We'll start by visiting the signup path using `get`:

```
get signup_path
```

In order to test the form submission, we need to issue a `POST` request to the `users_path` (Table 7.1), which we can do with the `post` function:

```
assert_no_difference 'User.count' do
  post users_path, params: { user: { name:  "",
                                     email: "user@invalid",
                                     password:              "foo",
                                     password_confirmation: "bar" } }
end
```

Here we've included the `params[:user]` hash expected by `User.new` in the `create` action (Listing 7.27). (In versions of Rails before 5, `params` was implicit, and only the `user` hash would be passed. This practice was deprecated in Rails 5.0, and now the recommended method is to include the full `params` hash explicitly.)

By wrapping the `post` in the `assert_no_difference` method with the string argument `'User.count'`, we arrange for a comparison between `User.count` before and after the contents inside the `assert_no_difference` block. This is equivalent to recording the user count, posting the data, and verifying that the count is the same:

```
before_count = User.count
post users_path, ...
after_count  = User.count
assert_equal before_count, after_count
```

Although the two are equivalent, using `assert_no_difference` is cleaner and is more idiomatically correct Ruby.

It's worth noting that the **get** and **post** steps above are technically unrelated, and it's actually not necessary to get the signup path before posting to the users path. I prefer to include both steps, though, both for conceptual clarity and to double-check that the signup form renders without error.

Putting the above ideas together leads to the test in Listing 7.23. We've also included a call to **assert_template** to check that a failed submission re-renders the **new** action. Adding lines to check for the appearance of error messages is left as an exercise (Section 7.3.4).

Listing 7.23: A test for an invalid signup. GREEN
test/integration/users_signup_test.rb

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

  test "invalid signup information" do
    get signup_path
    assert_no_difference 'User.count' do
      post users_path, params: { user: { name:  "",
                                         email: "user@invalid",
                                         password:              "foo",
                                         password_confirmation: "bar" } }
    end
    assert_template 'users/new'
  end
end
```

Because we wrote the application code before the integration test, the test suite should be GREEN:

Listing 7.24: 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. Write a test for the error messages implemented in Listing 7.20. How detailed you want to make your tests is up to you; a suggested template appears in Listing 7.25.

Listing 7.25: A template for tests of the error messages.
test/integration/users_signup_test.rb

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

  test "invalid signup information" do
    get signup_path
    assert_no_difference 'User.count' do
      post users_path, params: { user: { name:  "",
                                         email: "user@invalid",
                                         password:              "foo",
                                         password_confirmation: "bar" } }
    end
    assert_template 'users/new'
    assert_select 'div#<CSS id for error explanation>'
    assert_select 'div.<CSS class for field with error>'
  end
  .
  .
  .
end
```

7.4 Successful signups

Having handled invalid form submissions, now it's time to complete the signup form by actually saving a new user (if valid) to the database. First, we try to save the user; if the save succeeds, the user's information gets written to the database automatically, and we then *redirect* the browser to show the user's profile (together with a friendly greeting), as mocked up in Figure 7.20. If it fails, we simply fall back on the behavior developed in Section 7.3.

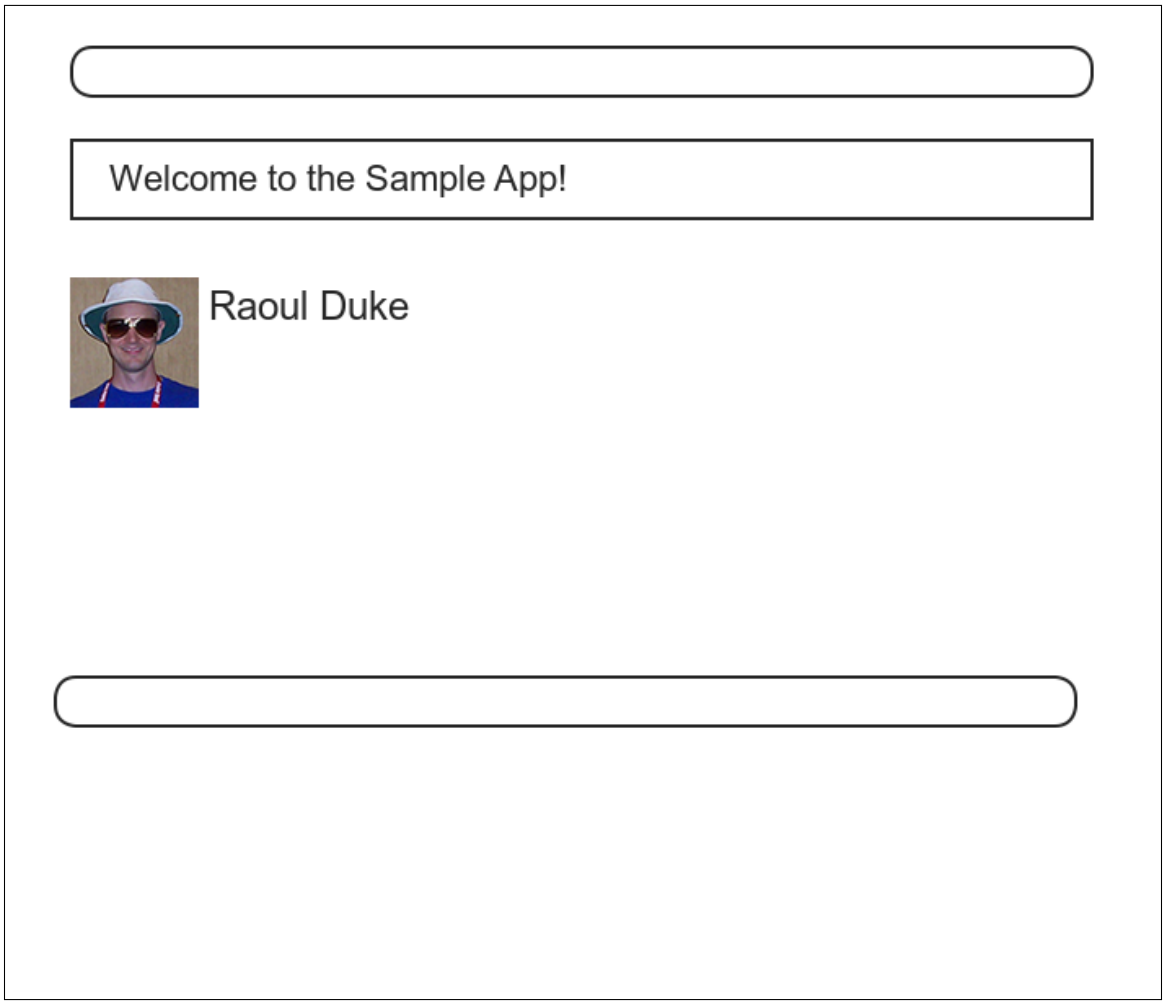


Figure 7.20: A mockup of successful signup.

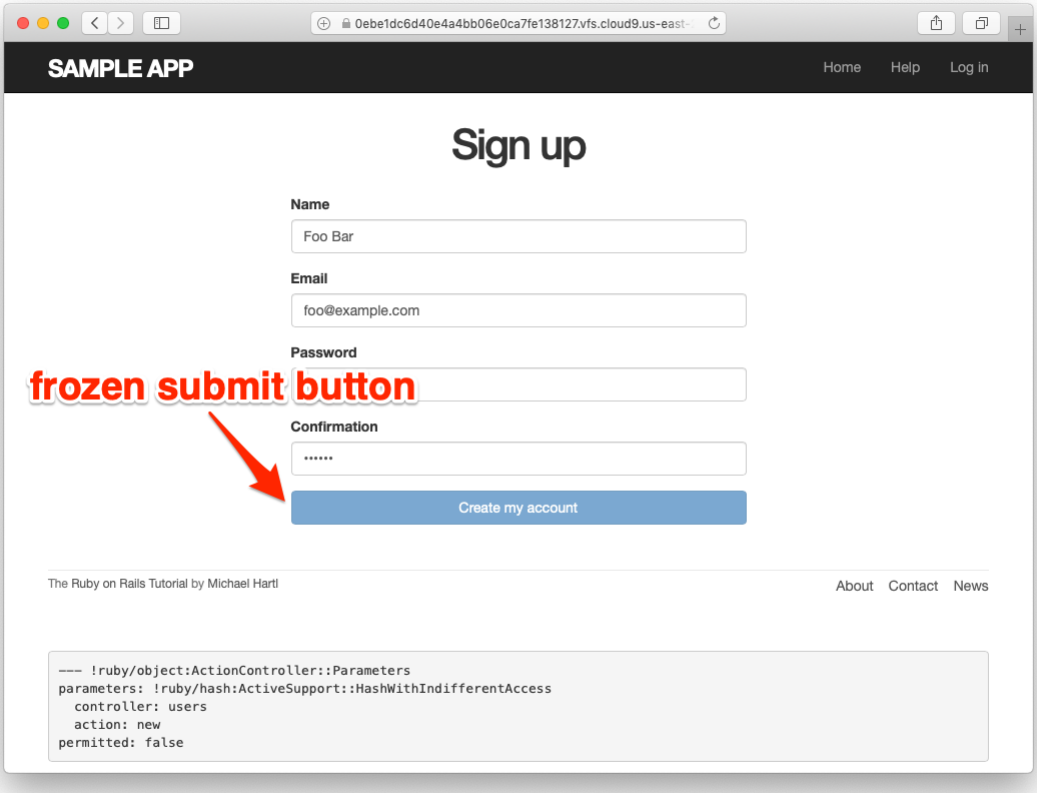


Figure 7.21: The frozen page on valid signup submission.

7.4.1 The finished signup form

To complete a working signup form, we need to fill in the commented-out section in Listing 7.19 with the appropriate behavior. Currently, the form simply freezes on valid submission, as indicated by the subtle color change in the submission button (Figure 7.21), although this behavior may be system-dependent. This is because the default behavior for a Rails action is to render the corresponding view, and there isn't a view template corresponding to the `create` action (Figure 7.22).

Although it's possible to render a template for the `create` action, the usual practice is to *redirect* to a different page instead when the creation is successful.

```
  ↳ app/controllers/users_controller.rb:13:in `create'
No template found for UsersController#create, rendering head :no_content
Completed 204 No Content in 332ms (ActiveRecord: 18.1ms | Allocations: 6720)
```

Figure 7.22: The **`create`** template error in the server log.

In particular, we'll follow the common convention of redirecting to the newly created user's profile, although the root path would also work. The application code, which introduces the **`redirect_to`** method, appears in Listing 7.26.

Listing 7.26: The user **`create`** action with a save and a redirect.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      redirect_to @user
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

Note that we've written

```
redirect_to @user
```

where we could have used the equivalent

```
redirect_to user_url(@user)
```

This is because Rails automatically infers from **`redirect_to @user`** that we want to redirect to **`user_url(@user)`**.

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 Rails console, verify that a user is in fact created when submitting valid information.

2. Confirm by updating Listing 7.26 and submitting a valid user that **`redirect_to user_url(@user)`** has the same effect as **`redirect_to @user`**.

7.4.2 The flash

With the code in Listing 7.26, our signup form is actually working, but before submitting a valid registration in a browser we're going to add a bit of polish common in web applications: a message that appears on the subsequent page (in this case, welcoming our new user to the application) and then disappears upon visiting a second page or on page reload.

The Rails way to display a temporary message is to use a special method called the *flash*, which we can treat like a hash. Rails adopts the convention of a **`:success`** key for a message indicating a successful result (Listing 7.27).

Listing 7.27: Adding a flash message to user signup.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

By assigning a message to the **flash**, we are now in a position to display the message on the first page after the redirect. Our method is to iterate through the **flash** and insert all relevant messages into the site layout. You may recall the console example in Section 4.3.3, where we saw how to iterate through a hash using the strategically named **flash** variable (Listing 7.28).

Listing 7.28: Iterating through a **flash** hash in the console.

```
$ rails console
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", danger: "It failed."}
>> flash.each do |key, value|
?>   puts "#{key}"
?>   puts "#{value}"
>> end
success
It worked!
danger
It failed.
```

By following this pattern, we can arrange to display the contents of the flash site-wide using code like this:

```
<% flash.each do |message_type, message| %>
  <div class="alert alert-<%= message_type %>"><%= message %></div>
<% end %>
```

(This code is a particularly ugly and difficult-to-read combination of HTML and ERb; making it prettier is left as an exercise (Section 7.4.4).) Here the embedded Ruby

```
alert-<%= message_type %>
```

makes a CSS class corresponding to the type of message, so that for a **`:success`** message the class is

```
alert-success
```

(The key **`:success`** is a symbol, but embedded Ruby automatically converts it to the string **`"success"`** before inserting it into the template.) Using a different class for each key allows us to apply different styles to different kinds of messages. For example, in Section 8.1.4 we'll use **`flash[:danger]`** to indicate a failed login attempt.[12] (In fact, we've already used **`alert-danger`** once, to style the error message div in Listing 7.21.) Bootstrap CSS supports styling for four such flash classes for increasingly urgent message types (**`success`**, **`info`**, **`warning`**, and **`danger`**), and we'll find occasion to use all of them in the course of developing the sample application (**`info`** in Section 11.2, **`warning`** in Section 11.3, and **`danger`** for the first time in Section 8.1.4).

Because the message is also inserted into the template, the full HTML result for

```
flash[:success] = "Welcome to the Sample App!"
```

appears as follows:

[12] Actually, we'll use the closely related **`flash.now`**, but we'll defer that subtlety until we need it.

```
<div class="alert alert-success">Welcome to the Sample App!</div>
```

Putting the embedded Ruby discussed above into the site layout leads to the code in Listing 7.29.

Listing 7.29: Adding the contents of the `flash` variable to the site layout.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  .
  .
  .
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <% flash.each do |message_type, message| %>
        <div class="alert alert-<%= message_type %>"><%= message %></div>
      <% end %>
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
    .
    .
    .
  </body>
</html>
```

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. In the console, confirm that you can use interpolation (Section 4.2.1) to interpolate a raw symbol. For example, what is the return value of `"#{:success}"`?

2. How does the previous exercise relate to the flash iteration shown in Listing 7.28?

7.4.3 The first signup

We can see the result of all this work by signing up the first user for the sample app. Even though previous submissions didn't work properly (as shown in Figure 7.21), the `user.save` line in the Users controller still works, so users might still have been created. To clear them out, we'll reset the database as follows:

```
$ rails db:migrate:reset
```

On some systems you might have to restart the webserver (using Ctrl-C) for the changes to take effect (Box 1.2).

We'll create the first user with the name "Rails Tutorial" and email address "example@railstutorial.org", as shown in Figure 7.23). The resulting page (Figure 7.24) shows a friendly flash message upon successful signup, including nice green styling for the `success` class, which comes included with the Bootstrap CSS framework from Section 5.1.2. Then, upon reloading the user show page, the flash message disappears as promised (Figure 7.25).

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 Rails console, find by the email address to double-check that the new user was actually created. The result should look something like Listing 7.30.

2. Create a new user with your primary email address. Verify that the Gravatar correctly appears.

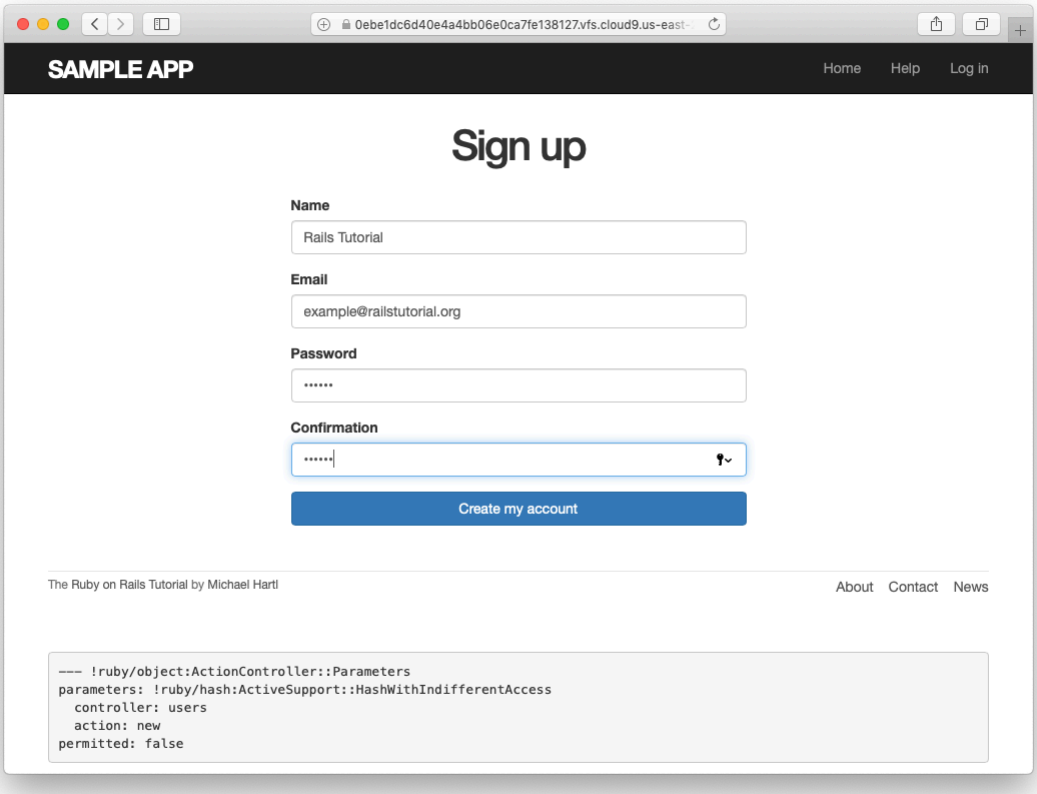


Figure 7.23: Filling in the information for the first signup.

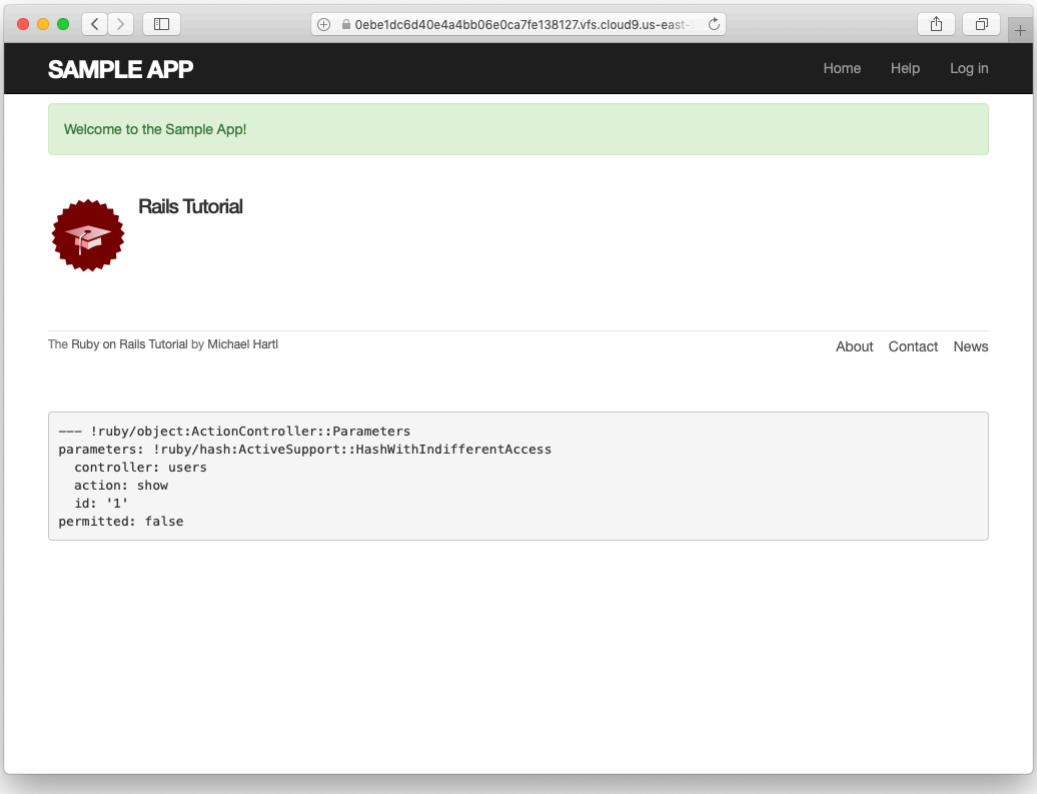


Figure 7.24: The results of a successful user signup, with flash message.

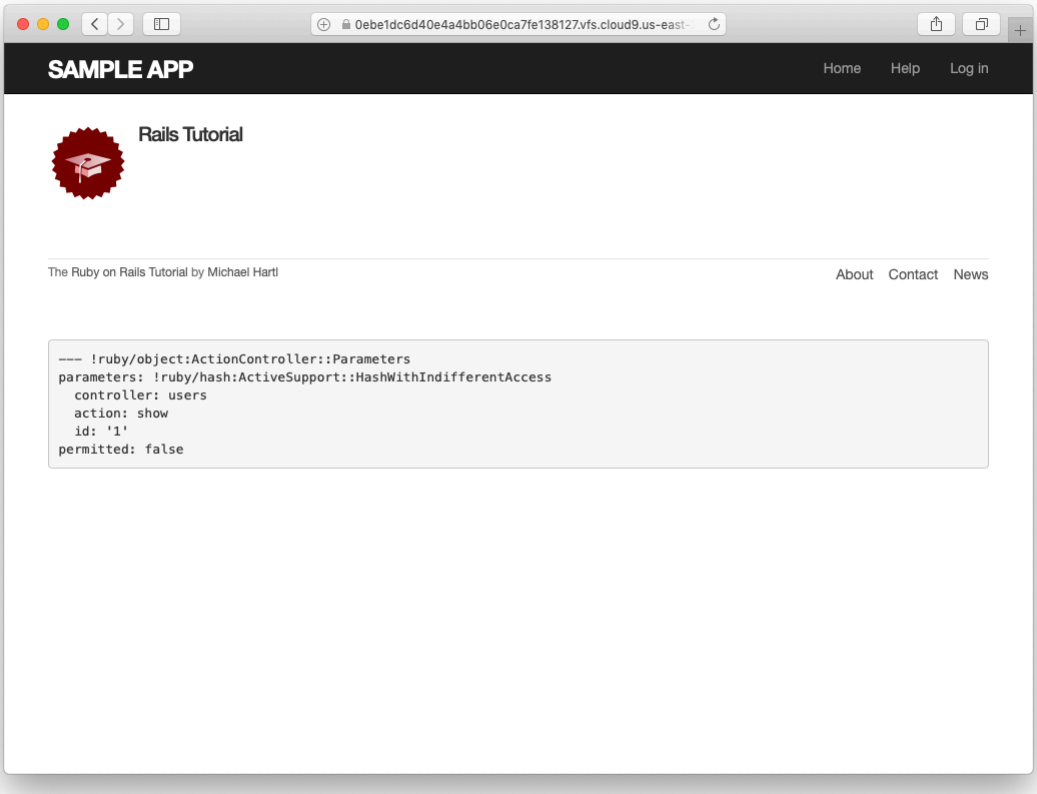


Figure 7.25: The `flash`-less profile page after a browser reload.

Listing 7.30: Finding the newly created user in the database.

```
$ rails console
>> User.find_by(email: "example@railstutorial.org")
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.
org", created_at: "2016-05-31 17:17:33", updated_at: "2016-05-31 17:17:33",
password_digest: "$2a$10$8MaeHdnOhZvMk3GmFdmpPOeG6a7u7/k2Z9TMjOanC9G...">
```

7.4.4 A test for valid submission

Before moving on, we'll write a test for valid submission to verify our application's behavior and catch regressions. As with the test for invalid submission in Section 7.3.4, our main purpose is to verify the contents of the database. In this case, we want to submit valid information and then confirm that a user *was* created. In analogy with Listing 7.23, which used

```
assert_no_difference 'User.count' do
  post users_path, ...
end
```

here we'll use the corresponding **`assert_difference`** method:

```
assert_difference 'User.count', 1 do
  post users_path, ...
end
```

As with **`assert_no_difference`**, the first argument is the string **`'User.count'`**, which arranges for a comparison between **`User.count`** before and after the contents of the **`assert_difference`** block. The second (optional) argument specifies the size of the difference (in this case, 1).

Incorporating **`assert_difference`** into the file from Listing 7.23 yields the test shown in Listing 7.31. Note that we've used the **`follow_redirect!`** method after posting to the users path. This simply arranges to follow the redirect after submission, resulting in a rendering of the **`'users/show'`** template. (It's probably a good idea to write a test for the flash as well, which is left as an exercise (Section 7.4.4).)

Listing 7.31: A test for a valid signup. GREEN
test/integration/users_signup_test.rb

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest
  .
  .
  .
  test "valid signup information" do
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, params: { user: { name:  "Example User",
                                         email: "user@example.com",
                                         password:              "password",
                                         password_confirmation: "password" } }
    end
    follow_redirect!
    assert_template 'users/show'
  end
end
```

Note that Listing 7.31 also verifies that the user show template renders following successful signup. For this test to work, it's necessary for the Users routes (Listing 7.3), the Users **show** action (Listing 7.5), and the **show.html.erb** view (Listing 7.8) to work correctly. As a result, the one line

```
assert_template 'users/show'
```

is a sensitive test for almost everything related to a user's profile page. This sort of end-to-end coverage of important application features illustrates one reason why integration tests are so useful.

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. Write a test for the flash implemented in Section 7.4.2. How detailed you want to make your tests is up to you; a suggested ultra-minimalist template appears in Listing 7.32, which you should complete by replacing **`FILL_IN`** with the appropriate code. (Even testing for the right key, much less the text, is likely to be brittle, so I prefer to test only that the flash isn't empty.)

2. As noted above, the flash HTML in Listing 7.29 is ugly. Verify by running the test suite that the cleaner code in Listing 7.33, which uses the Rails **`content_tag`** helper, also works.

3. Verify that the test fails if you comment out the redirect line in Listing 7.26.

4. Suppose we changed **`@user.save`** to **`false`** in Listing 7.26. How does this change verify that the **`assert_difference`** block is testing the right thing?

