Please see the post **Do not buy the print version of the Ruby on Rails Tutorial (yet)**

skip to content | view as single page

Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

Contents

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I’ve worked my way through many Rails books, this is the one that finally made me “get” it. Everything is done very much “the Rails way”—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it’s like to do a real-world project. The tutorial’s code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you’ll get the ultimate benefit.

Enjoy!

Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor Aurelius Prochazka. I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

About the author

Michael Hartl is the author of the *Ruby on Rails Tutorial*, the leading introduction to web development with Ruby on Rails. His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a Ruby Hero Award for his contributions to the Ruby community. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Copyright and license

Chapter 7
Sign up

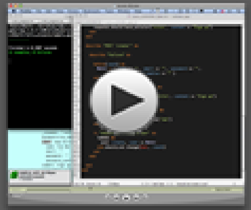
Now that we have a working User model, it's time to add an ability few websites can live without: letting users sign up for the site. We'll use an HTML *form* to submit user signup information to our application in Section 7.2, which will then be used to create a new user and save its attributes to the database in 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). As usual, we'll write tests as we develop, extending the theme of using RSpec and Capybara to write succinct and expressive integration tests.

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 6.3.5, we created a User record by hand using the Rails console. If you skipped that section, you should go there now and complete it before proceeding.

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

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

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 has our first 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 11.

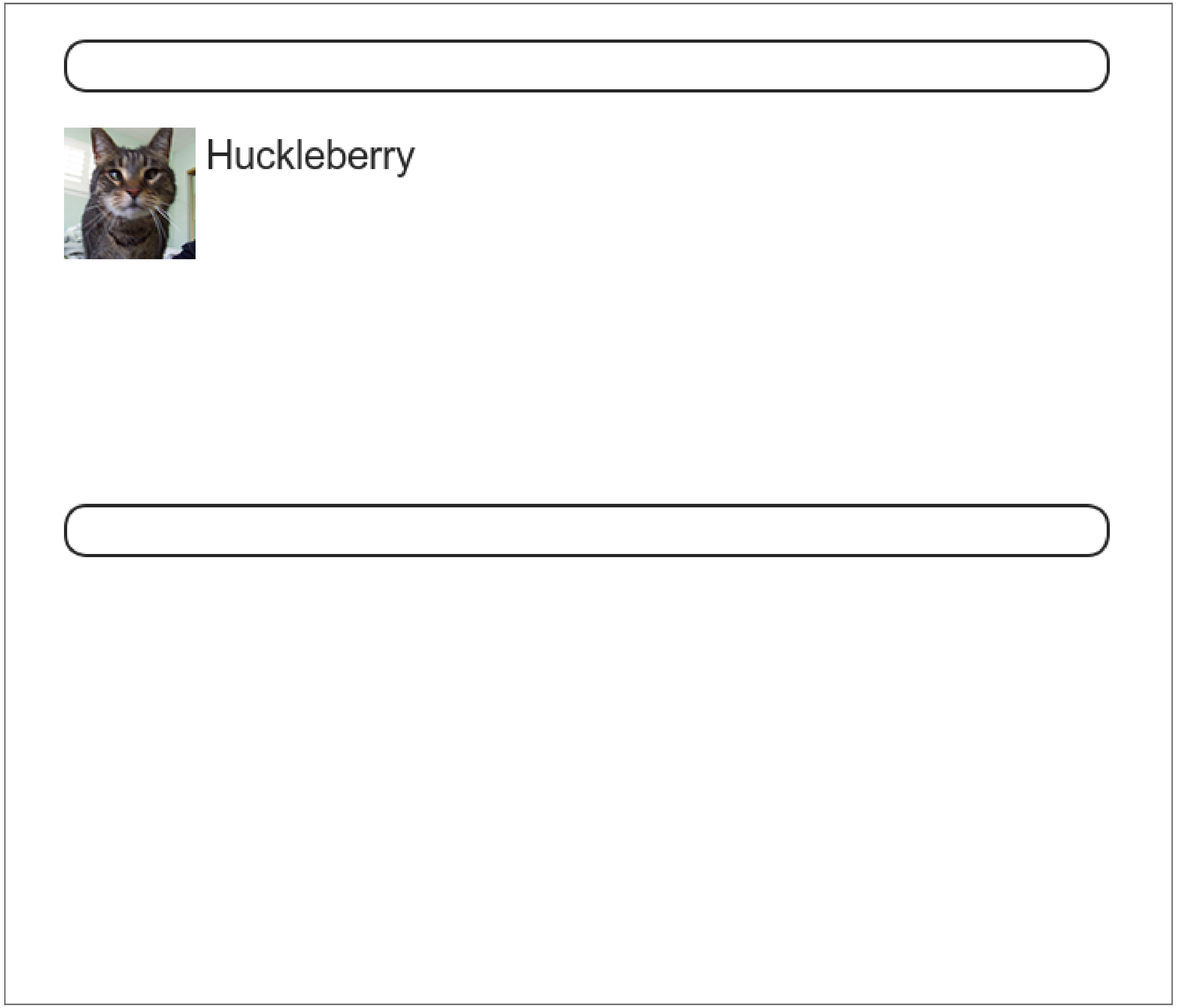


Figure 7.1: A mockup of the user profile made in this section. (full size)

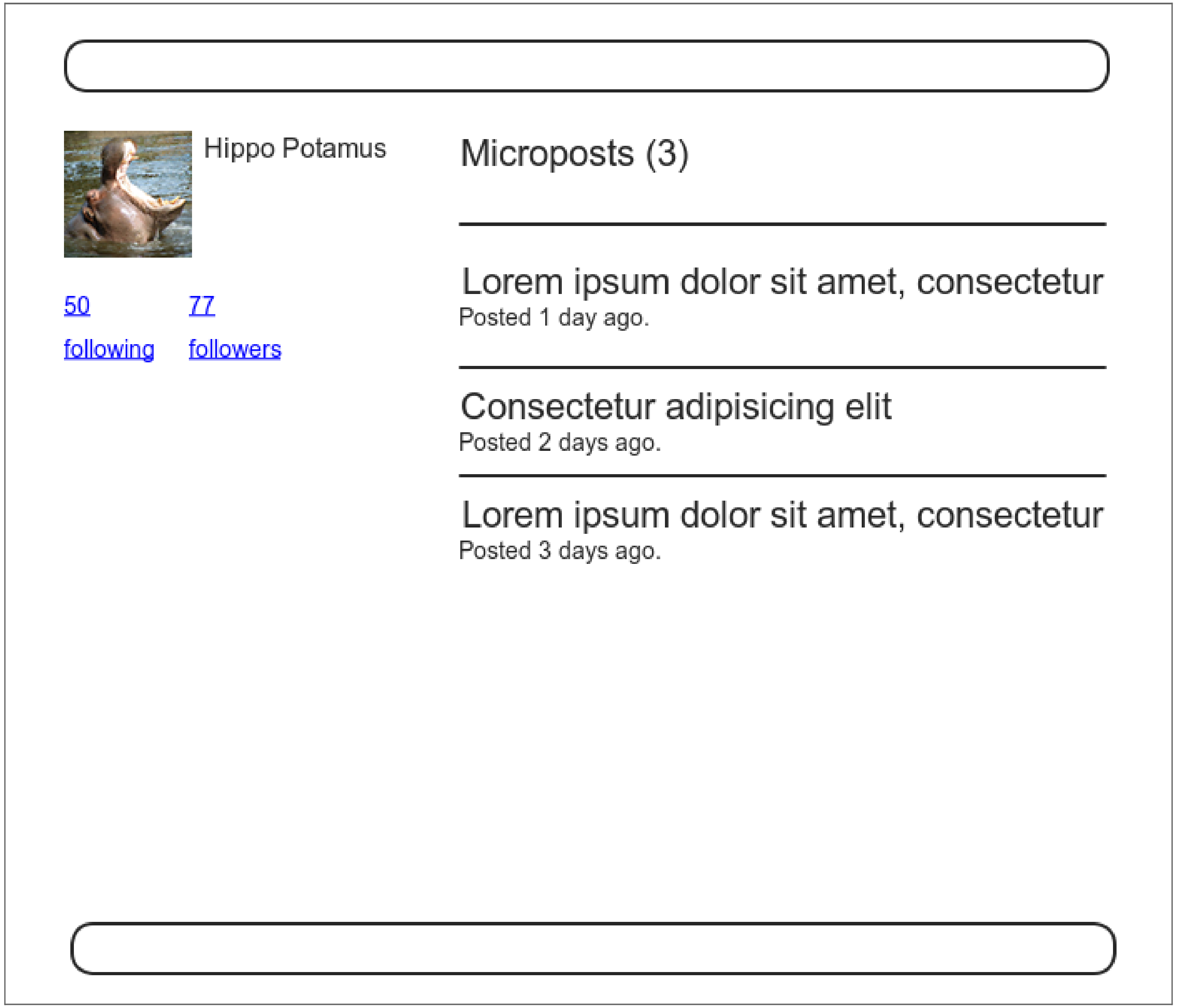

Hippo Potamus
50
following
77
followers
Microposts (3)
Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago.
Consectetur adipisicing elit
Posted 2 days ago.
Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago.

Figure 7.2: A mockup of our best guess at the final profile page. (full size)

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

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.css.scss`

```
@import "bootstrap";
```

```
/* mixins, variables, etc. */

$grayMediumLight: #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;
}
```

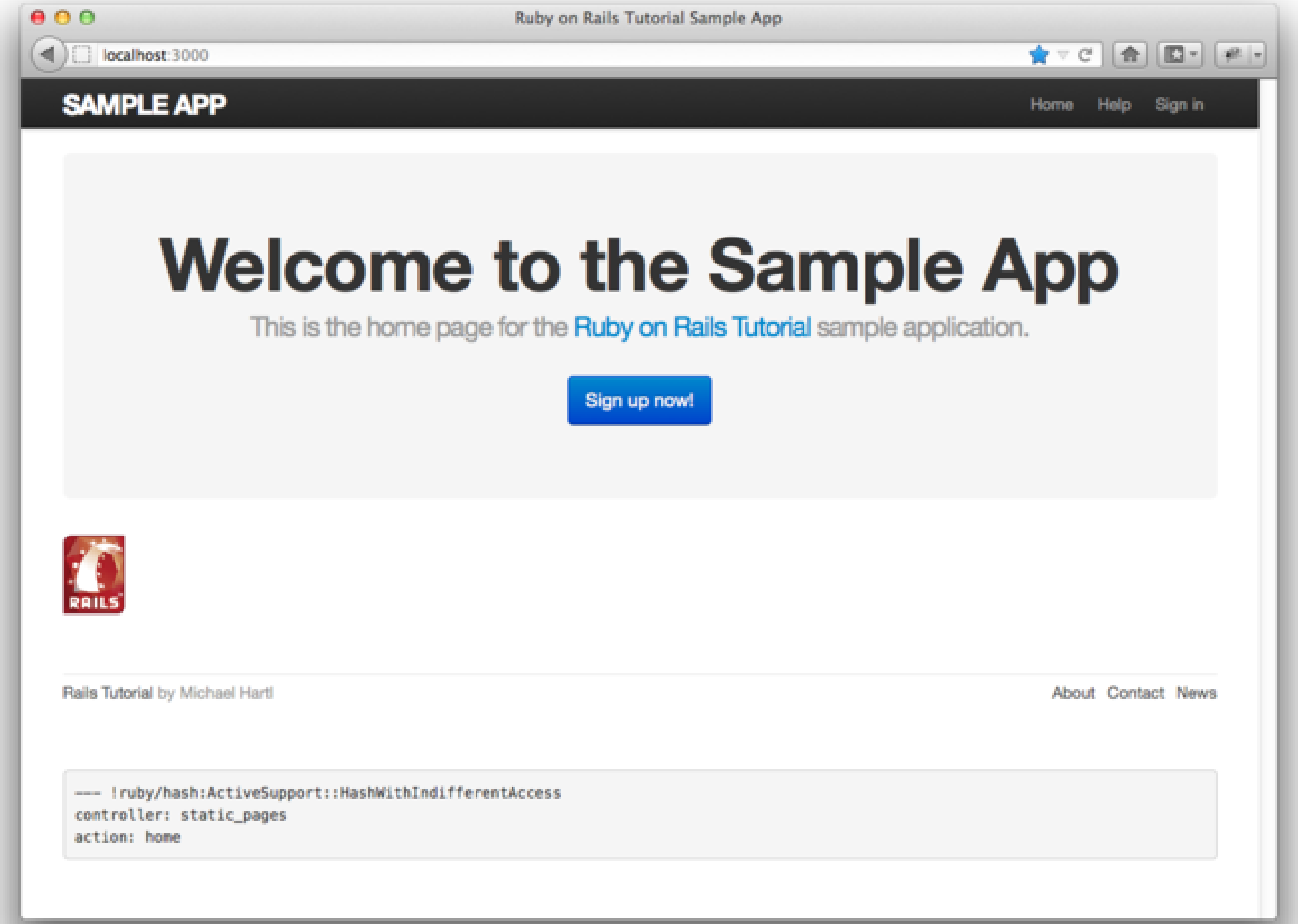
This introduces the Sass *mixin* facility, in this case called **`box_sizing`**. A mixin allows a group of CSS rules to be packaged up and used for multiple elements, converting

```
.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 7.2.2. The result in the case of the debug box is shown in Figure 7.3.



Welcome to the Sample App

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

Rails Tutorial by Michael Hartl

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: static_pages
action: home
```

Figure 7.3: The sample application Home page (/) with debug information. (full size)

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

```
---
```

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

This is a YAML[3] 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

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).[4] 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`:

```
  $ 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 local 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 `rake db:migrate` in production:

```
$ bundle exec rake db:migrate RAILS_ENV=production
```

(I find it confusing that the console, server, and migrate commands specify non-default environments in three mutually incompatible ways, which is why I bothered showing all three.)

By the way, if you have deployed your sample app to Heroku, you can see its environment using the `heroku` command, which provides its own (remote) console:

```
$ heroku run console
Ruby console for yourapp.herokuapp.com
>> Rails.env
=> "production"
>> Rails.env.production?
=> true
```

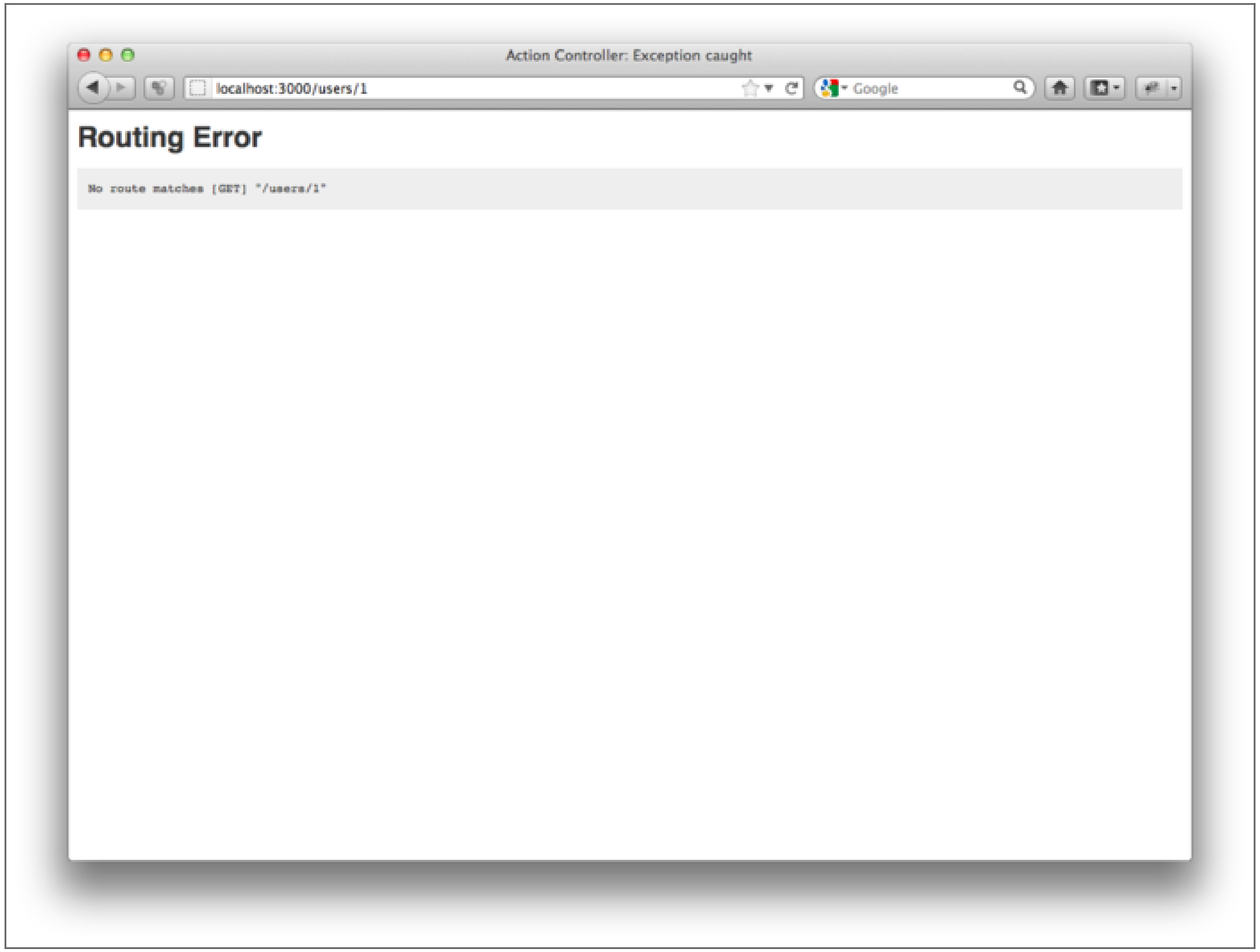
Naturally, since Heroku is a platform for production sites, it runs each application in a production environment.

7.1.2 A Users resource

At the end of Chapter 6, we created a new user in the database. As seen in Section 6.3.5, this 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`, `PUT`, 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 URI /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 URI /users/1. Unfortunately, visiting that URI right now just gives an error (Figure 7.4).



Routing Error

```
No route matches [GET] "/users/1"
```

Figure 7.4: The error page for /users/1. (full size)

We can get the REST-style URI 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`

```
SampleApp::Application.routes.draw do
  resources :users

  root to: 'static_pages#home'

  match '/signup',  to: 'users#new'
  .
  .
  .
end
```

You might have noticed that Listing 7.3 removes the line

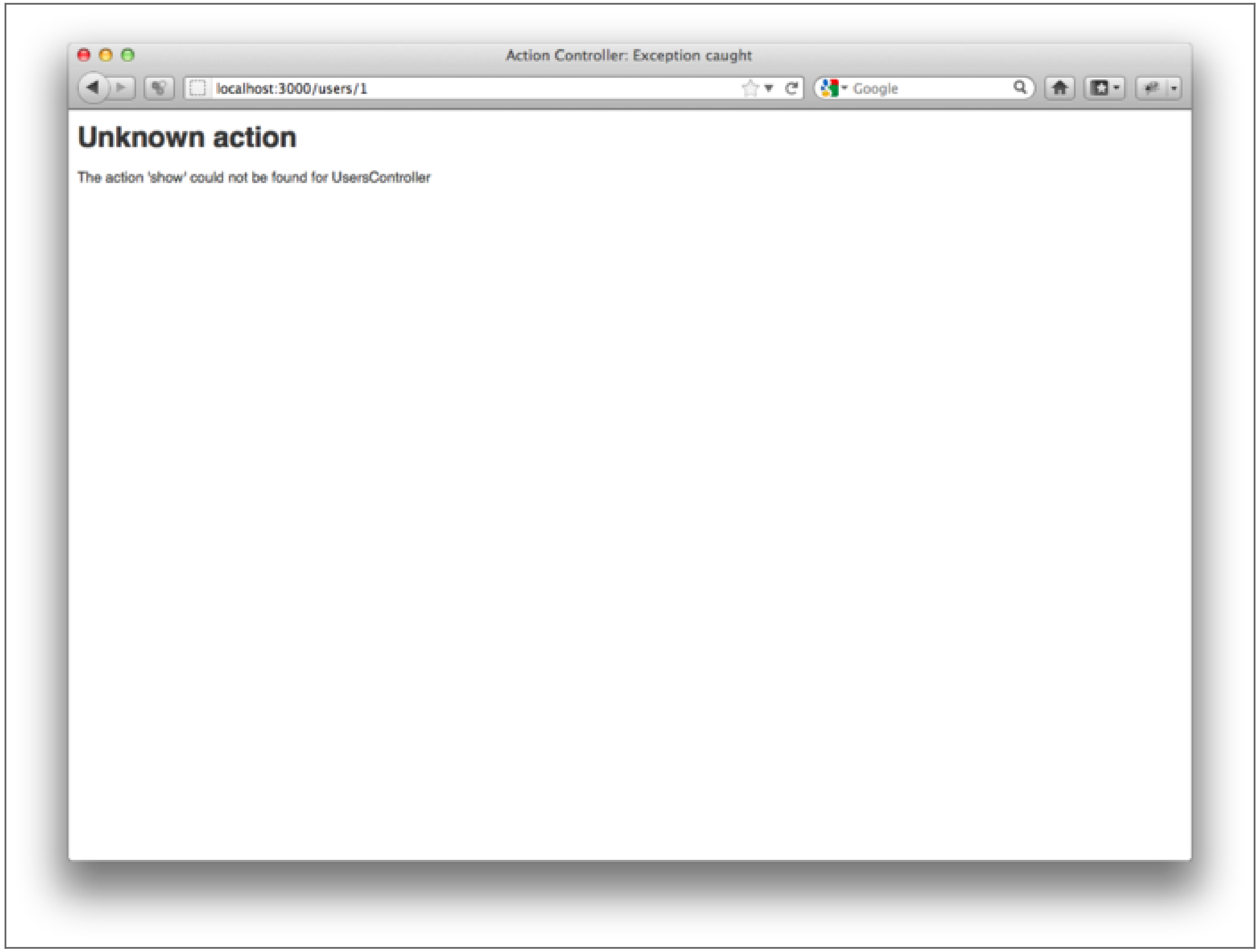
```
get "users/new"
```

last seen in Listing 5.32. This is because **`resources :users`** doesn't just add a working /users/1 URI; it endows our sample application with *all* the actions needed for a RESTful Users resource,[5] along with a large number of named routes (Section 5.3.3) for generating user URIs. The resulting correspondence of URIs, 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.

HTTP request	URI	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`
`PUT`	/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.

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.



Unknown action

The action 'show' could not be found for UsersController

Figure 7.5: The URI /users/1 with routing but no page. (full size)

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 5.28, the **`show.html.erb`** file doesn't currently exist, so you'll have to create it by hand, filling it with the 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.)

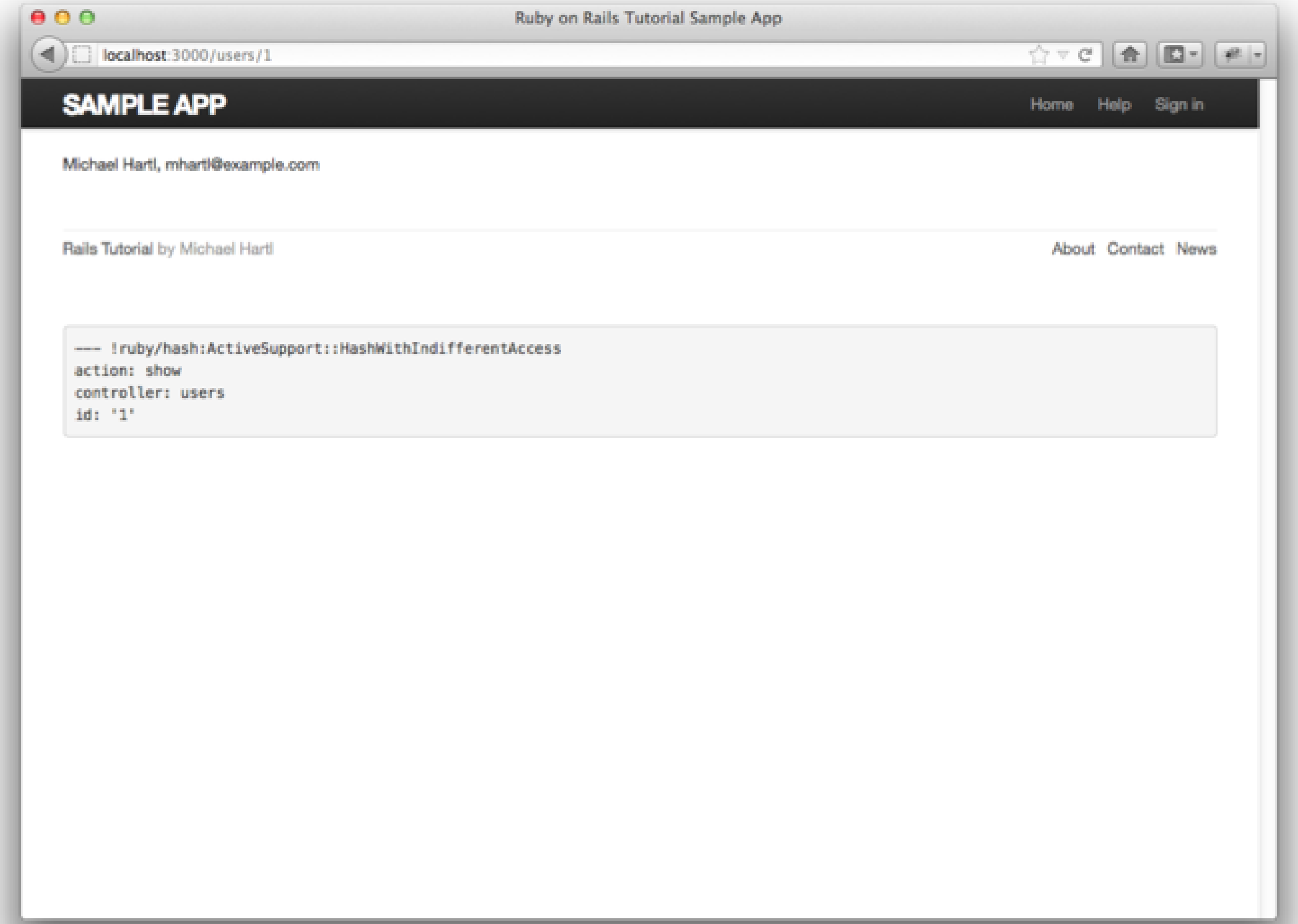
With the user view and action defined, the URI /users/1 works perfectly (Figure 7.6). Note that the debug information in Figure 7.6 confirms the value of **`params[:id]`**:

```
---
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`.



Michael Hartl, mhartl@example.com

Rails Tutorial by Michael Hartl

About Contact News

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.6: The user show page at /users/1 after adding a Users resource. (full size)

7.1.3 Testing the user show page (with factories)

Now that we have a minimal working profile, it's time to start working on the version mocked up in Figure 7.1. As with the creation of static pages (Chapter 3) and the User model (Chapter 6), we'll proceed using test-driven development.

Recall from Section 5.4.2 that we have elected to use integration tests for the pages associated with the Users resource. In the case of the signup page, our test first visits the **`signup_path`** and then checks for the right **`h1`** and **`title`** tags, as seen in Listing 5.31 and reproduced in Listing 7.6. (Note that we've omitted the **`full_title`** helper from Section 5.3.4 since the full title is already adequately tested there.)

Listing 7.6. A recap of the initial User pages spec.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "signup page" do
    before { visit signup_path }

    it { should have_selector('h1',    text: 'Sign up') }
    it { should have_selector('title', text: 'Sign up') }
  end
end
```

To test the user show page, we'll need a User model object so that the code in the **`show`** action (Listing 7.5) has something to find:

```
describe "profile page" do
  # Code to make a user variable
  before { visit user_path(user) }

  it { should have_selector('h1',    text: user.name) }
  it { should have_selector('title', text: user.name) }
end
```

where we need to fill in the comment with the appropriate code. This uses the **`user_path`** named route (Table 7.1) to generate the path to the show page for the given user. It then tests that the **`h1`** and **`title`** tags both contain the user's name.

In order to make the necessary User model object, we could use Active Record to create a user with **`User.create`**, but experience shows that user *factories* are a more convenient way to define user objects and insert them in the database. We'll be using the factories generated by Factory Girl,[6] a Ruby gem produced by the good people at thoughtbot. As with RSpec, Factory Girl defines a domain-specific language in Ruby, in this case specialized for defining Active Record objects. The syntax is simple, relying on Ruby blocks and custom methods to define the attributes of the desired object. For cases such as the one in this chapter, the advantage over Active Record may not be obvious, but we'll use more advanced features of factories in future chapters. For example, in Section 9.3.3 it will be important to create a sequence of users with unique email addresses, and factories make it easy to do this.

As with other Ruby gems, we can install Factory Girl by adding a line to the **`Gemfile`** used by Bundler (Listing 7.7). (Since Factory Girl is only needed in the tests, we've put it in the **`:test`** group.)

Listing 7.7. Adding Factory Girl to the **`Gemfile`**.

```
source 'https://rubygems.org'
  .
  .
  .
  group :test do
    .
    .
    .
    gem 'factory_girl_rails', '1.4.0'
  end
  .
  .
  .
end
```

Then install as usual:

```
$ bundle install
```

We'll put all our Factory Girl factories in the file **`spec/factories.rb`**, which automatically gets loaded by RSpec. The code needed to make a User factory appears in Listing 7.8.

Listing 7.8. A factory to simulate User model objects.
`spec/factories.rb`

```
FactoryGirl.define do
  factory :user do
    name     "Michael Hartl"
    email    "michael@example.com"
    password "foobar"
    password_confirmation "foobar"
  end
end
```

By passing the symbol **`:user`** to the **`factory`** command, we tell Factory Girl that the subsequent definition is for a User model object.

With the definition in Listing 7.8, we can create a User factory in the tests using the **`let`** command (Box 6.3) and the **`FactoryGirl`** method supplied by Factory Girl:

```
let(:user) { FactoryGirl.create(:user) }
```

The final result appears in Listing 7.9.

Listing 7.9. A test for the user show page.
spec/requests/user_pages_spec.rb

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "profile page" do
    let(:user) { FactoryGirl.create(:user) }
    before { visit user_path(user) }

    it { should have_selector('h1',    text: user.name) }
    it { should have_selector('title', text: user.name) }
  end
  .
  .
  .
end
```

You should verify at this point that the test suite is red:

```
$ bundle exec rspec spec/
```

We can get the tests to green with the code in Listing 7.10.

