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 2
A demo app

In this chapter, we'll develop a simple demonstration application to show off some of the power of Rails. The purpose is to get a high-level overview of Ruby on Rails programming (and web development in general) by rapidly generating an application using *scaffold generators*. As discussed in Box 1.1, the rest of the book will take the opposite approach, developing a full application incrementally and explaining each new concept as it arises, but for a quick overview (and some instant gratification) there is no substitute for scaffolding. The resulting demo app will allow us to interact with it through its URIs, giving us insight into the structure of a Rails application, including a first example of the *REST architecture* favored by Rails.

As with the forthcoming sample application, the demo app will consist of *users* and their associated *microposts* (thus constituting a minimalist Twitter-style app). The functionality will be utterly under-developed, and many of the steps will seem like magic, but worry not: the full sample app will develop a similar application from the ground up starting in Chapter 3, and I will provide plentiful forward-references to later material. In the mean time, have patience and a little faith—the whole point of this tutorial is to take you *beyond* this superficial, scaffold-driven approach to achieve a deeper understanding of Rails.

2.1 Planning the application

In this section, we'll outline our plans for the demo application. As in Section 1.2.3, we'll start by generating the application skeleton using the `rails` command:

```
$ cd ~/rails_projects
$ rails new demo_app
$ cd demo_app
```

Next, we'll use a text editor to update the **`Gemfile`** needed by Bundler with the contents of Listing 2.1.

Listing 2.1. A **`Gemfile`** for the demo app.

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

gem 'rails', '3.2.8'

group :development do
  gem 'sqlite3', '1.3.5'
end


# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '3.2.5'
  gem 'coffee-rails', '3.2.2'

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

group :production do
  gem 'pg', '0.12.2'
end
```

Note that Listing 2.1 is identical to Listing 1.5 except for the addition of a gem needed in production at Heroku:

```
group :production do
  gem 'pg', '0.12.2'
end
```

The **pg** gem is needed to access PostgreSQL ("post-gres-cue-ell"), the database used by Heroku.

We then install and include the gems using the **bundle install** command:

```
$ bundle install --without production
```

The `--without production` option prevents the installation of the production gems, which in this case is just the PostgreSQL gem `pg`. (If Bundler complains about

```
no such file to load -- readline (LoadError)
```

try adding **gem 'rb-readline'** to your **Gemfile**.)

Finally, we'll put the demo app under version control. Recall that the **rails** command generates a default **.gitignore** file, but depending on your system you may find the augmented file from Listing 1.7 to be more convenient. Then initialize a Git repository and make the first commit:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

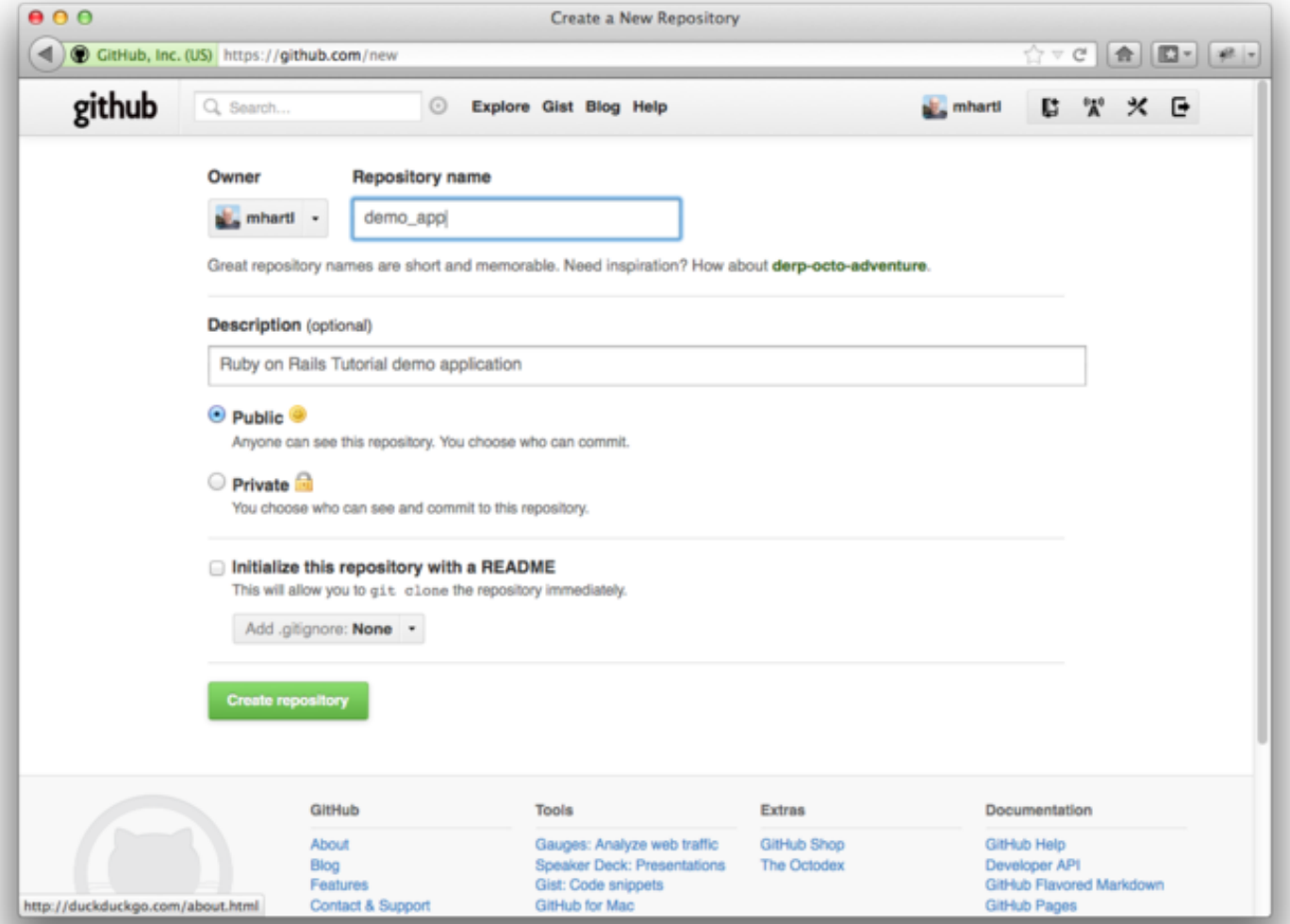

Figure 2.1: Creating a demo app repository at GitHub. (full size)

You can also optionally create a new repository (Figure 2.1) and push it up to GitHub:

