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 in 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` request 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'
```

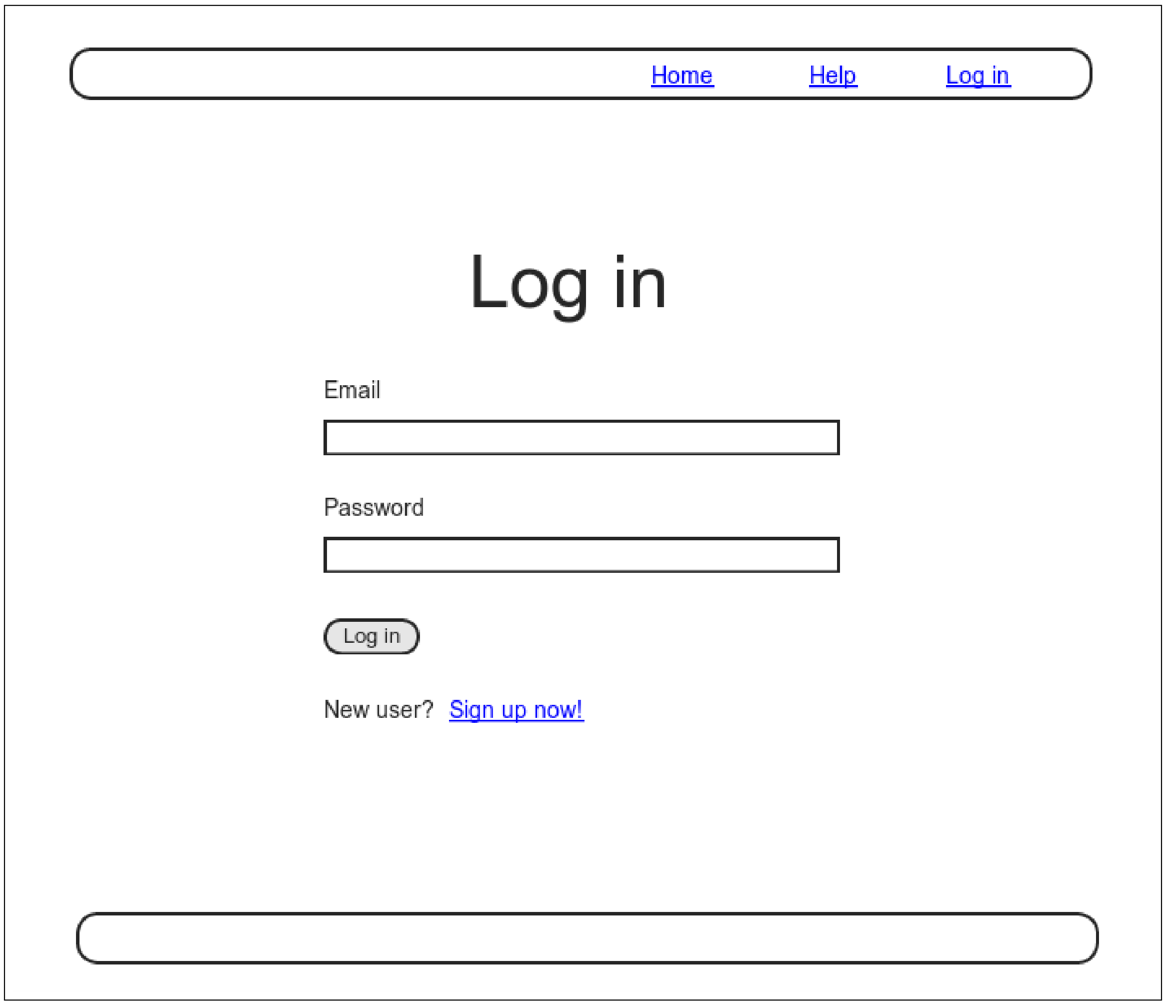


Figure 8.1: A mockup of the login form.