Listing 7.10. Adding a title and heading for the user profile page.
app/views/users/show.html.erb

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

Running the tests again should confirm that the test in Listing 7.9 is passing:

```
$ bundle exec rspec spec/
```

One thing you will quickly notice when running tests with Factory Girl is that they are *slow*. The reason is not Factory Girl's fault, and in fact it is a *feature*, not a bug. The issue is that the BCrypt algorithm used in Section 6.3.1 to create a secure password hash is slow by design: BCrypt's slow speed is part of what makes it so hard to attack. Unfortunately, this means that creating users can bog down the test suite; happily, there is an easy fix. BCrypt uses a *cost factor* to control how computationally costly it is to create the secure hash. The default value is designed for security, not for speed, which is perfect for production applications, but in tests our needs are reversed: we want *fast* tests, and don't care at all about the security of the test users' password hashes. The solution is to add a few lines to the test configuration file, **`config/environments/test.rb`**, redefining the cost factor from its secure default value to its fast minimum value, as shown in Listing 7.11. Even for a small test suite, the gains in speed from this step can be considerable, and I strongly recommend including Listing 7.11 in your **`test.rb`**.

Listing 7.11. Redefining the BCrypt cost factor in a test environment.
`config/environments/test.rb`

```
SampleApp::Application.configure do
  .
  .
  .
  # Speed up tests by lowering BCrypt's cost function.
  require 'bcrypt'
  silence_warnings do
```

```
    BCrypt::Engine::DEFAULT_COST = BCrypt::Engine::MIN_COST
  end
end
```

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. When making views, we'll focus on the visual appearance and not worry too much about the exact structure of the page, which means that (at least for now) we won't be writing tests. When we come to more error-prone aspects of view, such as pagination (Section 9.3.3), we'll resume test-driven development.

We'll start by adding a "globally recognized avatar", or Gravatar, to the user profile.[7] Originally created by Tom Preston-Werner (cofounder of GitHub) and later acquired by Automattic (the makers of WordPress), Gravatar is a free service that allows users to upload images and associate them with email addresses they control. 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 URI using the user's email address and the corresponding Gravatar image will automatically appear.[8]

Our plan is to define a **`gravatar_for`** helper function to return a Gravatar image for a given user, as shown in Listing 7.12.

Listing 7.12. 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>
```

You can verify at this point that the test suite is failing:

```
$ bundle exec rspec spec/
```

Because the **`gravatar_for`** method is undefined, the user show view is currently broken. (Catching errors of this nature is perhaps the most useful aspect of view specs. This is why having *some* test of the view, even a minimalist one, is so important.)

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 at the Gravatar home page, Gravatar URIs are based on an MD5 hash of the user's email 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. The resulting **`gravatar_for`** helper appears in Listing 7.13.

Listing 7.13. Defining a **`gravatar_for`** helper method.
`app/helpers/users_helper.rb`

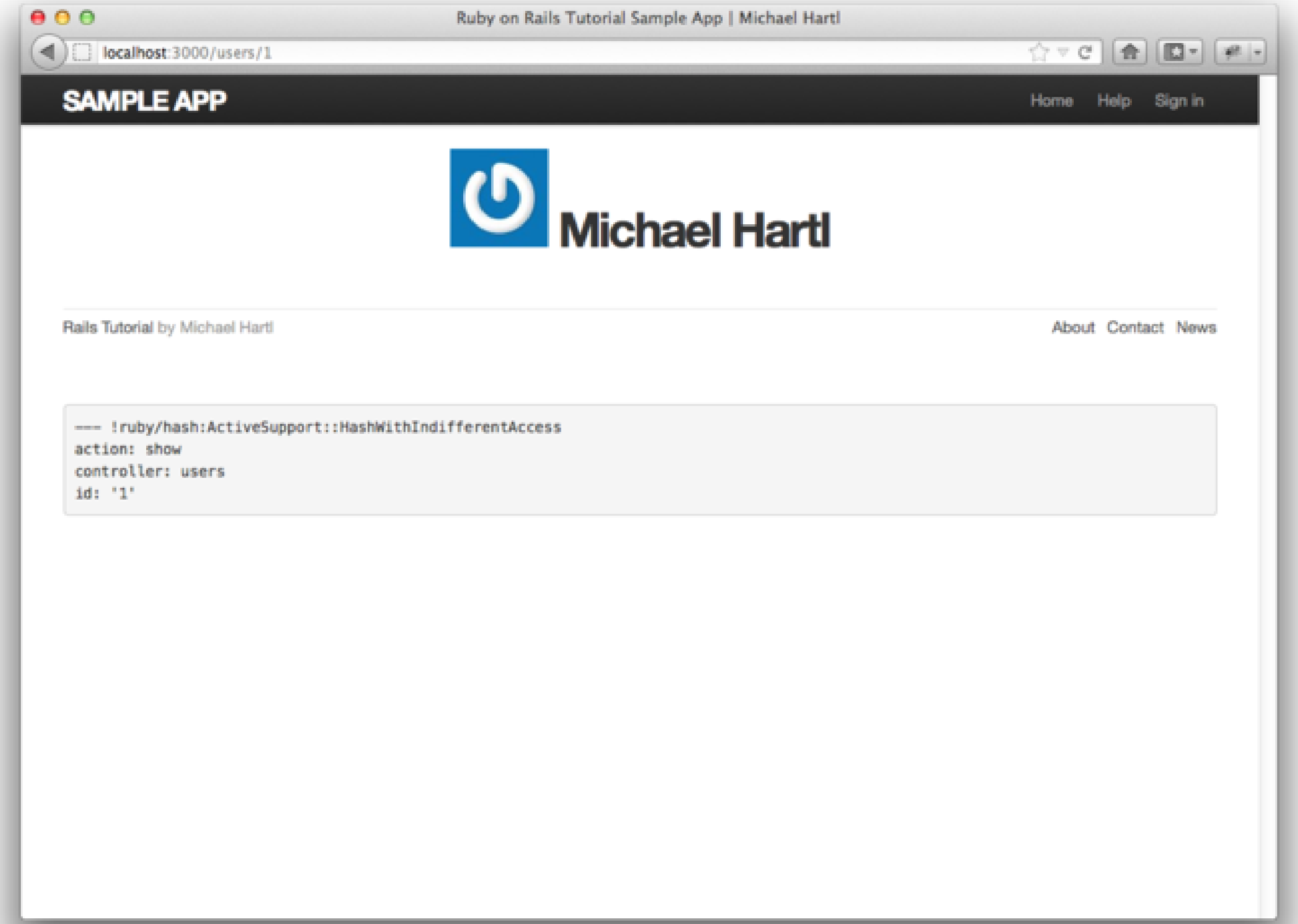
```
module UsersHelper

  # Returns the Gravatar (http://gravatar.com/) 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 7.13 returns an image tag for the Gravatar with a `"gravatar"` class and alt text equal to the user's name (which is especially convenient for sight-impaired browsers using a screen reader). You can confirm that the test suite is now passing:

```
$ bundle exec rspec spec/
```

The profile page appears as in Figure 7.7, which shows the default Gravatar image, which appears because `user@example.com` is an invalid email address (the example.com domain is reserved for examples).


Ruby on Rails Tutorial Sample App | Michael Hartl
localhost:3000/users/1
SAMPLE APP
Home
Help
Sign in
Michael Hartl
Rails Tutorial by Michael Hartl
About
Contact
News

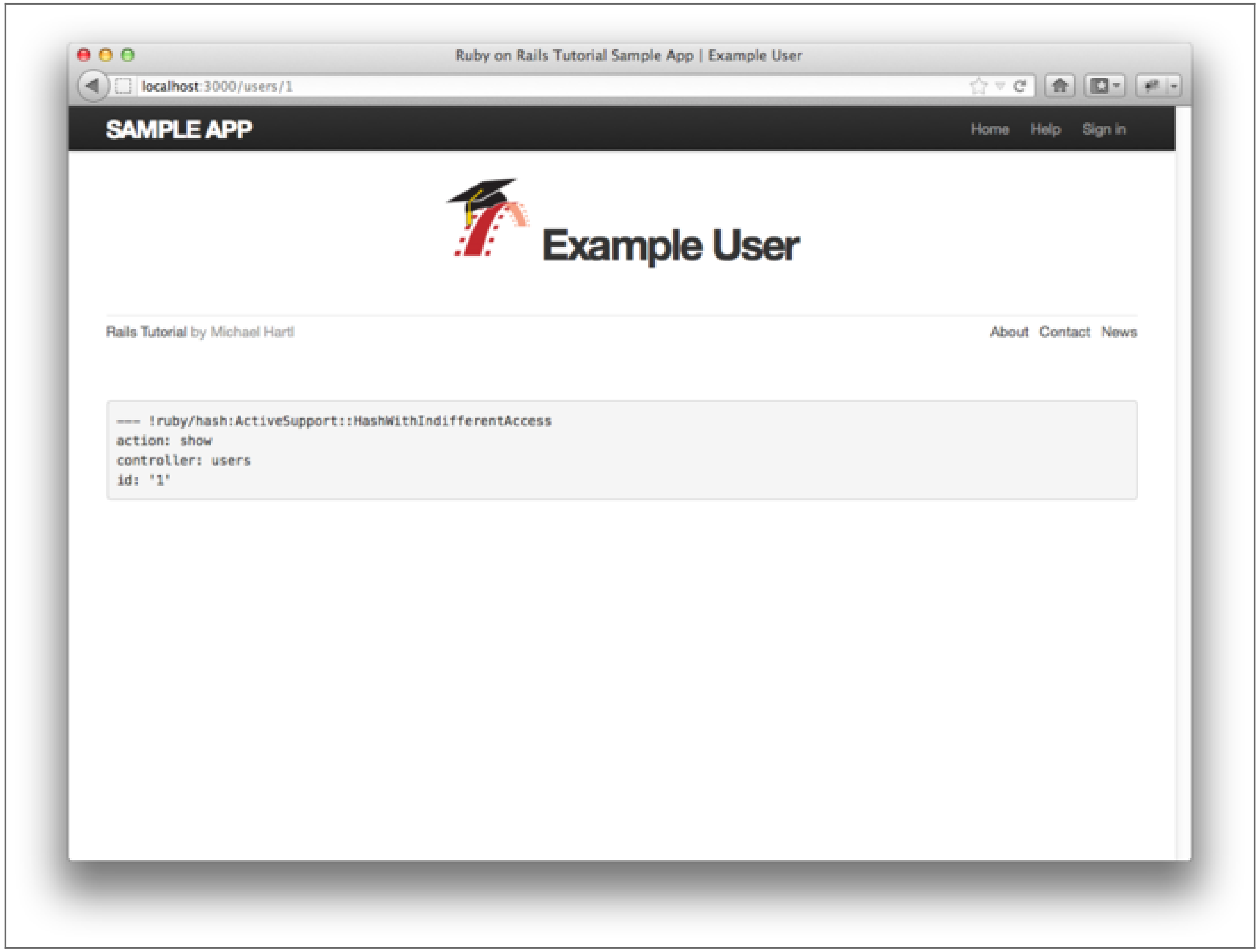
```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.7: The user profile page /users/1 with the default Gravatar. (full size)

To get our application to display a custom Gravatar, we'll use `update_attributes` (Section 6.1.5) to update the user in the database:

```
$ rails console
>> user = User.first
>> user.update_attributes(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 7.8.


Ruby on Rails Tutorial Sample App | Example User
localhost:3000/users/1
SAMPLE APP
Home
Help
Sign in
Example User
Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.8: The user show page with a custom Gravatar. (full size)

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 **`span4`** classes, which are both part of Bootstrap. The code for the modified user show page appears in Listing 7.14.

Listing 7.14. Adding a sidebar to the user `show` view.
`app/views/users/show.html.erb`

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="span4">
    <section>
      <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 7.15. (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 7.9.

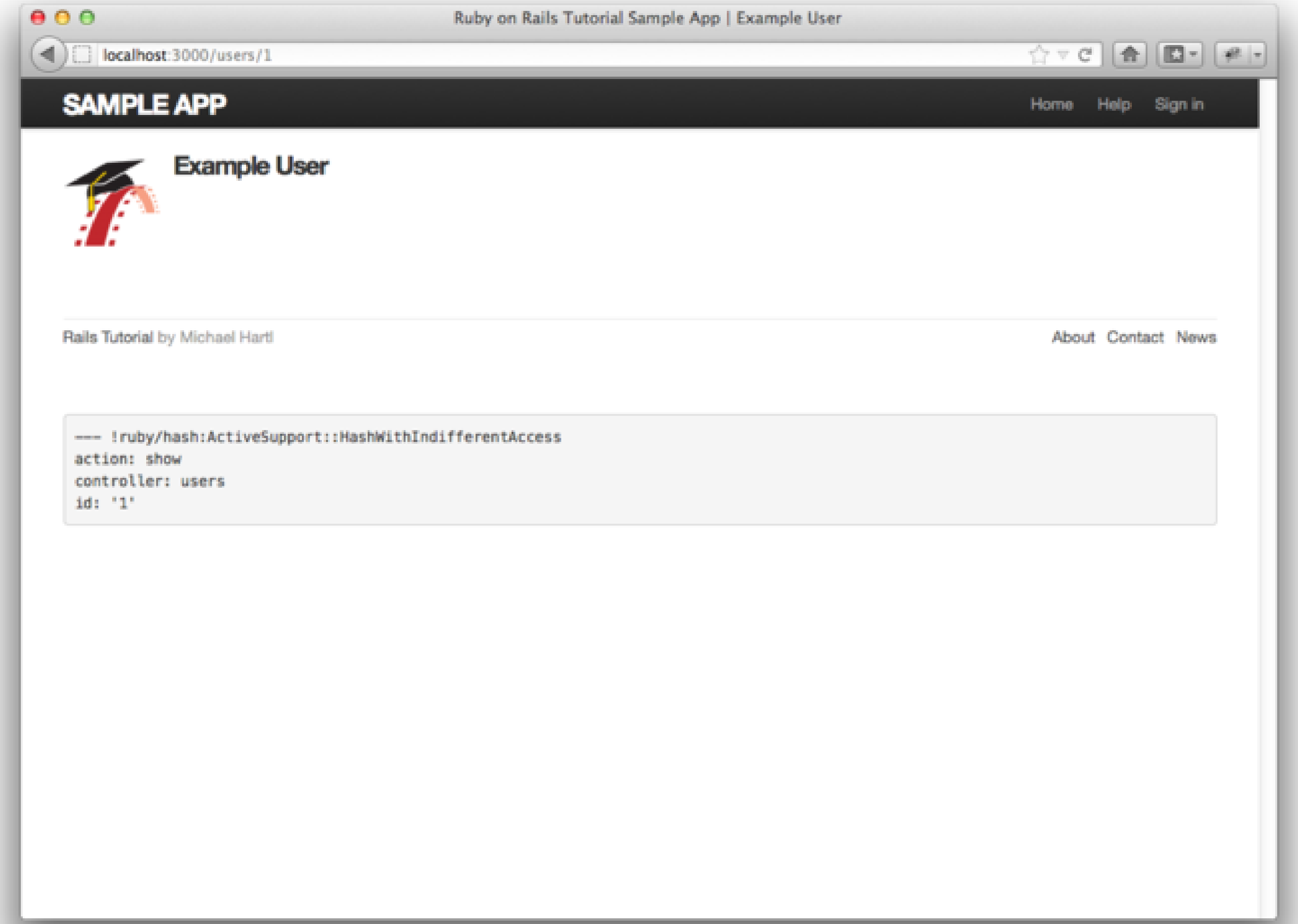
Listing 7.15. SCSS for styling the user show page, including the sidebar.
`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* sidebar */

aside {
  section {
    padding: 10px 0;
    border-top: 1px solid $grayLighter;
    &:first-child {
      border: 0;
      padding-top: 0;
    }
    span {
      display: block;
      margin-bottom: 3px;
      line-height: 1;
    }
    h1 {
      font-size: 1.6em;
      text-align: left;
      letter-spacing: -1px;
      margin-bottom: 3px;
    }
  }
}