```
$ git remote add origin git@github.com:<username>/demo_app.git
```

```
$ git push -u origin master
```

(As with the first app, take care *not* to initialize the GitHub repository with a **`README`** file.)

Now we're ready to start making the app itself. The typical first step when making a web application is to create a *data model*, which is a representation of the structures needed by our application. In our case, the demo app will be a microblog, with only users and short (micro)posts. Thus, we'll begin with a model for *users* of the app (Section 2.1.1), and then we'll add a model for *microposts* (Section 2.1.2).

2.1.1 Modeling demo users

There are as many choices for a user data model as there are different registration forms on the web; we'll go with a distinctly minimalist approach. Users of our demo app will have a unique **`integer`** identifier called **`id`**, a publicly viewable **`name`** (of type **`string`**), and an **`email`** address (also a **`string`**) that will double as a username. A summary of the data model for users appears in Figure 2.2.

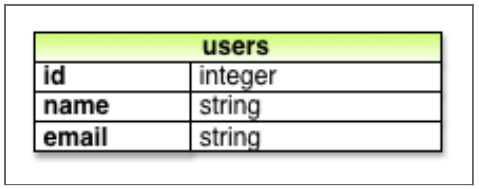

Figure 2.2: The data model for users.

As we'll see starting in Section 6.1.1, the label **`users`** in Figure 2.2 corresponds to a *table* in a database, and the **`id`**, **`name`**, and **`email`** attributes are *columns* in that table.

2.1.2 Modeling demo microposts

The core of the micropost data model is even simpler than the one for users: a micropost has only an `id` and a `content` field for the micropost's text (of type `string`).[1] There's an additional complication, though: we want to *associate* each micropost with a particular user; we'll accomplish this by recording the `user_id` of the owner of the post. The results are shown in Figure 2.3.

microposts	
id	integer
content	string
user_id	integer

Figure 2.3: The data model for microposts.

We'll see in Section 2.3.3 (and more fully in Chapter 10) how this `user_id` attribute allows us to succinctly express the notion that a user potentially has many associated microposts.

2.2 The Users resource

In this section, we'll implement the users data model in Section 2.1.1, along with a web interface to that model. The combination will constitute a *Users resource*, which will allow us to think of users as objects that can be created, read, updated, and deleted through the web via the HTTP protocol. As promised in the introduction, our Users resource will be created by a scaffold generator program, which comes standard with each Rails project. I urge you not to look too closely at the generated code; at this stage, it will only serve to confuse you.

Rails scaffolding is generated by passing the `scaffold` command to the `rails generate` script. The argument of the `scaffold` command is the singular version of the resource name (in this

case, **User**), together with optional parameters for the data model's attributes:[2]

```
$ rails generate scaffold User name:string email:string
      invoke  active_record
      create    db/migrate/20111123225336_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/unit/user_test.rb
      create      test/fixtures/users.yml
       route  resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/functional/users_controller_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      test_unit
      create        test/unit/helpers/users_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

By including **name:string** and **email:string**, we have arranged for the User model to have the form shown in Figure 2.2. (Note that there is no need to include a parameter for **id**; it is created automatically by Rails for use as the *primary key* in the database.)

To proceed with the demo application, we first need to *migrate* the database using *Rake* (Box 2.1):

```
$ bundle exec rake db:migrate
==  CreateUsers: migrating ==================================================
-- create_table(:users)
   -> 0.0017s
==  CreateUsers: migrated (0.0018s) =========================================
```

This simply updates the database with our new **`users`** data model. (We'll learn more about database migrations starting in Section 6.1.1.) Note that, in order to ensure that the command uses the version of Rake corresponding to our **`Gemfile`**, we need to run **`rake`** using **`bundle exec`**.

With that, we can run the local web server using **`rails s`**, which is a shortcut for **`rails server`**:

```
$ rails s
```

Now the demo application should be ready to go at http://localhost:3000/.

Box 2.1. Rake

In the Unix tradition, the *make* utility has played an important role in building executable programs from source code; many a computer hacker has committed to muscle memory the line

```
$ ./configure && make && sudo make install
```

commonly used to compile code on Unix systems (including Linux and Mac OS X).

Rake is *Ruby make*, a make-like language written in Ruby. Rails uses Rake extensively,

especially for the innumerable little administrative tasks necessary when developing database-backed web applications. The `rake db:migrate` command is probably the most common, but there are many others; you can see a list of database tasks using `-T db`:

```
$ bundle exec rake -T db
```

To see all the Rake tasks available, run

```
$ bundle exec rake -T
```

The list is likely to be overwhelming, but don't worry, you don't have to know all (or even most) of these commands. By the end of the *Rails Tutorial*, you'll know all the most important ones.

2.2.1 A user tour

Visiting the root url http://localhost:3000/ shows the same default Rails page shown in Figure 1.3, but in generating the Users resource scaffolding we have also created a large number of pages for manipulating users. For example, the page for listing all users is at /users, and the page for making a new user is at /users/new. The rest of this section is dedicated to taking a whirlwind tour through these user pages. As we proceed, it may help to refer to Table 2.1, which shows the correspondence between pages and URIs.

URI	Action	Purpose
/users	`index`	page to list all users
/users/1	`show`	page to show user with id `1`

/users/new	`new`	page to make a new user
/users/1/edit	`edit`	page to edit user with id `1`

Table 2.1: The correspondence between pages and URIs for the Users resource.

We start with the page to show all the users in our application, called `index`; as you might expect, initially there are no users at all (Figure 2.4).