Listing 7.32: A template for a test of the flash.
test/integration/users_signup_test.rb

```
require 'test_helper'
  .
  .
  .
  test "valid signup information" do
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, params: { user: { name:  "Example User",
                                         email: "user@example.com",
                                         password:              "password",
                                         password_confirmation: "password" } }
    end
    follow_redirect!
    assert_template 'users/show'
    assert_not flash.FILL_IN
  end
end
```

Listing 7.33: The `flash` ERb in the site layout using `content_tag`.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
      .
      .
      .
      <% flash.each do |message_type, message| %>
        <%= content_tag(:div, message, class: "alert alert-#{message_type}") %>
      <% end %>
      .
      .
      .
</html>
```

7.5 Professional-grade deployment

Now that we have a working signup page, it's time to deploy our application and get it working in production. Although we started deploying our application in Chapter 3, this is the first time it will actually *do* something, so we'll take this opportunity to make the deployment professional-grade. In particular, we'll add an important feature to the production application to make signup secure, we'll replace the default webserver with one suitable for real-world use, and we'll add some configuration for our production database.

As preparation for the deployment, you should merge your changes into the `master` branch at this point:

```
$ git add -A
$ git commit -m "Finish user signup"
$ git checkout master
$ git merge sign-up
```

7.5.1 SSL in production

When submitting the signup form developed in this chapter, the name, email address, and password get sent over the network, and hence are vulnerable to

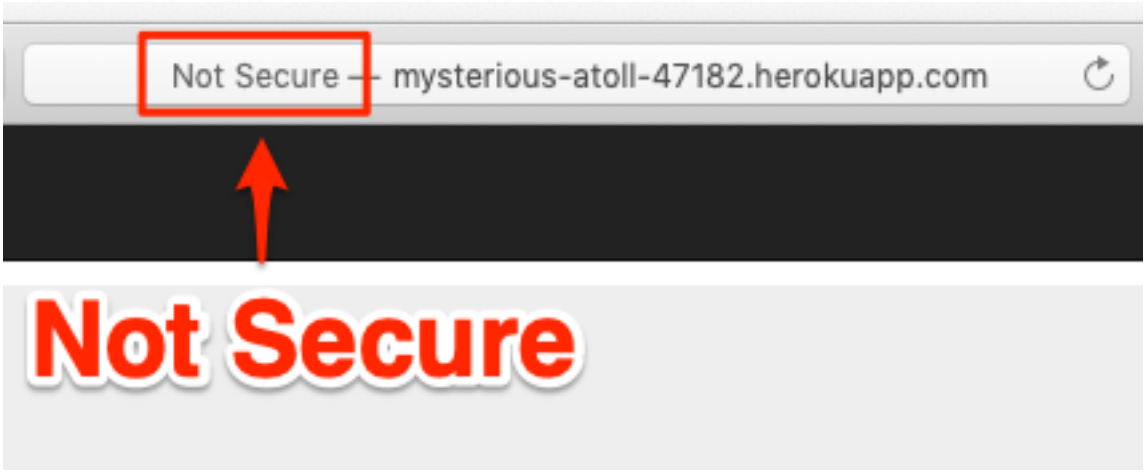


Figure 7.26: The result of using an insecure http URL in production.

being intercepted by malicious users. This is a potentially serious security flaw in our application, and the way to fix it is to use Secure Sockets Layer (SSL)[13] to encrypt all relevant information before it leaves the local browser. Although we could use SSL on just the signup page, it's actually easier to implement it site-wide, which has the additional benefits of securing user login (Chapter 8) and making our application immune to the critical *session hijacking* vulnerability discussed in Section 9.1.

Although Heroku uses SSL by default, it doesn't *force* browsers to use it, so any users hitting our application using regular http will be interacting insecurely with the site. You can see how this works by editing the URL in the address bar to change "https" to "http"; the result appears in Figure 7.26.

Luckily, forcing browsers to use SSL is as easy as uncommenting a single line in **`production.rb`**, the configuration file for production applications. As shown in Listing 7.34, all we need to do is set **`config.force_ssl`** to **`true`**.

Listing 7.34: Configuring the application to use SSL in production.
`config/environments/production.rb`

```
Rails.application.configure do
  .
```

[13] Technically, SSL is now TLS, for Transport Layer Security, but everyone I know still says "SSL".

```
  .
  .
  # Force all access to the app over SSL, use Strict-Transport-Security,
  # and use secure cookies.
  config.force_ssl = true
  .
  .
  .
end
```

At this stage, we need to set up SSL on the remote server. Setting up a production site to use SSL involves purchasing and configuring an *SSL certificate* for your domain. That's a lot of work, though, and luckily we won't need it here: for an application running on a Heroku domain (such as the sample application), we can piggyback on Heroku's SSL certificate. As a result, when we deploy the application in Section 7.5.2, SSL will automatically be enabled. (If you want to run SSL on a custom domain, such as www.example.com, refer to Heroku's documentation on SSL.)

7.5.2 Production webserver

Having added SSL, we now need to configure our application to use a webserver suitable for production applications. By default, Heroku uses a pure-Ruby webserver called WEBrick, which is easy to set up and run but isn't good at handling significant traffic. As a result, WEBrick isn't suitable for production use, so we'll replace WEBrick with Puma, an HTTP server that is capable of handling a large number of incoming requests.

To add the new webserver, we simply follow the Heroku Puma documentation. The first step is to include the `puma` gem in our **`Gemfile`**, but as of Rails 5 Puma is included by default (Listing 3.2). This means we can skip right to the second step, which is to replace the default contents of the file **`config/puma.rb`** with the configuration shown in Listing 7.35. The code in Listing 7.35 comes straight from the Heroku documentation,[14] and there is no need to understand it (Box 1.2).

[14] Listing 7.35 changes the formatting slightly so that the code fits in the standard 80 columns.

Listing 7.35: The configuration file for the production webserver.
config/puma.rb

```
# Puma configuration file.
max_threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }
min_threads_count = ENV.fetch("RAILS_MIN_THREADS") { max_threads_count }
threads min_threads_count, max_threads_count
port        ENV.fetch("PORT") { 3000 }
environment ENV.fetch("RAILS_ENV") { ENV['RACK_ENV'] || "development" }
pidfile ENV.fetch("PIDFILE") { "tmp/pids/server.pid" }
workers ENV.fetch("WEB_CONCURRENCY") { 2 }
preload_app!
plugin :tmp_restart
```

We also need to make a so-called **Procfile** to tell Heroku to run a Puma process in production, as shown in Listing 7.36. The **Procfile** should be created in your application's root directory (i.e., in the same directory as the **Gemfile**).

Listing 7.36: Defining a **Procfile** for Puma.
./Procfile

```
web: bundle exec puma -C config/puma.rb
```

7.5.3 Production database configuration

The final step in our production deployment is properly configuring the production database, which (as mentioned briefly in Section 2.3.5) is PostgreSQL. My testing indicates that PostgreSQL actually works on Heroku without any configuration, but the official Heroku documentation recommends explicit configuration nonetheless, so we'll err on the side of caution and include it.

The actual change is easy: all we have to do is update the **production** section of the database configuration file, **config/database.yml**. The result, which I adapted from the Heroku docs, is shown in Listing 7.37.

Listing 7.37: Configuring the database for production.
config/database.yml

```
# SQLite version 3.x
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem 'sqlite3'
#
default: &default
  adapter: sqlite3
  pool: 5
  timeout: 5000

development:
  <<: *default
  database: db/development.sqlite3

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: db/test.sqlite3

production:
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see Rails configuration guide
  # https://guides.rubyonrails.org/configuring.html#database-pooling
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  database: sample_app_production
  username: sample_app
  password: <%= ENV['SAMPLE_APP_DATABASE_PASSWORD'] %>
```

7.5.4 Production deployment

With the production webserver and database configuration completed, we're ready to commit and deploy:[15]

[15] We haven't changed the data model in this chapter, so running the migration at Heroku shouldn't be necessary, but only if you followed the steps in Section 6.4. Because several readers reported having trouble, I've added **`heroku run rails db:migrate`** as a final step just to be safe.

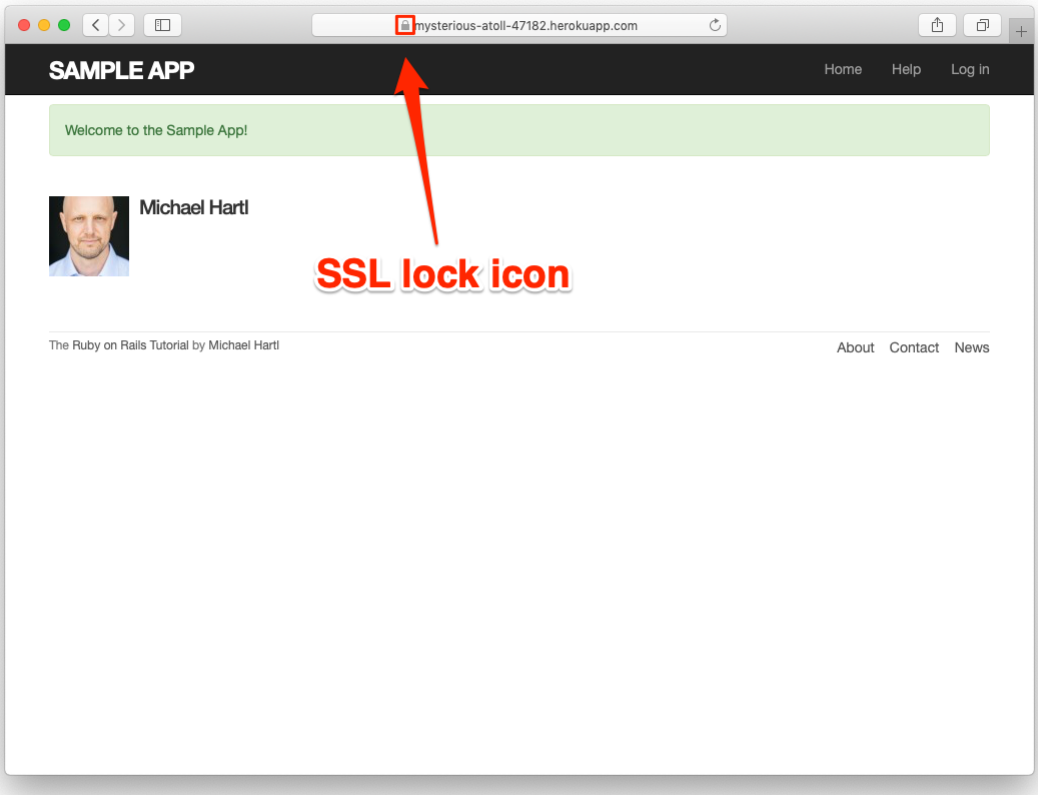


Figure 7.27: Signing up on the live Web.

```
$ rails test
$ git add -A
$ git commit -m "Use SSL and the Puma webserver in production"
$ git push && git push heroku
```

The signup form is now live, and the result of a successful signup is shown in Figure 7.27. Note the presence of a lock icon in the address bar of Figure 7.27, which indicate that SSL is working.

Ruby version number

When deploying to Heroku, you may get a warning message like this one:

```
###### WARNING:
       You have not declared a Ruby version in your Gemfile.
       To set your Ruby version add this line to your Gemfile:
       ruby '2.6.3'
```

Experience shows that, at the level of this tutorial, the costs associated with including such an explicit Ruby version number outweigh the (negligible) benefits, so you should ignore this warning for now. The main issue is that keeping your sample app and system in sync with the latest Ruby version can be a huge inconvenience,[16] and yet it almost never makes a difference which exact Ruby version number you use. Nevertheless, you should bear in mind that, should you ever end up running a mission-critical app on Heroku, specifying an exact Ruby version in the **`Gemfile`** is recommended to ensure maximum compatibility between development and production environments.

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. Confirm on your browser that the SSL lock and **`https`** appear.

2. Create a user on the production site using your primary email address. Does your Gravatar appear correctly?

[16] For example, at one point Heroku required Ruby 2.1.4, so I spent several hours trying unsuccessfully to install Ruby 2.1.4 on my local machine—only to discover that Ruby 2.1.5 had been released the previous day. (Attempts to install Ruby 2.1.5 then also failed.)

7.6 Conclusion

Being able to sign up users is a major milestone for our application. Although the sample app has yet to accomplish anything useful, we have laid an essential foundation for all future development. In Chapter 8 and Chapter 9, we will complete our authentication machinery by allowing users to log in and out of the application (with optional "remember me" functionality). In Chapter 10, we will allow all users to update their account information, and we will allow site administrators to delete users, thereby completing the full suite of Users resource REST actions from Table 7.1.

7.6.1 What we learned in this chapter

- Rails displays useful debug information via the `debug` method.
- Sass mixins allow a group of CSS rules to be bundled and reused in multiple places.
- Rails comes with three standard environments: `development`, `test`, and `production`.
- We can interact with users as a *resource* through a standard set of REST URLs.
- Gravatars provide a convenient way of displaying images to represent users.
- The `form_with` helper is used to generate forms for interacting with Active Record objects.
- Signup failure renders the new user page and displays error messages automatically determined by Active Record.
- Rails provides the `flash` as a standard way to display temporary messages.

- Signup success creates a user in the database and redirects to the user show page, and displays a welcome message.
- We can use integration tests to verify form submission behavior and catch regressions.
- We can configure our production application to use SSL for secure communications and Puma for high performance.

Chapter 8

Basic login

Now that new users can sign up for our site (Chapter 7), it's time to give them the ability to log in and log out. In this chapter, we'll implement a basic but still fully functional login system: the application will maintain the logged-in state until the browser is closed by the user. The resulting authentication system will allow us to customize the site and implement an authorization model based on login status and identity of the current user. For example, we'll be able to update the site header with login/logout links and a profile link.

In Chapter 10, we'll impose a security model in which only logged-in users can visit the user index page, only the correct user can access the page for editing their information, and only administrative users can delete other users from the database. Finally, in Chapter 13, we'll use the identity of a logged-in user to create microposts associated with that user, and in Chapter 14 we'll allow the current user to follow other users of the application (thereby receiving a feed of their microposts).

The authentication system from this chapter will also serve a foundation for the more advanced login system developed in Chapter 9. Instead of "forgetting" users on browser close, Chapter 9 will start by *automatically* remembering users, and will then *optionally* remember users based on the value of a "remember me" checkbox. As a result, taken together Chapter 8 and Chapter 9 develop all three of the most common types of login systems on the Web.

8.1 Sessions

HTTP is a *stateless protocol*, treating each request as an independent transaction that is unable to use information from any previous requests. This means there is no way within the Hypertext Transfer Protocol to remember a user's identity from page to page; instead, web applications requiring user login must use a *session*, which is a semi-permanent connection between two computers (such as a client computer running a web browser and a server running Rails).

The most common techniques for implementing sessions in Rails involve using *cookies*, which are small pieces of text placed on the user's browser. Because cookies persist from one page to the next, they can store information (such as a user id) that can be used by the application to retrieve the logged-in user from the database. In this section and in Section 8.2, we'll use the Rails method called `session` to make temporary sessions that expire automatically on browser close.[1] In Chapter 9, we'll learn how to make longer-lived sessions using the closely related `cookies` method.

It's convenient to model sessions as a RESTful resource: visiting the login page will render a form for *new* sessions, logging in will *create* a session, and logging out will *destroy* it. Unlike the Users resource, which uses a database back-end (via the User model) to persist data, the Sessions resource will use cookies, and much of the work involved in login comes from building this cookie-based authentication machinery. In this section and the next, we'll prepare for this work by constructing a Sessions controller, a login form, and the relevant controller actions. We'll then complete user login in Section 8.2 by adding the necessary session-manipulation code.

As in previous chapters, we'll do our work on a topic branch and merge in the changes at the end:

```
$ git checkout -b basic-login
```

[1] Some browsers offer an option to restore such sessions via a "continue where you left off" feature, but Rails has no control over this behavior. In such cases, the session cookie may persist even after logging out of the application.

8.1.1 Sessions controller

The elements of logging in and out correspond to particular REST actions of the Sessions controller: the login form is handled by the **new** action (covered in this section), actually logging in is handled by sending a POST request to the **create** action (Section 8.2), and logging out is handled by sending a DELETE request to the **destroy** action (Section 8.3). (Recall the association of HTTP verbs with REST actions from Table 7.1.)

To get started, we'll generate a Sessions controller with a **new** action (Listing 8.1).

Listing 8.1: Generating the Sessions controller.

```
$ rails generate controller Sessions new
```

(Including **new** actually generates *views* as well, which is why we don't include actions like **create** and **destroy** that don't correspond to views.) Following the model from Section 7.2 for the signup page, our plan is to create a login form for creating new sessions, as mocked up in Figure 8.1.

Unlike the Users resource, which used the special **resources** method to obtain a full suite of RESTful routes automatically (Listing 7.3), the Sessions resource will use only named routes, handling GET and POST requests with the **login** route and DELETE requests with the **logout** route. The result appears in Listing 8.2 (which also deletes the unneeded routes generated by **rails generate controller**).

Listing 8.2: Adding a resource to get the standard RESTful actions for sessions. RED

config/routes.rb

```
Rails.application.routes.draw do
  root   'static_pages#home'
  get    '/help',    to: 'static_pages#help'
  get    '/about',   to: 'static_pages#about'
  get    '/contact', to: 'static_pages#contact'
  get    '/signup',  to: 'users#new'
  get    '/login',   to: 'sessions#new'
```

Figure 8.1: A mockup of the login form.

HTTP request	URL	Named route	Action	Purpose
GET	/login	login_path	new	page for a new session (login)
POST	/login	login_path	create	create a new session (login)
DELETE	/logout	logout_path	destroy	delete a session (log out)

Table 8.1: Routes provided by the sessions rules in Listing 8.2.

```
  post   '/login',   to: 'sessions#create'
  delete '/logout',  to: 'sessions#destroy'
  resources :users
end
```

With the routes in Listing 8.2, we also need to update the test generated in Listing 8.1 with the new login route, as shown in Listing 8.3.

Listing 8.3: Updating the Sessions controller test to use the login route. GREEN
test/controllers/sessions_controller_test.rb

```
require 'test_helper'

class SessionsControllerTest < ActionDispatch::IntegrationTest

  test "should get new" do
    get login_path
    assert_response :success
  end
end
```

The routes defined in Listing 8.2 correspond to URLs and actions similar to those for users (Table 7.1), as shown in Table 8.1.

Since we've now added several custom named routes, it's useful to look at the complete list of the routes for our application, which we can generate using **`rails routes`**:

```
$ rails routes
   Prefix Verb   URI Pattern               Controller#Action
     root GET    /                         static_pages#home
     help GET    /help(.:format)           static_pages#help
    about GET    /about(.:format)          static_pages#about
```

```
  contact GET    /contact(.:format)         static_pages#contact
   signup GET    /signup(.:format)          users#new
    login GET    /login(.:format)           sessions#new
          POST   /login(.:format)           sessions#create
   logout DELETE /logout(.:format)          sessions#destroy
    users GET    /users(.:format)           users#index
          POST   /users(.:format)           users#create
 new_user GET    /users/new(.:format)       users#new
edit_user GET    /users/:id/edit(.:format)  users#edit
     user GET    /users/:id(.:format)       users#show
          PATCH  /users/:id(.:format)       users#update
          PUT    /users/:id(.:format)       users#update
          DELETE /users/:id(.:format)       users#destroy
```

It's not necessary to understand the results in detail, but viewing the routes in this manner gives us a high-level overview of the actions supported by our application.

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. What is the difference between **`GET login_path`** and **`POST login_path`**?

2. By piping the results of **`rails routes`** to **`grep`**, list all the routes associated with the Users resource. Do the same for Sessions. How many routes does each resource have? *Hint*: Refer to the section on grep in *Learn Enough Command Line to Be Dangerous*.

8.1.2 Login form

Having defined the relevant controller and route, now we'll fill in the view for new sessions, i.e., the login form. Comparing Figure 8.1 with Figure 7.12, we see that the login form is similar in appearance to the signup form, except with two fields (email and password) in place of four.

As seen in Figure 8.2, when the login information is invalid we want to re-render the login page and display an error message. In Section 7.3.3, we used an error-messages partial to display error messages, but we saw in that section that those messages are provided automatically by Active Record. This won't work for session creation errors because the session isn't an Active Record object, so we'll render the error as a flash message instead.

Recall from Listing 7.15 that the signup form uses the **form_with** helper, taking as an argument the user instance variable **@user**:

```
<%= form_with(model: @user, local: true) do |f| %>
  .
  .
  .
<% end %>
```

The main difference between the session form and the signup form is that we have no Session model, and hence no analogue for the **@user** variable. This means that, in constructing the new session form, we have to give **form_with** slightly different information; in particular, whereas

```
form_with(model: @user, local: true)
```

allows Rails to infer that the **action** of the form should be to `POST` to the URL /users, in the case of sessions we need to indicate the corresponding URL, along with the *scope* (in this case, the session):

```
form_with(url: login_path, scope: :session, local: true)
```

With the proper **form_with** in hand, it's easy to make a login form to match the mockup in Figure 8.1 using the signup form (Listing 7.15) as a model, as shown in Listing 8.4.

Figure 8.2: A mockup of login failure.

Listing 8.4: Code for the login form.
app/views/sessions/new.html.erb

```
<% provide(:title, "Log in") %>
<h1>Log in</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(url: login_path, scope: :session, local: true) do |f| %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.submit "Log in", class: "btn btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

Note that we've added a link to the signup page for convenience. With the code in Listing 8.4, the login form appears as in Figure 8.3. (Because the "Log in" navigation link hasn't yet been filled in, you'll have to type the /login URL directly into your address bar. We'll fix this blemish in Section 8.2.3.)

The generated form HTML appears in Listing 8.5.

Listing 8.5: HTML for the login form produced by Listing 8.4.

```
<form accept-charset="UTF-8" action="/login" method="post">
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />
  <label for="session_email">Email</label>
  <input class="form-control" id="session_email"
         name="session[email]" type="email" />
  <label for="session_password">Password</label>
  <input id="session_password" name="session[password]"
         type="password" />
  <input class="btn btn-primary" name="commit" type="submit"
       value="Log in" />
</form>
```

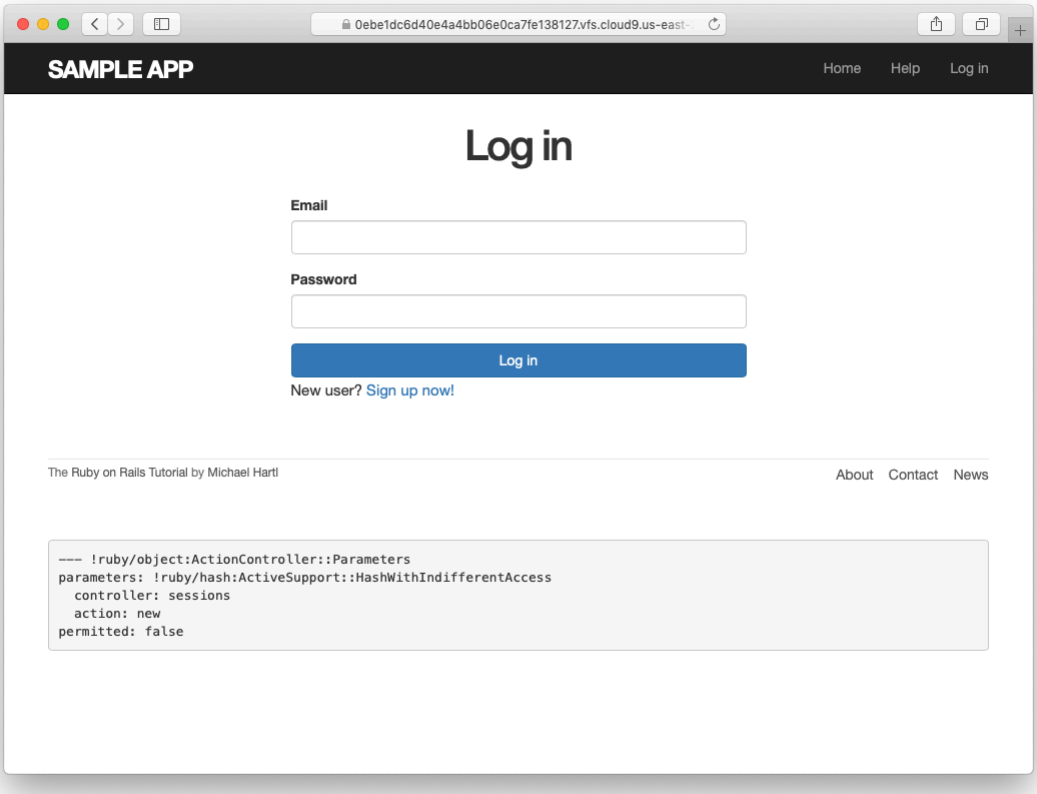


Figure 8.3: The login form.

Comparing Listing 8.5 with Listing 7.17, you might be able to guess that submitting this form will result in a `params` hash where `params[:session][:email]` and `params[:session][:password]` correspond to the email and password fields.

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. Submissions from the form defined in Listing 8.4 will be routed to the Session controller's `create` action. How does Rails know to do this? *Hint*: Refer to Table 8.1 and the first line of Listing 8.5.

8.1.3 Finding and authenticating a user

As in the case of creating users (signup), the first step in creating sessions (login) is to handle *invalid* input. We'll start by reviewing what happens when a form gets submitted, and then arrange for helpful error messages to appear in the case of login failure (as mocked up in Figure 8.2.) Then we'll lay the foundation for successful login (Section 8.2) by evaluating each login submission based on the validity of its email/password combination.

Let's start by defining a minimalist `create` action for the Sessions controller, along with empty `new` and `destroy` actions (Listing 8.6). The `create` action in Listing 8.6 does nothing but render the `new` view, but it's enough to get us started. Submitting the /sessions/new form then yields the result shown in Figure 8.4 and Figure 8.5.

Listing 8.6: A preliminary version of the Sessions `create` action.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
```

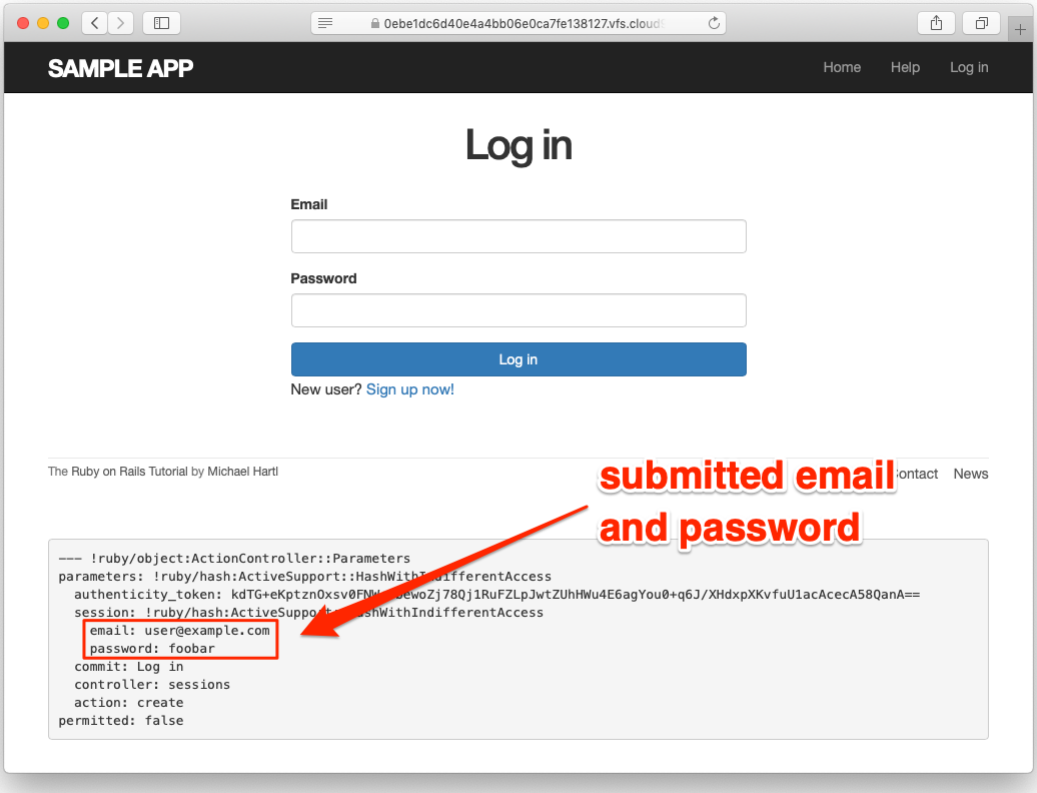


Figure 8.4: The initial failed login, with `create` as in Listing 8.6.

```
  end

  def create
    render 'new'
  end

  def destroy
  end
end
```

Carefully inspecting the debug information in Figure 8.5 shows that, as hinted at the end of Section 8.1.2, the submission results in a `params` hash containing the email and password under the key `session`, which (omitting

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  authenticity_token: QBa8IYb8XKOlp164MhLzDzxjSnPX6aZww4LYeEFyUeFJHz16YWWJpzIA+
CgjpEtmq2GPOcr5un/nfuHa8I3YwQ==
  session: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
    email: user@example.com
    password: foobar
  commit: Log in
  controller: sessions
  action: create
permitted: false
```

Figure 8.5: A closer look at the debug information from Figure 8.4.

some irrelevant details used internally by Rails) appears as follows:

```
---
session:
  email: 'user@example.com'
  password: 'foobar'
commit: Log in
action: create
controller: sessions
```

As with the case of user signup (Figure 7.16), these parameters form a *nested* hash like the one we saw in Listing 4.13. In particular, **`params`** contains a nested hash of the form

```
{ session: { password: "foobar", email: "user@example.com" } }
```

This means that

```
params[:session]
```

is itself a hash:

```
{ password: "foobar", email: "user@example.com" }
```

As a result,

```
params[:session][:email]
```

is the submitted email address and

```
params[:session][:password]
```

is the submitted password.