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.

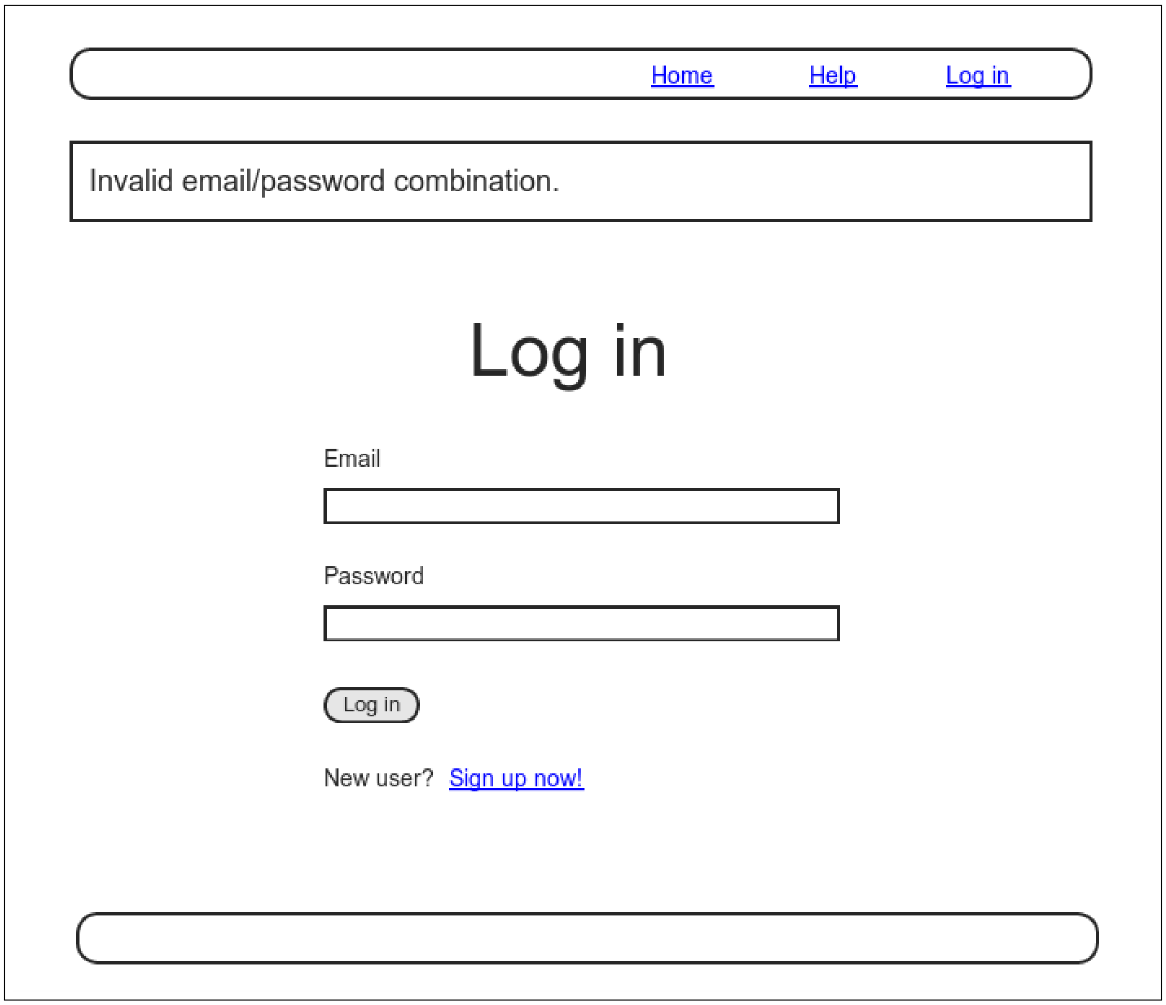


Figure 8.2: A mockup of login failure.

Listing 8.4: Code for the login form.
app/views/sessions/new.html.erb

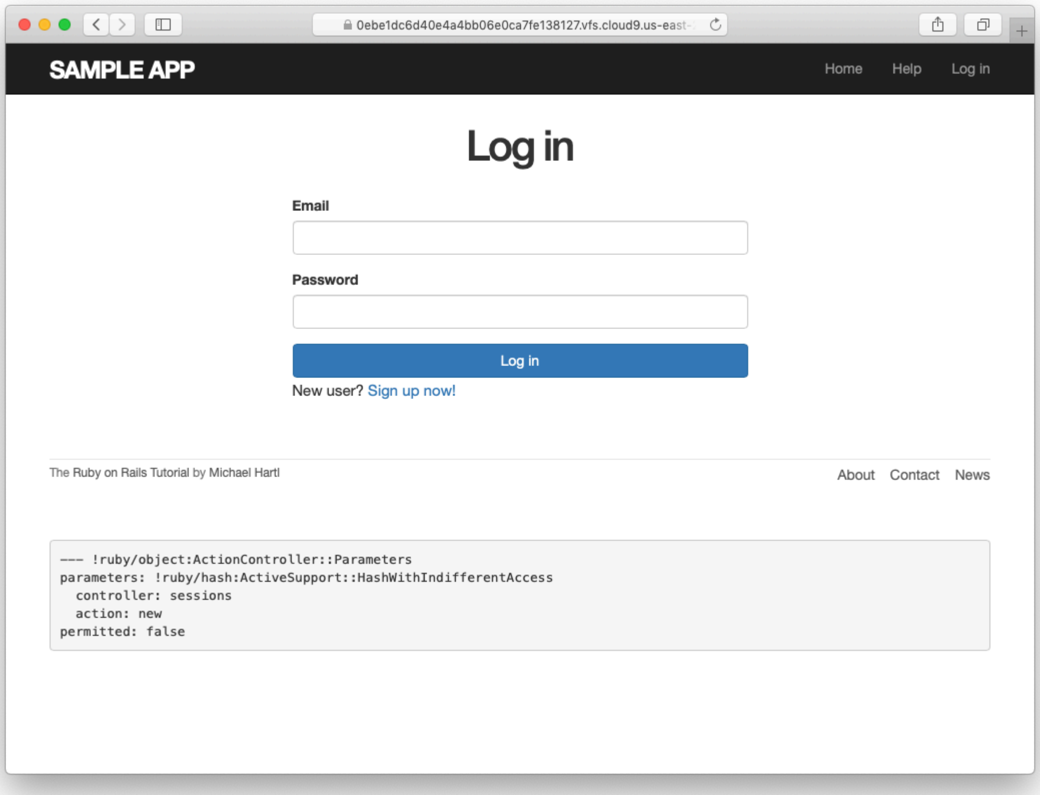
```
<% provide(:title, "Log in") %>
<h1>Log in</h1>