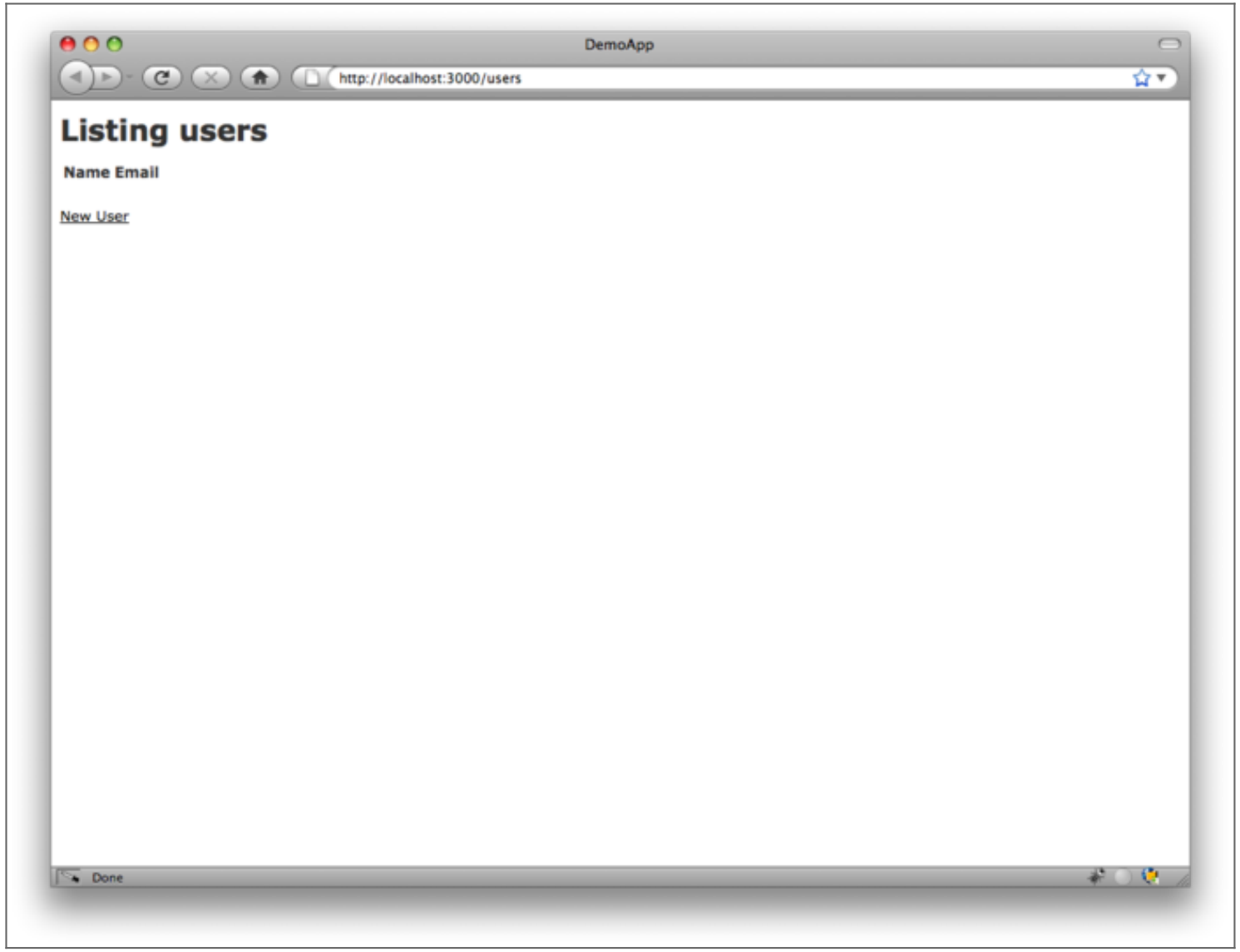

Figure 2.4: The initial index page for the Users resource (/users). (full size)

To make a new user, we visit the `new` page, as shown in Figure 2.5. (Since the http://localhost:3000 part of the address is implicit whenever we are developing locally, I'll usually omit it from now on.) In Chapter 7, this will become the user signup page.

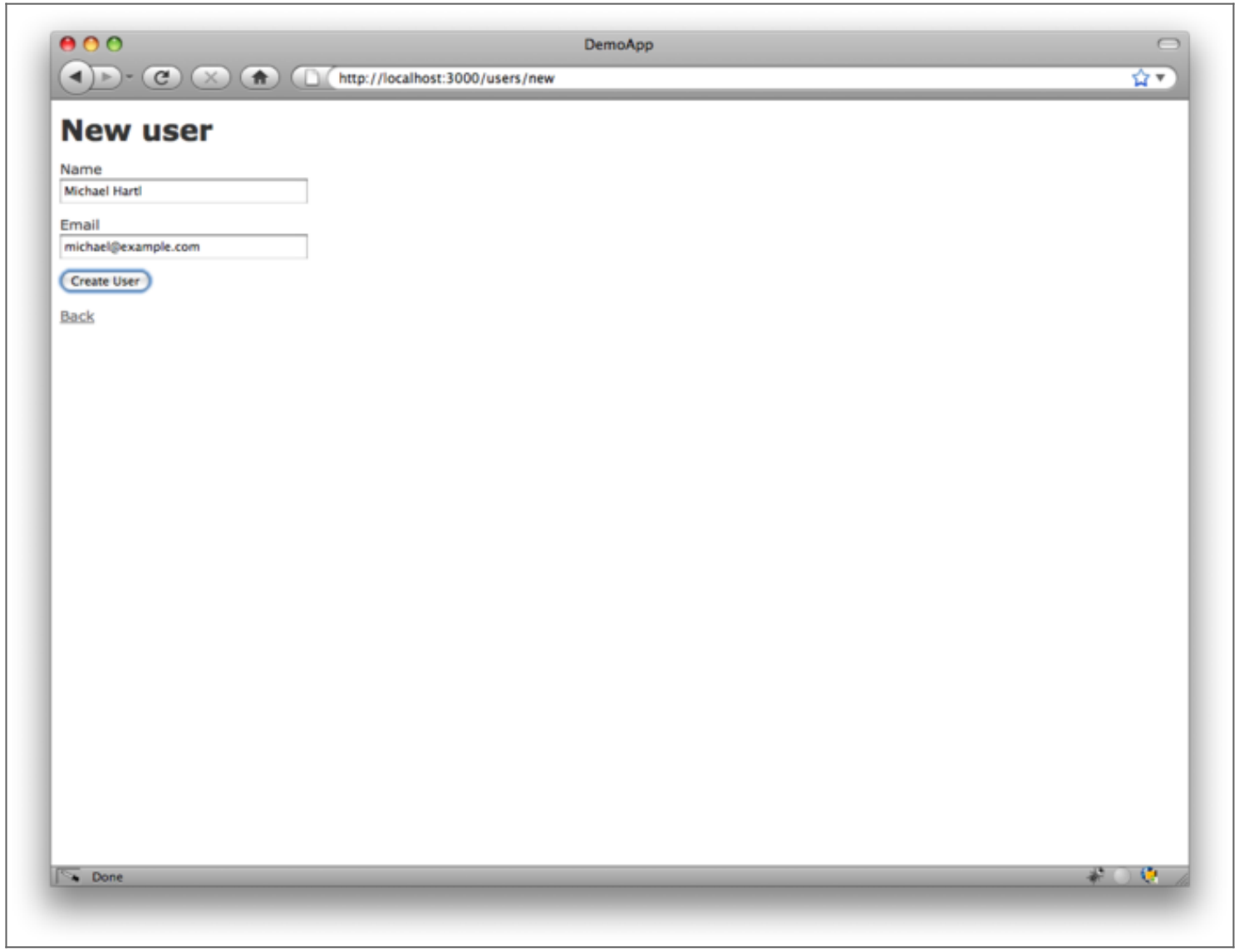

Figure 2.5: The new user page (/users/new). (full size)