In other words, inside the **create** action the **params** hash has all the information needed to authenticate users by email and password. Not coincidentally, we already have exactly the methods we need: the **User.find_by** method provided by Active Record (Section 6.1.4) and the **authenticate** method provided by **has_secure_password** (Section 6.3.4). Recalling that **authenticate** returns **false** for an invalid authentication (Section 6.3.4), our strategy for user login can be summarized as shown in Listing 8.7.

Listing 8.7: Finding and authenticating a user.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Log the user in and redirect to the user's show page.
    else
      # Create an error message.
      render 'new'
    end
  end

  def destroy
  end
end
```

User	Password	a && b
nonexistent	*anything*	`(nil && [anything]) == false`
valid user	wrong password	`(true && false) == false`
valid user	right password	`(true && true) == true`

Table 8.2: Possible results of `user && user.authenticate(…)`.

The first highlighted line in Listing 8.7 pulls the user out of the database using the submitted email address. (Recall from Section 6.2.5 that email addresses are saved as all lower-case, so here we use the `downcase` method to ensure a match when the submitted address is valid.) The next line can be a bit confusing but is fairly common in idiomatic Rails programming:

```
user && user.authenticate(params[:session][:password])
```

This uses `&&` (logical *and*) to determine if the resulting user is valid. Taking into account that any object other than `nil` and `false` itself is `true` in a boolean context (Section 4.2.2), the possibilities appear as in Table 8.2. We see from Table 8.2 that the `if` statement is `true` only if a user with the given email both exists in the database and has the given password, exactly as required.

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 Rails console, confirm each of the values in Table 8.2. Start with `user = nil`, and then use `user = User.first`. *Hint*: To coerce the result to a boolean value, use the bang-bang trick from Section 4.2.2, as in `!!(user && user.authenticate('foobar'))`.

8.1.4 Rendering with a flash message

Recall from Section 7.3.3 that we displayed signup errors using the User model error messages. These errors are associated with a particular Active Record object, but this strategy won't work here because the session isn't an Active Record model. Instead, we'll put a message in the flash to be displayed upon failed login. A first, slightly incorrect, attempt appears in Listing 8.8.

Listing 8.8: An (unsuccessful) attempt at handling failed login.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Log the user in and redirect to the user's show page.
    else
      flash[:danger] = 'Invalid email/password combination' # Not quite right!
      render 'new'
    end
  end

  def destroy
  end
end
```

Because of the flash message display in the site layout (Listing 7.29), the **`flash[:danger]`** message automatically gets displayed; because of the Bootstrap CSS, it automatically gets nice styling (Figure 8.6).

Unfortunately, as noted in the text and in the comment in Listing 8.8, this code isn't quite right. The page looks fine, though, so what's the problem? The issue is that the contents of the flash persist for one *request*, but—unlike a redirect, which we used in Listing 7.27—re-rendering a template with **`render`** doesn't count as a request. The result is that the flash message persists one request longer than we want. For example, if we submit invalid login information and then click on the Home page, the flash gets displayed a second time (Figure 8.7). Fixing this blemish is the task of Section 8.1.5.

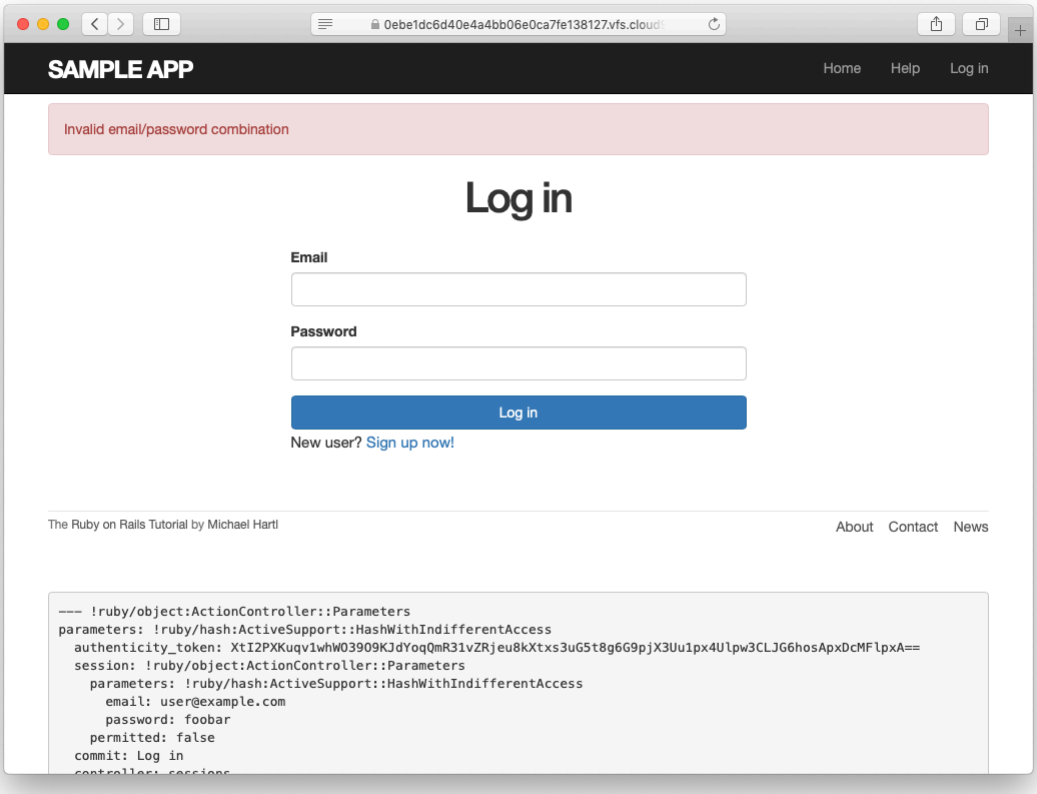


Figure 8.6: The flash message for a failed login.

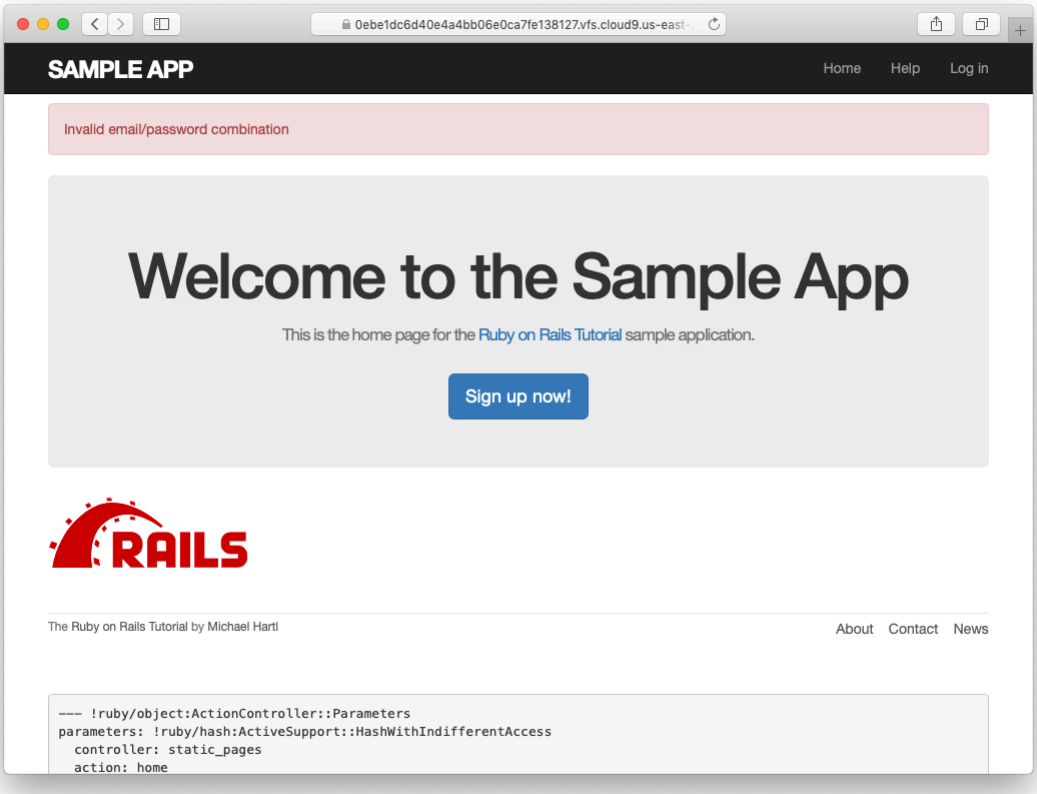


Figure 8.7: An example of flash persistence.

8.1.5 A flash test

The incorrect flash behavior is a minor bug in our application. According to the testing guidelines from Box 3.3, this is exactly the sort of situation where we should write a test to catch the error so that it doesn't recur. We'll thus write a short integration test for the login form submission before proceeding. In addition to documenting the bug and preventing a regression, this will also give us a good foundation for further integration tests of login and logout.

We start by generating an integration test for our application's login behavior:

```
$ rails generate integration_test users_login
      invoke  test_unit
      create    test/integration/users_login_test.rb
```

Next, we need a test to capture the sequence shown in Figure 8.6 and Figure 8.7. The basic steps appear as follows:

1. Visit the login path.
2. Verify that the new sessions form renders properly.
3. Post to the sessions path with an invalid `params` hash.
4. Verify that the new sessions form gets re-rendered and that a flash message appears.
5. Visit another page (such as the Home page).
6. Verify that the flash message *doesn't* appear on the new page.

A test implementing the above steps appears in Listing 8.9.

Listing 8.9: A test to catch unwanted flash persistence. RED
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  test "login with invalid information" do
    get login_path
    assert_template 'sessions/new'
    post login_path, params: { session: { email: "", password: "" } }
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end
end
```

After adding the test in Listing 8.9, the login test should be RED:

Listing 8.10: RED

```
$ rails test test/integration/users_login_test.rb
```

This shows how to run one (and only one) test file using **`rails test`** and the full path to the file.

The way to get the failing test in Listing 8.9 to pass is to replace **`flash`** with the special variant **`flash.now`**, which is specifically designed for displaying flash messages on rendered pages. Unlike the contents of **`flash`**, the contents of **`flash.now`** disappear as soon as there is an additional request, which is exactly the behavior we've tested in Listing 8.9. With this substitution, the corrected application code appears as in Listing 8.11.

Listing 8.11: Correct code for failed login. GREEN
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
```

```
      # Log the user in and redirect to the user's show page.
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

We can then verify that both the login integration test and the full test suite are GREEN:

Listing 8.12: GREEN

```
$ rails test test/integration/users_login_test.rb
$ 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 in your browser that the sequence from Section 8.1.4 works correctly, i.e., that the flash message disappears when you click on a second page.

8.2 Logging in

Now that our login form can handle invalid submissions, the next step is to handle valid submissions correctly by actually logging a user in. In this section, we'll log the user in with a temporary session cookie that expires automatically upon browser close. In Section 9.1, we'll add sessions that persist even after closing the browser.

Implementing sessions will involve defining a large number of related functions for use across multiple controllers and views. You may recall from Section 4.2.4 that Ruby provides a *module* facility for packaging such functions in one place. Conveniently, a Sessions helper module was generated automatically when generating the Sessions controller (Section 8.1.1). Moreover, such helpers are automatically included in Rails views; by including the module into the base class of all controllers (the Application controller), we arrange to make them available in our controllers as well (Listing 8.13).[2]

Listing 8.13: Including the Sessions helper module into the Application controller.
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  include SessionsHelper
end
```

With this configuration complete, we're now ready to write the code to log users in.

8.2.1 The `log_in` method

Logging a user in is simple with the help of the **`session`** method defined by Rails. (This method is separate and distinct from the Sessions controller generated in Section 8.1.1.) We can treat **`session`** as if it were a hash, and assign to it as follows:

```
session[:user_id] = user.id
```

This places a temporary cookie on the user's browser containing an encrypted version of the user's id, which allows us to retrieve the id on subsequent pages

[2] I like this technique because it connects to the pure Ruby way of including modules, but Rails 4 introduced a technique called *concerns* that can also be used for this purpose. To learn how to use concerns, run a search for "how to use concerns in Rails".

using `session[:user_id]`. In contrast to the persistent cookie created by the `cookies` method (Section 9.1), the temporary cookie created by the `session` method expires immediately when the browser is closed.

Because we'll want to use the same login technique in a couple of different places, we'll define a method called `log_in` in the Sessions helper, as shown in Listing 8.14.

Listing 8.14: The `log_in` function.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
end
```

Because temporary cookies created using the `session` method are automatically encrypted, the code in Listing 8.14 is secure, and there is no way for an attacker to use the session information to log in as the user. This applies only to temporary sessions initiated with the `session` method, though, and is *not* the case for persistent sessions created using the `cookies` method. Permanent cookies are vulnerable to a *session hijacking* attack, so in Chapter 9 we'll have to be much more careful about the information we place on the user's browser.

With the `log_in` method defined in Listing 8.14, we're now ready to complete the session `create` action by logging the user in and redirecting to the user's profile page. The result appears in Listing 8.15.[3]

Listing 8.15: Logging in a user.
app/controllers/sessions_controller.rb

```
1 class SessionsController < ApplicationController
2
3   def new
4   end
```

[3]The `log_in` method is available in the Sessions controller because of the module inclusion in Listing 8.13.

```
5
6     def create
7       user = User.find_by(email: params[:session][:email].downcase)
8       if user && user.authenticate(params[:session][:password])
9         log_in user
10        redirect_to user
11      else
12        flash.now[:danger] = 'Invalid email/password combination'
13        render 'new'
14      end
15    end
16
17    def destroy
18    end
19  end
```

Note the compact redirect

```
redirect_to user
```

which we saw before in Section 7.4.1. Rails automatically converts this to the route for the user's profile page:

```
user_url(user)
```

With the **`create`** action defined in Listing 8.15, the login form defined in Listing 8.4 should now be working. It doesn't have any effects on the application display, though, so short of inspecting the browser session directly there's no way to tell that you're logged in. As a first step toward enabling more visible changes, in Section 8.2.2 we'll retrieve the current user from the database using the id in the session. In Section 8.2.3, we'll change the links on the application layout, including a URL to the current user's profile.

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. Log in with a valid user and inspect your browser's cookies. What is the value of the session content? *Hint*: If you don't know how to view your browser's cookies, Google for it (Box 1.2).

2. What is the value of the **Expires** attribute from the previous exercise?

8.2.2 Current user

Having placed the user's id securely in the temporary session, we are now in a position to retrieve it on subsequent pages, which we'll do by defining a **current_user** method to find the user in the database corresponding to the session id. The purpose of **current_user** is to allow constructions such as

```
<%= current_user.name %>
```

and

```
redirect_to current_user
```

To find the current user, one possibility is to use the **find** method, as on the user profile page (Listing 7.5):

```
User.find(session[:user_id])
```

But recall from Section 6.1.4 that **find** raises an exception if the user id doesn't exist. This behavior is appropriate on the user profile page because it will only happen if the id is invalid, but in the present case **session[:user_id]** will often be **nil** (i.e., for non-logged-in users). To handle this possibility, we'll use the same **find_by** method used to find by email address in the **create** method, with **id** in place of **email**:

```
User.find_by(id: session[:user_id])
```

Rather than raising an exception, this method returns **nil** (indicating no such user) if the id is invalid.

We could now define the **current_user** method as follows:

```
def current_user
  if session[:user_id]
    User.find_by(id: session[:user_id])
  end
end
```

(If the session user id doesn't exist, the function just falls off the end and returns **nil** automatically, which is exactly what we want.) This would work fine, but it would hit the database multiple times if, e.g., **current_user** appeared multiple times on a page. Instead, we'll follow a common Ruby convention by storing the result of **User.find_by** in an instance variable, which hits the database the first time but returns the instance variable immediately on subsequent invocations:[4]

```
if @current_user.nil?
  @current_user = User.find_by(id: session[:user_id])
else
  @current_user
end
```

Recalling the *or* operator || seen in Section 4.2.2, we can rewrite this as follows:

```
@current_user = @current_user || User.find_by(id: session[:user_id])
```

[4]This practice of remembering variable assignments from one method invocation to the next is known as *memoization*. (Note that this is a technical term; in particular, it's *not* a misspelling of "memorization"—a subtlety lost on the hapless copyeditor of a previous edition of this book.)

Because a User object is true in a boolean context, the call to **find_by** only gets executed if **@current_user** hasn't yet been assigned.

Although the preceding code would work, it's not idiomatically correct Ruby; instead, the proper way to write the assignment to **@current_user** is like this:

```
@current_user ||= User.find_by(id: session[:user_id])
```

This uses the potentially confusing but frequently used **||=** ("or equals") operator (Box 8.1).

Box 8.1. What the *$@! is ||= ?

The ||= ("or equals") assignment operator is a common Ruby idiom and is thus important for aspiring Rails developers to recognize. Although at first it may seem mysterious, *or equals* is easy to understand by analogy.

We start by noting the common pattern of incrementing a variable:

```
x = x + 1
```

Many languages provide a syntactic shortcut for this operation; in Ruby (and in C, C++, Perl, Python, Java, etc.), it can also appear as follows:

```
x += 1
```

Analogous constructs exist for other operators as well:

```
$ rails console
>> x = 1
=> 1
>> x += 1
=> 2
>> x *= 3
```

```
=> 6
>> x -= 8
=> -2
>> x /= 2
=> -1
```

In each case, the pattern is that `x = x O y` and `x O= y` are equivalent for any operator `O`.

Another common Ruby pattern is assigning to a variable if it's `nil` but otherwise leaving it alone. Recalling the *or* operator `||` seen in Section 4.2.2, we can write this as follows:

```
>> @foo
=> nil
>> @foo = @foo || "bar"
=> "bar"
>> @foo = @foo || "baz"
=> "bar"
```

Since `nil` is false in a boolean context, the first assignment to `@foo` is `nil || "bar"`, which evaluates to `"bar"`. Similarly, the second assignment is `@foo || "baz"`, i.e., `"bar" || "baz"`, which also evaluates to `"bar"`. This is because anything other than `nil` or `false` is `true` in a boolean context, and the series of `||` expressions terminates after the first true expression is evaluated. (This practice of evaluating `||` expressions from left to right and stopping on the first true value is known as *short-circuit evaluation*. The same principle applies to `&&` statements, except in this case evaluation stops on the first *false* value.)

Comparing the console sessions for the various operators, we see that `@foo = @foo || "bar"` follows the `x = x O y` pattern with `||` in the place of `O`:

```
x    =   x   +   1      ->     x     +=   1
x    =   x   *   3      ->     x     *=   3
x    =   x   -   8      ->     x     -=   8
```

```
x     =    x    /    2        ->      x      /=    2
@foo = @foo || "bar"     ->      @foo ||= "bar"
```

Thus we see that `@foo = @foo || "bar"` and `@foo ||= "bar"` are equivalent. In the context of the current user, this suggests the following construction:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Voilà !

(Technically, Ruby evaluates the expression `@foo || @foo = "bar"`, which avoids an unnecessary assignment when `@foo` is not `nil` or `false`. But this expression doesn't explain the `||=` notation as well, so the above discussion uses the nearly equivalent `@foo = @foo || "bar"`.)

Applying the results of the above discussion yields the succinct **`current_user`** method shown in Listing 8.16. (There's a slight amount of repetition in the use of **`session[:user_id]`**, which we'll eliminate in Section 9.1.2.)

Listing 8.16: Finding the current user in the session.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end
end
```

With the working `current_user` method in Listing 8.16, we're now in a position to make changes to our application based on user login status.

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. Confirm at the console that `User.find_by(id: ...)` returns `nil` when the corresponding user doesn't exist.
2. In a Rails console, create a `session` hash with key `:user_id`. By following the steps in Listing 8.17, confirm that the `||=` operator works as required.

Listing 8.17: Simulating `session` in the console.

```
>> session = {}
>> session[:user_id] = nil
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
>> session[:user_id]= User.first.id
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
>> @current_user ||= User.find_by(id: session[:user_id])
<What happens here?>
```

8.2.3 Changing the layout links

The first practical application of logging in involves changing the layout links based on login status. In particular, as seen in the Figure 8.8 mockup,[5] we'll add links for logging out, for user settings, for listing all users, and for the current user's profile page. Note in Figure 8.8 that the logout and profile links appear in a dropdown "Account" menu; we'll see in Listing 8.19 how to make such a menu with Bootstrap.

[5] Image retrieved from https://www.flickr.com/photos/elevy/14730820387 on 2016-06-03. Copyright © 2014 by Elias Levy and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

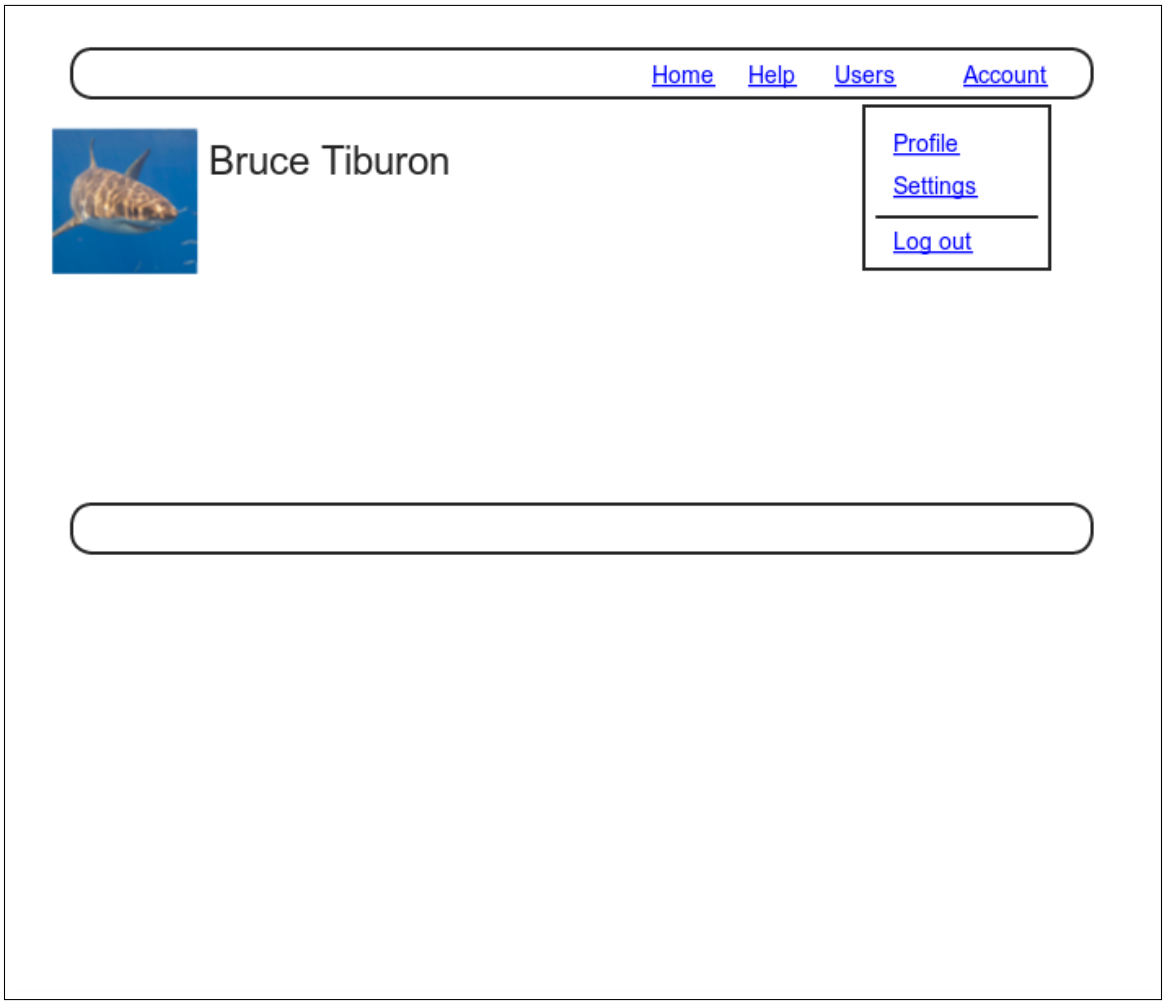


Figure 8.8: A mockup of the user profile after a successful login.

At this point, in real life I would consider writing an integration test to capture the behavior described above. As noted in Box 3.3, as you become more familiar with the testing tools in Rails you may find yourself more inclined to write tests first. In this case, though, such a test involves several new ideas, so for now it's best deferred to its own section (Section 8.2.4).

The way to change the links in the site layout involves using an if-else statement inside embedded Ruby to show one set of links if the user is logged in and another set of links otherwise:

```
<% if logged_in? %>
  # Links for logged-in users
<% else %>
  # Links for non-logged-in-users
<% end %>
```

This kind of code requires the existence of a **`logged_in?`** boolean method, which we'll now define.

A user is logged in if there is a current user in the session, i.e., if **`current_user`** is not **`nil`**. Checking for this requires the use of the "not" operator (Section 4.2.2), written using an exclamation point **`!`** and usually read as "bang". The resulting **`logged_in?`** method appears in Listing 8.18.

Listing 8.18: The **`logged_in?`** helper method.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end

  # Returns true if the user is logged in, false otherwise.
```

```
  def logged_in?
    !current_user.nil?
  end
end
```

With the addition in Listing 8.18, we're now ready to change the layout links if a user is logged in. There are four new links, two of which are stubbed out (to be completed in Chapter 10):

```
<%= link_to "Users",    '#' %>
<%= link_to "Settings", '#' %>
```

The logout link, meanwhile, uses the logout path defined in Listing 8.2:

```
<%= link_to "Log out", logout_path, method: :delete %>
```

Notice that the logout link passes a hash argument indicating that it should submit with an HTTP `DELETE` request.[6] We'll also add a profile link as follows:

```
<%= link_to "Profile", current_user %>
```

Here we could write

```
<%= link_to "Profile", user_path(current_user) %>
```

but as usual Rails allows us to link directly to the user by automatically converting **`current_user`** into **`user_path(current_user)`** in this context. Finally, when users *aren't* logged in, we'll use the login path defined in Listing 8.2 to make a link to the login form:

[6] Web browsers can't actually issue `DELETE` requests; Rails fakes it with JavaScript.

```
<%= link_to "Log in", login_path %>
```

Putting everything together gives the updated header partial shown in Listing 8.19.

Listing 8.19: Changing the layout links for logged-in users.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Account <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li><%= link_to "Profile", current_user %></li>
              <li><%= link_to "Settings", '#' %></li>
              <li class="divider"></li>
              <li>
                <%= link_to "Log out", logout_path, method: :delete %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

As part of including the new links into the layout, Listing 8.19 takes advantage of Bootstrap's ability to make dropdown menus.[7] Note in particular the inclusion of the special Bootstrap CSS classes such as **dropdown**, **dropdown-menu**, etc. To activate the dropdown menu, we need to include Bootstrap's custom JavaScript library into our application (which is *not* included

[7] See the Bootstrap components page for more information.

automatically as part of the `bootstrap-sass` gem in Listing 5.5), as well as the jQuery library.

Section 5.2 mentioned briefly that the Rails asset pipeline works in parallel with Webpack and Yarn, and we need to put both to work in order to include the above JavaScript. The first step is to install both jQuery and Bootstrap's JavaScript library in our application, which coincidentally needs the same version number for each:

```
$ yarn add jquery@3.4.1 bootstrap@3.4.1
```

In order to make jQuery available in our application, we need to edit Webpack's environment file and add the content shown in Listing 8.20.