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

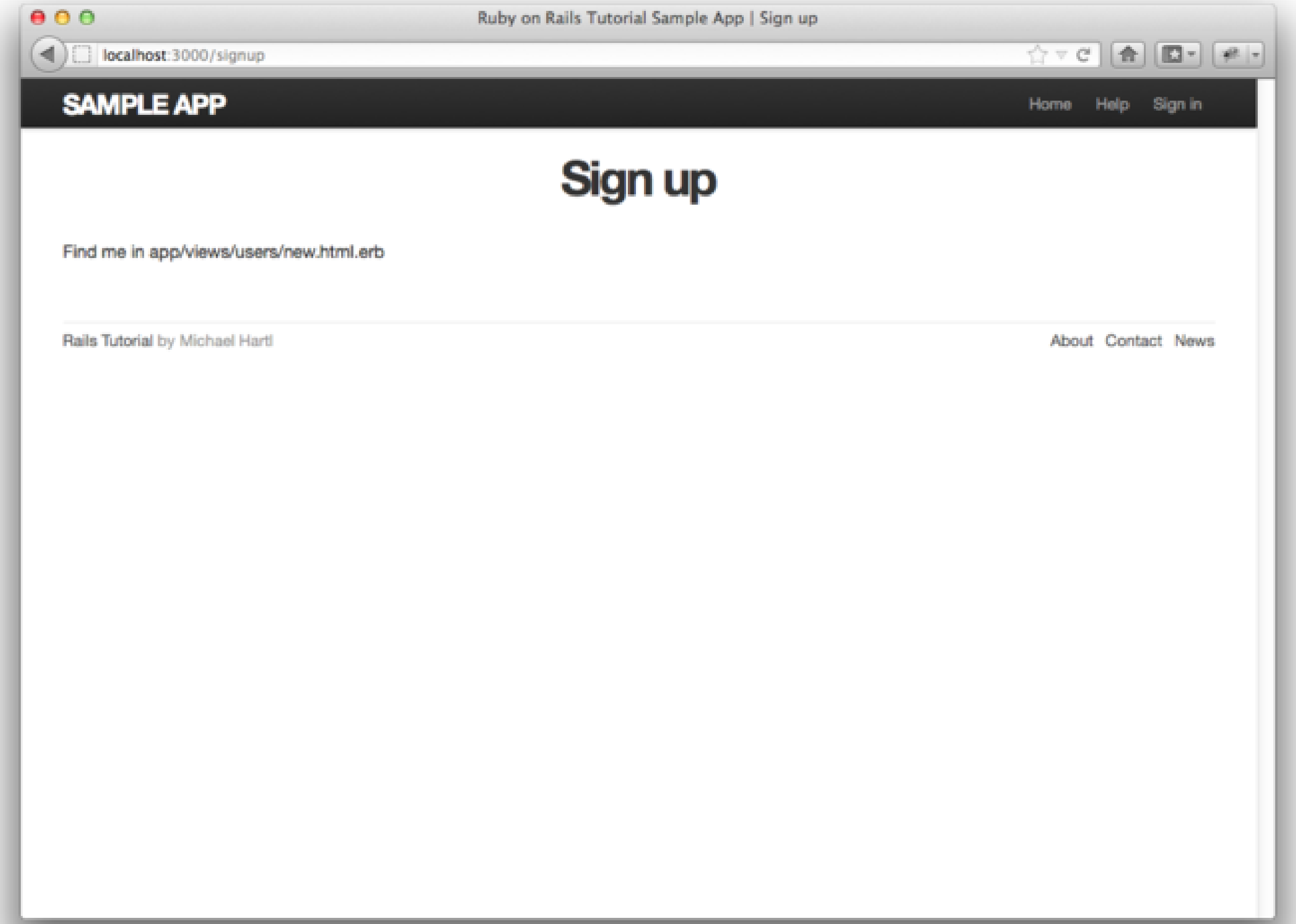

Ruby on Rails Tutorial Sample App | Example User
localhost:3000/users/1
SAMPLE APP
Home
Help
Sign in
Example User
Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.9: The user show page /users/1 with a sidebar and CSS. (full size)

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 5.9 (shown again in Figure 7.10) 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 7.11.


Ruby on Rails Tutorial Sample App | Sign up
localhost:3000/signup
SAMPLE APP
Home
Help
Sign in
Rails Tutorial by Michael Hartl
About
Contact
News

Sign up

Find me in app/views/users/new.html.erb

Figure 7.10: The current state of the signup page /signup. (full size)

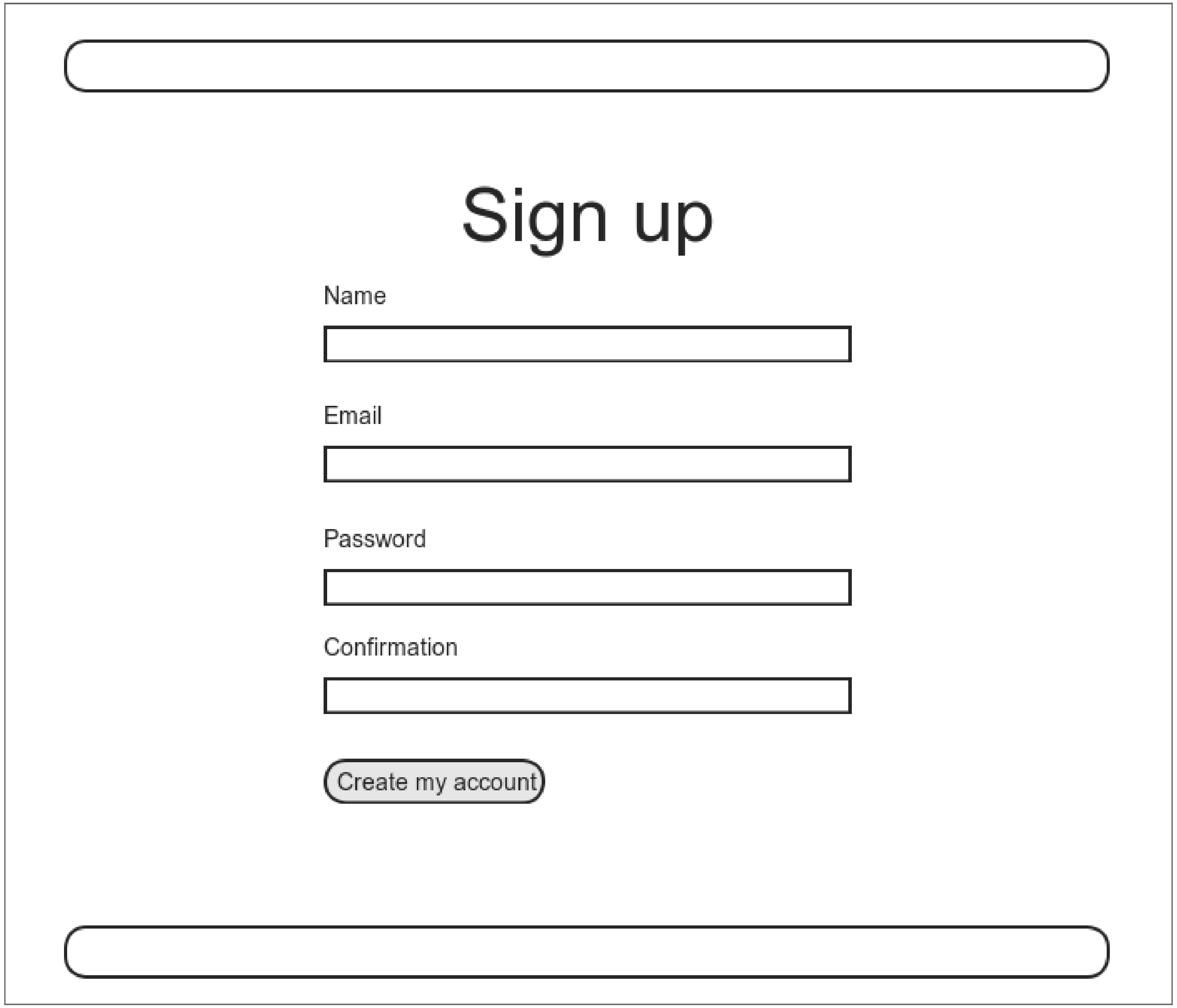


Figure 7.11: A mockup of the user signup page. (full size)

Since we’re about to add the ability to create new users through the web, let’s remove the user created at the console in Section 6.3.5. The cleanest way to do this is to reset the database with the **`db:reset`** Rake task:

```
$ bundle exec rake db:reset
```

After resetting the database, on some systems the test database needs to be re-prepared as well:

```
$ bundle exec rake db:test:prepare
```

Finally, on some systems you might have to restart the web server for the changes to take effect.[9]

7.2.1 Tests for user signup

In the days before powerful web frameworks with full testing capabilities, testing was often painful and error-prone. 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. With RSpec and Capybara, we will be able to write expressive tests to automate tasks that used to have to be done by hand.

We've already seen how Capybara supports an intuitive web-navigation syntax. So far, we've mostly used **`visit`** to visit particular pages, but Capybara can do a lot more, including filling in the kind of fields we see in Figure 7.11 and clicking on the button. The syntax looks like this:

```
visit signup_path
fill_in "Name", with: "Example User"
.
.
```

```
.
click_button "Create my account"
```

Our goal now is to write tests for the right behavior given invalid and valid signup information. Because these tests are fairly advanced, we'll build them up piece by piece. If you want to see how they work (including which file to put them in), you can skip ahead to Listing 7.16.

Our first task is to test for a failing signup form, and we can simulate the submission of invalid data by visiting the page and clicking the button using **`click_button`**:

```
visit signup_path
click_button "Create my account"
```

This is equivalent to visiting the signup page and submitting blank signup information (which is invalid). Similarly, to simulate the submission of valid data, we fill in valid information using **`fill_in`**:

```
visit signup_path
fill_in "Name",         with: "Example User"
fill_in "Email",        with: "user@example.com"
fill_in "Password",     with: "foobar"
fill_in "Confirmation", with: "foobar"
click_button "Create my account"
```

The purpose of our tests is to verify that clicking the signup button results in the correct behavior, creating a new user when the information is valid and not creating a user when it's invalid. 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
=> 0
```

Here **`User.count`** is **`0`** because we reset the database at the beginning of this section.

When submitting invalid data, we expect the user count not to change; when submitting valid data, we expect it to change by 1. We can express this in RSpec by combining the **`expect`** method with either the **`to`** method or the **`not_to`** method. We'll start with the invalid case since it is simpler; we visit the signup path and click the button, and we expect it *not to* change the user count:

```
visit signup_path
expect { click_button "Create my account" }.not_to change(User, :count)
```

Note that, as indicated by the curly braces, **`expect`** wraps **`click_button`** in a block (Section 4.3.2). This is for the benefit of the **`change`** method, which takes as arguments an object and a symbol and then calculates the result of calling that symbol as a method on the object both before and after the block. In other words, the code

```
expect { click_button "Create my account" }.not_to change(User, :count)
```

calculates

```
User.count
```

before and after the execution of

```
click_button "Create my account"
```

In the present case, we want the given code *not* to change the count, which we express using the **`not_to`** method. In effect, by enclosing the button click in a block we are able to replace

```
initial = User.count
click_button "Create my account"
final = User.count
initial.should == final
```

with the single line

```
expect { click_button "Create my account" }.not_to change(User, :count)
```

which reads like natural language and is much more compact.

The case of valid data is similar, but instead of verifying that the user count doesn't change, we check that clicking the button changes the count by 1:

```
visit signup_path
fill_in "Name",         with: "Example User"
fill_in "Email",        with: "user@example.com"
fill_in "Password",     with: "foobar"
fill_in "Confirmation", with: "foobar"
expect do
  click_button "Create my account"
end.to change(User, :count).by(1)
```

This uses the **`to`** method because we expect a click on the signup button with valid data *to* change the user count by one.

Combining the two cases with the appropriate `describe` blocks and pulling the common code into `before` blocks yields good basic tests for signing up users, as shown in Listing 7.16. Here we've factored out the common text for the submit button using the `let` method to define a `submit` variable.

Listing 7.16. Good basic tests for signing up users.
`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }
  .
  .
  .
  describe "signup" do

    before { visit signup_path }

    let(:submit) { "Create my account" }

    describe "with invalid information" do
      it "should not create a user" do
        expect { click_button submit }.not_to change(User, :count)
      end
    end

    describe "with valid information" do
      before do
        fill_in "Name",         with: "Example User"
        fill_in "Email",        with: "user@example.com"
        fill_in "Password",     with: "foobar"
        fill_in "Confirmation", with: "foobar"
      end

      it "should create a user" do
```

```
        expect { click_button submit }.to change(User, :count).by(1)
      end
    end
  end
end
```

We'll add a few more tests as needed in the sections that follow, but the basic tests in Listing 7.16 already cover an impressive amount of functionality. To get them to pass, we have to create a signup page with just the right elements, arrange for the page's submission to be routed to the right place, and successfully create a new user in the database only if the resulting user data is valid.

Of course, at this point the tests should fail:

```
$ bundle exec rspec spec/
```

7.2.2 Using `form_for`

Now that we have good failing tests for user signup, we'll start getting them to pass by making a *form* for signing up users. We can accomplish this in Rails with the **`form_for`** helper method, which takes in an Active Record object and constructs a form using the object's attributes. The result appears in Listing 7.17. (Readers familiar with Rails 2.x should note that **`form_for`** uses the "percent-equals" ERb syntax for inserting content; that is, where Rails 2.x used `<% form_for ... %>`, Rails 3 uses `<%= form_for ... %>` instead.)

Listing 7.17. 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="span6 offset3">
    <%= form_for(@user) do |f| %>

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

      <%= f.label :email %>
      <%= f.text_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-large btn-primary" %>
    <% end %>
  </div>
</div>
```

Let's break this down into pieces. The presence of the **`do`** keyword indicates that **`form_for`** takes a block with one variable, which we've called **`f`** for "form":

```
<%= form_for(@user) do |f| %>
  .
  .
  .
<% end %>
```

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—it 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. (We'll take a look at the HTML itself in Section 7.2.3.)

To see this in action, we need to drill down and look at the actual HTML produced by this form, but here we have a problem: the page currently breaks, because we have not set the **`@user`** variable—like all undefined instance variables (Section 4.4.5), **`@user`** is currently **`nil`**. Appropriately, if you run your test suite at this point, you'll see that the tests for the structure of the signup page from Listing 7.6 (i.e., the **`h1`** and the **`title`**) now fail:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "signup page"
```

(The **`-e`** here arranges to run just the examples whose description strings match **`"signup page"`**. Note in particular that this is *not* the substring **`"signup"`**, which would run all the test in Listing 7.16.) To get these tests to pass again and to get our form to render, we must define an **`@user`** variable in the controller action corresponding to **`new.html.erb`**, i.e., the **`new`** action in the Users controller. The **`form_for`** helper expects **`@user`** to be a User object, and since we're creating a *new* user we simply use **`User.new`**, as seen in Listing 7.18.