We can create a user by entering name and email values in the text fields and then clicking the Create User button. The result is the user `show` page, as seen in Figure 2.6. (The green welcome

message is accomplished using the *flash*, which we'll learn about in Section 7.4.2.) Note that the URI is /users/1; as you might suspect, the number `1` is simply the user's `id` attribute from Figure 2.2. In Section 7.1, this page will become the user's profile.

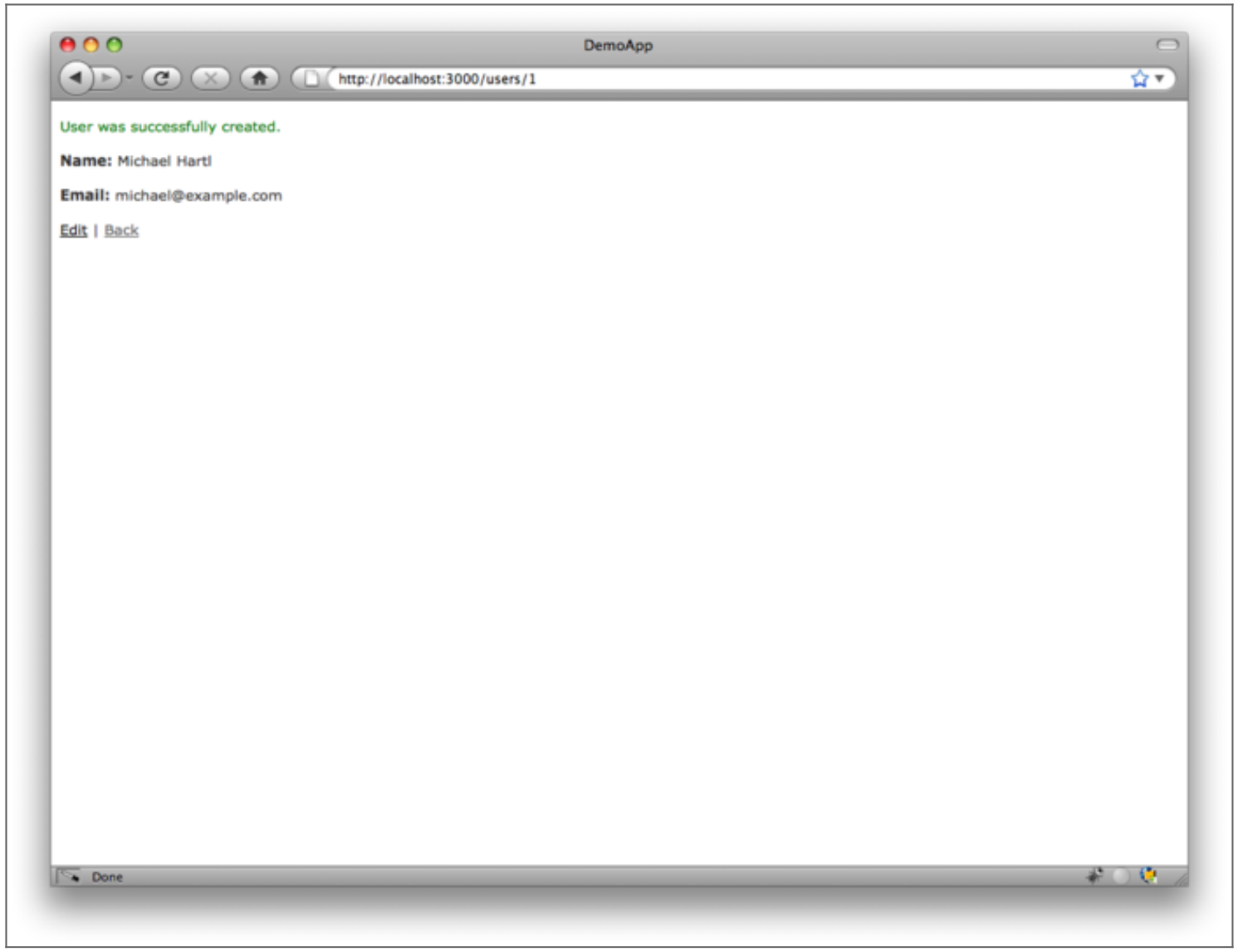

Figure 2.6: The page to show a user (/users/1). (full size)

To change a user's information, we visit the `edit` page (Figure 2.7). By modifying the user information and clicking the Update User button, we arrange to change the information for the user in the demo application (Figure 2.8). (As we'll see in detail starting in Chapter 6, this user data is stored in a database back-end.) We'll add user edit/update functionality to the sample application in Section 9.1.