<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>
```


0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-
SAMPLE APP
Home
Help
Log in
Log in
Email
Password
Log in
New user? Sign up now!
The Ruby on Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  controller: sessions
  action: new
permitted: false
```

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.

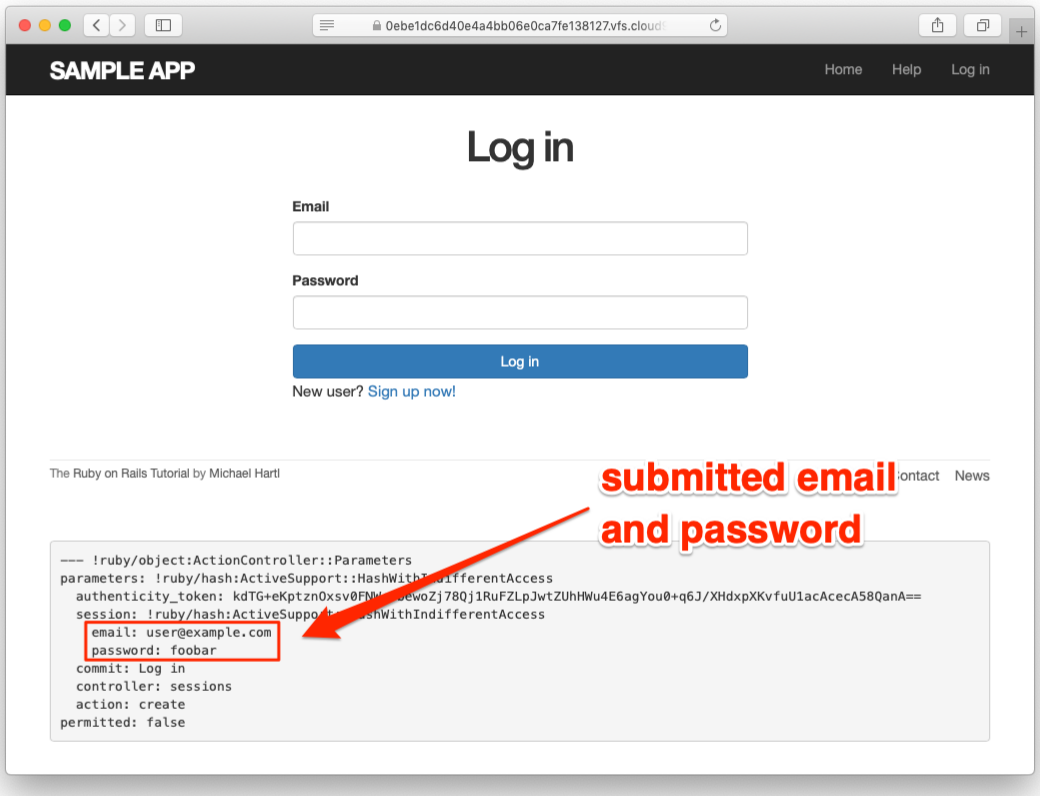
8.1.3 Finding and authenticating a user

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

```ruby
class SessionsController < ApplicationController

  def new
```