Listing 8.20: Adding jQuery configuration to Webpack.
config/webpack/environment.js

```
const { environment } = require('@rails/webpacker')

const webpack = require('webpack')
environment.plugins.prepend('Provide',
  new webpack.ProvidePlugin({
    $: 'jquery/src/jquery',
    jQuery: 'jquery/src/jquery'
  })
)

module.exports = environment
```

Finally, we need to require jQuery and import Bootstrap in our **`application.js`** file, as shown in Listing 8.21.[8]

Listing 8.21: Requiring and importing the necessary JavaScript libraries.
app/javascript/packs/application.js

```
require("@rails/ujs").start()
require("turbolinks").start()
```

[8]For what it's worth, I don't know offhand why one uses **`require`** and the other uses **`import`**.

```
require("@rails/activestorage").start()
require("channels")
require("jquery")
import "bootstrap"
```

At this point, you should visit the login path and log in as a valid user (username **example@railstutorial.org**, password **foobar**), which effectively tests the code in the previous three sections.[9] With the code in Listing 8.19 and Listing 8.21, you should see the dropdown menu and links for logged-in users, as shown in Figure 8.9.

If you quit your browser completely, you should also be able to verify that the application forgets your login status, requiring you to log in again to see the changes described above.[10]

Mobile styling

Now that we've got the dropdown menu working, we're going to take an opportunity to add a few design tweaks for mobile devices, thereby fulfilling a promise made in Section 5.1.[11] As noted in that section, this is not a tutorial on web design, so we'll be making the minimum set of changes needed to make the mobile app look nice, but you can find a much more detailed treatment of mobile styling in *Learn Enough CSS & Layout to Be Dangerous*, especially Chapter 9, "Mobile media queries".

Our first step in applying some mobile-friendly design is to view our current app as it appears in a mobile device. One possibility is simply using a smartphone to view the site, but this can be inconvenient, especially if the development app is running behind a cloud IDE login wall or on a local network (which is often inaccessible to outside devices). A more convenient alternative is to use a feature of the Safari web browser known as "Responsive Design Mode", which can be activated as shown in Figure 8.10.[12] This mode gives us

[9] You may have to restart the webserver to get this to work (Box 1.2).

[10] If you're using the cloud IDE, I recommend using a different browser to test the login behavior so that you don't have to close down the browser running the IDE.

[11] Several of these changes were based on or inspired by the excellent work of *Rails Tutorial* reader Craig Zeise.

[12] See "Mobile viewport" in *Learn Enough CSS & Layout to Be Dangerous* for yet another alternative.

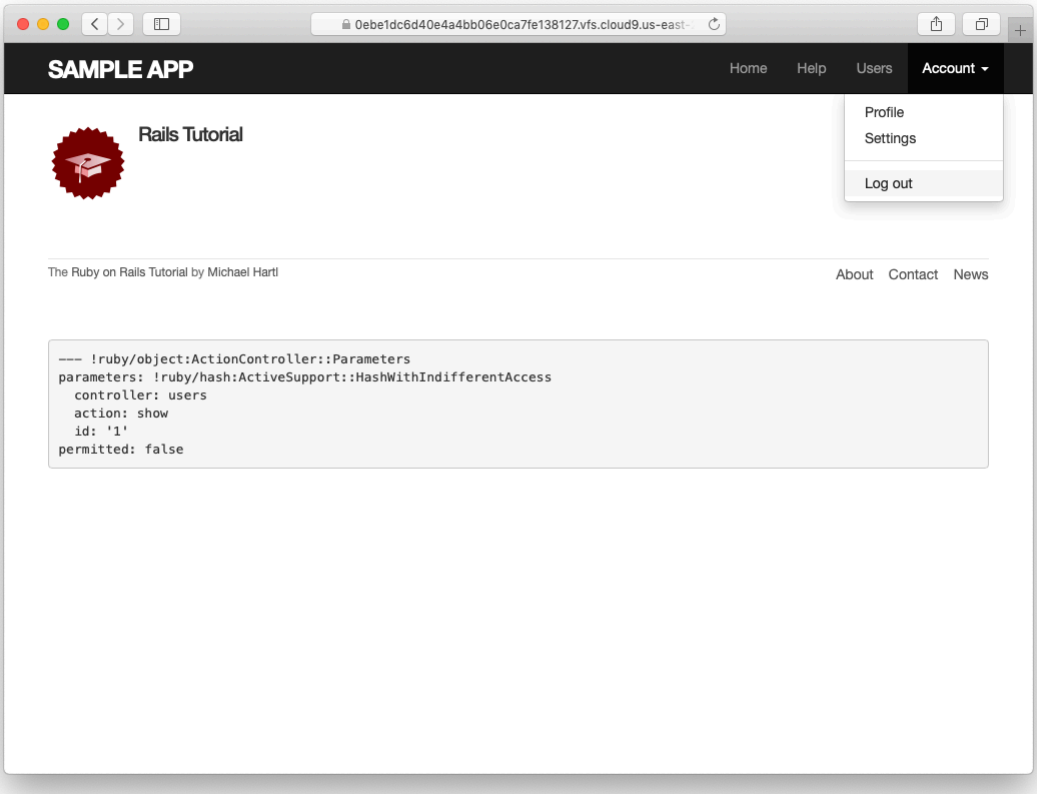


Figure 8.9: A logged-in user with new links and a dropdown menu.

the ability to view the app as it appears in any of a number of different mobile devices, as seen in Figure 8.11.[13]

Now let's take a look at the profile page from Figure 8.9 as it appears in a mobile device. As seen in Figure 8.12, the menu items currently aren't nicely aligned, and the menu itself takes over a large part of the top of the screen. Meanwhile, the footer navigation links appear in an awkward location, squished up against the other link in the footer. These are the only major issues we need to address, and we'll be able to fix both of them with a surprisingly small amount of code.

Our first step is to add a special **`meta`** tag called the *viewport*, which lets developers switch between desktop and mobile modes.[14] The result, which goes in the application template, appears in Listing 8.22.

Listing 8.22: Adding the viewport **`meta`** tag.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    .
    .
    .
```

To fix the menu, we'll use a so-called "hamburger menu" (named for its unintentional resemblance to a hamburger) when in mobile mode. This involves adding some markup to the header file from Listing 8.19, as shown in Listing 8.23.[15]

[13] To change the aspect ratio from portrait to landscape for any of the devices, simply double-click on the corresponding icon. Back when we were making *Learn Enough CSS & Layout to Be Dangerous*, it took me and my coauthor Lee Donahoe a surprisingly long time to figure this out.

[14] See "Mobile viewport" in *Learn Enough CSS & Layout to Be Dangerous* for more details.

[15] I might never have figured this out on my own. Thanks again to Craig Zeise for his fantastic volunteer effort.

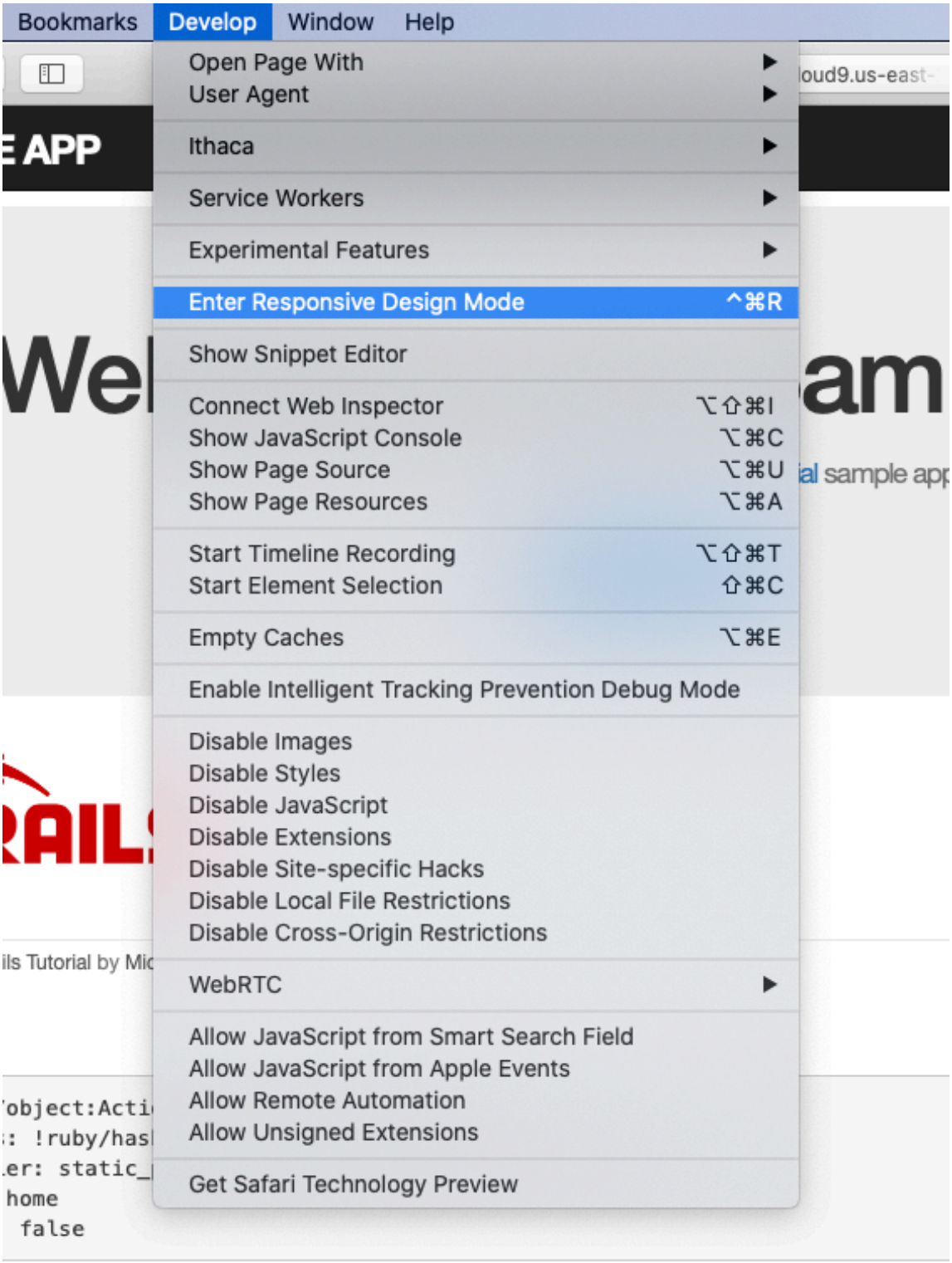


Figure 8.10: The menu item to enter Safari's Responsive Design Mode.

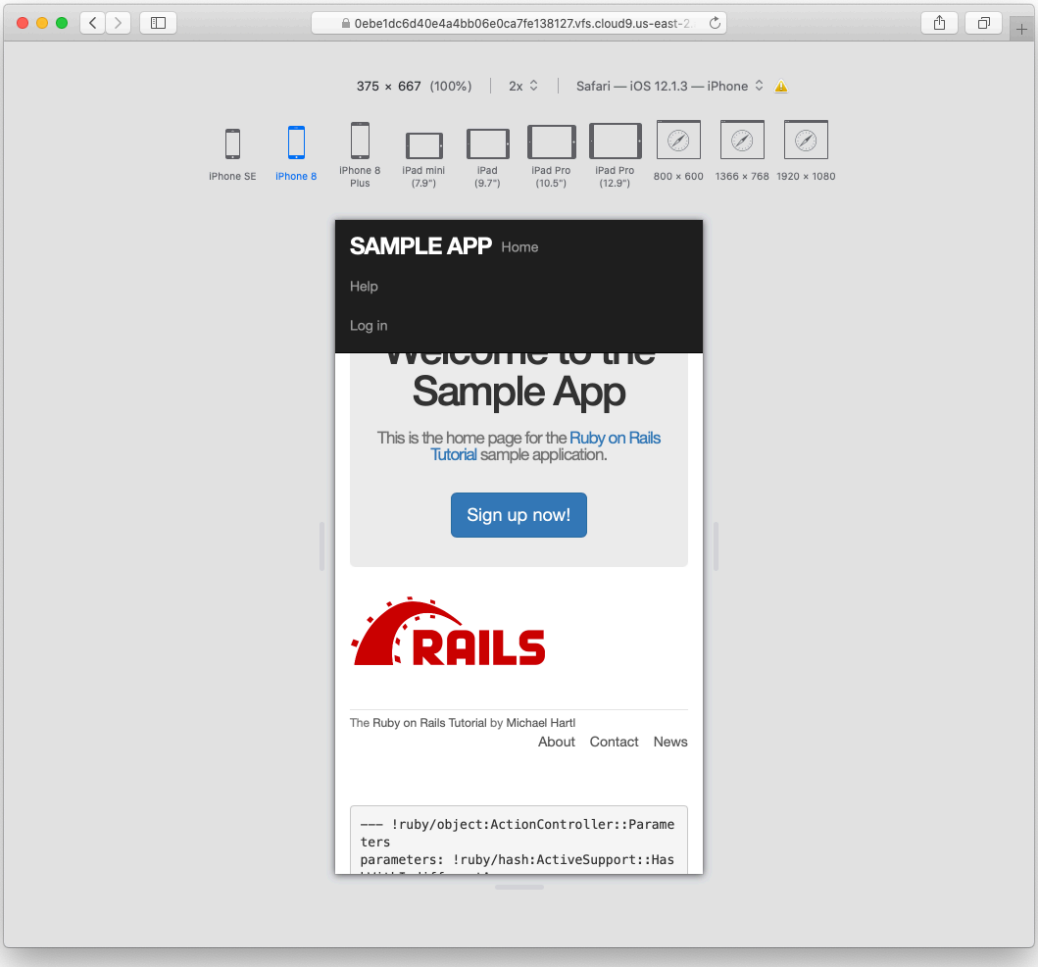
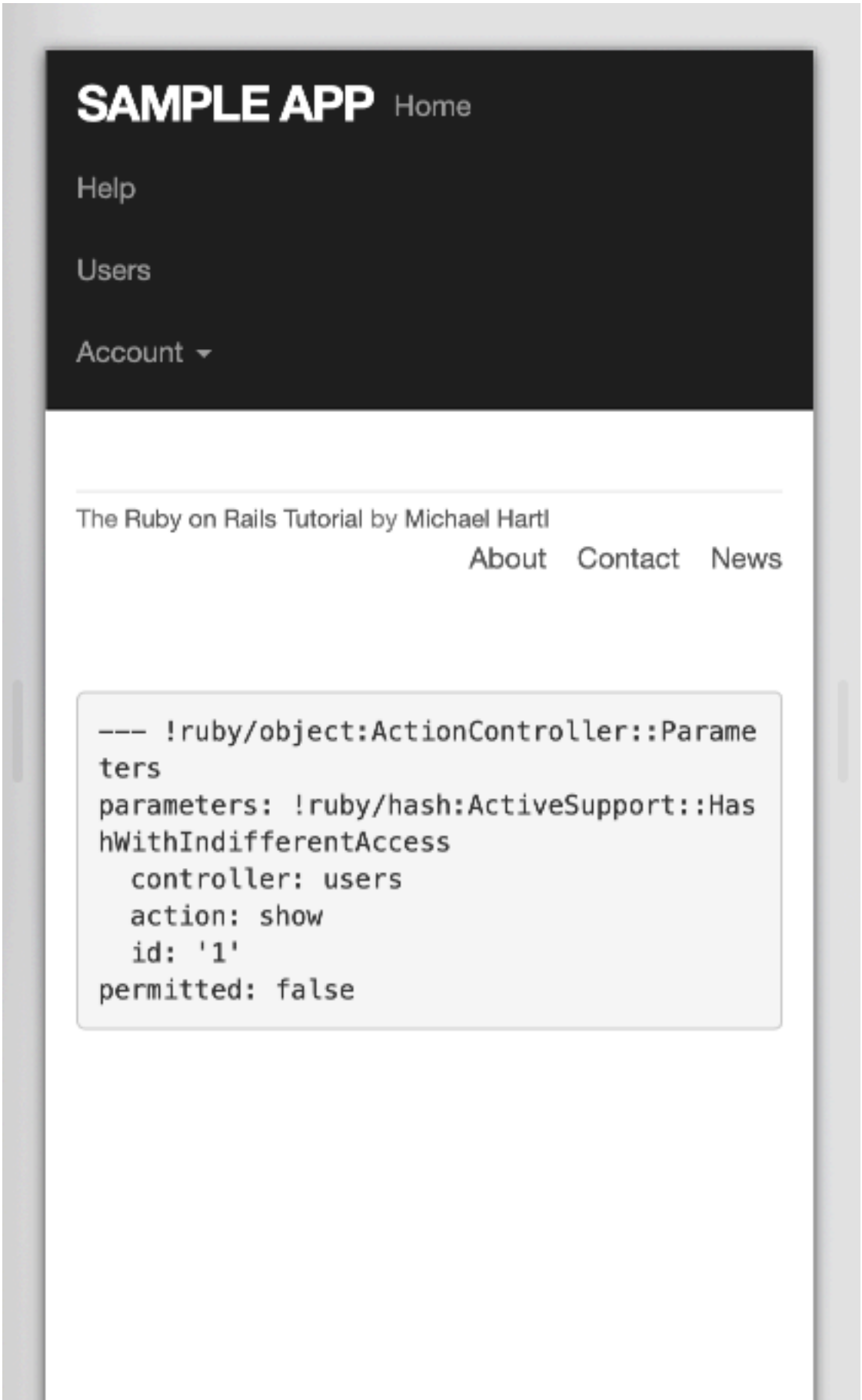


Figure 8.11: Safari's Responsive Design Mode.


SAMPLE APP
Home
Help
Users
Account
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: users
action: show
id: '1'
permitted: false

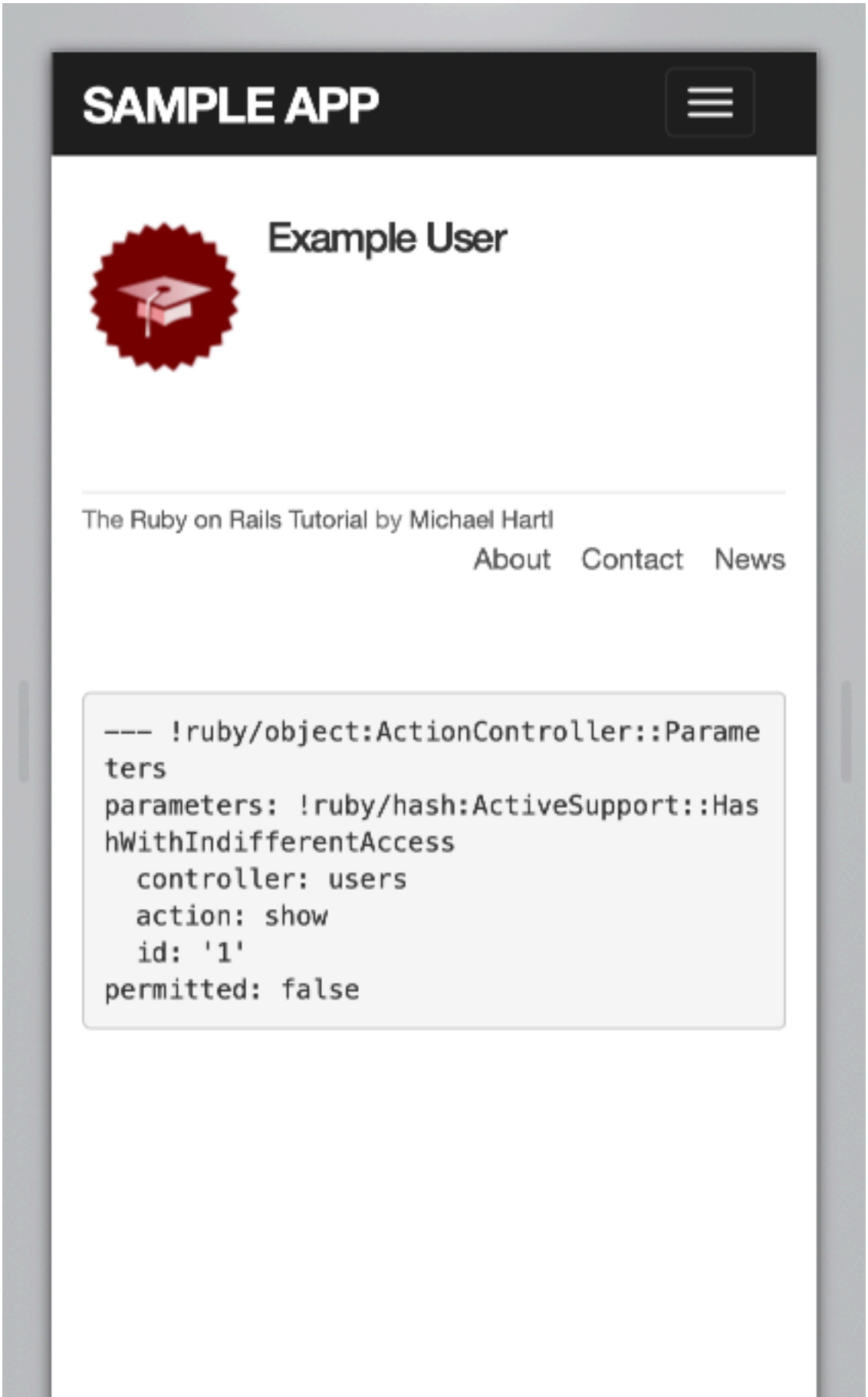
Listing 8.23: Adding the markup for a hamburger menu.
app/views/layouts/_header.html.erb

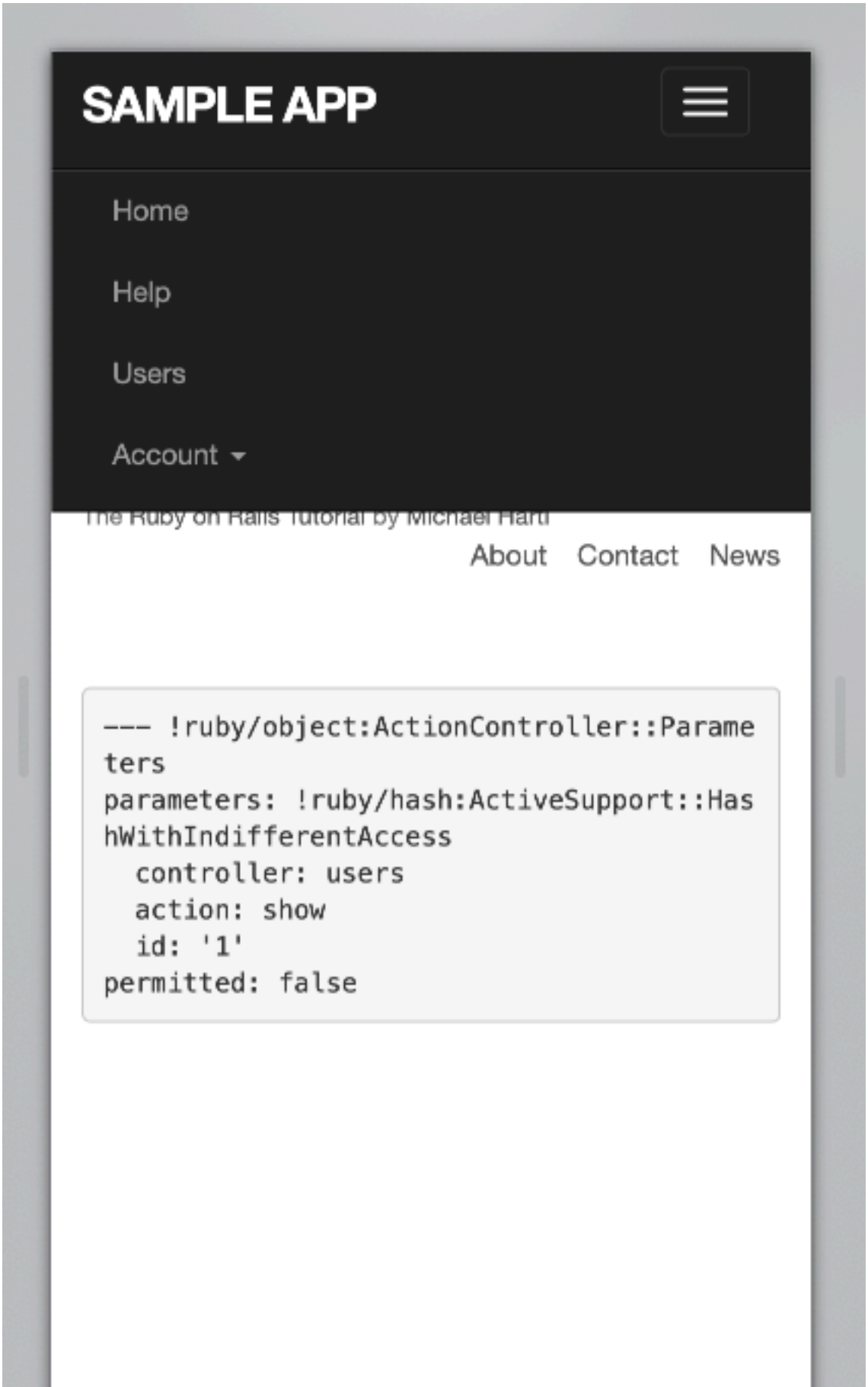
```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed"
                data-toggle="collapse"
                data-target="#bs-example-navbar-collapse-1"
                aria-expanded="false">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
      </div>
      <ul class="nav navbar-nav navbar-right collapse navbar-collapse"
          id="bs-example-navbar-collapse-1">
        .
        .
        .
      </ul>
    </nav>
  </div>
</header>
```

The result of Listing 8.23 appears in Figure 8.13. Pressing or clicking the hamburger icon then opens the menu, as shown in Figure 8.14. Note what a significant improvement this is compared to the default menu shown in Figure 8.12.

As a final step, we'll add a little CSS to fix the footer issue. The trick is to use a *media query* to apply different CSS for mobile devices than for desktop computers. The result, obtained after some trial-and-error in a web inspector, appears in Listing 8.24.[16] Note that the new styles will be applied when the width of the device is less than 800 pixels, which is a common threshold (known as a "breakpoint") for mobile devices. After refreshing the browser to apply the style in Listing 8.24, our app appears as shown in Figure 8.15.

[16] Thanks to Learn Enough cofounder and *Learn Enough CSS & Layout to Be Dangerous* coauthor Lee Donahoe for his assistance.


SAMPLE APP
Example User
The Ruby on Rails Tutorial by Michael Hartl
About Contact News
--- !ruby/object:ActionController::Parame
ters
parameters: !ruby/hash:ActiveSupport::Has
hWithIndifferentAccess
controller: users
action: show
id: '1'
permitted: false


SAMPLE APP
Home
Help
Users
Account
About
Contact
News
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: users
action: show
id: '1'
permitted: false

Listing 8.24: Updating the footer CSS.
app/assets/stylesheets/custom.scss