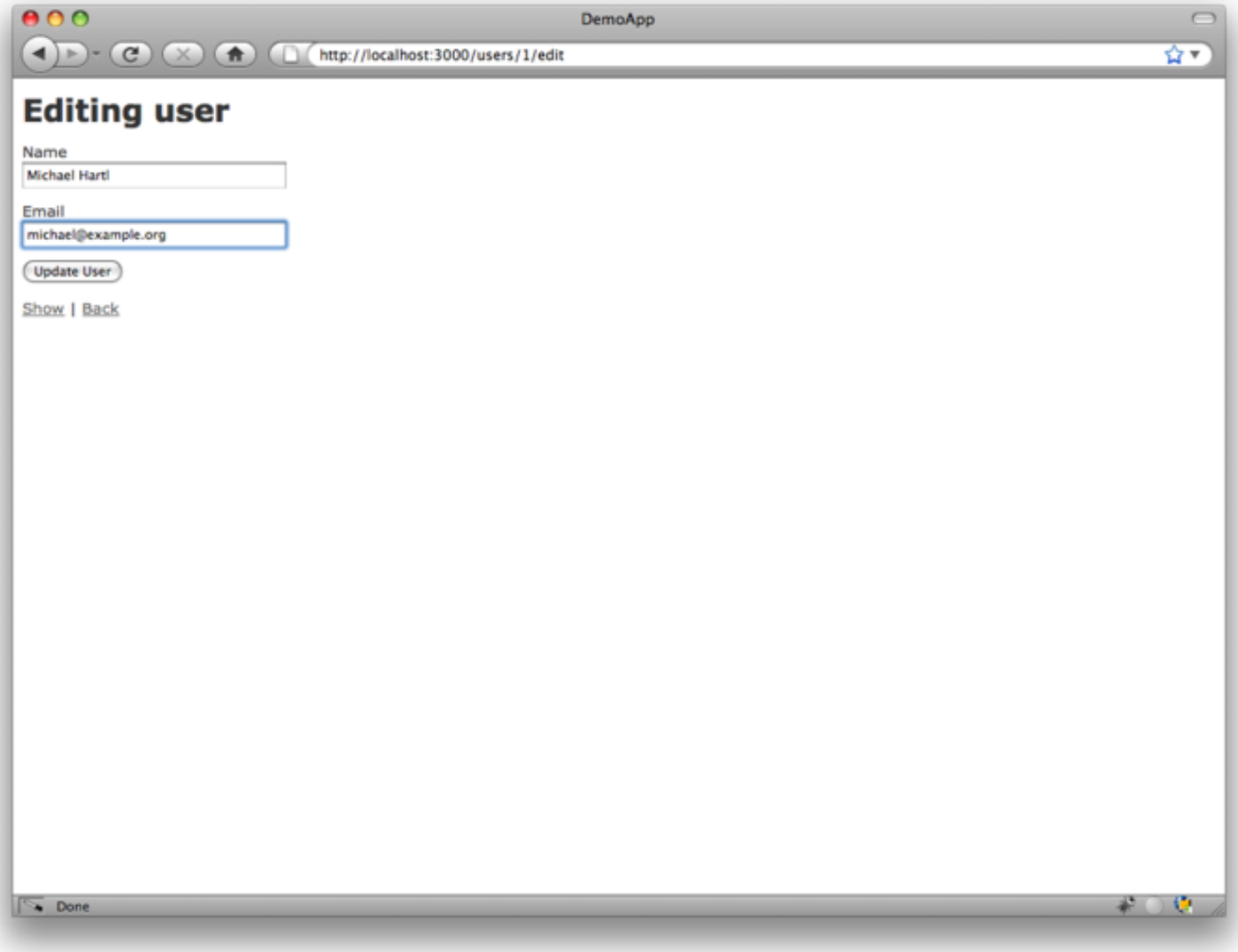

Figure 2.7: The user edit page (/users/1/edit). (full size)

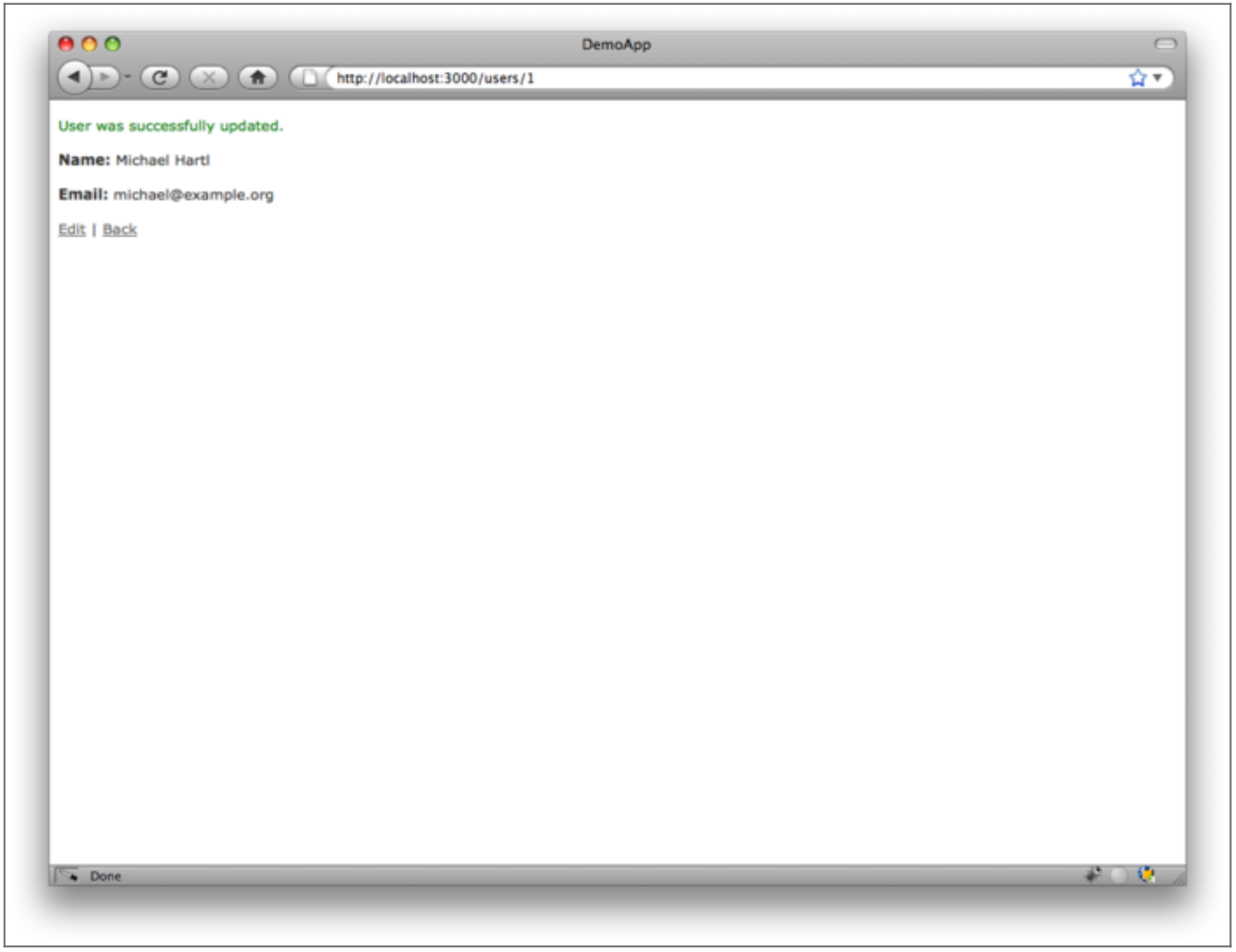

Figure 2.8: A user with updated information. (full size)

Now we'll create a second user by revisiting the `new` page and submitting a second set of user information; the resulting user `index` is shown in Figure 2.9. Section 7.1 will develop the user index into a more polished page for showing all users.