Listing 7.18. Adding an **`@user`** variable to the **`new`** action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def new
```

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

With the **`@user`** variable so defined, the test for the signup page should be passing again:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb -e "signup page"
```

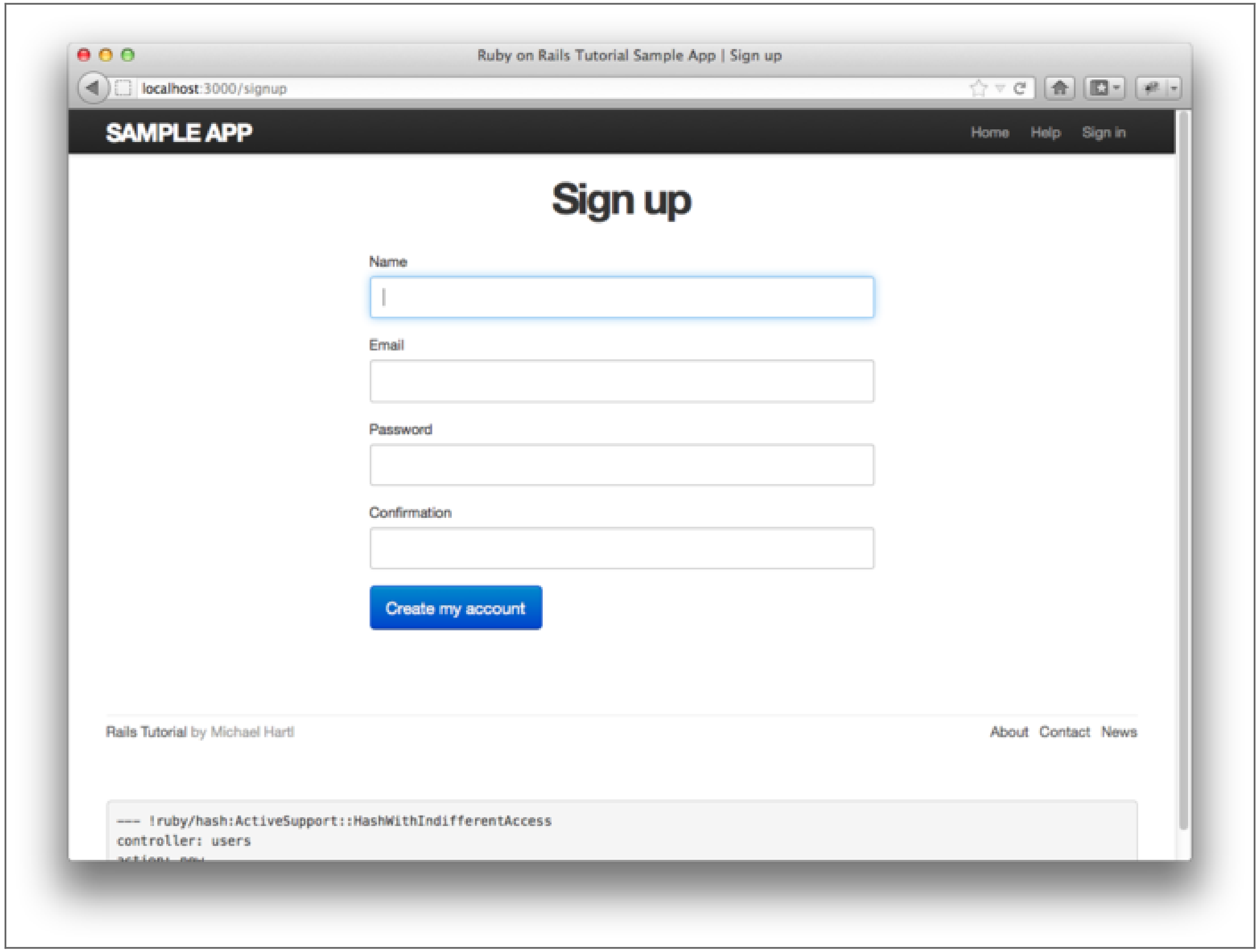
At this point, the form (with the styling from Listing 7.19) appears as in Figure 7.12. Note the reuse of the **`box_sizing`** mixin from Listing 7.2.

Listing 7.19. CSS for the signup form.
`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* forms */

input, textarea, select, .uneditable-input {
  border: 1px solid #bbb;
  width: 100%;
  padding: 10px;
  height: auto !important;
  margin-bottom: 15px;
  @include box_sizing;
}
```



```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: users
```

Figure 7.12: The signup form /signup for new users. (full size)

7.2.3 The form HTML

As indicated by Figure 7.12, the signup page now renders properly, indicating that the `form_for` code in Listing 7.17 is producing valid HTML. If you look at the HTML for the generated form (using either Firebug or the “view page source” feature of your browser), you should see markup as in Listing 7.20. Although many of the details are irrelevant for our purposes, let’s take a moment to highlight the most important parts of its structure.

Listing 7.20. The HTML for the form in Figure 7.12.

```
<form accept-charset="UTF-8" action="/users" class="new_user"
      id="new_user" method="post">

  <label for="user_name">Name</label>
  <input id="user_name" name="user[name]" size="30" type="text" />

  <label for="user_email">Email</label>
  <input id="user_email" name="user[email]" size="30" type="text" />

  <label for="user_password">Password</label>
  <input id="user_password" name="user[password]" size="30"
         type="password" />

  <label for="user_password_confirmation">Confirmation</label>
  <input id="user_password_confirmation"
         name="user[password_confirmation]" size="30" type="password" />

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

(Here I’ve omitted some HTML related to the *authenticity token*, which Rails automatically includes to thwart a particular kind of attack called a *cross-site request forgery* (CSRF). See the Stack Overflow entry on the Rails authenticity token if you’re interested in the details of how this works and why it’s important.)

We’ll start with the internal structure of the document. Comparing Listing 7.17 with Listing 7.20, 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]" size="30" type="text" />
```

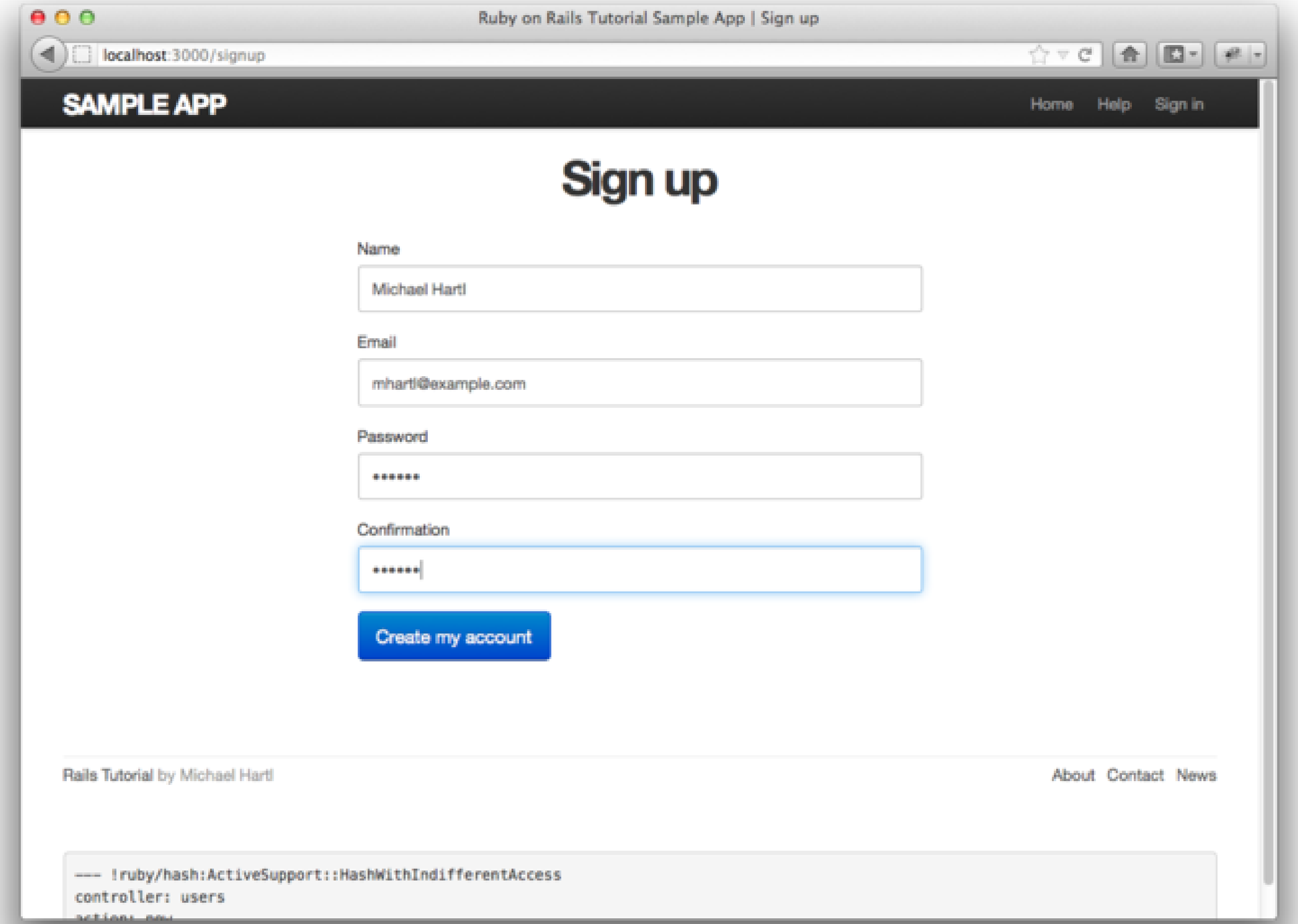
and

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

produces the HTML

```
<label for="user_password">Password</label><br />
<input id="user_password" name="user[password]" size="30" type="password" />
```

As seen in Figure 7.13, text fields (`type="text"`) simply display their contents, whereas password fields (`type="password"`) obscure the input for security purposes, as seen in Figure 7.13.



Name: Michael Hartl

Email: mhartl@example.com

Password: ••••••

Confirmation: ••••••

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
controller: users
```

Figure 7.13: A filled-in form with `text` and `password` fields. (full size)

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.

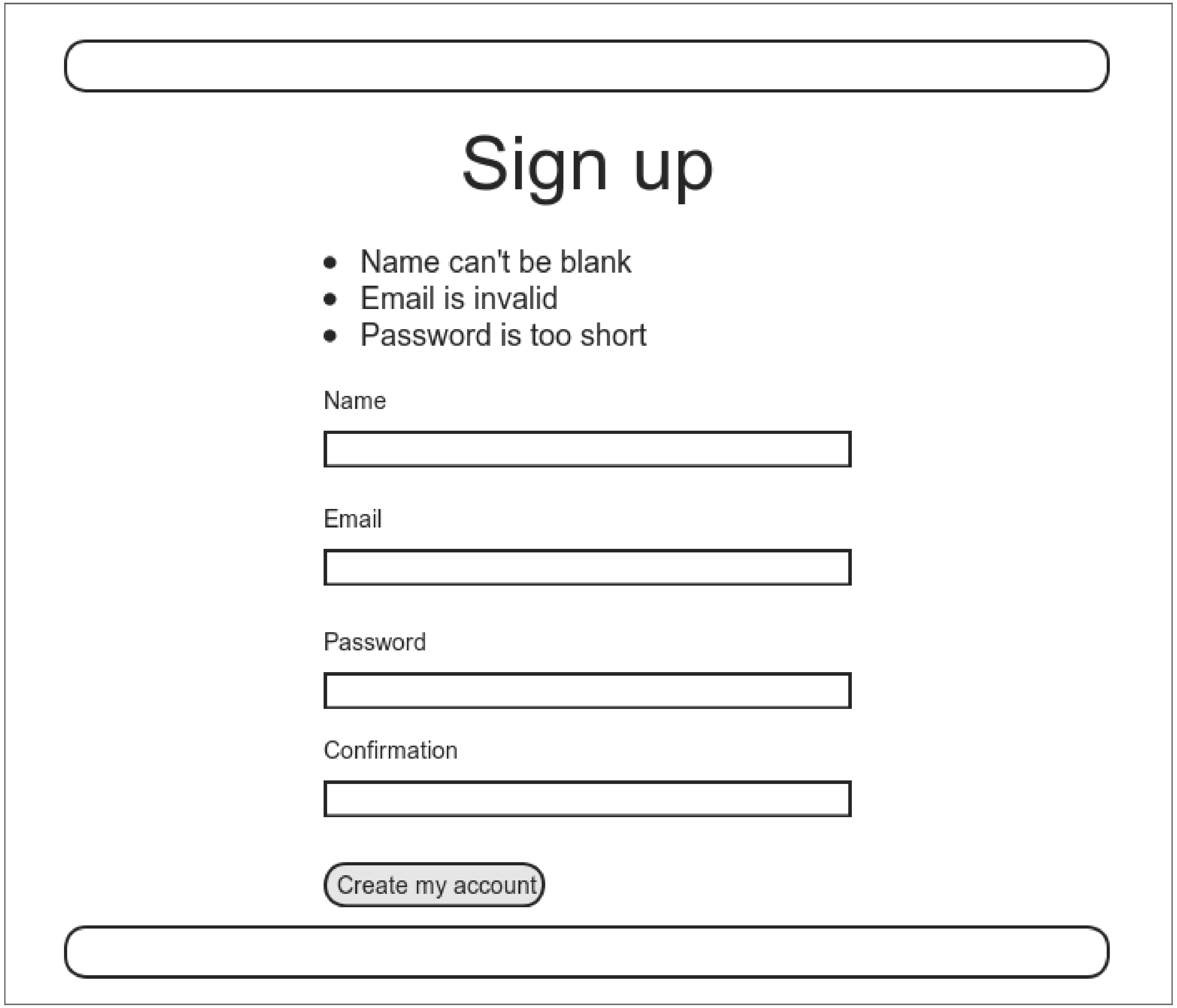
The second important element is the **form** tag itself. Rails creates the **form** tag using the **@user** object: because every Ruby object knows its own class (Section 4.4.1), Rails figures out that **@user** is of class **User**; moreover, since **@user** is a *new* user, Rails knows to construct 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 URI. We'll see in the next two sections what effects this has.

7.3 Signup failure

Although we've briefly examined the HTML for the form in Figure 7.12 (shown in Listing 7.20), it's 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 7.14.