```
.
.
.
/* footer */

footer {
  .
  .
  .
}
@media (max-width: 800px) {
  footer {
    small {
      display: block;
      float: none;
      margin-bottom: 1em;
    }
    ul {
      float: none;
      padding: 0;
      li {
        float: none;
        margin-left: 0;
      }
    }
  }
}
```

With that, our mobile styling is done, and we're ready to add some tests for the layout links added in this section.

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 cookie inspector in your browser (Section 8.2.1), remove the session cookie and confirm that the layout links revert to the non-logged-in state.


SAMPLE APP
Example User
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News
--- !ruby/object:ActionController::Parame
ters
parameters: !ruby/hash:ActiveSupport::Has
hWithIndifferentAccess
controller: users
action: show
id: '1'
permitted: false

2. Log in again, confirming that the layout links change correctly. Then quit your browser and start it again to confirm that the layout links revert to the non-logged-in state. (If your browser has a "remember where I left off" feature that automatically restores the session, be sure to disable it in this step (Box 1.2).)

8.2.4 Testing layout changes

Having verified by hand that the application is behaving properly upon successful login, before moving on we'll write an integration test to capture that behavior and catch regressions. We'll build on the test from Listing 8.9 and write a series of steps to verify the following sequence of actions:

1. Visit the login path.
2. Post valid information to the sessions path.
3. Verify that the login link disappears.
4. Verify that a logout link appears
5. Verify that a profile link appears.

In order to see these changes, our test needs to log in as a previously registered user, which means that such a user must already exist in the database. The default Rails way to do this is to use *fixtures*, which are a way of organizing data to be loaded into the test database. We discovered in Section 6.2.5 that we needed to delete the default fixtures so that our email uniqueness tests would pass (Listing 6.31). Now we're ready to start filling in that empty file with custom fixtures of our own.

In the present case, we need only one user, whose information should consist of a valid name and email address. Because we'll need to log the user in, we also have to include a valid password to compare with the password submitted to the Sessions controller's `create` action. Referring to the data model in Figure 6.9, we see that this means creating a `password_digest` attribute for

the user fixture, which we'll accomplish by defining a **digest** method of our own.

As discussed in Section 6.3.1, the password digest is created using bcrypt (via **has_secure_password**), so we'll need to create the fixture password using the same method. By inspecting the secure password source code, we find that this method is

```
BCrypt::Password.create(string, cost: cost)
```

where **string** is the string to be hashed and **cost** is the *cost parameter* that determines the computational cost to calculate the hash. Using a high cost makes it computationally intractable to use the hash to determine the original password, which is an important security precaution in a production environment, but in tests we want the **digest** method to be as fast as possible. The secure password source code has a line for this as well:

```
cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                              BCrypt::Engine.cost
```

This rather obscure code, which you don't need to understand in detail, arranges for precisely the behavior described above: it uses the minimum cost parameter in tests and a normal (high) cost parameter in production. (We'll learn more about the strange **?-:** notation in Section 9.2.)

There are several places we could put the resulting **digest** method, but we'll have an opportunity in Section 9.1.1 to reuse **digest** in the User model. This suggests placing the method in **user.rb**. Because we won't necessarily have access to a user object when calculating the digest (as will be the case in the fixtures file), we'll attach the **digest** method to the User class itself, which (as we saw briefly in Section 4.4.1) makes it a *class method*. The result appears in Listing 8.25.

Listing 8.25: Adding a digest method for use in fixtures.
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end
end
```

With the **`digest`** method from Listing 8.25, we are now ready to create a user fixture for a valid user, as shown in Listing 8.26.[17]

Listing 8.26: A fixture for testing user login.
test/fixtures/users.yml

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>
```

Note in particular that fixtures support embedded Ruby, which allows us to use

```
<%= User.digest('password') %>
```

to create the valid password digest for the test user.

[17] It's worth noting that indentation in fixture files must take the form of spaces, not tabs, so take care when copying code like that shown in Listing 8.26.

Although we've defined the **password_digest** attribute required by **has_secure_password**, sometimes it's convenient to refer to the plain (virtual) password as well. Unfortunately, this is impossible to arrange with fixtures, and adding a **password** attribute to Listing 8.26 causes Rails to complain that there is no such column in the database (which is true). We'll make do by adopting the convention that all fixture users have the same password (**'password'**).

Having created a fixture with a valid user, we can retrieve it inside a test as follows:

```
user = users(:michael)
```

Here **users** corresponds to the fixture filename **users.yml**, while the symbol **:michael** references user with the key shown in Listing 8.26.

With the fixture user as above, we can now write a test for the layout links by converting the sequence enumerated at the beginning of this section into code, as shown in Listing 8.27.

Listing 8.27: A test for user logging in with valid information. GREEN
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid information" do
    get login_path
    post login_path, params: { session: { email:    @user.email,
                                          password: 'password' } }
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
  end
end
```

Here we've used

```
assert_redirected_to @user
```

to check the right redirect target and

```
follow_redirect!
```

to actually visit the target page. Listing 8.27 also verifies that the login link disappears by verifying that there are *zero* login path links on the page:

```
assert_select "a[href=?]", login_path, count: 0
```

By including the extra **`count: 0`** option, we tell **`assert_select`** that we expect there to be zero links matching the given pattern. (Compare this to **`count: 2`** in Listing 5.32, which checks for exactly two matching links.)

Because the application code was already working, this test should be GREEN:

Listing 8.28: GREEN

```
$ rails test test/integration/users_login_test.rb
```

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. Confirm by commenting out everything after **`if user`** in Line 8 of Listing 8.15 that the tests still pass even if we don't authenticate the user by email and password, as shown in Listing 8.29. This is because Listing 8.9

doesn't test the case of a correct user email but incorrect password. Fix this serious omission in our test suite by adding a valid email to the Users login test (Listing 8.30). Verify that the tests are RED, then remove the Line 8 comment to get back to GREEN. (Because it's so important, we'll add this test to the main code in Section 8.3.)

2. Use the "safe navigation" operator `&.` to simplify the boolean test in Line 8 of Listing 8.15, as shown in Line 8 of Listing 8.31.[18] This Ruby feature allows us to condense the common pattern of `obj && obj.method` into `obj&.method`. Confirm that the tests in Listing 8.30 still pass after the change.

Listing 8.29: Commenting out the authentication code, but tests still GREEN.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user # && user.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Listing 8.30: Testing the case of valid user email, invalid password.
test/integration/users_login_test.rb

```
require 'test_helper'
```

[18] Thanks to reader Aviv Levinsky for suggesting this addition.

```
class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid email/invalid password" do
    get login_path
    assert_template 'sessions/new'
    post login_path, params: { session: { email:    FILL_IN,
                                          password: "invalid" } }
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end
  .
  .
  .
end
```

Listing 8.31: Using the "safe navigation" operator `&.` to simplify the login code.

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user&.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out
    redirect_to root_url
  end
end
```

8.2.5 Login upon signup

Although our authentication system is now working, newly registered users might be confused, as they are not logged in by default. Because it would be strange to force users to log in immediately after signing up, we'll log in new users automatically as part of the signup process. To arrange this behavior, all we need to do is add a call to **log_in** in the Users controller **create** action, as shown in Listing 8.32.[19]

Listing 8.32: Logging in the user upon signup.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

To test the behavior from Listing 8.32, we can add a line to the test from List-

[19] As with the Sessions controller, the **log_in** method is available in the Users controller because of the module inclusion in Listing 8.13.

ing 7.31 to check that the user is logged in. It's helpful in this context to define an **is_logged_in?** helper method to parallel the **logged_in?** helper defined in Listing 8.18, which returns **true** if there's a user id in the (test) session and false otherwise (Listing 8.33). (Because helper methods aren't available in tests, we can't use the **current_user** as in Listing 8.18, but the **session** method is available, so we use that instead.) Here we use **is_logged_in?** instead of **logged_in?** so that the test helper and Sessions helper methods have different names, which prevents them from being mistaken for each other.[20] (In this case we could actually just include the Sessions helper and use **logged_in?** directly, but this technique would fail in Chapter 9 due to details of how cookies are handled in tests, so instead we define a test-specific method that will work in all cases.)

Listing 8.33: A boolean method for login status inside tests.
test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end
end
```

With the code in Listing 8.33, we can assert that the user is logged in after signup using the line shown in Listing 8.34.

Listing 8.34: A test of login after signup. GREEN
test/integration/users_signup_test.rb

[20] For example, I once had a test suite that was GREEN even after I accidentally deleted the main **log_in** method in the Sessions helper. The reason is that the tests were happily using a test helper with the same name, thereby passing even though the application was completely broken. As with **is_logged_in?**, we'll avoid this issue by defining the test helper **log_in_as** in Listing 9.24.

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest
  .
  .
  .
  test "valid signup information" do
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, params: { user: { name:  "Example User",
                                         email: "user@example.com",
                                         password:              "password",
                                         password_confirmation: "password" } }
    end
    follow_redirect!
    assert_template 'users/show'
    assert is_logged_in?
  end
end
```

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

Listing 8.35: 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. Is the test suite RED or GREEN if you comment out the **`log_in`** line in Listing 8.32?

2. By using your text editor's ability to comment out code, toggle back and forth between commenting out code in Listing 8.32 and confirm that the test suite toggles between RED and GREEN. (You will need to save the file between toggles.)

8.3 Logging out

As discussed in Section 8.1, our authentication model is to keep users logged in until they log out explicitly. In this section, we'll add this necessary logout capability. Because the "Log out" link has already been defined (Listing 8.19), all we need is to write a valid controller action to destroy user sessions.

So far, the Sessions controller actions have followed the RESTful convention of using **`new`** for a login page and **`create`** to complete the login. We'll continue this theme by using a **`destroy`** action to delete sessions, i.e., to log out. Unlike with the login functionality, which we use in both Listing 8.15 and Listing 8.32, we'll be logging out only in one place, so we'll put the relevant code directly in the **`destroy`** action. As we'll see in Section 9.3, this design (with a little refactoring) will also make the authentication machinery easier to test.

Logging out involves undoing the effects of the **`log_in`** method from Listing 8.14, which involves deleting the user id from the session.[21] To do this, we use the **`delete`** method as follows:

```
session.delete(:user_id)
```

We'll also set the current user to **`nil`**, although in the present case this won't matter because of an immediate redirect to the root URL.[22] As with **`log_in`** and associated methods, we'll put the resulting **`log_out`** method in the Sessions helper module, as shown in Listing 8.36.

Listing 8.36: The **`log_out`** method.
app/helpers/sessions_helper.rb

[21] Some browsers offer a "remember where I left off" feature, which restores the session automatically, so be sure to disable any such feature before trying to log out.

[22] Setting **`@current_user`** to **`nil`** would matter only if **`@current_user`** were created before the **`destroy`** action (which it isn't) *and* if we didn't issue an immediate redirect (which we do). This is an unlikely combination of events, and with the application as presently constructed it isn't necessary, but because it's security-related I include it for completeness.

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
  .
  .
  .
  # Logs out the current user.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

We can put the **log_out** method to use in the Sessions controller's **destroy** action, as shown in Listing 8.37.

Listing 8.37: Destroying a session (user logout).
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out
    redirect_to root_url
  end
end
```

To test the logout machinery, we can add some steps to the user login test from Listing 8.27. After logging in, we use **delete** to issue a DELETE request

to the logout path (Table 8.1) and verify that the user is logged out and redirected to the root URL. We also check that the login link reappears and that the logout and profile links disappear. The new steps appear in Listing 8.38.

Listing 8.38: A test for user logout (and an improved test for invalid login).
GREEN
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid email/invalid password" do
    get login_path
    assert_template 'sessions/new'
    post login_path, params: { session: { email:    @user.email,
                                          password: "invalid" } }
    assert_not is_logged_in?
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end

  test "login with valid information followed by logout" do
    get login_path
    post login_path, params: { session: { email:    @user.email,
                                          password: 'password' } }
    assert is_logged_in?
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
    delete logout_path
    assert_not is_logged_in?
    assert_redirected_to root_url
    follow_redirect!
    assert_select "a[href=?]", login_path
    assert_select "a[href=?]", logout_path,      count: 0
    assert_select "a[href=?]", user_path(@user), count: 0
  end
end
```

(Now that we have `is_logged_in?` available in tests, we've also thrown in a bonus `assert is_logged_in?` immediately after posting valid information to the sessions path. We've also added a similar assertion and the solution to the exercise from Section 8.2.4 by adding the results of Listing 8.30.)

With the session `destroy` action thus defined and tested, the initial sign-up/login/logout triumvirate is complete, and the test suite should be GREEN:

Listing 8.39: 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. Confirm in a browser that the "Log out" link causes the correct changes in the site layout. What is the correspondence between these changes and the final three steps in Listing 8.38?

2. By checking the site cookies, confirm that the session is correctly removed after logging out.

8.4 Conclusion

With the material in this chapter, our sample application has a fully functional login and authentication system. In the next chapter, we'll take our app to the next level by adding the ability to remember users for longer than a single browser session.

Before moving on, merge your changes back into the master branch:

```
$ rails test
$ git add -A
$ git commit -m "Implement basic login"
$ git checkout master
$ git merge basic-login
```

Then push up to the remote repository:

```
$ rails test
$ git push
```

Finally, deploy to Heroku as usual:

```
$ git push heroku
```

8.4.1 What we learned in this chapter

- Rails can maintain state from one page to the next using temporary cookies via the **`session`** method.
- The login form is designed to create a new session to log a user in.
- The **`flash.now`** method is used for flash messages on rendered pages.
- Test-driven development is useful when debugging by reproducing the bug in a test.
- Using the **`session`** method, we can securely place a user id on the browser to create a temporary session.
- We can change features such as links on the layouts based on login status.
- Integration tests can verify correct routes, database updates, and proper changes to the layout.

Chapter 9

Advanced login

The basic login system developed in Chapter 8 is fully functional, but most modern websites include the ability to "remember" users when they visit the site again even if they've closed their browsers in the interim. In this chapter, we use *permanent cookies* to implement this behavior. We'll start by automatically remembering users when they log in (Section 9.1), a common model used by sites such as Bitbucket and GitHub. We'll then add the ability to *optionally* remember users using a "remember me" checkbox, a model used by sites such as Twitter and Facebook.

Because the Chapter 8 login system is complete by itself, the core of the sample application will work fine without it, and if desired you can skip right to Chapter 10 (and from there to Chapter 13). On the other hand, learning how to implement the "remember me" feature is both highly instructive by itself and lays an essential foundation for account activation (Chapter 11) and password reset (Chapter 12). Moreover, the result is an outstanding example of computer magic: You've seen a billion of these "remember me" login forms on the Web, and now's your chance to learn how to make one.

9.1 Remember me

In this section, we'll add the ability to remember our users' login state even after they close and reopen their browsers. This "remember me" behavior will

happen automatically, and users will automatically stay logged in until they explicitly log out. As we'll see, the resulting machinery will make it easy to add an optional "remember me" checkbox as well (Section 9.2).

As usual, I suggest switching to a topic branch before proceeding:

```
$ git checkout -b advanced-login
```

9.1.1 Remember token and digest

In Section 8.2, we used the Rails `session` method to store the user's id, but this information disappears when the user closes their browser. In this section, we'll take the first step toward persistent sessions by generating a *remember token* appropriate for creating permanent cookies using the `cookies` method, together with a secure *remember digest* for authenticating those tokens.

As noted in Section 8.2.1, information stored using `session` is automatically secure, but this is not the case with information stored using `cookies`. In particular, persistent cookies are vulnerable to session hijacking, in which an attacker uses a stolen remember token to log in as a particular user. There are four main ways to steal cookies: (1) using a packet sniffer to detect cookies being passed over insecure networks,[1] (2) compromising a database containing remember tokens, (3) using cross-site scripting (XSS), and (4) gaining physical access to a machine with a logged-in user.

We prevented the first problem in Section 7.5 by using Secure Sockets Layer (SSL) site-wide, which protects network data from packet sniffers. We'll prevent the second problem by storing a hash digest of the remember tokens instead of the token itself, in much the same way that we stored password digests instead of raw passwords in Section 6.3.[2] Rails automatically prevents the third problem by escaping any content inserted into view templates. Finally, although there's no iron-clad way to stop attackers who have physical access to a logged-in computer, we'll minimize the fourth problem by changing tokens every time

[1] Session hijacking was widely publicized by the Firesheep application, which showed that remember tokens at many high-profile sites were visible from devices connected to public Wi-Fi networks.

[2] Rails 5 introduced a `has_secure_token` method that automatically generates random tokens, but it stores the *unhashed* values in the database, and hence is unsuitable for our present purposes.

a user logs out and by taking care to *cryptographically sign* any potentially sensitive information we place on the browser.

With these design and security considerations in mind, our plan for creating persistent sessions appears as follows:

1. Create a random string of digits for use as a remember token.
2. Place the token in the browser cookies with an expiration date far in the future.
3. Save the hash digest of the token to the database.
4. Place an encrypted version of the user's id in the browser cookies.
5. When presented with a cookie containing a persistent user id, find the user in the database using the given id, and verify that the remember token cookie matches the associated hash digest from the database.

Note how similar the final step is to logging a user in, where we retrieve the user by email address and then verify (using the **`authenticate`** method) that the submitted password matches the password digest (Listing 8.7). As a result, our implementation will parallel aspects of **`has_secure_password`**.

We'll start by adding the required **`remember_digest`** attribute to the User model, as shown in Figure 9.1.
To add the data model from Figure 9.1 to our application, we'll generate a migration:

```
$ rails generate migration add_remember_digest_to_users remember_digest:string
```

(Compare to the password digest migration in Section 6.3.1.) As in previous migrations, we've used a migration name that ends in **`_to_users`** to tell Rails that the migration is designed to alter the **`users`** table in the database. Because we also included the attribute (**`remember_digest`**) and type (**`string`**), Rails generates a default migration for us, as shown in Listing 9.1.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string

Figure 9.1: The User model with an added `remember_digest` attribute.

Listing 9.1: The generated migration for the remember digest.
db/migrate/[timestamp]_add_remember_digest_to_users.rb

```
class AddRememberDigestToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :remember_digest, :string
  end
end
```

Because we don't expect to retrieve users by remember digest, there's no need to put an index on the `remember_digest` column, and we can use the default migration as generated above:

```
$ rails db:migrate
```

Now we have to decide what to use as a remember token. There are many mostly equivalent possibilities—essentially, any long random string will do. The `urlsafe_base64` method from the `SecureRandom` module in the Ruby standard library fits the bill:[3] it returns a random string of length 22 composed

[3]This choice is based on the RailsCast on remember me.

of the characters A–Z, a–z, 0–9, "-", and "_" (for a total of 64 possibilities, thus "base64"). A typical base64 string appears as follows:

```
$ rails console
>> SecureRandom.urlsafe_base64
=> "brl_446-8bqHv87AQzUj_Q"
```

Just as it's perfectly fine if two users have the same password,[4] there's no need for remember tokens to be unique, but it's more secure if they are.[5] In the case of the base64 string above, each of the 22 characters has 64 possibilities, so the probability of two remember tokens colliding is a negligibly small $1/64^{22} = 2^{-132} \approx 10^{-40}$.[6] As a bonus, by using base64 strings specifically designed to be safe in URLs (as indicated by the name **`urlsafe_base64`**), we'll be able to use the same token generator to make account activation and password reset links in Chapter 12.

Remembering users involves creating a remember token and saving the digest of the token to the database. We've already defined a **`digest`** method for use in the test fixtures (Listing 8.25), and we can use the results of the discussion above to create a **`new_token`** method to create a new token. As with **`digest`**, the new token method doesn't need a user object, so we'll make it a class method.[7] The result is the User model shown in Listing 9.2.

Listing 9.2: Adding a method for generating tokens.
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
```

[4] In any case, with bcrypt's salted hashes there's no way for us to tell if two users' passwords match.

[5] With unique remember tokens, an attacker always needs *both* the user id and the remember token cookies to hijack the session.

[6] This hasn't stopped some developers from adding a check to verify that no collision has occurred, but such efforts result from failing to grasp just how small 10^{-40} is. For example, if we generated a billion tokens a second for the entire age of the Universe (4.4×10^7 s), the expected number of collisions would still be on the order of 2×10^{-23}, which is zero in any operational sense of the word.

[7] As a general rule, if a method doesn't need an instance of an object, it should be a class method. Indeed, this decision will prove to be wise in Section 11.2.

```
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end
end
```

Our plan for the implementation is to make a **`user.remember`** method that associates a remember token with the user and saves the corresponding remember digest to the database. Because of the migration in Listing 9.1, the User model already has a **`remember_digest`** attribute, but it doesn't yet have a **`remember_token`** attribute. We need a way to make a token available via **`user.remember_token`** (for storage in the cookies) *without* storing it in the database. We solved a similar issue with secure passwords in Section 6.3, which paired a virtual **`password`** attribute with a secure **`password_digest`** attribute in the database. In that case, the virtual **`password`** attribute was created automatically by **`has_secure_password`**, but we'll have to write the code for a **`remember_token`** ourselves. The way to do this is to use **`attr_accessor`** to create an accessible attribute, which we saw before in Section 4.4.5:

```
class User < ApplicationRecord
  attr_accessor :remember_token
  .
  .
  .
  def remember
    self.remember_token = ...
    update_attribute(:remember_digest, ...)
  end
end
```

Note the form of the assignment in the first line of the **remember** method. Because of the way Ruby handles assignments inside objects, without **self** the assignment would create a *local* variable called **remember_token**, which isn't what we want. Using **self** ensures that assignment sets the user's **remember_token** attribute. (Now you know why the **before_save** callback from Listing 6.32 uses **self.email** instead of just **email**.) Meanwhile, the second line of **remember** uses the **update_attribute** method to update the remember digest. (As noted in Section 6.1.5, this method bypasses the validations, which is necessary in this case because we don't have access to the user's password or confirmation.)

With these considerations in mind, we can create a valid token and associated digest by first making a new remember token using **User.new_token**, and then updating the remember digest with the result of applying **User.digest**. This procedure gives the **remember** method shown in Listing 9.3.

Listing 9.3: Adding a **remember** method to the User model. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
```

```
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end
end
```

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. In the console, assign **`user`** to the first user in the database, and verify by calling it directly that the **`remember`** method works. How do **`remember_token`** and **`remember_digest`** compare?

2. In Listing 9.3, we defined the new token and digest class methods by explicitly prefixing them with **`User`**. This works fine and, because they are actually *called* using **`User.new_token`** and **`User.digest`**, it is probably the clearest way to define them. But there are two perhaps more idiomatically correct ways to define class methods, one slightly confusing and one extremely confusing. By running the test suite, verify that the implementations in Listing 9.4 (slightly confusing) and Listing 9.5 (extremely confusing) are correct. (Note that, in the context of Listing 9.4 and Listing 9.5, **`self`** is the **`User`** class, whereas the other uses of **`self`** in the User model refer to a user object *instance*. This is part of what makes them confusing.)

Listing 9.4: Defining the new token and digest methods using **`self`**. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  # Returns the hash digest of the given string.
  def self.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
```

```
                                                    BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def self.new_token
    SecureRandom.urlsafe_base64
  end
  .
  .
  .
end
```

Listing 9.5: Defining the new token and digest methods using `class << self`. GREEN

app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  class << self
    # Returns the hash digest of the given string.
    def digest(string)
      cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                    BCrypt::Engine.cost
      BCrypt::Password.create(string, cost: cost)
    end

    # Returns a random token.
    def new_token
      SecureRandom.urlsafe_base64
    end
  end
  .
  .
  .
```

9.1.2 Login with remembering

Having created a working `user.remember` method, we can now create a persistent session by storing a user's (encrypted) id and remember token as permanent cookies on the browser. The way to do this is with the `cookies` method,

which (as with **session**) we can treat as a hash. A cookie consists of two pieces of information, a **value** and an optional **expires** date. For example, we could make a persistent session by creating a cookie with value equal to the remember token that expires 20 years from now:

```
cookies[:remember_token] = { value:   remember_token,
                             expires: 20.years.from_now.utc }
```

(This uses one of the convenient Rails time helpers, as discussed in Box 9.1.) This pattern of setting a cookie that expires 20 years in the future is so common that Rails has a special **permanent** method to implement it, so that we can simply write

```
cookies.permanent[:remember_token] = remember_token
```

This causes Rails to set the expiration to **20.years.from_now** automatically.

Box 9.1. Cookies expire `20.years.from_now`

You may recall from Section 4.4.2 that Ruby lets you add methods to *any* class, even built-in ones. In that section, we added a `palindrome?` method to the `String` class (and discovered as a result that `"deified"` is a palindrome), and we also saw how Rails adds a `blank?` method to class `Object` (so that `"".blank?`, `" ".blank?`, and `nil.blank?` are all `true`). The `cookies.permanent` method, which creates "permanent" cookies with an expiration `20.years.from_now`, gives yet another example of this practice through one of Rails' *time helpers*, which are methods added to `Fixnum` (the base class for integers):

```
$ rails console
>> 1.year.from_now
=> Wed, 21 Jun 2017 19:36:29 UTC +00:00
>> 10.weeks.ago
=> Tue, 12 Apr 2016 19:36:44 UTC +00:00
```

Rails adds other helpers, too:

```
>> 1.kilobyte
=> 1024
>> 5.megabytes
=> 5242880
```

These are useful for upload validations, making it easy to restrict, say, image uploads to `5.megabytes`.

Although it should be used with caution, the flexibility to add methods to built-in classes allows for extraordinarily natural additions to plain Ruby. Indeed, much of the elegance of Rails ultimately derives from the malleability of the underlying Ruby language.

To store the user's id in the cookies, we could follow the pattern used with the **`session`** method (Listing 8.14) using something like

```
cookies[:user_id] = user.id
```

Because it places the id as plain text, this method exposes the form of the application's cookies and makes it easier for an attacker to compromise user accounts. To avoid this problem, we'll use an *encrypted* cookie, which securely encrypts the cookie before placing it on the browser:

```
cookies.encrypted[:user_id] = user.id
```

Because we want the user id to be paired with the permanent remember token, we should make it permanent as well, which we can do by chaining the **`encrypted`** and **`permanent`** methods:

```
cookies.permanent.encrypted[:user_id] = user.id
```

After the cookies are set, on subsequent page views we can retrieve the user with code like

```
User.find_by(id: cookies.encrypted[:user_id])
```

where **`cookies.encrypted[:user_id]`** automatically decrypts the user id cookie. We can then use bcrypt to verify that **`cookies[:remember_token]`** matches the **`remember_digest`** generated in Listing 9.3. (In case you're wondering why we don't just use the encrypted user id, without the remember token, this would allow an attacker with possession of the encrypted id to log in as the user in perpetuity. In the present design, an attacker with both cookies can log in as the user only until the user logs out.)

The final piece of the puzzle is to verify that a given remember token matches the user's remember digest, and in this context there are a couple of equivalent ways to use bcrypt to verify a match. If you look at the secure password source code, you'll find a comparison like this:[8]

```
BCrypt::Password.new(password_digest) == unencrypted_password
```

In our case, the analogous code would look like this:

```
BCrypt::Password.new(remember_digest) == remember_token
```

If you think about it, this code is really strange: it appears to be comparing a bcrypt password digest directly with a token, which would imply *decrypting* the digest in order to compare using **`==`**. But the whole point of using bcrypt is for hashing to be irreversible, so this can't be right. Indeed, digging into

[8] As noted in Section 6.3.1, "unencrypted password" is a misnomer, as the secure password is *hashed*, not encrypted.

the source code of the bcrypt gem verifies that the comparison operator **==** is being *redefined*, and under the hood the comparison above is equivalent to the following:

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

Instead of **==**, this uses the boolean method **is_password?** to perform the comparison. Because its meaning is a little clearer, we'll prefer this second comparison form in the application code.

The above discussion suggests putting the digest–token comparison into an **authenticated?** method in the User model, which plays a role similar to that of the **authenticate** method provided by **has_secure_password** for authenticating a user (Listing 8.15). The implementation appears in Listing 9.6. (Although the **authenticated?** method in Listing 9.6 is tied specifically to the remember digest, it will turn out to be useful in other contexts as well, and we'll generalize it in Chapter 11.)

Listing 9.6: Adding an **authenticated?** method to the User model.
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
```

```
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end

  # Returns true if the given token matches the digest.
  def authenticated?(remember_token)
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end
end
```

Note that the **remember_token** argument in the **authenticated?** method defined in Listing 9.6 is not the same as the accessor that we defined in Listing 9.3 using **attr_accessor :remember_token**; instead, it is a variable local to the method. (Because the argument refers to the remember token, it is not uncommon to use a method argument that has the same name.) Also note the use of the **remember_digest** attribute, which is the same as **self.remember_digest** and, like **name** and **email** in Chapter 6, is created automatically by Active Record based on the name of the corresponding database column (Listing 9.1).