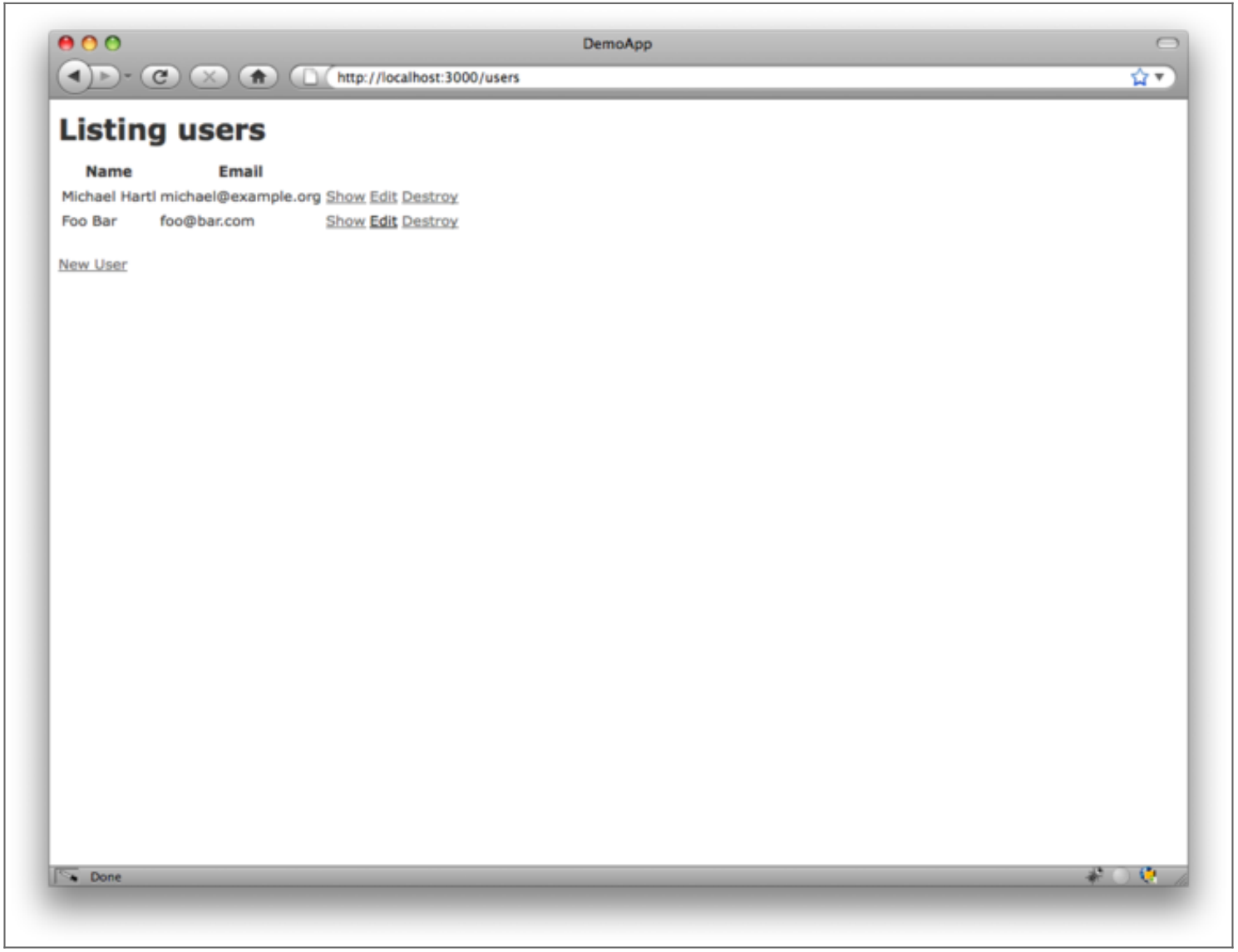

Figure 2.9: The user index page (/users) with a second user. (full size)

Having shown how to create, show, and edit users, we come finally to destroying them (Figure 2.10). You should verify that clicking on the link in Figure 2.10 destroys the second user, yielding an index page with only one user. (If it doesn't work, be sure that JavaScript is enabled in your browser; Rails uses JavaScript to issue the request needed to destroy a user.) Section 9.4 adds user deletion to the sample app, taking care to restrict its use to a special class of administrative users.