Sign up

- Name can't be blank
- Email is invalid
- Password is too short

Name

Email

Password

Confirmation

Figure 7.14: A mockup of the signup failure page. (full size)

7.3.1 A working form

Our first step is to eliminate the error that currently results when submitting the signup form, as you can verify in your browser or by running the test for signup with invalid information:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
-e "signup with invalid information"
```

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 URIs 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 7.2.3, this HTML issues a `POST` request to the /users URI.

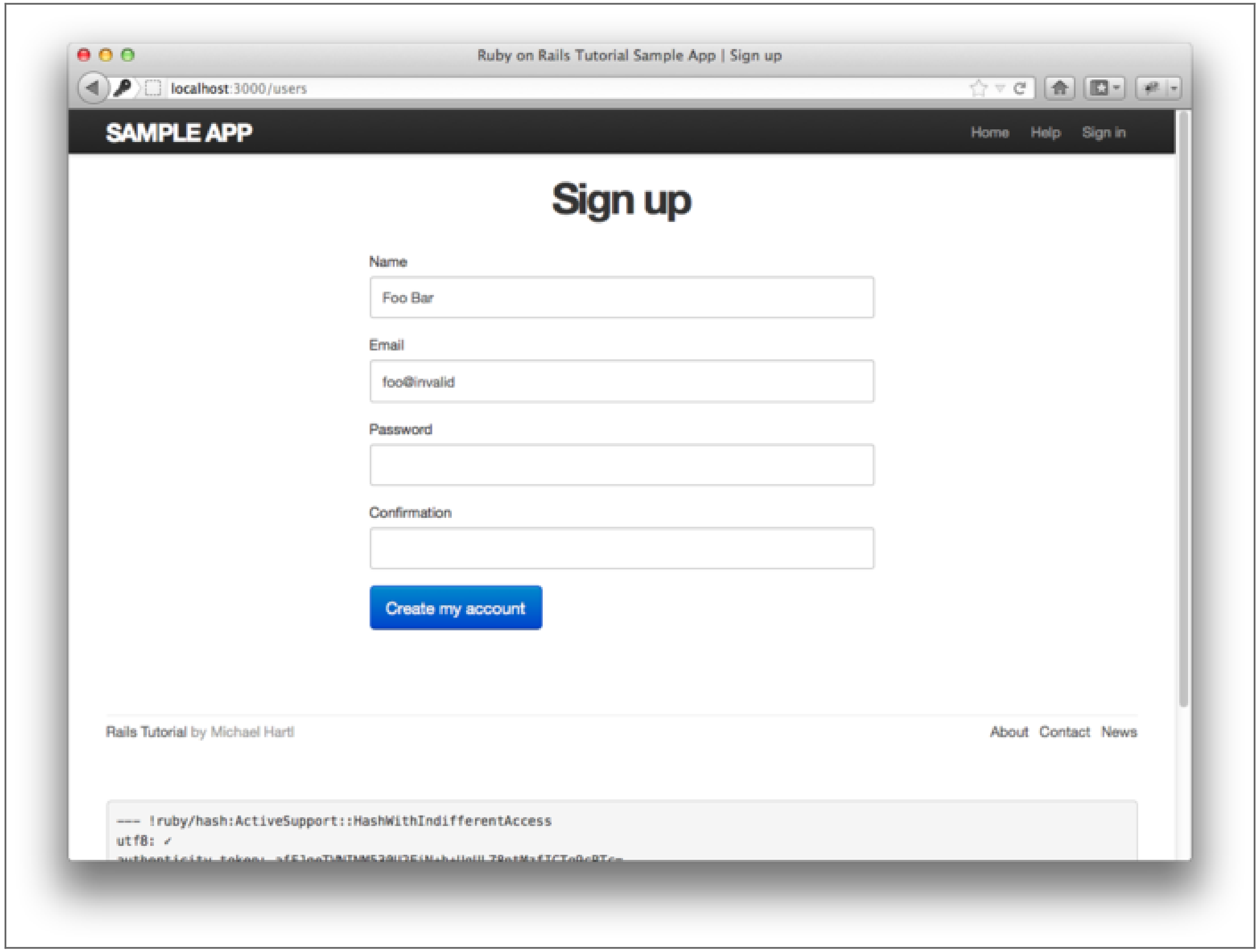
We can get the test for invalid information from Listing 7.16 to pass with the code in Listing 7.21. This listing includes a second use of the **`render`** method, which we first saw in the context of partials (Section 5.1.3); as you can see, **`render`** works in controller actions as well. Note that 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 the save succeeds.

Listing 7.21. A **`create`** action that can handle signup failure (but not success).

app/controllers/users_controller.rb

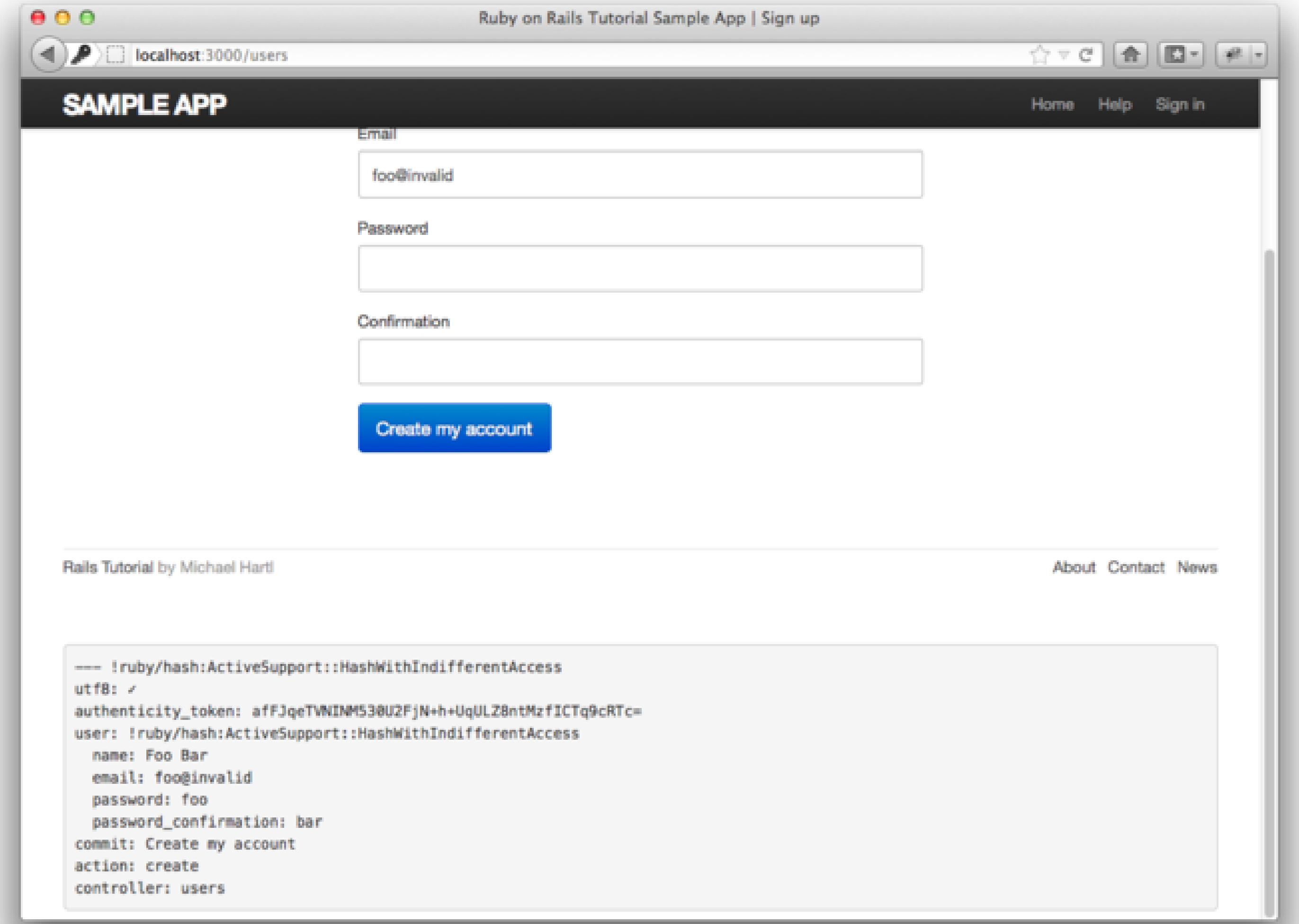
```ruby
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(params[:user])
    if @user.save
      # Handle a successful save.
    else
      render 'new'
    end
  end
end
```

The best way to understand how the code in Listing 7.21 works is to *submit* the form with some invalid signup data; the result appears in Figure 7.15, and the full debug information appears in Figure 7.16.



```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
utf8: ✓
```

Figure 7.15: Signup failure. (full size)



```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
utf8: ✓
authenticity_token: afFJqeTVNINM530U2FjN+h+UqULZ8ntMzfICTq9cRTc=
user: !ruby/hash:ActiveSupport::HashWithIndifferentAccess
  name: Foo Bar
  email: foo@invalid
  password: foo
  password_confirmation: bar
commit: Create my account
action: create
controller: users
```

Figure 7.16: Signup failure debug information. (full size)

To get a clearer picture of how Rails handles the submission, let's take a closer look at the **`params`** hash from the debug information (Figure 7.16):

```
---
user:
  name: Foo Bar
  password_confirmation: foo
  password: bar
  email: foo@invalid
commit: Create my account
action: create
controller: users
```

We saw starting in Section 7.1.2 that the **`params`** hash contains information about each request; in the case of a URI 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 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]" size="30" type="text" />
```

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

Although the hash keys appear as strings in the debug output, internally Rails uses 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 7.21. This means that the line

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

is equivalent to

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

Of course, instantiating such a variable has implications for successful signup—as we'll see in Section 7.4, once `@user` is defined properly, calling `@user.save` is all that's needed to complete the registration—but it has consequences even in the failed signup considered here. Note in Figure 7.15 that the fields are *pre-filled* with the data from the failed submission. This is because `form_for` automatically fills in the fields with the attributes of the `@user` object, so that, for example, if `@user.name` is `"Foo"` then

```
<%= form_for(@user) do |f| %>
  <%= f.label :name %>
  <%= f.text_field :name %>
  .
  .
  .
```

will produce the HTML

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

  <label for="user_name">Name</label><br />
  <input id="user_name" name="user[name]" size="30" type="text" value="Foo"/>
  .
  .
  .
```

Here the `value` of the `input` tag is `"Foo"`, so that's what appears in the text field.

As you might guess, now that we can submit a form without generating an error, the test for invalid submission should pass:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
-e "signup with invalid information"
```

7.3.2 Signup error messages

Though not strictly necessary, it's helpful to output error messages on failed signup to indicate the problems that prevented successful user registration. Rails provides just 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 in Section 6.2.2) contains an array of error messages.

As in the console session above, the failed save in Listing 7.21 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, as shown in Listing 7.22. (Writing a test for the error messages first is a good idea and is left as an exercise; see Section 7.6.) It's worth noting that this error messages partial is only a first attempt; the final version appears in Section 10.3.2.

Listing 7.22. 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="span6 offset3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>
      .
      .
      .
    <% 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 9.1.1.) This means that we have to create both the new `app/views/shared` directory and the `_error_messages.html.erb` partial file. The partial itself appears in Listing 7.23.

Listing 7.23. 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-error">
      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 4.2.3 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 10.2.)

The other new idea is the **`pluralize`** text helper. It isn't available in the console by default, but we can include it explicitly through the **`ActionView::Helpers::TextHelper`** module:[10]

```
>> include ActionView::Helpers::TextHelper
>> pluralize(1, "error")
=> "1 error"
```

```
>> 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:

```
>> pluralize(2, "woman")
=> "2 women"
>> 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 on teh interwebs).

Note that Listing 7.23 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, on error pages 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 7.24, which makes use of Sass's `@extend` function to include the functionality of two Bootstrap classes `control-group` and `error`. As a result, on failed submission the error messages appear surrounded by red, as seen in Figure 7.17. 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.

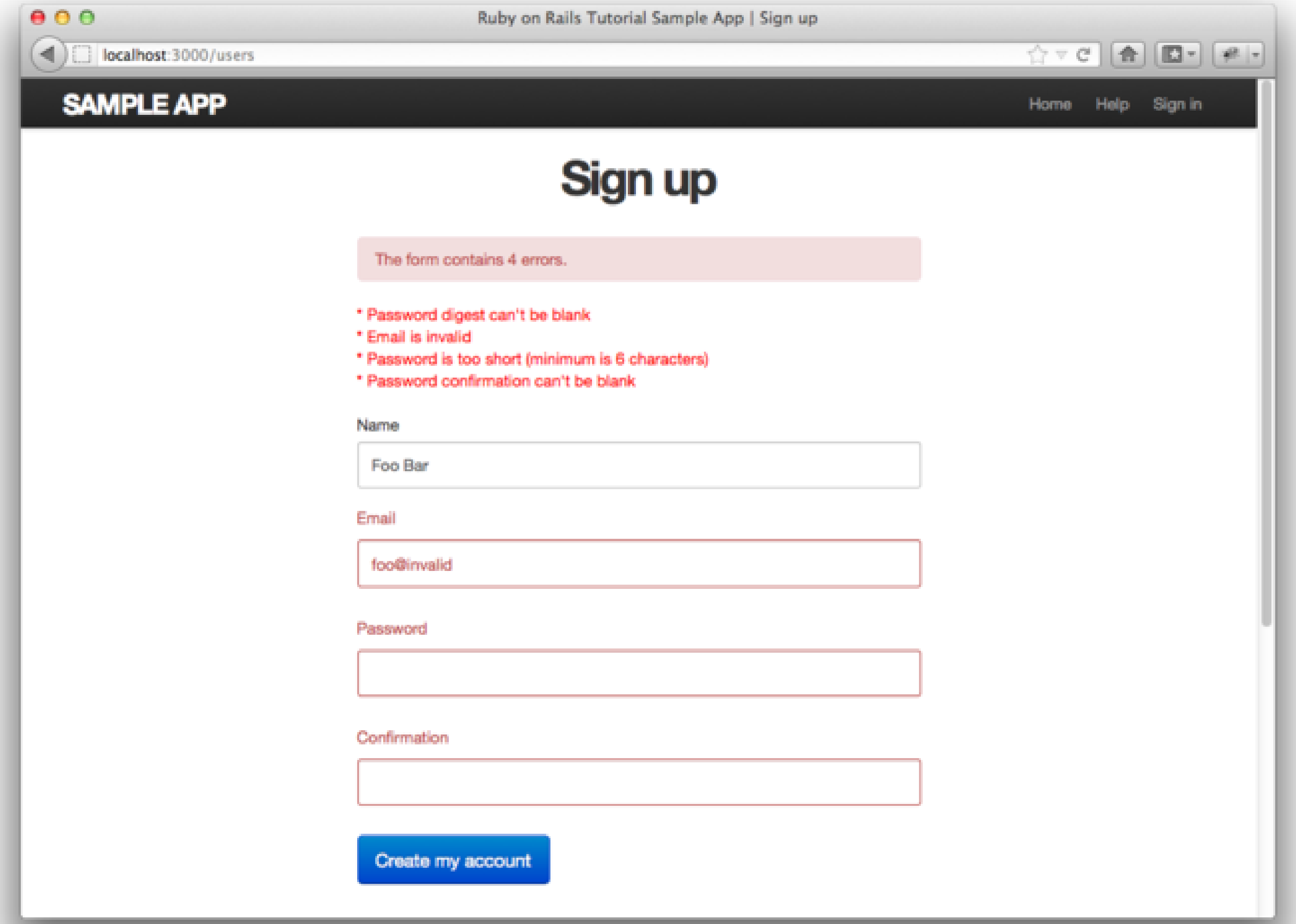
Listing 7.24. CSS for styling error messages.