We're now in a position to remember a logged-in user, which we'll do by adding a **remember** helper to go along with **log_in**, as shown in Listing 9.7.

Listing 9.7: Logging in and remembering a user. RED
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      remember user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
```

```
    end
  end

  def destroy
    log_out
    redirect_to root_url
  end
end
```

As with **log_in**, Listing 9.7 defers the real work to the Sessions helper, where we define a **remember** method that calls **user.remember**, thereby generating a remember token and saving its digest to the database. It then uses **cookies** to create permanent cookies for the user id and remember token as described above. The result appears in Listing 9.8.

Listing 9.8: Remembering the user. GREEN
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.encrypted[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the current logged-in user (if any).
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end

  # Returns true if the user is logged in, false otherwise.
  def logged_in?
    !current_user.nil?
  end

  # Logs out the current user.
  def log_out
```

```
    session.delete(:user_id)
    @current_user = nil
  end
end
```

With the code in Listing 9.8, a user logging in will be remembered in the sense that their browser will get a valid remember token, but it doesn't yet do us any good because the **`current_user`** method defined in Listing 8.16 knows only about the temporary session:

```
@current_user ||= User.find_by(id: session[:user_id])
```

In the case of persistent sessions, we want to retrieve the user from the temporary session if **`session[:user_id]`** exists, but otherwise we should look for **`cookies[:user_id]`** to retrieve (and log in) the user corresponding to the persistent session. We can accomplish this as follows:

```
if session[:user_id]
  @current_user ||= User.find_by(id: session[:user_id])
elsif cookies.encrypted[:user_id]
  user = User.find_by(id: cookies.encrypted[:user_id])
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

(This follows the same **`user && user.authenticated`** pattern we saw in Listing 8.7.) The code above will work, but note the repeated use of both **`session`** and **`cookies`**. We can eliminate this duplication as follows:

```
if (user_id = session[:user_id])
  @current_user ||= User.find_by(id: user_id)
elsif (user_id = cookies.encrypted[:user_id])
  user = User.find_by(id: user_id)
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

This uses the common but potentially confusing construction

```
if (user_id = session[:user_id])
```

Despite appearances, this is *not* a comparison (which would use double-equals `==`), but rather is an *assignment*. If you were to read it in words, you wouldn't say "If user id equals session of user id…", but rather something like "If session of user id exists (while setting user id to session of user id)…".[9]

Defining the `current_user` helper as discussed above leads to the implementation shown in Listing 9.9.

Listing 9.9: Updating `current_user` for persistent sessions. RED
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.encrypted[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.encrypted[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
```

[9] I generally use the convention of putting such assignments in parentheses, which is a visual reminder that it's not a comparison.

```
  # Returns true if the user is logged in, false otherwise.
  def logged_in?
    !current_user.nil?
  end

  # Logs out the current user.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

With the code as in Listing 9.9, newly logged in users are correctly remembered, as you can verify by logging in, closing the browser, and checking that you're still logged in when you restart the sample application and revisit the sample application.[10] If you want, you can even inspect the browser cookies to see the result directly (Figure 9.2).[11]

There's only one problem with our application as it stands: short of clearing their browser cookies (or waiting 20 years), there's no way for users to log out. This is exactly the sort of thing our test suite should catch, and indeed the tests should currently be RED:

[10] Alert reader Jack Fahnestock has noted that there is an edge case that isn't covered by the current design:

1. Log in with "remember me" checked in browser A (saving hashed remember token A to **`remember_digest`**).
2. Log in with "remember me" checked in browser B (saving hashed **`remember_token`** B to **`remember_digest`**, overwriting remember token A saved in browser A).
3. Close browser A (now relying on permanent cookies for login—second conditional in **`current_user`** method).
4. Reopen browser A (**`logged_in?`** returns false, even though permanent cookies are on the browser).

Although this is arguably a more secure design than remembering the user in multiple places, it violates the expectation that users can be permanently remembered on more than one browser. The solution, which is substantially more complicated than the present design, is to factor the remember digest into a separate table, where each row has a user id and a digest. Checking for the current user would then look through the table for a digest corresponding to a particular remember token. Furthermore, the **`forget`** in Listing 9.11 method would delete only the row corresponding to the digest of the current browser. For security purposes, logging out would remove all digests for that user.

[11] Google "<your browser name> inspect cookies" to learn how to inspect the cookies on your system.

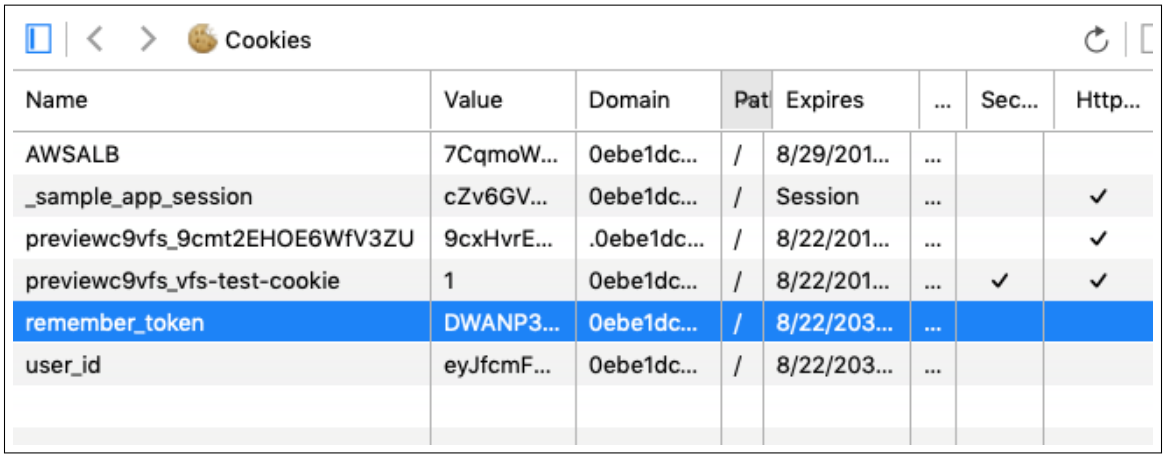


Figure 9.2: The remember token cookie in the local browser.

Listing 9.10: RED

```
$ 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. By finding the cookie in your local browser, verify that a remember token and encrypted user id are present after logging in.

2. At the console, verify directly that the **`authenticated?`** method defined in Listing 9.6 works correctly.

9.1.3 Forgetting users

To allow users to log out, we'll define methods to forget users in analogy with the ones to remember them. The resulting **`user.forget`** method just undoes

`user.remember` by updating the remember digest with **`nil`**, as shown in Listing 9.11.

Listing 9.11: Adding a **`forget`** method to the User model. RED
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end

  # Returns true if the given token matches the digest.
  def authenticated?(remember_token)
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end

  # Forgets a user.
  def forget
    update_attribute(:remember_digest, nil)
  end
end
```

With the code in Listing 9.11, we're now ready to forget a permanent session by adding a **`forget`** helper and calling it from the **`log_out`** helper (List-

ing 9.12). As seen in Listing 9.12, the **forget** helper calls **user.forget** and then deletes the **user_id** and **remember_token** cookies.

Listing 9.12: Logging out from a persistent session. GREEN
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
  .
  .
  .
  # Forgets a persistent session.
  def forget(user)
    user.forget
    cookies.delete(:user_id)
    cookies.delete(:remember_token)
  end

  # Logs out the current user.
  def log_out
    forget(current_user)
    session.delete(:user_id)
    @current_user = nil
  end
end
```

At this point, the tests suite should be GREEN:

Listing 9.13: 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. After logging out, verify that the corresponding cookies have been removed from your browser.

9.1.4 Two subtle bugs

There are two closely related subtleties left to address. The first subtlety is that, even though the "Log out" link appears only when logged-in, a user could potentially have multiple browser windows open to the site. If the user logged out in one window, thereby setting **`current_user`** to **`nil`**, clicking the "Log out" link in a second window would result in an error because of **`forget(current_user)`** in the **`log_out`** method (Listing 9.12).[12] We can avoid this by logging out only if the user is logged in.

The second subtlety is that a user could be logged in (and remembered) in multiple browsers, such as Chrome and Firefox, which causes a problem if the user logs out in the first browser but not the second, and then closes and reopens the second one.[13] For example, suppose that the user logs out in Firefox, thereby setting the remember digest to **`nil`** (via **`user.forget`** in Listing 9.11). The application will still work in Firefox; because the **`log_out`** method in Listing 9.12 deletes the user's id, both highlighted conditionals are **`false`**:

```
# Returns the user corresponding to the remember token cookie.
def current_user
  if (user_id = session[:user_id])
    @current_user ||= User.find_by(id: user_id)
  elsif (user_id = cookies.encrypted[:user_id])
    user = User.find_by(id: user_id)
    if user && user.authenticated?(cookies[:remember_token])
      log_in user
      @current_user = user
    end
  end
end
```

As a result, evaluation falls off the end of the **`current_user`** method, thereby returning **`nil`** as required.

[12] Thanks to reader Paulo Célio Júnior for pointing this out.

[13] Thanks to reader Niels de Ron for pointing this out.

In contrast, if we close Chrome, we set **`session[:user_id]`** to **`nil`** (because all **`session`** variables expire automatically on browser close), but the **`user_id`** *cookie* will still be present. This means that the corresponding user will still be pulled out of the database when Chrome is relaunched:

```
# Returns the user corresponding to the remember token cookie.
def current_user
  if (user_id = session[:user_id])
    @current_user ||= User.find_by(id: user_id)
  elsif (user_id = cookies.encrypted[:user_id])
    user = User.find_by(id: user_id)
    if user && user.authenticated?(cookies[:remember_token])
      log_in user
      @current_user = user
    end
  end
end
```

Consequently, the inner **`if`** conditional will be evaluated:

```
user && user.authenticated?(cookies[:remember_token])
```

In particular, because **`user`** isn't **`nil`**, the *second* expression will be evaluated, which raises an error. This is because the user's remember digest was deleted as part of logging out (Listing 9.11) in Firefox, so when we access the application in Chrome we end up calling

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

with a **`nil`** remember digest, thereby raising an exception inside the bcrypt library. To fix this, we want **`authenticated?`** to return **`false`** instead.

These are exactly the sorts of subtleties that benefit from test-driven development, so we'll write tests to catch the two errors before correcting them. We first get the integration test from Listing 8.38 to RED, as shown in Listing 9.14.

Listing 9.14: A test for logging out in a second window. RED
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest
  .
  .
  .
  test "login with valid information followed by logout" do
    get login_path
    post login_path, params: { session: { email:    @user.email,
                                          password: 'password' } }
    assert is_logged_in?
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
    delete logout_path
    assert_not is_logged_in?
    assert_redirected_to root_url
    # Simulate a user clicking logout in a second window.
    delete logout_path
    follow_redirect!
    assert_select "a[href=?]", login_path
    assert_select "a[href=?]", logout_path,      count: 0
    assert_select "a[href=?]", user_path(@user), count: 0
  end
end
```

The second call to **`delete logout_path`** in Listing 9.14 should raise an error due to the missing **`current_user`**, leading to a RED test suite:

Listing 9.15: RED

```
$ rails test
```

The application code simply involves calling **`log_out`** only if **`logged_in?`** is true, as shown in Listing 9.16.

Listing 9.16: Only logging out if logged in. GREEN
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

The second case, involving a scenario with two different browsers, is harder to simulate with an integration test, but it's easy to check in the User model test directly. All we need is to start with a user that has no remember digest (which is true for the **`@user`** variable defined in the **`setup`** method) and then call **`authenticated?`**, as shown in Listing 9.17. (Note that we've just left the remember token blank; it doesn't matter what its value is, because the error occurs before it ever gets used.)

Listing 9.17: A test of **`authenticated?`** with a nonexistent digest. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "authenticated? should return false for a user with nil digest" do
    assert_not @user.authenticated?('')
  end
end
```

Because **`BCrypt::Password.new(nil)`** raises an error, the test suite should now be RED:

Listing 9.18: RED

```
$ rails test
```

To fix the error and get to GREEN, all we need to do is return **`false`** if the remember digest is **`nil`**, as shown in Listing 9.19.

Listing 9.19: Updating **`authenticated?`** to handle a nonexistent digest. GREEN

app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  # Returns true if the given token matches the digest.
  def authenticated?(remember_token)
    return false if remember_digest.nil?
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end

  # Forgets a user.
  def forget
    update_attribute(:remember_digest, nil)
  end
end
```

This uses the **`return`** keyword to return immediately if the remember digest is **`nil`**, which is a common way to emphasize that the rest of the method gets ignored in that case. The equivalent code

```
if remember_digest.nil?
  false
else
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

would also work fine, but I prefer the explicitness of the version in Listing 9.19 (which also happens to be slightly shorter).

With the code in Listing 9.19, our full test suite should be GREEN, and both subtleties should now be addressed:

Listing 9.20: 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. Comment out the fix in Listing 9.16 and then verify that the first subtle bug is present by opening two logged-in tabs, logging out in one, and then clicking "Log out" link in the other.

2. Comment out the fix in Listing 9.19 and verify that the second subtle bug is present by logging out in one browser and closing and opening the second browser.

3. Uncomment the fixes and confirm that the test suite goes from RED to GREEN.

9.2 "Remember me" checkbox

With the code in Section 9.1.3, our application has a complete, professional-grade authentication system. As a final step, we'll see how to make staying logged in optional using a "remember me" checkbox. A mockup of the login form with such a checkbox appears in Figure 9.3.

To write the implementation, we start by adding a checkbox to the login form from Listing 8.4. As with labels, text fields, password fields, and submit buttons, checkboxes can be created with a Rails helper method. In order to get the styling right, though, we have to *nest* the checkbox inside the label, as follows:

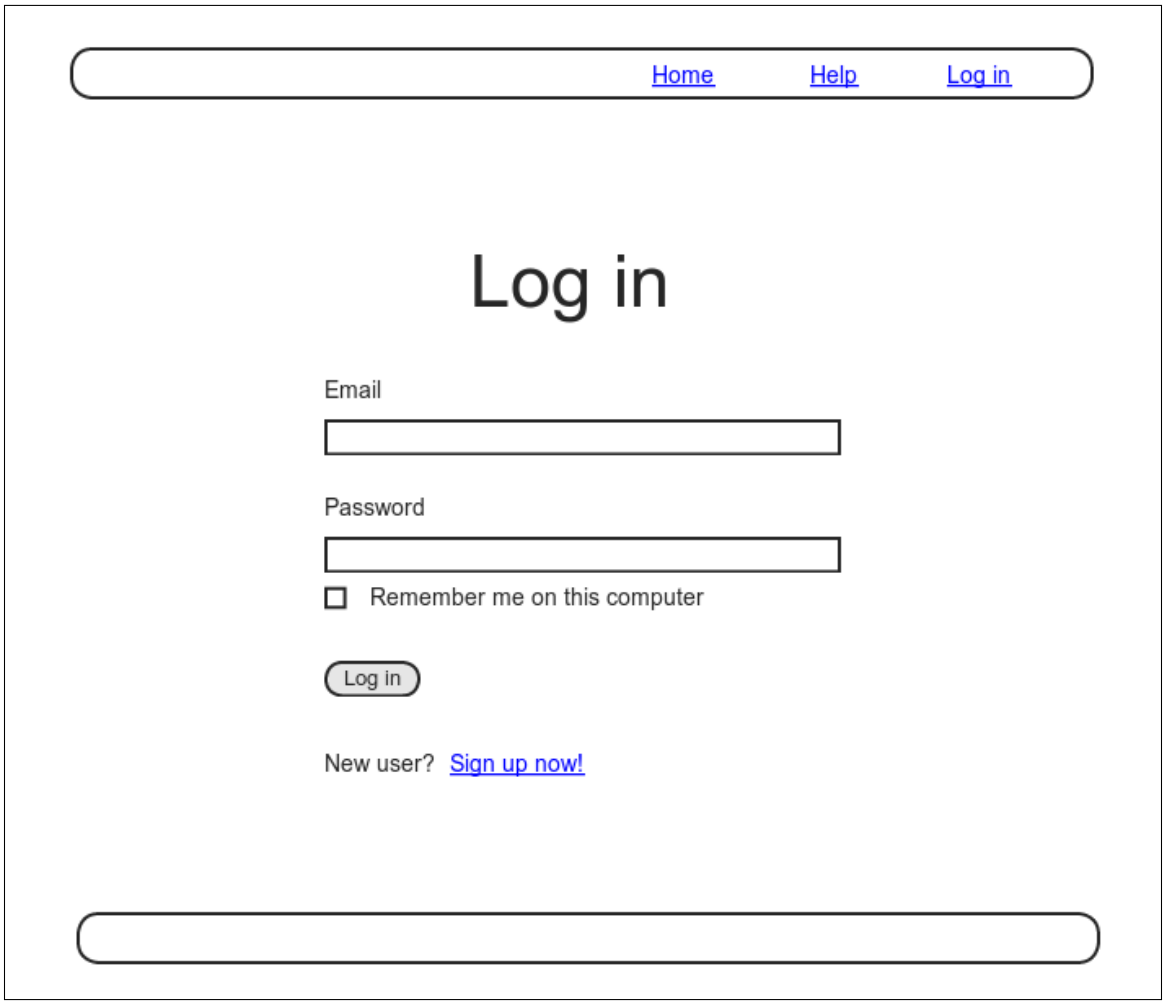


Figure 9.3: A mockup of a “remember me” checkbox.

```
<%= f.label :remember_me, class: "checkbox inline" do %>
  <%= f.check_box :remember_me %>
  <span>Remember me on this computer</span>
<% end %>
```

Putting this into the login form gives the code shown in Listing 9.21.

Listing 9.21: Adding a "remember me" checkbox to the login form.
app/views/sessions/new.html.erb

```
<% provide(:title, "Log in") %>
<h1>Log in</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(url: login_path, scope: :session, local: true) do |f| %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :remember_me, class: "checkbox inline" do %>
        <%= f.check_box :remember_me %>
        <span>Remember me on this computer</span>
      <% end %>

      <%= f.submit "Log in", class: "btn btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

In Listing 9.21, we've included the CSS classes **`checkbox`** and **`inline`**, which Bootstrap uses to put the checkbox and the text ("Remember me on this computer") in the same line. In order to complete the styling, we need just a few more CSS rules, as shown in Listing 9.22. The resulting login form appears in Figure 9.4.

Listing 9.22: CSS for the "remember me" checkbox.
app/assets/stylesheets/custom.scss

```
.
.
.
/* forms */
.
.
.
.checkbox {
  margin-top: -10px;
  margin-bottom: 10px;
  span {
    margin-left: 20px;
    font-weight: normal;
  }
}

#session_remember_me {
  width: auto;
  margin-left: 0;
}
```

Having edited the login form, we're now ready to remember users if they check the checkbox and forget them otherwise. Incredibly, because of all our work in the previous sections, the implementation can be reduced to one line. We start by noting that the **params** hash for submitted login forms now includes a value based on the checkbox (as you can verify by submitting the form in Listing 9.21 with invalid information and inspecting the values in the debug section of the page). In particular, the value of

```
params[:session][:remember_me]
```

is **'1'** if the box is checked and **'0'** if it isn't.

By testing the relevant value of the **params** hash, we can now remember or forget the user based on the value of the submission:[14]

[14]Note that this means unchecking the box will log out the user on all browsers on all computers. The alternate design of remembering user login sessions on each browser independently is potentially more convenient for users, but it's less secure, and is also more complicated to implement. Ambitious readers are invited to try their hand at implementing it.

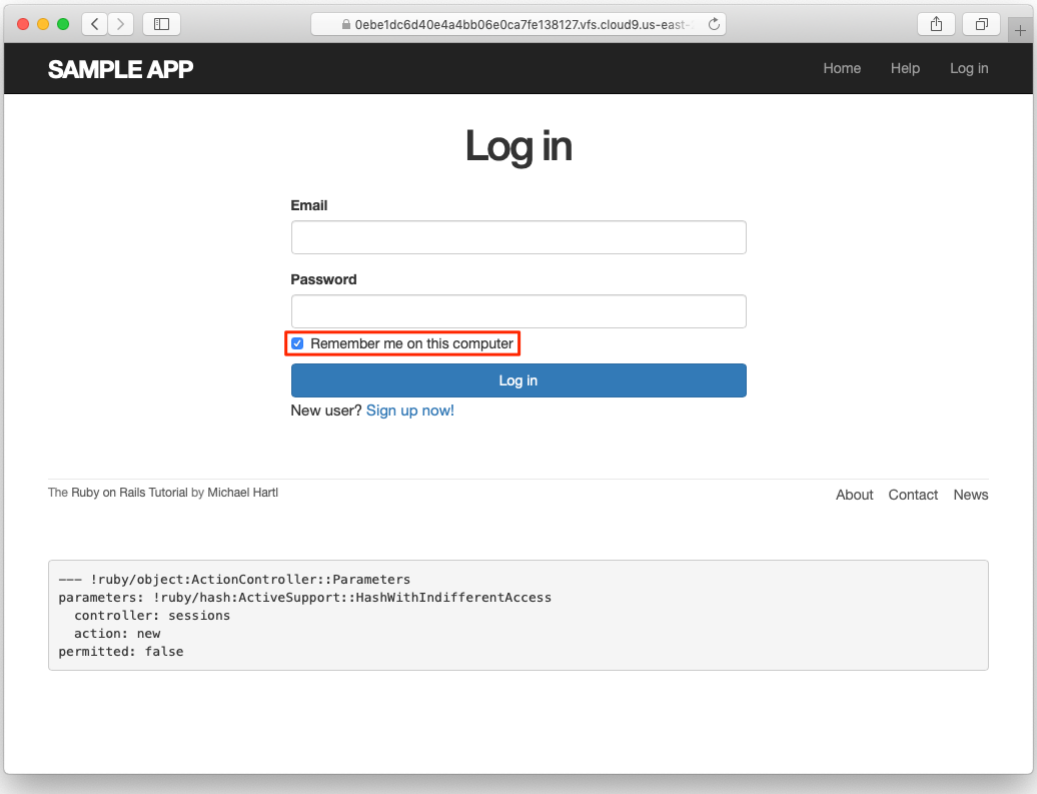


Figure 9.4: The login form with an added "remember me" checkbox.

```
if params[:session][:remember_me] == '1'
  remember(user)
else
  forget(user)
end
```

As explained in Box 9.2, this sort of **if-then** branching structure can be converted to one line using the *ternary operator* as follows:[15]

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

Using this to replace **remember user** in the Sessions controller's **create** method (Listing 9.7) leads to the amazingly compact code shown in Listing 9.23. (Now you're in a position to understand the code in Listing 8.25, which uses the ternary operator to define the bcrypt **cost** variable.)

Listing 9.23: Handling the submission of the "remember me" checkbox.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      params[:session][:remember_me] == '1' ? remember(user) : forget(user)
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

[15] Before we wrote **remember user** without parentheses, but when used with the ternary operator omitting them results in a syntax error.

With the implementation in Listing 9.23, our login system is complete, as you can verify by checking or unchecking the box in your browser.

Box 9.2. 10 types of people

There's an old joke that there are 10 kinds of people in the world: those who understand binary and those who don't (10, of course, being 2 in binary). In this spirit, we can say that there are 10 kinds of people in the world: those who like the ternary operator, those who don't, and those who don't yet know about it. (If you happen to be in the third category, soon you won't be any longer.)

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

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

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

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

You can also use the ternary operator to replace assignment, so that

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

becomes

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

Finally, it's often convenient to use the ternary operator in a function's return value:

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

Since Ruby implicitly returns the value of the last expression in a function, here the `foo` method returns `"bar"` or `"baz"` depending on whether `boolean?` is `true` or `false`.

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. By inspecting your browser's cookies directly, verify that the "remember me" checkbox is having its intended effect.

2. At the console, invent examples showing both possible behaviors of the ternary operator (Box 9.2).

9.3 Remember tests

Although our "remember me" functionality is now working, it's important to write some tests to verify its behavior. One reason is to catch implementation errors, as discussed in a moment. Even more important, though, is that the core user persistence code is in fact completely untested at present. Fixing these

issues will require some trickery, but the result will be a far more powerful test suite.

9.3.1 Testing the "remember me" box

When I originally implemented the checkbox handling in Listing 9.23, instead of the correct

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

I actually used

```
params[:session][:remember_me] ? remember(user) : forget(user)
```

In this context, **`params[:session][:remember_me]`** is either **`'0'`** or **`'1'`**, both of which are **`true`** in a boolean context, so the resulting expression is *always true*, and the application acts as if the checkbox is always checked. This is exactly the kind of error a test should catch.

Because remembering users requires that they be logged in, our first step is to define a helper to log users in inside tests. In Listing 8.27, we logged a user in using the **`post`** method and a valid **`session`** hash, but it's cumbersome to do this every time. To avoid needless repetition, we'll write a helper method called **`log_in_as`** to log in for us.

Our method for logging a user in depends on the type of test. Inside controller tests, we can manipulate the **`session`** method directly, assigning **`user.id`** to the **`:user_id`** key (as first seen in Listing 8.14):

```
def log_in_as(user)
  session[:user_id] = user.id
end
```

We call the method **`log_in_as`** to avoid any confusion with the application code's **`log_in`** method as defined in Listing 8.14. Its location is in the **`ActiveSupport::TestCase`** class inside the **`test_helper`** file, the same location as the **`is_logged_in?`** helper from Listing 8.33:

```
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end

  # Log in as a particular user.
  def log_in_as(user)
    session[:user_id] = user.id
  end
end
```

We won't actually need this version of the method in this chapter, but we'll put it to use in Chapter 10.

Inside integration tests, we can't manipulate **`session`** directly, but we can **`post`** to the sessions path as in Listing 8.27, which leads to the **`log_in_as`** method shown here:

```
class ActionDispatch::IntegrationTest

  # Log in as a particular user.
  def log_in_as(user, password: 'password', remember_me: '1')
    post login_path, params: { session: { email: user.email,
                                          password: password,
                                          remember_me: remember_me } }
  end
end
```

Because it's located inside the **`ActionDispatch::IntegrationTest`** class, this is the version of **`log_in_as`** that will be called inside integration tests. We use the same method name in both cases because it lets us do things like use code from a controller test in an integration without making any changes to the login method.

Putting these two methods together yields the parallel **`log_in_as`** helpers shown in Listing 9.24.

Listing 9.24: Adding a `log_in_as` helper.
test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end

  # Log in as a particular user.
  def log_in_as(user)
    session[:user_id] = user.id
  end
end

class ActionDispatch::IntegrationTest

  # Log in as a particular user.
  def log_in_as(user, password: 'password', remember_me: '1')
    post login_path, params: { session: { email: user.email,
                                          password: password,
                                          remember_me: remember_me } }
  end
end
```

Note that, for maximum flexibility, the second `log_in_as` method in Listing 9.24 accepts keyword arguments (as in Listing 7.13), with default values for the password and for the "remember me" checkbox set to `'password'` and `'1'`, respectively.

To verify the behavior of the "remember me" checkbox, we'll write two tests, one each for submitting with and without the checkbox checked. This is easy using the login helper defined in Listing 9.24, with the two cases appearing as

```
log_in_as(@user, remember_me: '1')
```

and

```
log_in_as(@user, remember_me: '0')
```

(Because `'1'` is the default value of `remember_me`, we could omit the corresponding option in the first case above, but I've included it to make the parallel structure more apparent.)

After logging in, we can check if the user has been remembered by looking for the `remember_token` key in the `cookies`. Ideally, we would check that the cookie's value is equal to the user's remember token, but as currently designed there's no way for the test to get access to it: the `user` variable in the controller has a remember token attribute, but (because `remember_token` is virtual) the `@user` variable in the test doesn't. Fixing this minor blemish is left as an exercise (Section 9.3.1), but for now we can just test to see if the relevant cookie is `nil` or not. The results appear in Listing 9.25. (Recall from Listing 8.27 that `users(:michael)` references the fixture user from Listing 8.26.)

Listing 9.25: A test of the "remember me" checkbox. GREEN
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
    assert_not_empty cookies[:remember_token]
  end

  test "login without remembering" do
    # Log in to set the cookie.
    log_in_as(@user, remember_me: '1')
    # Log in again and verify that the cookie is deleted.
    log_in_as(@user, remember_me: '0')
    assert_empty cookies[:remember_token]
  end
end
```

Assuming you didn't make the same implementation mistake I did, the tests should be GREEN:

Listing 9.26: 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. As mentioned above, the application currently doesn't have any way to access the virtual **`remember_token`** attribute in the integration test in Listing 9.25. It is possible, though, using a special test method called **`assigns`**. Inside a test, you can access *instance* variables defined in the controller by using **`assigns`** with the corresponding symbol. For example, if the **`create`** action defines an **`@user`** variable, we can access it in the test using **`assigns(:user)`**. Right now, the Sessions controller **`create`** action defines a normal (non-instance) variable called **`user`**, but if we change it to an instance variable we can test that **`cookies`** correctly contains the user's remember token. By filling in the missing elements in Listing 9.27 and Listing 9.28 (indicated with question marks **`?`** and **`FILL_IN`**), complete this improved test of the "remember me" checkbox.