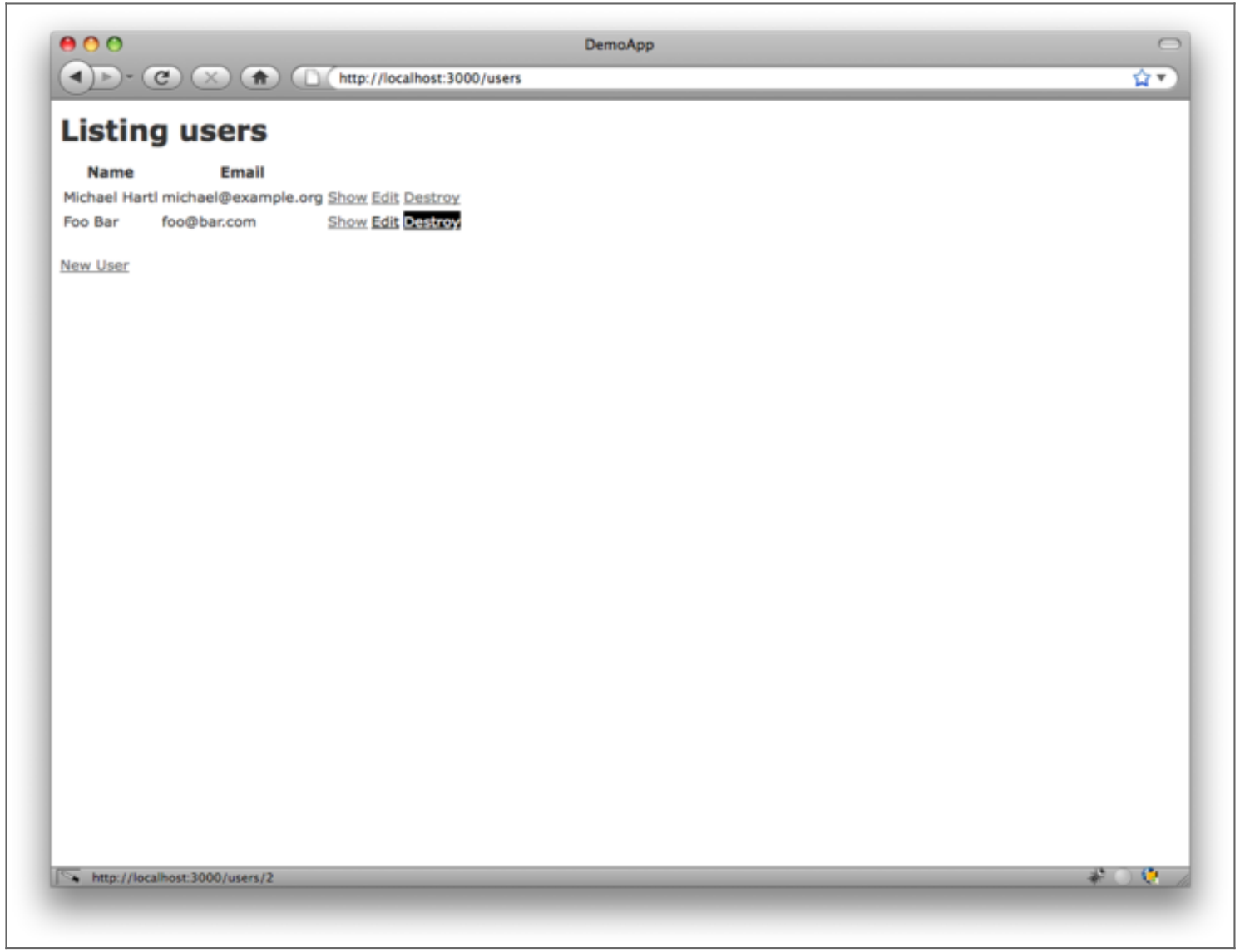

Figure 2.10: Destroying a user. (full size)

2.2.2 MVC in action

Now that we've completed a quick overview of the Users resource, let's examine one particular part

of it in the context of the Model-View-Controller (MVC) pattern introduced in Section 1.2.6. Our strategy will be to describe the results of a typical browser hit—a visit to the user index page at /users—in terms of MVC (Figure 2.11).

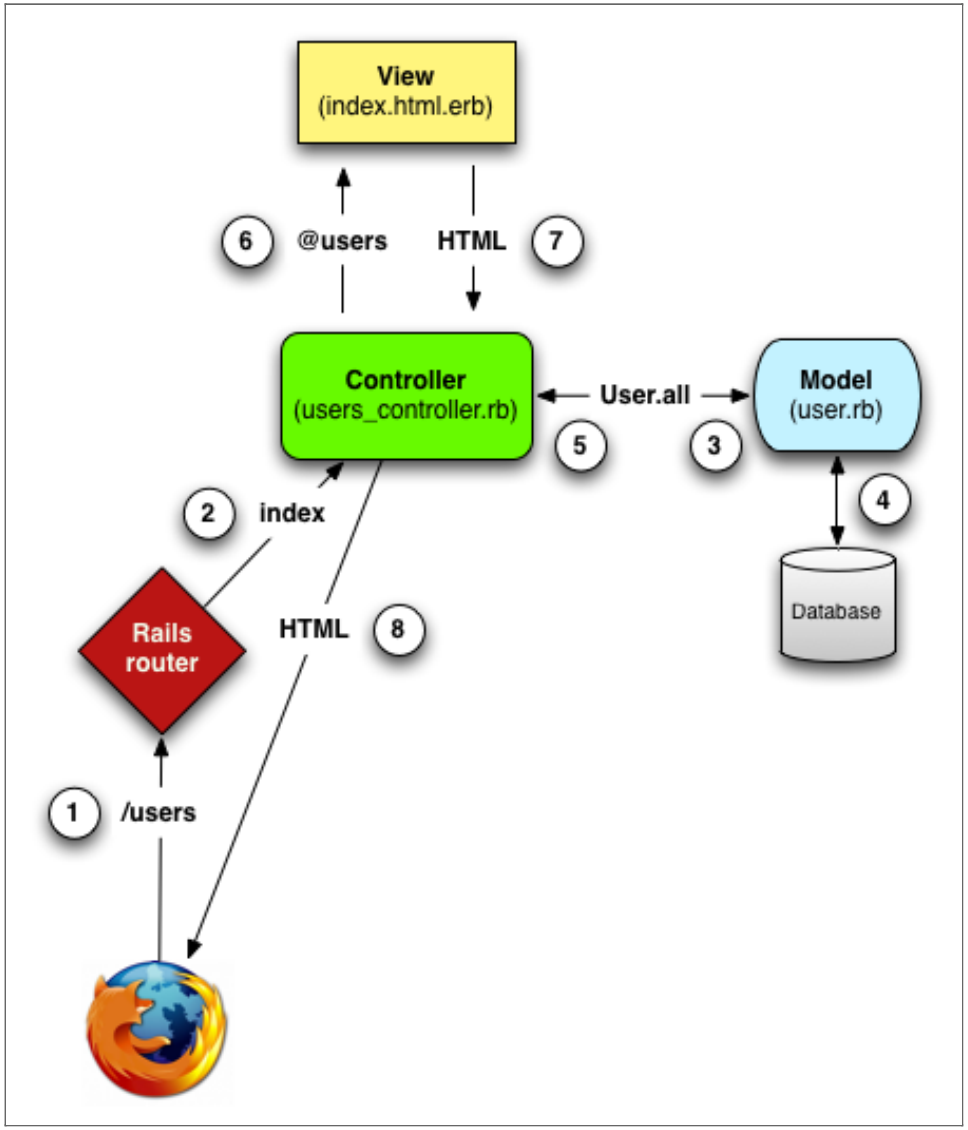

Figure 2.11: A detailed diagram of MVC in Rails. (full size)

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

We start with a request issued from the browser—i.e., the result of typing a URI in the address bar or clicking on a link (Step 1 in Figure 2.11). This request hits the *Rails router* (Step 2), which dispatches to the proper *controller action* based on the URI (and, as we'll see in Box 3.2, the type of request). The code to create the mapping of user URIs to controller actions for the Users resource appears in Listing 2.2; this code effectively sets up the table of URI/action pairs seen in Table 2.1. (The strange notation `:users` is a *symbol*, which we'll learn about in Section 4.3.3.)