`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* forms */
.
.
.
#error_explanation {
  color: #f00;
  ul {
    list-style: none;
    margin: 0 0 18px 0;
  }
}

.field_with_errors {
  @extend .control-group;
  @extend .error;
 }
```


Ruby on Rails Tutorial Sample App | Sign up

localhost:3000/users

Sign up

The form contains 4 errors.

- Password digest can't be blank
- Email is invalid
- Password is too short (minimum is 6 characters)
- Password confirmation can't be blank

Name: Foo Bar

Email: foo@invalid

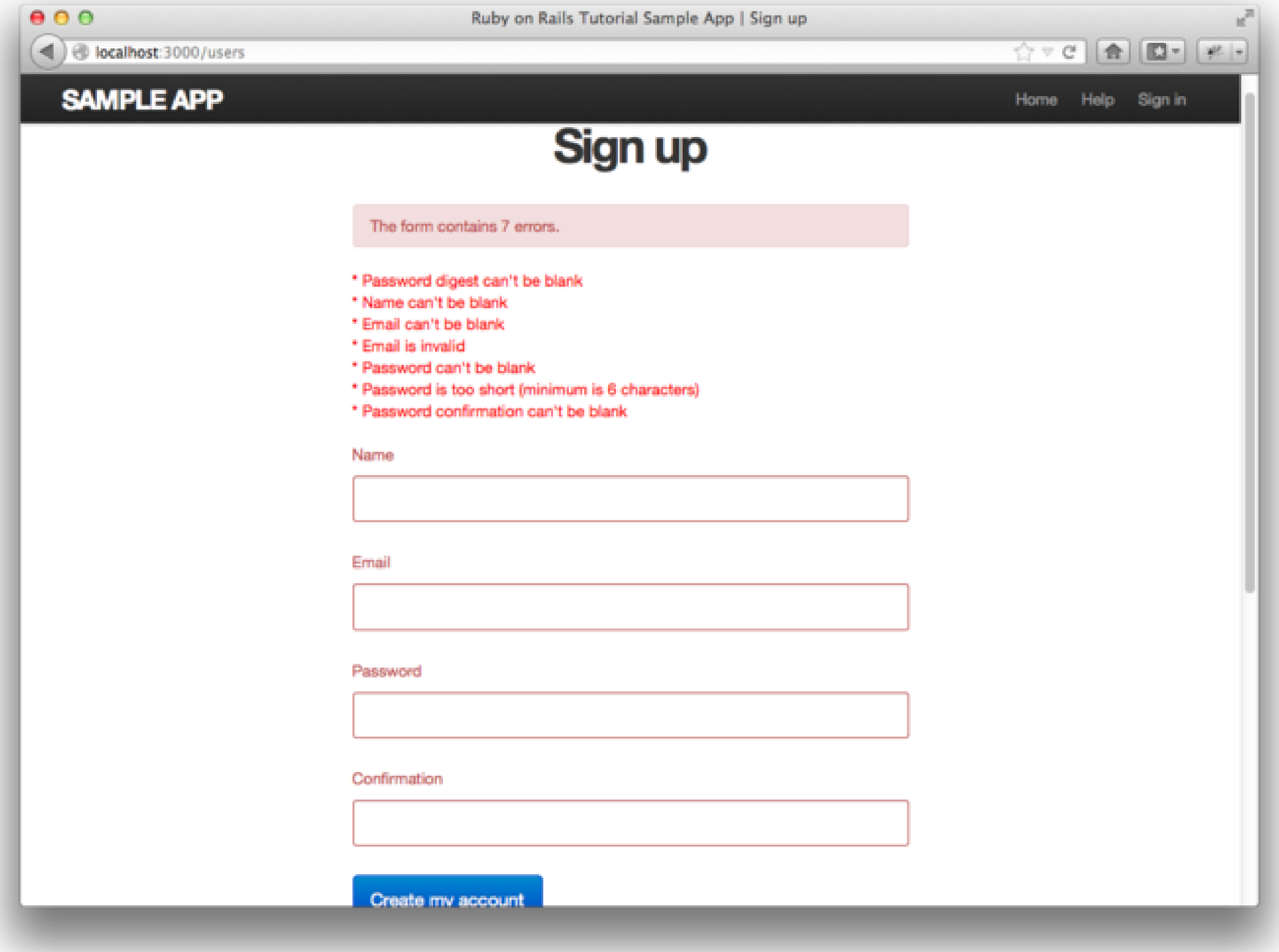
Password:

Confirmation:

Figure 7.17: Failed signup with error messages. (full size)

To see the results of our work in this section, we'll recapitulate the steps in the failed signup test from Listing 7.16 by visiting the signup page and clicking "Create my account" with blank input fields. The result is shown in Figure 7.18. As you might guess from the working page, at this point the corresponding test should also pass:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
> -e "signup with invalid information"
```



Sign up

The form contains 7 errors.

- Password digest can't be blank
- Name can't be blank
- Email can't be blank
- Email is invalid
- Password can't be blank
- Password is too short (minimum is 6 characters)
- Password confirmation can't be blank

Figure 7.18: The result of visiting /signup and just clicking “Create my account”. (full size)

Unfortunately, there’s a minor blemish in the error messages shown in Figure 7.18: the error for a missing password reads as “Password digest can’t be blank” instead of the more sensible “Password can’t be blank”. This is due to the password digest presence validation hiding in **`has_secure_password`**, as mentioned briefly in Section 6.3.4. Fixing this problem is left as an exercise (Section 7.6).

7.4 Signup success

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 7.19. If it fails, we simply fall back on the behavior developed in Section 7.3.

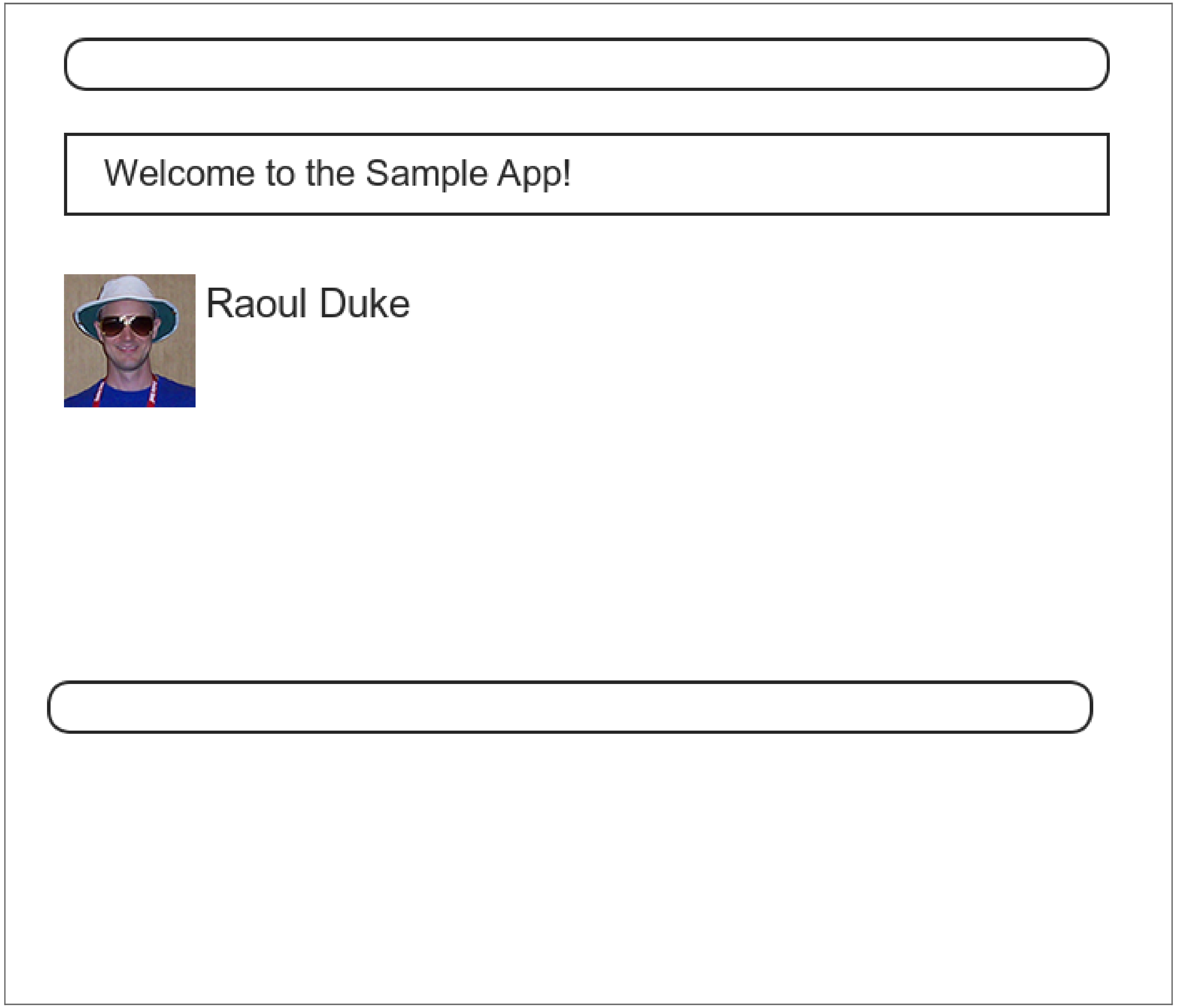


Figure 7.19: A mockup of successful signup. (full size)

7.4.1 The finished signup form

To complete a working signup form, we need to fill in the commented-out section in Listing 7.21 with the appropriate behavior. Currently, the test for valid submission should be failing:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb \
> -e "signup with valid information"
```

This is because the default behavior for a Rails action is to render the corresponding view, but there is not (nor should there be) a view template corresponding to the `create` action. Instead, we need to redirect to a different page, and it makes sense for that page to be the newly created user's profile. Testing that the proper page gets rendered is left as an exercise (Section 7.6); the application code appears in Listing 7.25.

Listing 7.25. The user `create` action with a save and a redirect.
`app/controllers/users_controller.rb`

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

Note that we can omit the `user_url` in the redirect, writing simply `redirect_to @user` to redirect to the user show page.

With the code in Listing 7.25, our signup form is working, as you can verify by running the test suite:

```
$ bundle exec rspec spec/
```

7.4.2 The flash

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 accomplish this is to use a special variable called the *flash*, which operates like flash memory in that it stores its data temporarily. The **`flash`** variable is effectively a hash; you may even recall the console example in Section 4.3.3, where we saw how to iterate through a hash using a strategically named **`flash`** hash:

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

We can arrange to display the contents of the flash site-wide by including it in our application layout, as in Listing 7.26. (This code is a particularly ugly combination of HTML and ERb; an exercise in Section 7.6 shows how to make it prettier.)

Listing 7.26. 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 |key, value| %>
        <div class="alert alert-<%= key %>"><%= value %></div>
      <% end %>
      <%= yield %>
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
    .
    .
    .
  </body>
</html>
```

The code in Listing 7.26 arranges to insert a `div` tag for each element in the flash, with a CSS class indicating the type of message. For example, if `flash[:success] = "Welcome to the Sample App!"`, then the code

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

will produce this HTML:

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

(Note that the key `:success` is a symbol, but embedded Ruby automatically converts it to the string `"success"` before inserting it into the template.) The reason we iterate through all possible key/value pairs is so that we can include other kinds of flash messages. For example, in Section 8.1.5 we'll see `flash[:error]` used to indicate a failed signin attempt.[11]

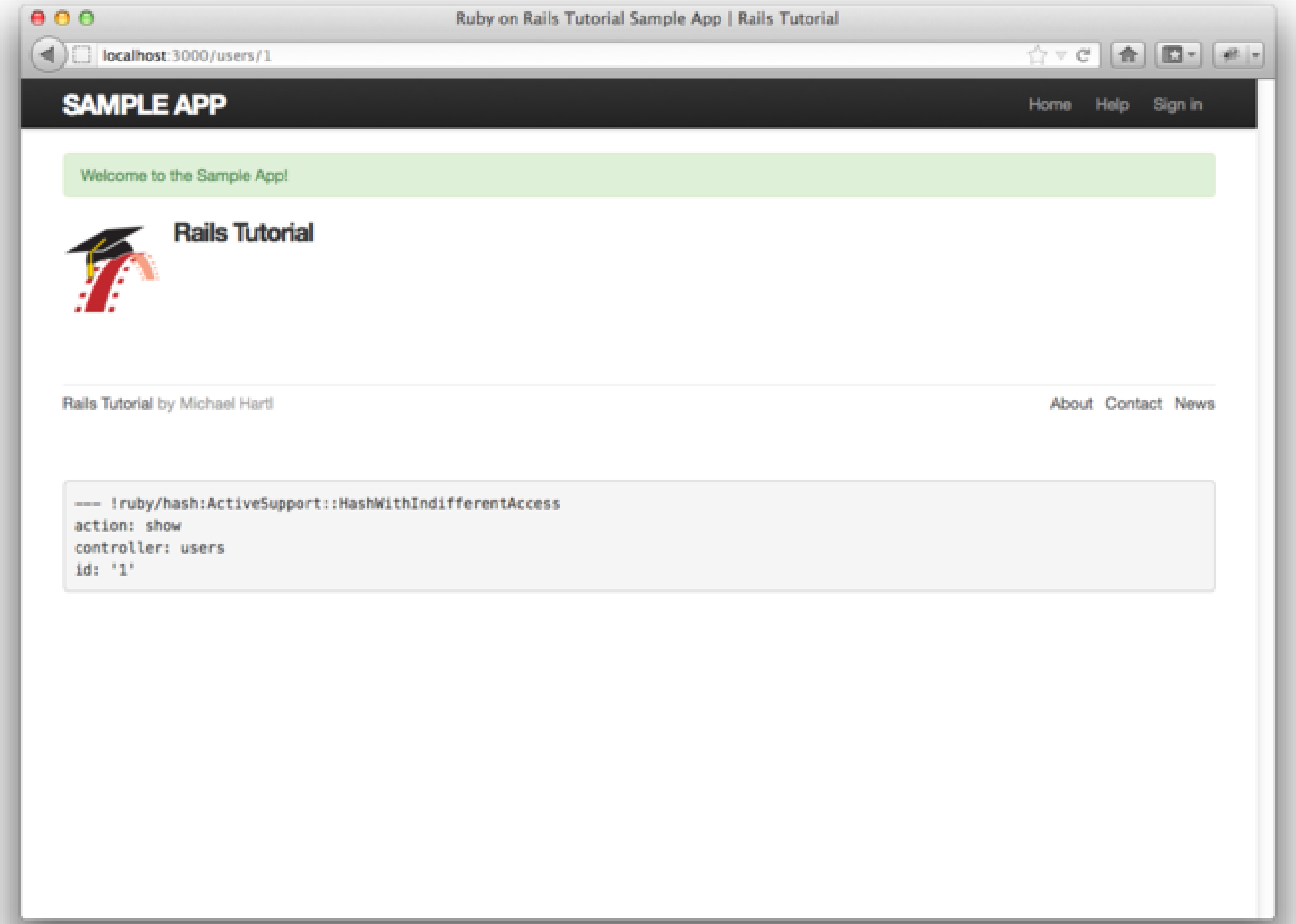
Writing a test for the right flash message is left as an exercise (Section 7.6), and we can get the test to pass by assigning `flash[:success]` a welcome message in the `create` action, as shown in Listing 7.27.