Listing 9.27: A template for using an instance variable in the **`create`** action.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    ?user = User.find_by(email: params[:session][:email].downcase)
```

```
    if ?user && ?user.authenticate(params[:session][:password])
      log_in ?user
      params[:session][:remember_me] == '1' ? remember(?user) : forget(?user)
      redirect_to ?user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

Listing 9.28: A template for an improved "remember me" test. GREEN
test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
    assert_equal FILL_IN, assigns(:user).FILL_IN
  end

  test "login without remembering" do
    # Log in to set the cookie.
    log_in_as(@user, remember_me: '1')
    # Log in again and verify that the cookie is deleted.
    log_in_as(@user, remember_me: '0')
    assert_empty cookies[:remember_token]
  end
  .
  .
  .
end
```

9.3.2 Testing the remember branch

In Section 9.1.2, we verified by hand that the persistent session implemented in the preceding sections is working, but in fact the relevant branch in the **`current_user`** method is currently completely untested. My favorite way to handle this kind of situation is to raise an exception in the suspected untested block of code: if the code isn't covered, the tests will still pass; if it is covered, the resulting error will identify the relevant test. The result in the present case appears in Listing 9.29.

Listing 9.29: Raising an exception in an untested branch. GREEN
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.encrypted[:user_id])
      raise       # The tests still pass, so this branch is currently untested.
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

At this point, the tests are GREEN:

Listing 9.30: GREEN

```
$ rails test
```

This is a problem, of course, because the code in Listing 9.29 is broken. Moreover, persistent sessions are cumbersome to check by hand, so if we ever want

to refactor the `current_user` method (as we will in Chapter 11) it's important to test it.

Because both versions of the `log_in_as` helper method defined in Listing 9.24 automatically set `session[:user_id]` (either explicitly or by posting to the login path), testing the "remember" branch of the `current_user` method is difficult in an integration test. Luckily, we can bypass this restriction by testing the `current_user` method directly in a Sessions helper test, whose file we have to create:

```
$ touch test/helpers/sessions_helper_test.rb
```

The test sequence is simple:

1. Define a `user` variable using the fixtures.
2. Call the `remember` method to remember the given user.
3. Verify that `current_user` is equal to the given user.

Because the `remember` method doesn't set `session[:user_id]`, this procedure will test the desired "remember" branch. The result appears in Listing 9.31.

Listing 9.31: A test for persistent sessions. RED
test/helpers/sessions_helper_test.rb

```
require 'test_helper'

class SessionsHelperTest < ActionView::TestCase

  def setup
    @user = users(:michael)
    remember(@user)
  end

  test "current_user returns right user when session is nil" do
    assert_equal @user, current_user
    assert is_logged_in?
  end
```

```
  test "current_user returns nil when remember digest is wrong" do
    @user.update_attribute(:remember_digest, User.digest(User.new_token))
    assert_nil current_user
  end
end
```

Note that we've added a second test, which checks that the current user is **`nil`** if the user's remember digest doesn't correspond correctly to the remember token, thereby testing the **`authenticated?`** expression in the nested **`if`** statement:

```
if user && user.authenticated?(cookies[:remember_token])
```

Incidentally, in Listing 9.31 we could write

```
assert_equal current_user, @user
```

instead, and it would work just the same, but (as mentioned briefly in Section 5.3.4) the conventional order for the arguments to **`assert_equal`** is *expected*, *actual*:

```
assert_equal <expected>, <actual>
```

which in the case of Listing 9.31 gives

```
assert_equal @user, current_user
```

With the code as in Listing 9.31, the test is RED as required:

Listing 9.32: RED

```
$ rails test test/helpers/sessions_helper_test.rb
```

We can get the tests in Listing 9.31 to pass by removing the **`raise`** and restoring the original **`current_user`** method, as shown in Listing 9.33.

Listing 9.33: Removing the raised exception. GREEN
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.encrypted[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

At this point, the test suite should be GREEN:

Listing 9.34: GREEN

```
$ rails test
```

Now that the "remember" branch of **`current_user`** is tested, we can be confident of catching regressions without having to check by hand.

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 removing the **`authenticated?`** expression in Listing 9.33 that the second test in Listing 9.31 fails, thereby confirming that it tests the right thing.

9.4 Conclusion

We've covered a lot of ground in the last three chapters, transforming our promising but unformed application into a site capable of the full suite of signup and login behaviors. All that is needed to complete the authentication functionality is to restrict access to pages based on login status and user identity. We'll accomplish this task en route to giving users the ability to edit their information, which is the main goal of Chapter 10.

Before moving on, merge your changes back into the master branch:

```
$ rails test
$ git add -A
$ git commit -m "Implement advanced login"
$ git checkout master
$ git merge advanced-login
$ git push
```

Before deploying to Heroku, it's worth noting that the application will briefly be in an invalid state after pushing but before the migration is finished. On a production site with significant traffic, it's a good idea to turn *maintenance mode* on before making the changes:

```
$ heroku maintenance:on
$ git push heroku
$ heroku run rails db:migrate
$ heroku maintenance:off
```

This arranges to show a standard error page during the deployment and migration (Figure 9.5). (We won't bother with this step again, but it's good to see it at least once.) For more information, see the Heroku documentation on error pages.

9.4.1 What we learned in this chapter

- Rails can maintain state from one page to the next using persistent cookies via the `cookies` method.

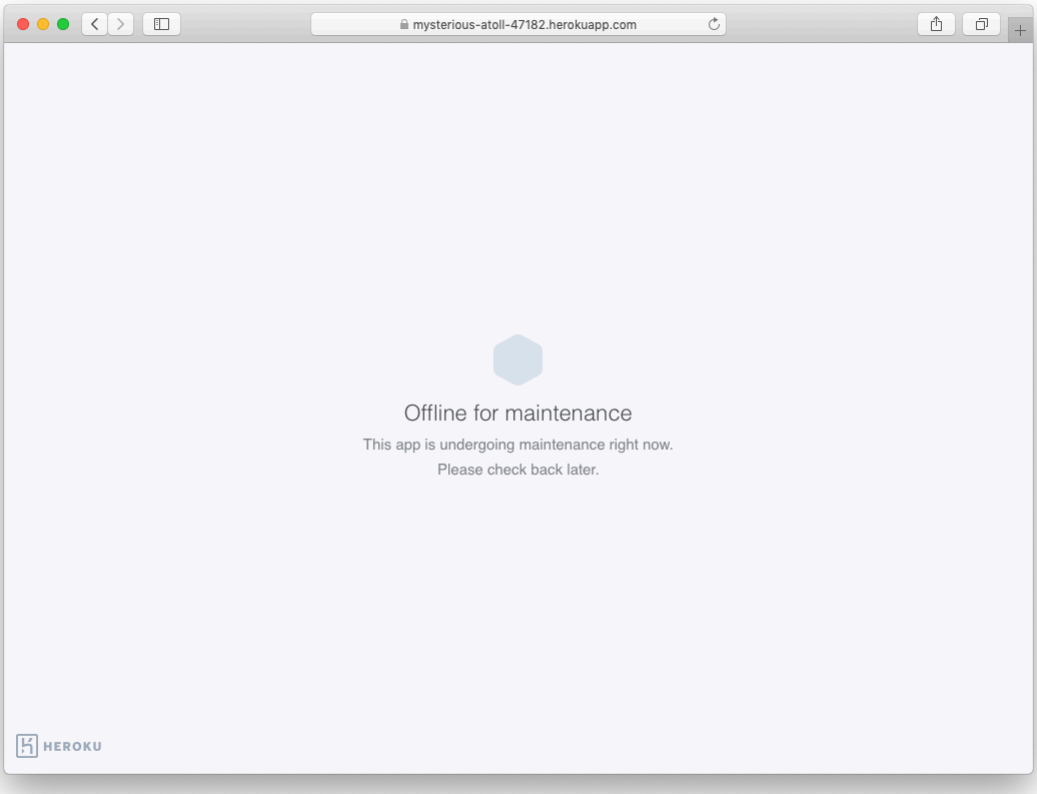


Figure 9.5: The production app in maintenance mode.

- We associate to each user a remember token and a corresponding remember digest for use in persistent sessions.
- Using the `cookies` method, we create a persistent session by placing a permanent remember token cookie on the browser.
- Login status is determined by the presence of a current user based on the temporary session's user id or the permanent session's unique remember token.
- The application signs users out by deleting the session's user id and removing the permanent cookie from the browser.
- The ternary operator is a compact way to write simple if-then statements.

Chapter 10

Updating, showing, and deleting users

In this chapter, we will complete the REST actions for the Users resource (Table 7.1) by adding `edit`, `update`, `index`, and `destroy` actions. We'll start by giving users the ability to update their profiles, which will also provide a natural opportunity to enforce an authorization model (made possible by the authentication code in Chapter 8). Then we'll make a listing of all users (also requiring authentication), which will motivate the introduction of sample data and pagination. Finally, we'll add the ability to destroy users, wiping them clear from the database. Since we can't allow just any user to have such dangerous powers, we'll take care to create a privileged class of administrative users authorized to delete other users.

10.1 Updating users

The pattern for editing user information closely parallels that for creating new users (Chapter 7). Instead of a `new` action rendering a view for new users, we have an `edit` action rendering a view to edit users; instead of `create` responding to a `POST` request, we have an `update` action responding to a `PATCH` request (Box 3.2). The biggest difference is that, while anyone can sign up, only the current user should be able to update their information. The authentication

machinery from Chapter 8 will allow us to use a *before filter* to ensure that this is the case.

To get started, let's start work on an **updating-users** topic branch:

```
$ git checkout -b updating-users
```

10.1.1 Edit form

We start with the edit form, whose mockup appears in Figure 10.1.[1] To turn the mockup in Figure 10.1 into a working page, we need to fill in both the Users controller **edit** action and the user edit view. We start with the **edit** action, which requires pulling the relevant user out of the database. Note from Table 7.1 that the proper URL for a user's edit page is /users/1/edit (assuming the user's id is 1). Recall that the id of the user is available in the **params[:id]** variable, which means that we can find the user with the code in Listing 10.1.

Listing 10.1: The user **edit** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
```

[1] Image retrieved from https://www.flickr.com/photos/sashawolff/4598355045/ on 2014-08-25. Copyright © 2010 by Sasha Wolff and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

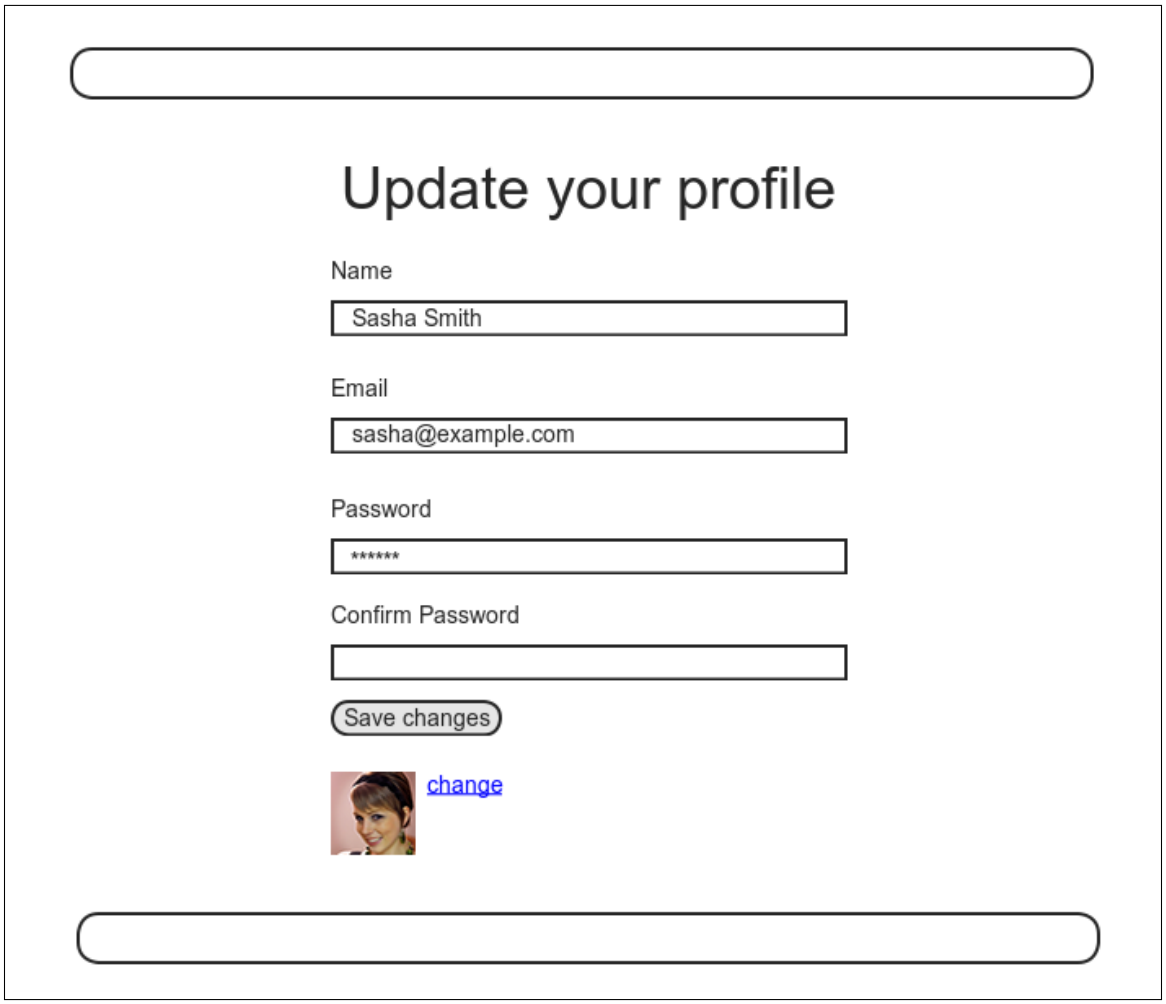


Figure 10.1: A mockup of the user edit page.

```
    end
  end

  def edit
    @user = User.find(params[:id])
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

The corresponding user edit view (which you will have to create by hand) is shown in Listing 10.2. Note how closely this resembles the new user view from Listing 7.15; the large overlap suggests factoring the repeated code into a partial, which is left as an exercise (Section 10.1.1).

Listing 10.2: The user edit view.
app/views/users/edit.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, local: true) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%= f.submit "Save changes", class: "btn btn-primary" %>
    <% end %>
```

```
    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="https://gravatar.com/emails" target="_blank">change</a>
    </div>
  </div>
</div>
```

Here we have reused the shared `error_messages` partial introduced in Section 7.3.3. By the way, the use of `target="_blank"` in the Gravatar link is a neat trick to get the browser to open the page in a new window or tab, which is sometimes convenient behavior when linking to third-party sites. (There's a minor security issue associated with `target="_blank"`; dealing with this detail is left as an exercise (Section 10.1.1).)

With the `@user` instance variable from Listing 10.1, the edit page should render properly, as shown in Figure 10.2. The "Name" and "Email" fields in Figure 10.2 also shows how Rails automatically pre-fills the Name and Email fields using the attributes of the existing `@user` variable.

Looking at the HTML source for Figure 10.2, we see a form tag as expected, as in Listing 10.3 (slight details may differ).

Listing 10.3: HTML for the edit form defined in Listing 10.2 and shown in Figure 10.2.

```
<form accept-charset="UTF-8" action="/users/1" class="edit_user"
      id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="patch" />
  .
  .
  .
</form>
```

Note here the hidden input field:

```
<input name="_method" type="hidden" value="patch" />
```

Since web browsers can't natively send `PATCH` requests (as required by the REST conventions from Table 7.1), Rails fakes it with a `POST` request and a

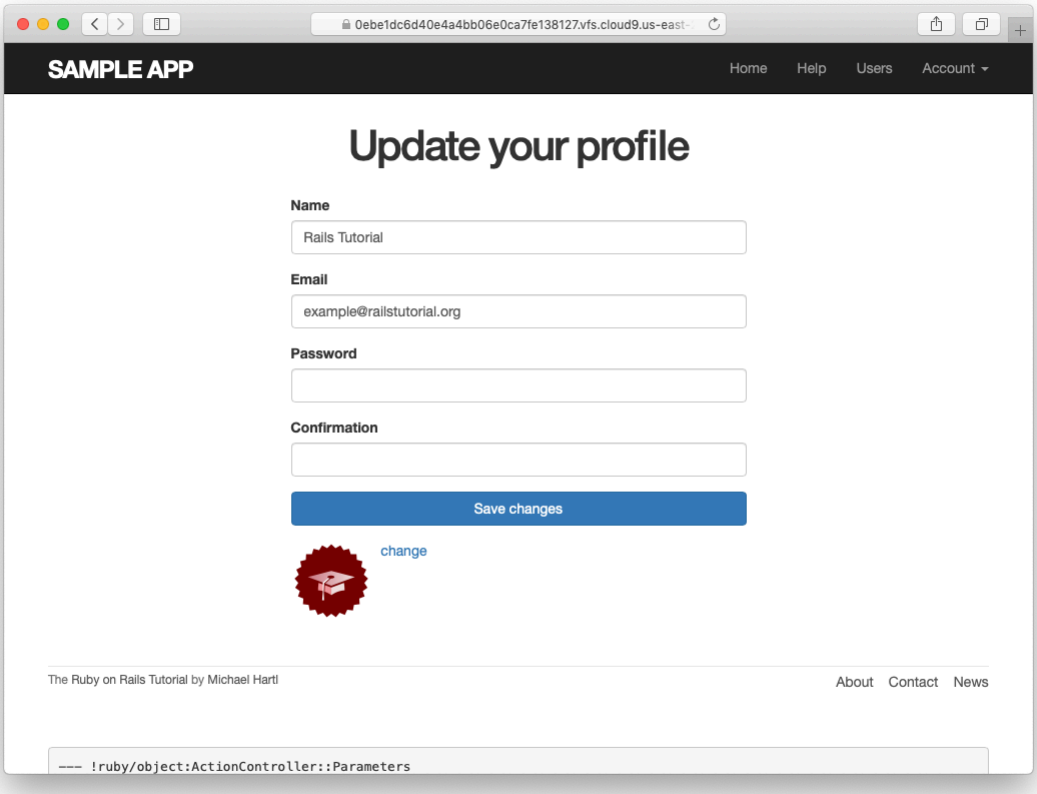


Figure 10.2: The initial user edit page with pre-filled name and email.

hidden **input** field.[2]

There's another subtlety to address here: the code **form_with(@user)** in Listing 10.2 is *exactly* the same as the code in Listing 7.15—so how does Rails know to use a POST request for new users and a PATCH for editing users? The answer is that it is possible to tell whether a user is new or already exists in the database via Active Record's **new_record?** boolean method:

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

When constructing a form using **form_with(@user)**, Rails uses POST if **@user.new_record?** is **true** and PATCH if it is **false**.

As a final touch, we'll fill in the URL of the settings link in the site navigation. This is easy using the named route **edit_user_path** from Table 7.1, together with the handy **current_user** helper method defined in Listing 9.9:

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

The full application code appears in Listing 10.4).

Listing 10.4: Adding a URL to the "Settings" link in the site layout.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
```

[2]Don't worry about how this works; the details are of interest to developers of the Rails framework itself, and by design are not important for Rails application developers.

```
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Account <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li><%= link_to "Profile", current_user %></li>
              <li><%= link_to "Settings", edit_user_path(current_user) %></li>
              <li class="divider"></li>
              <li>
                <%= link_to "Log out", logout_path, method: :delete %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

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. As noted above, there's a minor security issue associated with using **`target="_blank"`** to open URLs, which is that the target site gains control of what's known as the "**`window`** object" associated with the HTML document. The result is that the target site could potentially introduce malicious content, such as a phishing page. This is extremely unlikely to happen when linking to a reputable site like Gravatar, but it turns out that we can eliminate the risk entirely by setting the **`rel`** attribute ("relationship") to **`"noopener"`** in the origin link. Add this attribute to the Gravatar edit link in Listing 10.2.

2. Remove the duplicated form code by refactoring the **`new.html.erb`** and **`edit.html.erb`** views to use the partial in Listing 10.5, as shown in Listing 10.6 and Listing 10.7. Note the use of the **`provide`** method,

which we used before in Section 3.4.3 to eliminate duplication in the layout.[3]

Listing 10.5: A partial for the `new` and `edit` form.
app/views/users/_form.html.erb

```
<%= form_with(model: @user, local: true) do |f| %>
  <%= render 'shared/error_messages', object: @user %>

  <%= f.label :name %>
  <%= f.text_field :name, class: 'form-control' %>

  <%= f.label :email %>
  <%= f.email_field :email, class: 'form-control' %>

  <%= f.label :password %>
  <%= f.password_field :password, class: 'form-control' %>

  <%= f.label :password_confirmation %>
  <%= f.password_field :password_confirmation, class: 'form-control' %>

  <%= f.submit yield(:button_text), class: "btn btn-primary" %>
<% end %>
```

Listing 10.6: The signup view with partial.
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<% provide(:button_text, 'Create my account') %>
<h1>Sign up</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
  </div>
</div>
```

Listing 10.7: The edit view with partial.
app/views/users/edit.html.erb

[3] Thanks to Jose Carlos Montero Gómez for a suggestion that further reduced duplication in the `new` and `edit` partials.

```
<% provide(:title, 'Edit user') %>
<% provide(:button_text, 'Save changes') %>
<h1>Update your profile</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="https://gravatar.com/emails" target="_blank">Change</a>
    </div>
  </div>
</div>
```

10.1.2 Unsuccessful edits

In this section we'll handle unsuccessful edits, following similar ideas to unsuccessful signups (Section 7.3). We start by creating an **update** action, which uses **update** (Section 6.1.5) to update the user based on the submitted **params** hash, as shown in Listing 10.8. With invalid information, the update attempt returns **false**, so the **else** branch renders the edit page. We've seen this pattern before; the structure closely parallels the first version of the **create** action (Listing 7.18).

Listing 10.8: The initial user **update** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
```

```
      render 'new'
    end
  end

  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update(user_params)
      # Handle a successful update.
    else
      render 'edit'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

Note the use of **`user_params`** in the call to **`update`**, which uses strong parameters to prevent mass assignment vulnerability (as described in Section 7.3.2).

Because of the existing User model validations and the error-messages partial in Listing 10.2, submission of invalid information results in helpful error messages (Figure 10.3).

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. Confirm by submitting various invalid combinations of username, email, and password that the edit form won't accept invalid submissions.

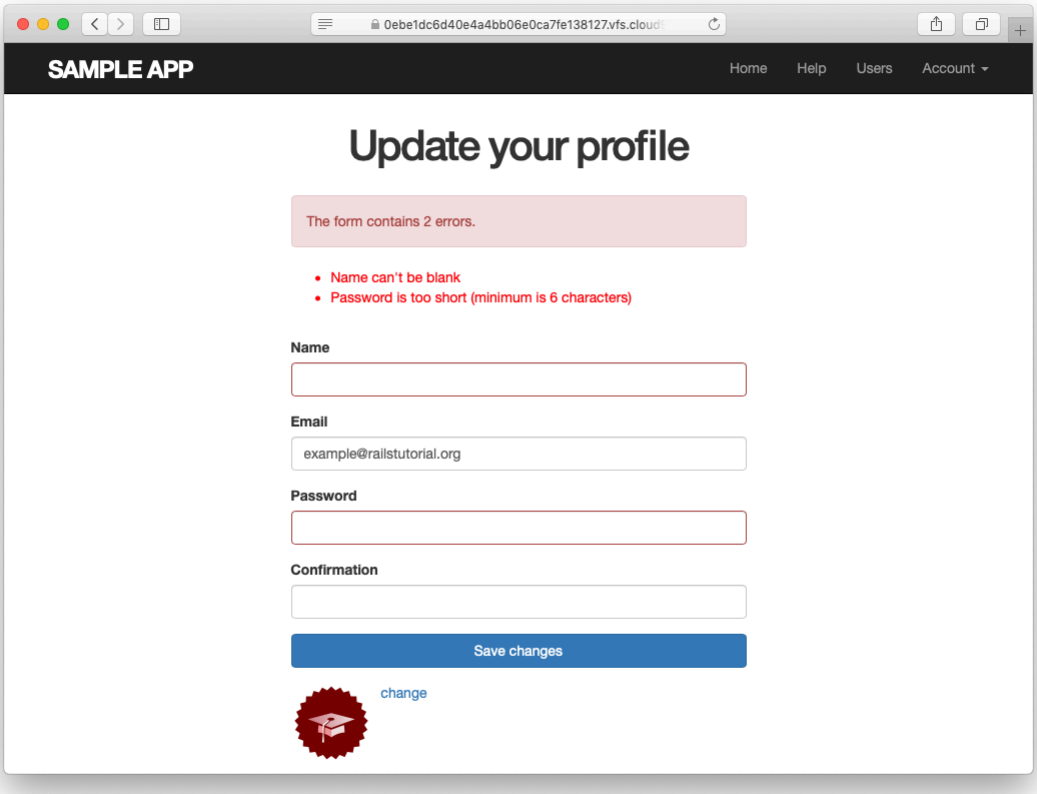


Figure 10.3: Error messages from submitting the update form.

10.1.3 Testing unsuccessful edits

We left Section 10.1.2 with a working edit form. Following the testing guidelines from Box 3.3, we'll now write an integration test to catch any regressions. Our first step is to generate an integration test as usual:

```
$ rails generate integration_test users_edit
      invoke  test_unit
      create    test/integration/users_edit_test.rb
```

Then we'll write a simple test of an unsuccessful edit, as shown in Listing 10.9. The test in Listing 10.9 checks for the correct behavior by verifying that the edit template is rendered after getting the edit page and re-rendered upon submission of invalid information. Note the use of the **patch** method to issue a PATCH request, which follows the same pattern as **get**, **post**, and **delete**.

Listing 10.9: A test for an unsuccessful edit. GREEN
test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    patch user_path(@user), params: { user: { name:  "",
                                              email: "foo@invalid",
                                              password:              "foo",
                                              password_confirmation: "bar" } }

    assert_template 'users/edit'
  end
end
```

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

Listing 10.10: 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. Add a line in Listing 10.9 to test for the correct *number* of error messages. *Hint*: Use an `assert_select` (Table 5.2) that tests for a `div` with class `alert` containing the text "The form contains 4 errors."

10.1.4 Successful edits (with TDD)

Now it's time to get the edit form to work. Editing the profile images is already functional since we've outsourced image upload to the Gravatar website: we can edit a Gravatar by clicking on the "change" link in Figure 10.2, which opens the Gravatar site in a new tab (due to `target="_blank"` in Listing 10.2), as shown in Figure 10.4. Let's get the rest of the user edit functionality working as well.