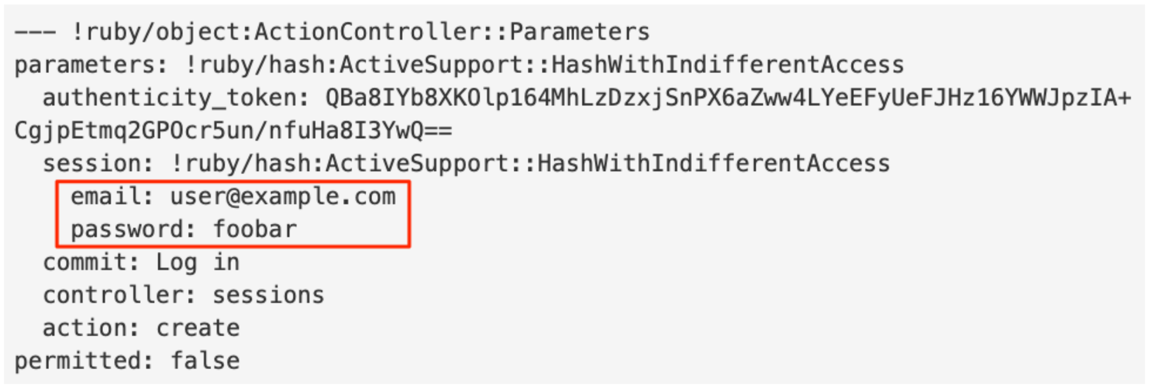

0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud
SAMPLE APP
Home
Help
Log in
Log in
Email
Password
Log in
New user? Sign up now!
The Ruby on Rails Tutorial by Michael Hartl
Contact
News
submitted email and password

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  authenticity_token: kdTG+eKptznOxsv0FNW[illegible]ewoZj78Qj1RuFZLpJwtZUhHWu4E6agYou0+q6J/XHdxpXKvfuU1acAcecA58QanA==
  session: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
    email: user@example.com
    password: foobar
  commit: Log in
  controller: sessions
  action: create
permitted: false
```

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

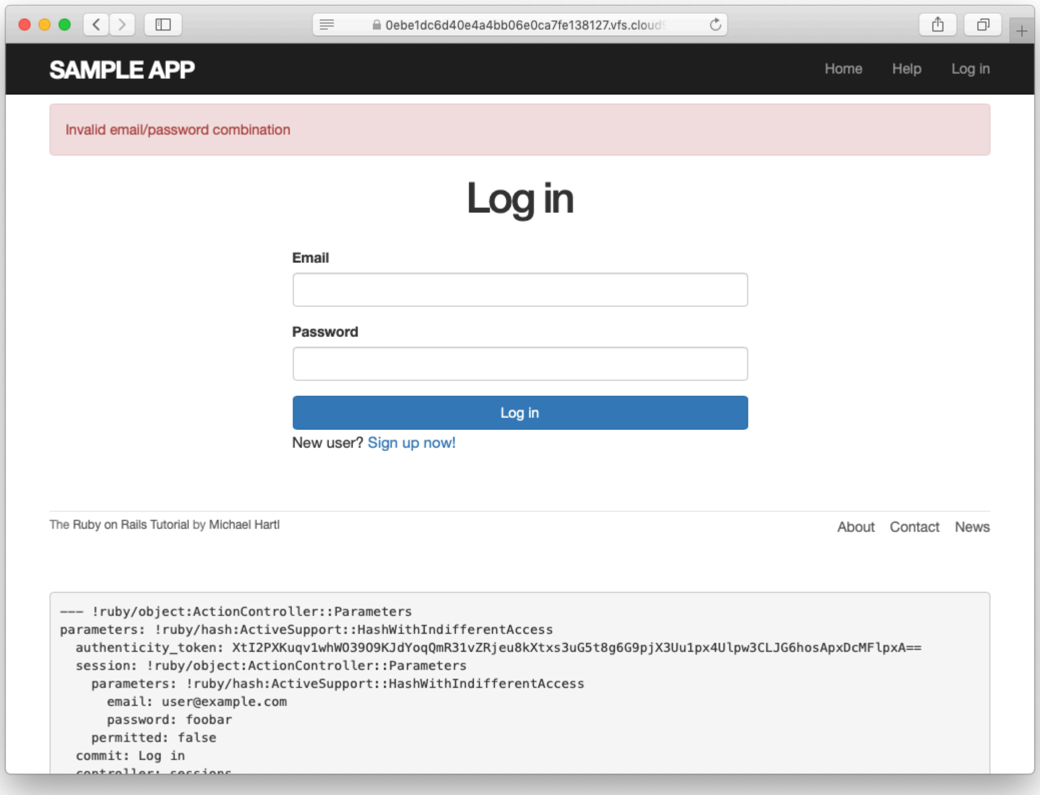
Recall from Section 7.3.3 that we displayed signup errors using the User model error messages. These errors are associated with a particular Active Record object, but this strategy won't work here because the session isn't an Active Record model. Instead, we'll put a message in the flash to be displayed 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