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

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

7.4.3 The first signup

We can see the result of all this work by signing up our first user under the name "Rails Tutorial" and email address "`example@railstutorial.org`". The resulting page (Figure 7.20) shows a friendly 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. (If instead you get an error message indicating that the email address has already been taken, be sure to run the **`db:reset`** Rake task as indicated in Section 7.2.) Then, upon reloading the user show page, the flash message disappears as promised (Figure 7.21).

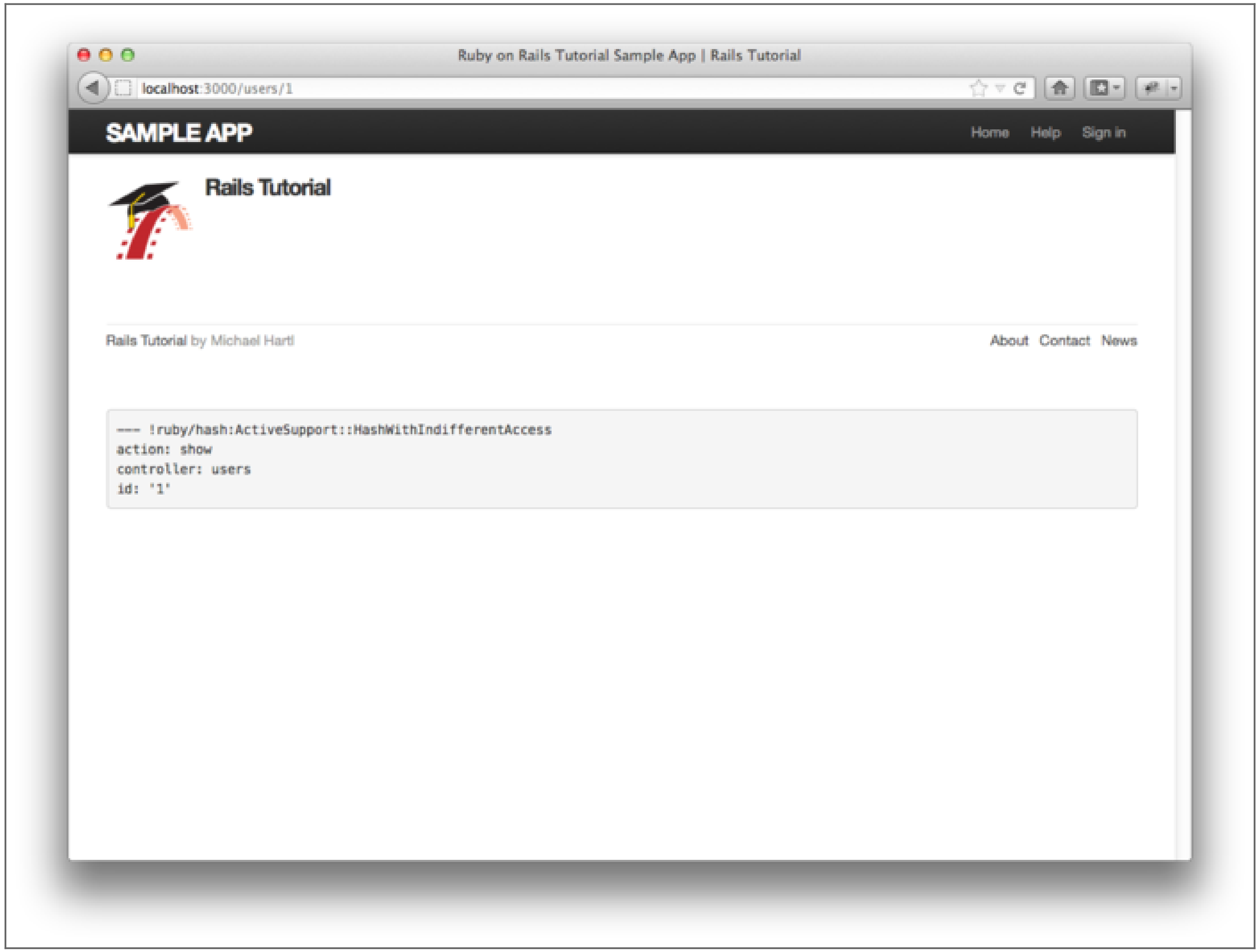


Welcome to the Sample App!

Rails Tutorial by Michael Hartl

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.20: The results of a successful user signup, with flash message. (full size)


Ruby on Rails Tutorial Sample App | Rails Tutorial
localhost:3000/users/1
SAMPLE APP
Home
Help
Sign in
Rails Tutorial
Rails Tutorial by Michael Hartl
About
Contact
News

```
--- !ruby/hash:ActiveSupport::HashWithIndifferentAccess
action: show
controller: users
id: '1'
```

Figure 7.21: The flash-less profile page after a browser reload. (full size)

We can now check our database just to be double-sure that the new user was actually created:

```
$ rails console
>> User.find_by_email("example@railstutorial.org")
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.org",
created_at: "2011-12-13 05:51:34", updated_at: "2011-12-13 05:51:34",
password_digest: "$2a$10$A58/j7wwh3aAffGkMAO9Q.jjh3jshd.6akhDKtchAz/R...">
```

7.4.4 Deploying to production with SSL

Having developed the User model and the signup functionality, now is a good time to deploy the sample application to production. (If you didn't follow the setup steps in the introduction to Chapter 3, you should go back and do them now.) As part of this, we will add Secure Sockets Layer (SSL)[12] to the production application, thereby making signup secure. Since we'll implement SSL site-wide, the sample application will also be secure during user signin (Chapter 8) and will also be immune to the *session hijacking* vulnerability (Section 8.2.2).

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

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

To get the deployment to work, we first need to add a line forcing the use of SSL in production. The result, which involves editing the production configuration file **`config/environments/production.rb`**, appears in Listing 7.28.

Listing 7.28. Configuring the application to use SSL in production.

config/environments/production.rb

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

To get the production site working, we have to commit the change to the configuration file and push the result up to Heroku:

```
$ git commit -a -m "Add SSL in production"
$ git push heroku
```

Next, we need to run the migration on the production database to tell Heroku about the User data model:[13]

```
$ heroku run rake db:migrate
```

(You might see some deprecation warnings at this point, which you should ignore.)

Finally, we need to set up SSL on the remote server. Configuring a production site to use SSL is painful and error-prone, and among other things it involves purchasing an *SSL certificate* for your domain. Luckily, for an application running on a Heroku domain (such as the sample application), we can piggyback on Heroku's SSL certificate, a feature that is included automatically as part of the

Heroku platform. If you want to run SSL on a custom domain, such as **`example.com`**, you'll have no choice but to endure some pain, which you can read about on Heroku's page on SSL.

The result of all this work is a working signup form on the production server (Figure 7.22):

```
$ heroku open
```

Note in Figure 7.22 the `https://` in place of the usual `http://`. The extra 's' is an indication that SSL is working.

You should feel free to visit the signup page and create a new user at this time. If you have trouble, try running

```
$ heroku logs
```

to debug the error using the Heroku logfile.

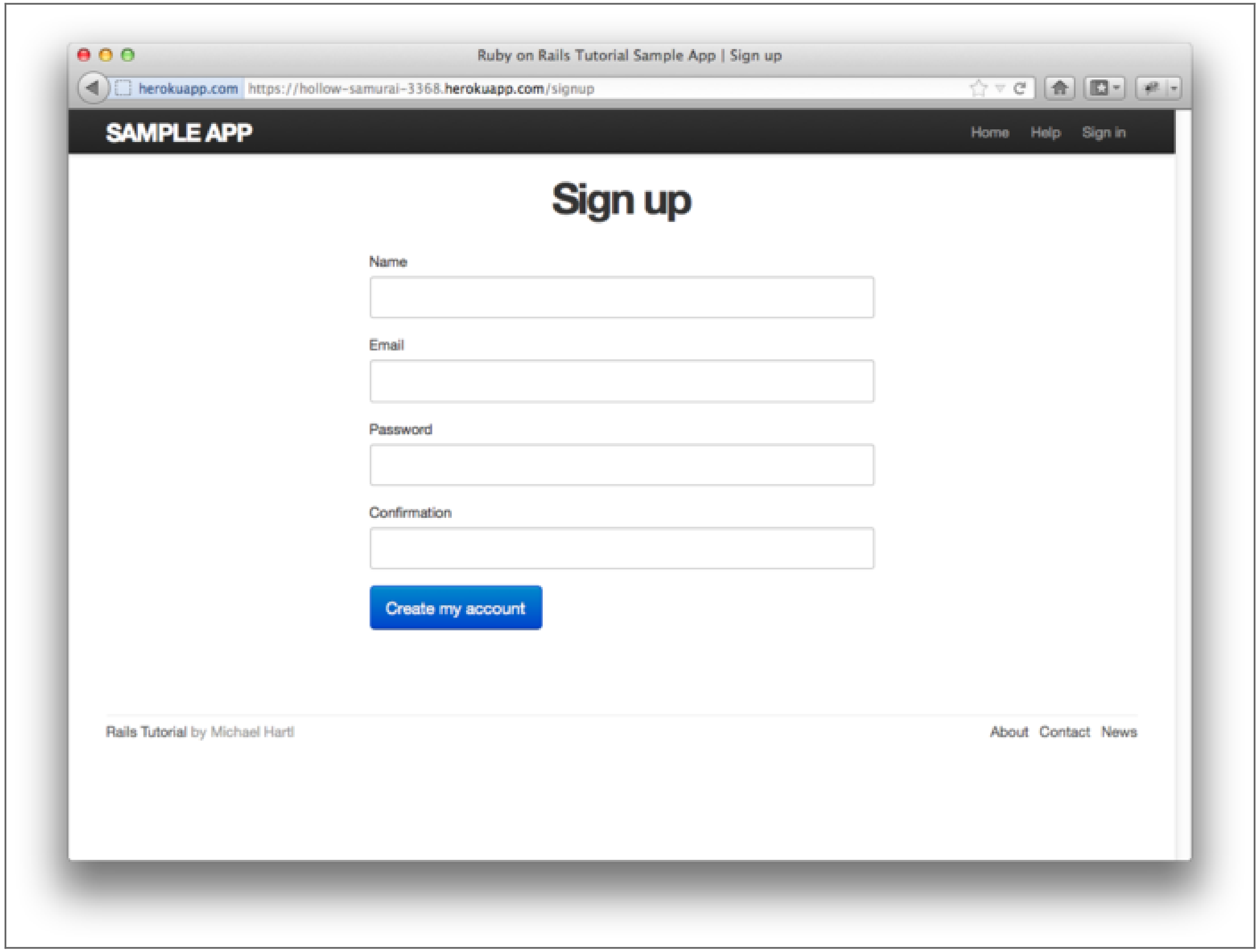

Ruby on Rails Tutorial Sample App | Sign up
herokuapp.com https://hollow-samurai-3368.herokuapp.com/signup
SAMPLE APP
Home
Help
Sign in
Sign up
Name
Email
Password
Confirmation
Create my account
Rails Tutorial by Michael Hartl
About
Contact
News

Figure 7.22: A working signup page on the live Web. (full size)

7.5 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, we will complete our authentication machinery by allowing users to sign in and out of the application. In Chapter 9, we will allow all users to update their account information, and will allow site administrators to delete users, thereby completing the full suite of the Users resource REST actions from Table 7.1. Finally, we'll add authorization methods to our actions to enforce a site security model.

7.6 Exercises

1. Verify that the code in Listing 7.29 allows the `gravatar_for` helper defined in Section 7.1.4 to take an optional `size` parameter, allowing code like `gravatar_for user, size: 40` in the view.

2. Write tests for the error messages implemented in Listing 7.22. A suggested start appears in Listing 7.31.

3. Using the code in Listing 7.30, replace the error message for a missing password, currently "Password digest can't be blank", with the more understandable "Password can't be blank". (This uses Rails' internationalization support to produce a functional but rather hacky solution. Take special care to use spaces and not tab characters; the YAML format forbids tabs.) Note that, to avoid duplication of error messages, you should also remove the password's `presence: true` validation in the User model.

4. By writing the test first or by intentionally breaking and then fixing the application code, verify that the tests in Listing 7.32 correctly specify the desired behavior after saving the user in the `create` action.

5. As noted before, the flash HTML in Listing 7.26 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.

Listing 7.29. Defining an optional `:size` parameter for the `gravatar_for` helper.
`app/helpers/users_helper.rb`

```ruby
module UsersHelper

  # Returns the Gravatar (http://gravatar.com/) for the given user.
  def gravatar_for(user, options = { size: 50 })
    gravatar_id = Digest::MD5::hexdigest(user.email.downcase)
    size = options[:size]
    gravatar_url = "https://secure.gravatar.com/avatar/#{gravatar_id}?s=#{size}"
    image_tag(gravatar_url, alt: user.name, class: "gravatar")
  end
end
```

Listing 7.30. Hacking a better error message for missing passwords.
config/locales/en.yml

```yaml
en:
  activerecord:
    attributes:
      user:
        password_digest: "Password"
```

Listing 7.31. Suggested error messages tests.
spec/requests/user_pages_spec.rb

```ruby
  .
  .
  .
  describe "signup" do

    before { visit signup_path }
    .
    .
    .
    describe "with invalid information" do
      .
      .
      .
      describe "after submission" do
```

```
        before { click_button submit }

        it { should have_selector('title', text: 'Sign up') }
        it { should have_content('error') }
      end
      .
      .
      .
```

Listing 7.32. Tests for the post-save behavior in the **`create`** action.

`spec/requests/user_pages_spec.rb`

```
    .
    .
    .
    describe "with valid information" do
      .
      .
      .
      describe "after saving the user" do
        before { click_button submit }
        let(:user) { User.find_by_email('user@example.com') }

        it { should have_selector('title', text: user.name) }
        it { should have_selector('div.alert.alert-success', text: 'Welcome') }
      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 |key, value| %>
        <%= content_tag(:div, value, class: "alert alert-#{key}") %>
      <% end %>
      .
      .
      .
</html>
```

[« Chapter 6 Modeling users](#) [Chapter 8 Sign in, sign out »](#)

---

1. Mockingbird doesn't support custom images like the profile photo in Figure 7.1; I put that in by hand using Adobe Fireworks. ↑
2. The hippo here is from http://www.flickr.com/photos/43803060@N00/24308857/. ↑
3. 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. ↑
4. You can define your own custom environments as well; see the RailsCast on adding an environment for details. ↑
5. 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 URI will return an error. ↑
6. Presumably "Factory Girl" is a reference to the movie of the same name. ↑
7. 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. But you've seen the movie, so you already knew this. ↑
8. If your application does need to handle custom images or other file uploads, I recommend the

Paperclip gem. ↑

9. Weird, right? I don't get it either. ↑
10. I figured this out by looking up `pluralize` in the Rails API. ↑
11. Actually, we'll use the closely related `flash.now`, but we'll defer that subtlety until we need it. ↑
12. Technically, SSL is now TLS, for Transport Layer Security, but everyone I know still says "SSL". ↑
13. Readers interested in using Heroku for real-life production applications might be interested in Kumade, which handles things like database migrations automatically. ↑