As you get more comfortable with testing, you might find that it's useful to write integration tests before writing the application code instead of after. In this context, such tests are sometimes known as *acceptance tests*, since they determine when a particular feature should be accepted as complete. To see how this works, we'll complete the user edit feature using test-driven development.

We'll test for the correct behavior of updating users by writing a test similar to the one shown in Listing 10.9, only this time we'll submit valid information. Then we'll check for a nonempty flash message and a successful redirect to the profile page, while also verifying that the user's information correctly changed in the database. The result appears in Listing 10.11. Note that the password and confirmation in Listing 10.11 are blank, which is convenient for

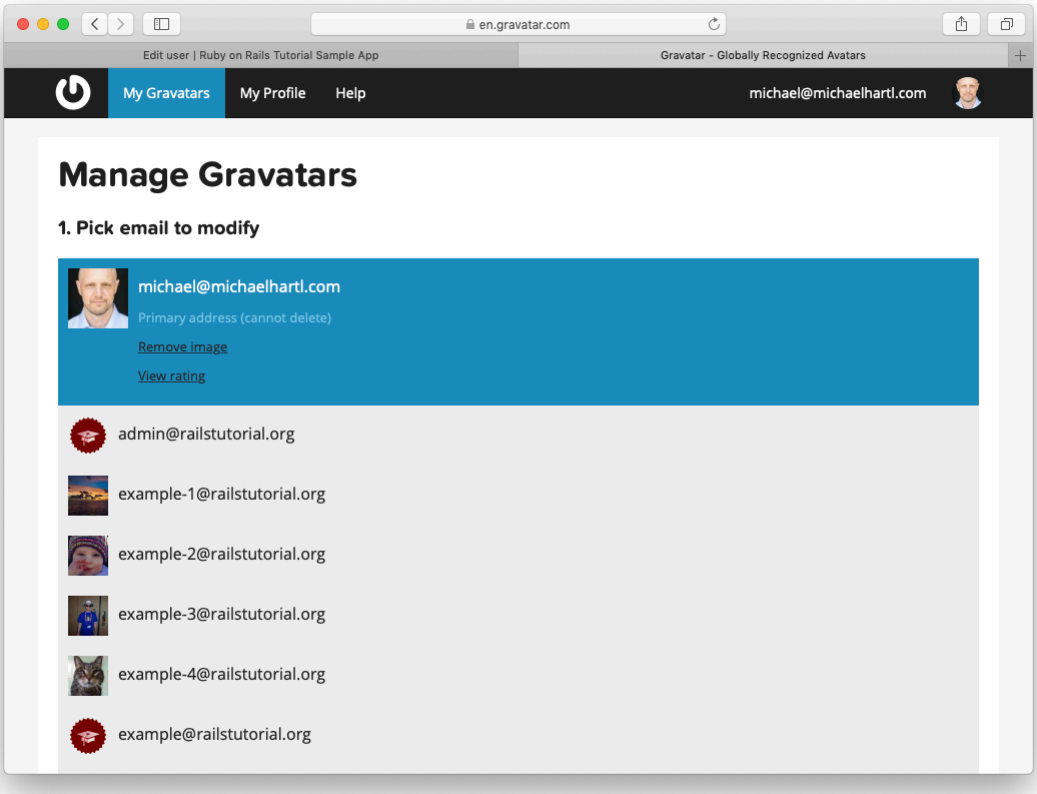


Figure 10.4: The Gravatar image-editing interface.

users who don't want to update their passwords every time they update their names or email addresses. Note also the use of **`@user.reload`** (first seen in Section 6.1.5) to reload the user's values from the database and confirm that they were successfully updated. (This is the kind of detail you could easily forget initially, which is why acceptance testing (and TDD generally) require a certain level of experience to be effective.)

Listing 10.11: A test of a successful edit. RED
test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "successful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    name  = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), params: { user: { name:  name,
                                              email: email,
                                              password:              "",
                                              password_confirmation: "" } }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name,  @user.name
    assert_equal email, @user.email
  end
end
```

The **`update`** action needed to get the tests in Listing 10.11 to pass is similar to the final form of the **`create`** action (Listing 8.32), as seen in Listing 10.12.

Listing 10.12: The user **`update`** action. RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def update
    @user = User.find(params[:id])
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
end
```

As indicated in the caption to Listing 10.12, the test suite is still RED, which is the result of the password length validation (Listing 6.43) failing due to the empty password and confirmation in Listing 10.11. To get the tests to GREEN, we need to make an exception to the password validation if the password is empty. We can do this by passing the **`allow_nil: true`** option to **`validates`**, as seen in Listing 10.13.

Listing 10.13: Allowing empty passwords on update. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }, allow_nil: true
  .
  .
  .
end
```

In case you're worried that Listing 10.13 might allow new users to sign up with empty passwords, recall from Section 6.3.3 that **`has_secure_password`**

includes a separate presence validation that specifically catches `nil` passwords. (Because `nil` passwords now bypass the main presence validation but are still caught by `has_secure_password`, this also fixes the duplicate error message mentioned in Section 7.3.3.)

With the code in this section, the user edit page should be working (Figure 10.5), as you can double-check by re-running the test suite, which should now be GREEN:

Listing 10.14: 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. Double-check that you can now make edits by making a few changes on the development version of the application.

2. What happens when you change the email address to one without an associated Gravatar?

10.2 Authorization

In the context of web applications, *authentication* allows us to identify users of our site, while *authorization* lets us control what they can do. One nice effect of building the authentication machinery in Chapter 8 is that we are now in a position to implement authorization as well.

Although the edit and update actions from Section 10.1 are functionally complete, they suffer from a ridiculous security flaw: they allow anyone (even non-logged-in users) to access either action and update the information for any

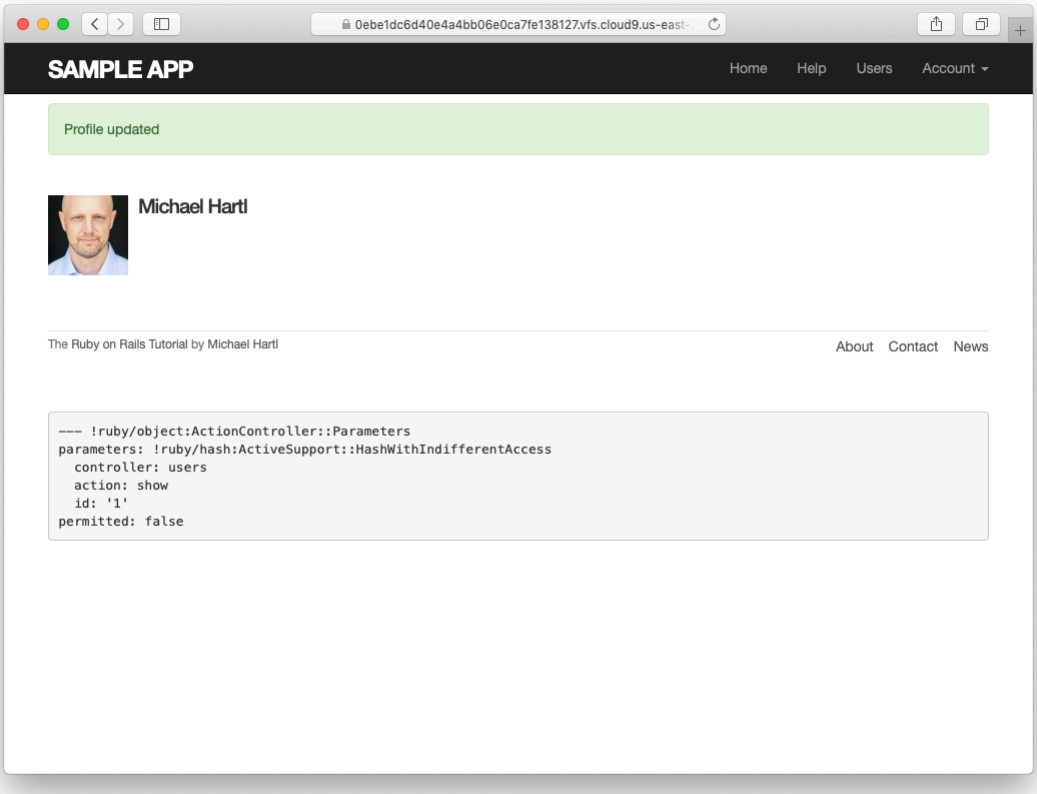


Figure 10.5: The result of a successful edit.

user. In this section, we'll implement a security model that requires users to be logged in and prevents them from updating any information other than their own.

In Section 10.2.1, we'll handle the case of non-logged-in users who try to access a protected page to which they might normally have access. Because this could easily happen in the normal course of using the application, such users will be forwarded to the login page with a helpful message, as mocked up in Figure 10.6. On the other hand, users who try to access a page for which they would never be authorized (such as a logged-in user trying to access a different user's edit page) will be redirected to the root URL (Section 10.2.2).

10.2.1 Requiring logged-in users

To implement the forwarding behavior shown in Figure 10.6, we'll use a *before filter* in the Users controller. Before filters use the **`before_action`** command to arrange for a particular method to be called before the given actions.[4] To require users to be logged in, we define a **`logged_in_user`** method and invoke it using **`before_action :logged_in_user`**, as shown in Listing 10.15.

Listing 10.15: Adding a **`logged_in_user`** before filter. RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Before filters

    # Confirms a logged-in user.
```

[4] The command for before filters used to be called **`before_filter`**, but the Rails core team decided to rename it to emphasize that the filter takes place before particular controller actions.

Figure 10.6: A mockup of the result of visiting a protected page

```
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end
end
```

By default, before filters apply to *every* action in a controller, so here we restrict the filter to act only on the **`:edit`** and **`:update`** actions by passing the appropriate **`only:`** options hash.

We can see the result of the before filter in Listing 10.15 by logging out and attempting to access the user edit page /users/1/edit, as seen in Figure 10.7.

As indicated in the caption of Listing 10.15, our test suite is currently RED:

Listing 10.16: RED

```
$ rails test
```

The reason is that the edit and update actions now require a logged-in user, but no user is logged in inside the corresponding tests.

We'll fix our test suite by logging the user in before hitting the edit or update actions. This is easy using the **`log_in_as`** helper developed in Section 9.3 (Listing 9.24), as shown in Listing 10.17.

Listing 10.17: Logging in a test user. GREEN

test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
```

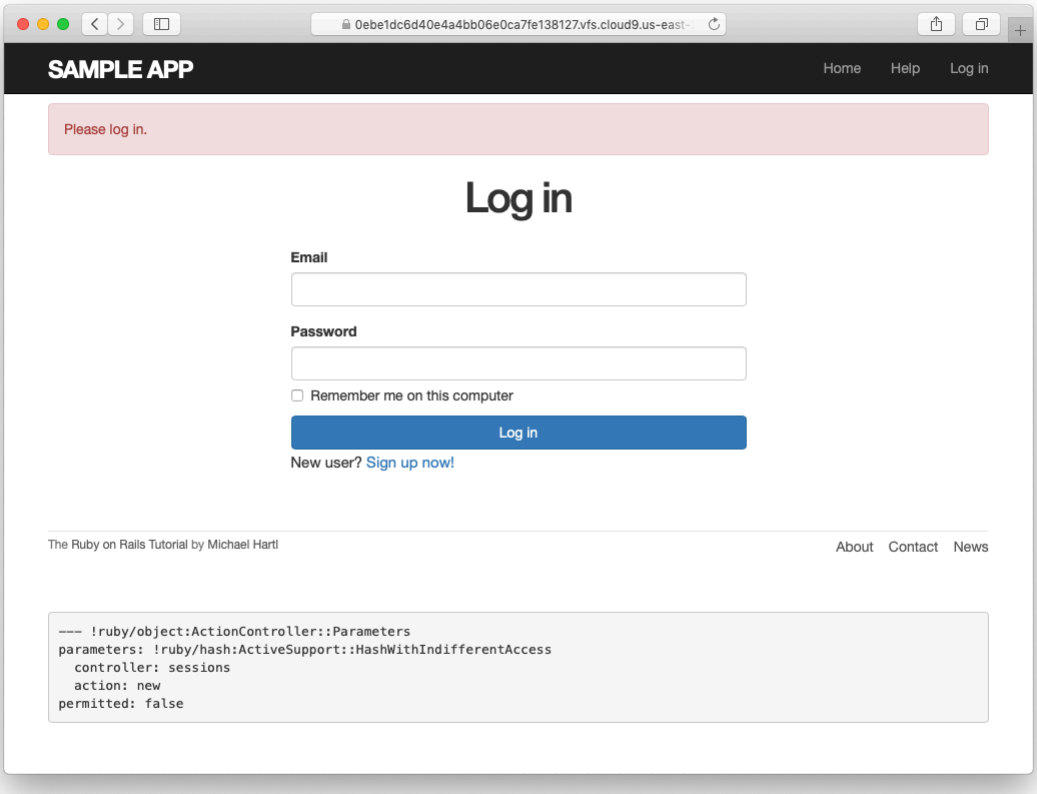


Figure 10.7: The login form after trying to access a protected page.

```
    .
    .
  end

  test "successful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
    .
    .
  end
end
```

(We could eliminate some duplication by putting the test login in the **setup** method of Listing 10.17, but in Section 10.2.3 we'll change one of the tests to visit the edit page *before* logging in, which isn't possible if the login step happens during the test setup.)

At this point, our test suite should be green:

Listing 10.18: GREEN

```
$ rails test
```

Even though our test suite is now passing, we're not finished with the before filter, because the suite is still GREEN even if we remove our security model, as you can verify by commenting it out (Listing 10.19). This is a Bad Thing—of all the regressions we'd like our test suite to catch, a massive security hole is probably #1, so the code in Listing 10.19 should definitely be RED. Let's write tests to arrange that.

Listing 10.19: Commenting out the before filter to test our security model. GREEN

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  # before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

Because the before filter operates on a per-action basis, we'll put the corresponding tests in the Users controller test. The plan is to hit the **edit** and **update** actions with the right kinds of requests and verify that the flash is set and that the user is redirected to the login path. From Table 7.1, we see that the proper requests are GET and PATCH, respectively, which means using the **get** and **patch** methods inside the tests. The results (which include adding a **setup** method to define an **@user** variable) appear in Listing 10.20.

Listing 10.20: Testing that **edit** and **update** are protected. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "should redirect edit when not logged in" do
    get edit_user_path(@user)
    assert_not flash.empty?
    assert_redirected_to login_url
  end

  test "should redirect update when not logged in" do
    patch user_path(@user), params: { user: { name: @user.name,
                                              email: @user.email } }
    assert_not flash.empty?
    assert_redirected_to login_url
  end
end
```

Note that the second test shown in Listing 10.20 involves using the **patch** method to send a PATCH request to **user_path(@user)**. According to Table 7.1, such a request gets routed to the **update** action in the Users controller, as required.

The test suite should now be RED, as required. To get it to GREEN, just uncomment the before filter (Listing 10.21).

Listing 10.21: Uncommenting the before filter. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

With that, our test suite should be GREEN:

Listing 10.22: GREEN

```
$ rails test
```

Any accidental exposure of the edit methods to unauthorized users will now be caught immediately by our test suite.

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. As noted above, by default before filters apply to every action in a controller, which in our cases is an error (requiring, e.g., that users log in to hit the signup page, which is absurd). By commenting out the `only:` hash in Listing 10.15, confirm that the test suite catches this error.

10.2.2 Requiring the right user

Of course, requiring users to log in isn't quite enough; users should only be allowed to edit their *own* information. As we saw in Section 10.2.1, it's easy to have a test suite that misses an essential security flaw, so we'll proceed using test-driven development to be sure our code implements the security model

correctly. To do this, we'll add tests to the Users controller test to complement the ones shown in Listing 10.20.

In order to make sure users can't edit other users' information, we need to be able to log in as a second user. This means adding a second user to our users fixture file, as shown in Listing 10.23.

Listing 10.23: Adding a second user to the fixture file.
test/fixtures/users.yml

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>
```

By using the `log_in_as` method defined in Listing 9.24, we can test the `edit` and `update` actions as in Listing 10.24. Note that we expect to redirect users to the root path instead of the login path because a user trying to edit a different user would already be logged in.

Listing 10.24: Tests for trying to edit as the wrong user. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user       = users(:michael)
    @other_user = users(:archer)
  end
  .
  .
  .
  test "should redirect edit when logged in as wrong user" do
    log_in_as(@other_user)
    get edit_user_path(@user)
    assert flash.empty?
```

```
    assert_redirected_to root_url
  end

  test "should redirect update when logged in as wrong user" do
    log_in_as(@other_user)
    patch user_path(@user), params: { user: { name: @user.name,
                                              email: @user.email } }
    assert flash.empty?
    assert_redirected_to root_url
  end
end
```

To redirect users trying to edit another user's profile, we'll add a second method called **`correct_user`**, together with a before filter to call it (Listing 10.25). Note that the **`correct_user`** before filter defines the **`@user`** variable, so Listing 10.25 also shows that we can eliminate the **`@user`** assignments in the **`edit`** and **`update`** actions.

Listing 10.25: A before filter to protect the edit/update pages. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
```

```
    end

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless @user == current_user
    end
end
```

At this point, our test suite should be GREEN:

Listing 10.26: GREEN

```
$ rails test
```

As a final refactoring, we'll adopt a common convention and define a **`current_user?`** boolean method for use in the **`correct_user`** before filter. We'll use this method to replace code like

```
unless @user == current_user
```

with the more expressive

```
unless current_user?(@user)
```

The result appears in Listing 10.27. Note that by writing **`user && user == current_user`**, we also catch the edge case where **`user`** is **`nil`**.[5]

[5] Thanks to reader Andrew Moor for pointing this out. Andrew also noted that we can use the safe navigation operator introduced in Section 8.2.4 to write this as **`user&. == current_user`**.

Listing 10.27: The `current_user?` method.
app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.encrypted[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.encrypted[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end

  # Returns true if the given user is the current user.
  def current_user?(user)
    user && user == current_user
  end
  .
  .
  .
end
```

Replacing the direct comparison with the boolean method gives the code shown in Listing 10.28.

Listing 10.28: The final `correct_user` before filter. GREEN
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
```

```
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end

  def update
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

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. Why is it important to protect both the **`edit`** and **`update`** actions?
2. Which action could you more easily test in a browser?

10.2.3 Friendly forwarding

Our site authorization is complete as written, but there is one minor blemish: when users try to access a protected page, they are currently redirected to their profile pages regardless of where they were trying to go. In other words, if a non-logged-in user tries to visit the edit page, after logging in the user will be redirected to /users/1 instead of /users/1/edit. It would be much friendlier to redirect them to their intended destination instead.

The application code will turn out to be relatively complicated, but we can write a ridiculously simple test for friendly forwarding just by reversing the order of logging in and visiting the edit page in Listing 10.17. As seen in Listing 10.29, the resulting test tries to visit the edit page, then logs in, and then checks that the user is redirected to the *edit* page instead of the default profile page. (Listing 10.29 also removes the test for rendering the edit template since that's no longer the expected behavior.)

Listing 10.29: A test for friendly forwarding. RED
test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "successful edit with friendly forwarding" do
    get edit_user_path(@user)
    log_in_as(@user)
    assert_redirected_to edit_user_url(@user)
    name  = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), params: { user: { name:  name,
```

```
                                                email: email,
                                                password:              "",
                                                password_confirmation: "" } }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name,  @user.name
    assert_equal email, @user.email
  end
end
```

Now that we have a failing test, we're ready to implement friendly forwarding.[6] In order to forward users to their intended destination, we need to store the location of the requested page somewhere, and then redirect to that location instead of to the default. We accomplish this with a pair of methods, **`store_location`** and **`redirect_back_or`**, both defined in the Sessions helper (Listing 10.30).

Listing 10.30: Code to implement friendly forwarding. RED
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Redirects to stored location (or to the default).
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
  end

  # Stores the URL trying to be accessed.
  def store_location
    session[:forwarding_url] = request.original_url if request.get?
  end
end
```

Here the storage mechanism for the forwarding URL is the same **`session`** facility we used in Section 8.2.1 to log the user in. Listing 10.30 also uses the **`request`** object (via **`request.original_url`**) to get the URL of the requested page.

[6] The code in this section is adapted from the Clearance gem by thoughtbot.

The **store_location** method in Listing 10.30 puts the requested URL in the **session** variable under the key **:forwarding_url**, but only for a **GET** request. This prevents storing the forwarding URL if a user, say, submits a form when not logged in (which is an edge case but could happen if, e.g., a user deleted the session cookies by hand before submitting the form). In such a case, the resulting redirect would issue a **GET** request to a URL expecting **POST**, **PATCH**, or **DELETE**, thereby causing an error. Including **if request.get?** prevents this from happening.[7]

To make use of **store_location**, we need to add it to the **logged_in_user** before filter, as shown in Listing 10.31.

Listing 10.31: Adding **store_location** to the logged-in user before filter.
RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]
  .
  .
  .
  def edit
  end
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
```

[7] Thanks to reader Yoel Adler for pointing out this subtle issue, and for discovering the solution.

```
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

To implement the forwarding itself, we use the **redirect_back_or** method to redirect to the requested URL if it exists, or some default URL otherwise, which we add to the Sessions controller **create** action to redirect after successful login (Listing 10.32). The **redirect_back_or** method uses the or operator **||** through

```
session[:forwarding_url] || default
```

This evaluates to **session[:forwarding_url]** unless it's **nil**, in which case it evaluates to the given default URL. Note that Listing 10.30 is careful to remove the forwarding URL (via **session.delete(:forwarding_url)**); otherwise, subsequent login attempts would forward to the protected page until the user closed their browser. (Testing for this is left as an exercise (Section 10.2.3).) Also note that the session deletion occurs even though the line with the redirect appears first; redirects don't happen until an explicit **return** or the end of the method, so any code appearing after the redirect is still executed.

Listing 10.32: The Sessions **create** action with friendly forwarding. GREEN
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
```

```
      params[:session][:remember_me] == '1' ? remember(user) : forget(user)
      redirect_back_or user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end
  .
  .
  .
end
```

With that, the friendly forwarding integration test in Listing 10.29 should pass, and the basic user authentication and page protection implementation is complete. As usual, it's a good idea to verify that the test suite is GREEN before proceeding:

Listing 10.33: 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. Write a test to make sure that friendly forwarding only forwards to the given URL the first time. On subsequent login attempts, the forwarding URL should revert to the default (i.e., the profile page). *Hint*: Add to the test in Listing 10.29 by checking for the right value of **`session[:forwarding_url]`**.

2. Put a **`debugger`** (Section 7.1.3) in the Sessions controller's **`new`** action, then log out and try to visit /users/1/edit. Confirm in the debugger that the value of **`session[:forwarding_url]`** is correct. What is the value of **`request.get?`** for the **`new`** action? (Sometimes the terminal can freeze

up or act strangely when you're using the debugger; use your technical sophistication (Box 1.2) to resolve any issues.)

10.3 Showing all users

In this section, we'll add the penultimate user action, the **index** action, which is designed to display *all* the users instead of just one. Along the way, we'll learn how to seed the database with sample users and how to *paginate* the user output so that the index page can scale up to display a potentially large number of users. A mockup of the result—users, pagination links, and a "Users" navigation link—appears in Figure 10.8.[8] In Section 10.4, we'll add an administrative interface to the users index so that users can also be destroyed.

10.3.1 Users index

To get started with the users index, we'll first implement a security model. Although we'll keep individual user **show** pages visible to all site visitors, the user **index** will be restricted to logged-in users so that there's a limit to how much unregistered users can see by default.[9]

To protect the **index** page from unauthorized access, we'll first add a short test to verify that the **index** action is redirected properly (Listing 10.34).

Listing 10.34: Testing the **index** action redirect. RED
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  def setup
    @user       = users(:michael)
    @other_user = users(:archer)
  end
```

[8] Image retrieved from https://www.flickr.com/photos/glasgows/338937124/ on 2014-08-25. Copyright © 2008 by M&R Glasgow and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

[9] This is the same authorization model used by Twitter.

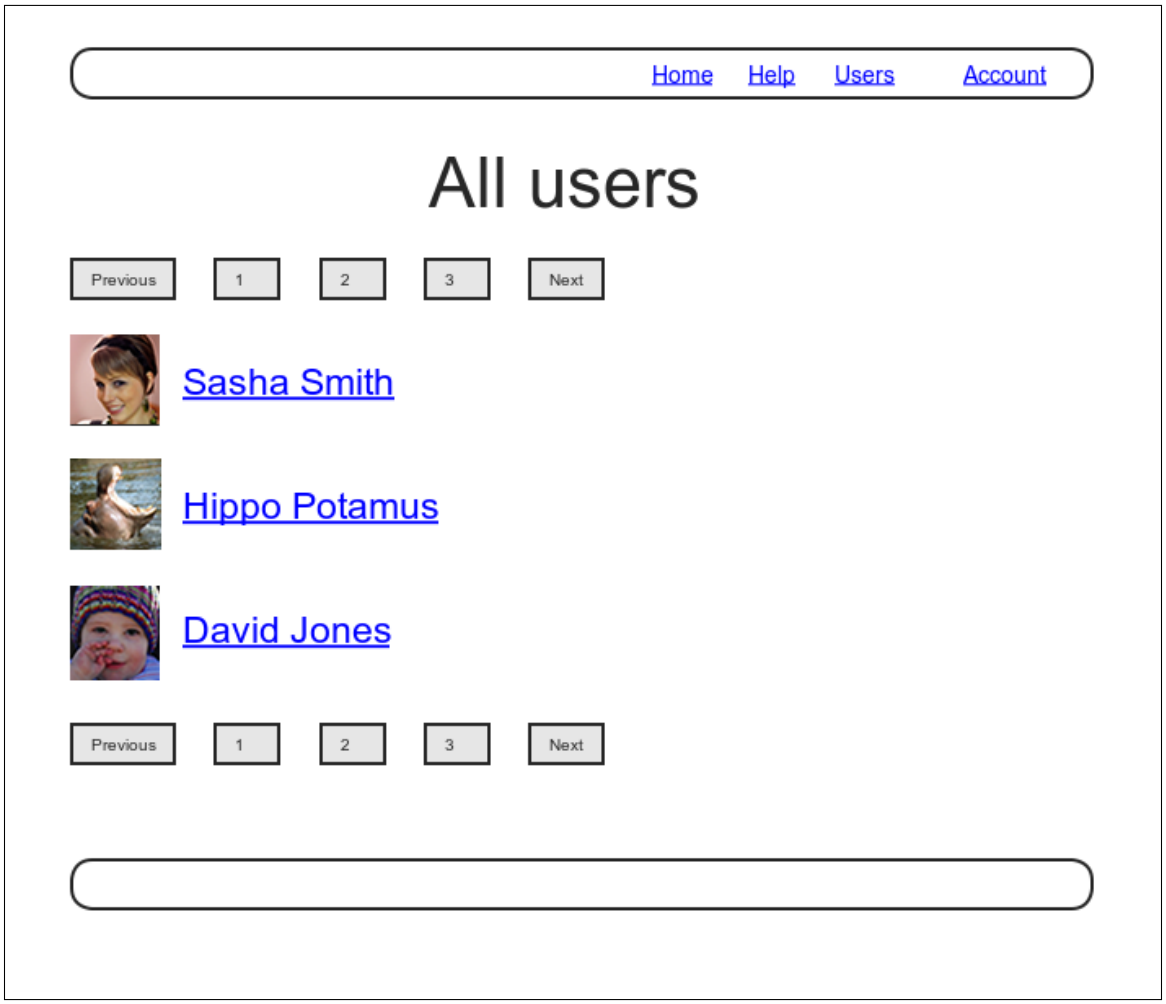


Figure 10.8: A mockup of the users index page.