```ruby
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.



Invalid email/password combination

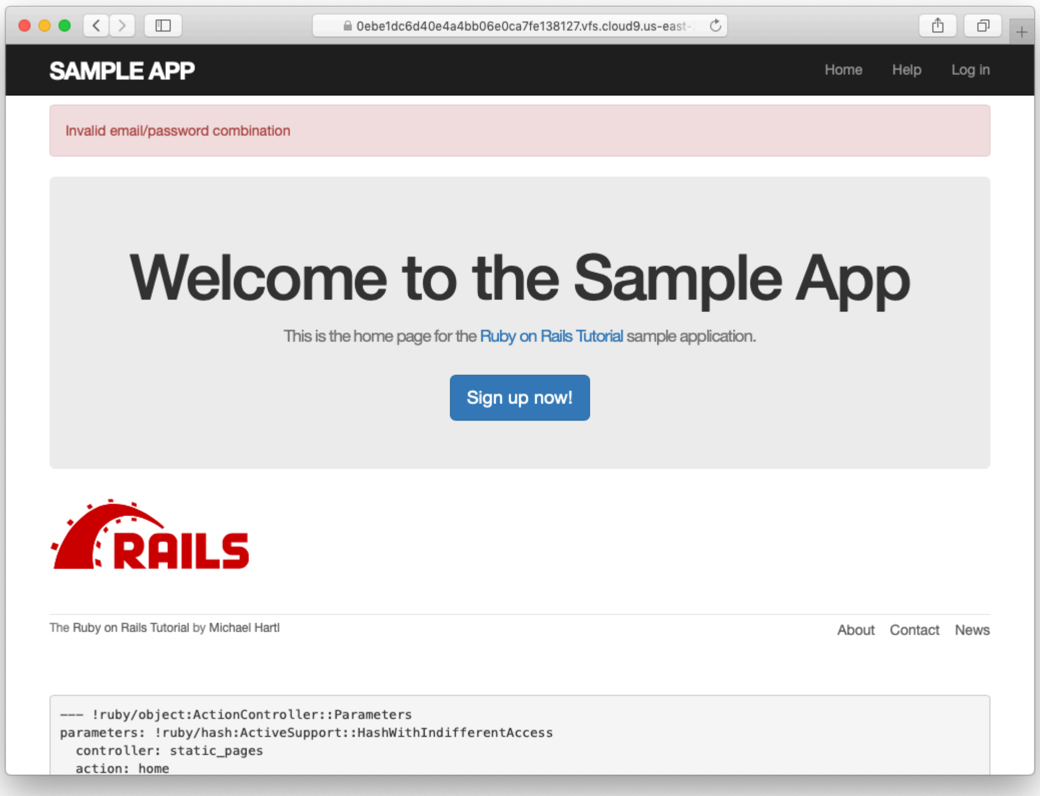
Log in

New user? Sign up now!

The Ruby on Rails Tutorial by Michael Hartl

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  authenticity_token: XtI2PXKuqv1whWO39O9KJdYoqQmR31vZRjeu8kXtxs3uG5t8g6G9pjX3Uu1px4Ulpw3CLJG6hosApxDcMFlpxA==
  session: !ruby/object:ActionController::Parameters
    parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
      email: user@example.com
      password: foobar
    permitted: false
  commit: Log in
  controller: sessions
```

Figure 8.6: The flash message for a failed login.



Invalid email/password combination

Welcome to the Sample App

This is the home page for the Ruby on Rails Tutorial sample application.

The Ruby on Rails Tutorial by Michael Hartl

```
--- !ruby/object:ActionController::Parameters
parameters: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  controller: static_pages
  action: home
```

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.