Listing 2.2. The Rails routes, with a rule for the Users resource.

`config/routes.rb`

```
DemoApp::Application.routes.draw do
  resources :users
  .
  .
  .
end
```

The pages from the tour in Section 2.2.1 correspond to *actions* in the Users *controller*, which is a collection of related actions; the controller generated by the scaffolding is shown schematically in Listing 2.3. Note the notation `class UsersController < ApplicationController`; this is an example of a Ruby *class* with *inheritance*. (We'll discuss inheritance briefly in Section 2.3.4 and cover both subjects in more detail in Section 4.4.)

Listing 2.3. The Users controller in schematic form.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

  def index
    .
    .
    .
  end

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def edit
    .
    .
    .
```

```
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

You may notice that there are more actions than there are pages; the **`index`**, **`show`**, **`new`**, and **`edit`** actions all correspond to pages from Section 2.2.1, but there are additional **`create`**, **`update`**, and **`destroy`** actions as well. These actions don't typically render pages (although they sometimes do); instead, their main purpose is to modify information about users in the database. This full suite of controller actions, summarized in Table 2.2, represents the implementation of the REST architecture in Rails (Box 2.2), which is based on the ideas of *representational state transfer* identified and named by computer scientist Roy Fielding.[4] Note from Table 2.2 that there is some overlap in the URIs; for example, both the user **`show`** action and the **`update`** action correspond to the URI /users/1. The difference between them is the HTTP request method they respond to. We'll learn more about HTTP request methods starting in Section 3.2.1.

HTTP request	URI	Action	Purpose
GET	/users	**`index`**	page to list all users
GET	/users/1	**`show`**	page to show user with id **`1`**
GET	/users/new	**`new`**	page to make a new user

POST	/users	`create`	create a new user
GET	/users/1/edit	`edit`	page to edit user with id `1`
PUT	/users/1	`update`	update user with id `1`
DELETE	/users/1	`destroy`	delete user with id `1`

Table 2.2: RESTful routes provided by the Users resource in Listing 2.2.

Box 2.2. REpresentational State Transfer (REST)

If you read much about Ruby on Rails web development, you'll see a lot of references to "REST", which is an acronym for REpresentational State Transfer. REST is an architectural style for developing distributed, networked systems and software applications such as the World Wide Web and web applications. Although REST theory is rather abstract, in the context of Rails applications REST means that most application components (such as users and microposts) are modeled as *resources* that can be created, read, updated, and deleted—operations that correspond both to the CRUD operations of relational databases and the four fundamental HTTP request methods: `POST`, `GET`, `PUT`, and `DELETE`. (We'll learn more about HTTP requests in Section 3.2.1 and especially Box 3.2.)

As a Rails application developer, the RESTful style of development helps you make choices about which controllers and actions to write: you simply structure the application using resources that get created, read, updated, and deleted. In the case of users and microposts, this process is straightforward, since they are naturally resources in their own right. In Chapter 11, we'll see an example where REST principles allow us to model a subtler problem, "following users", in a natural and convenient way.

To examine the relationship between the Users controller and the User model, let's focus on a simplified version of the `index` action, shown in Listing 2.4. (The scaffold code is ugly and confusing, so I've suppressed it.)

Listing 2.4. The simplified user `index` action for the demo application.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def index
    @users = User.all
  end
  .
  .
  .
end
```

This `index` action has the line `@users = User.all` (Step 3), which asks the User model to retrieve a list of all the users from the database (Step 4), and then places them in the variable `@users` (pronounced "at-users") (Step 5). The User model itself appears in Listing 2.5; although it is rather plain, it comes equipped with a large amount of functionality because of inheritance (Section 2.3.4 and Section 4.4). In particular, by using the Rails library called *Active Record*, the code in Listing 2.5 arranges for `User.all` to return all the users. (We'll learn about the `attr_accessible` line in Section 6.1.2.2. *Note*: This line will not appear if you are using Rails 3.2.2 or earlier.)

Listing 2.5. The User model for the demo application.

app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
```

```
end
```

Once the `@users` variable is defined, the controller calls the *view* (Step 6), shown in Listing 2.6. Variables that start with the `@` sign, called *instance variables*, are automatically available in the view; in this case, the `index.html.erb` view in Listing 2.6 iterates through the `@users` list and outputs a line of HTML for each one. (Remember, you aren't supposed to understand this code right now. It is shown only for purposes of illustration.)

Listing 2.6. The view for the user index.
`app/views/users/index.html.erb`

```
<h1>Listing users</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Email</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.email %></td>
    <td><%= link_to 'Show', user %></td>
    <td><%= link_to 'Edit', edit_user_path(user) %></td>
    <td><%= link_to 'Destroy', user, method: :delete,
                                     data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New User', new_user_path %>
```

The view converts its contents to HTML (Step 7), which is then returned by the controller to the browser for display (Step 8).

2.2.3 Weaknesses of this Users resource

Though good for getting a general overview of Rails, the scaffold Users resource suffers from a number of severe weaknesses.

- **No data validations.** Our User model accepts data such as blank names and invalid email addresses without complaint.
- **No authentication.** We have no notion signing in or out, and no way to prevent any user from performing any operation.
- **No tests.** This isn't technically true—the scaffolding includes rudimentary tests—but the generated tests are ugly and inflexible, and they don't test for data validation, authentication, or any other custom requirements.
- **No layout.** There is no consistent site styling or navigation.
- **No real understanding.** If you understand the scaffold code, you probably shouldn't be reading this book.

2.3 The Microposts resource

Having generated and explored the Users resource, we turn now to the associated Microposts resource. Throughout this section, I recommend comparing the elements of the Microposts resource with the analogous user elements from Section 2.2; you should see that the two resources parallel each other in many ways. The RESTful structure of Rails applications is best absorbed by this sort of repetition of form; indeed, seeing the parallel structure of Users and Microposts even at this early stage is one of the prime motivations for this chapter. (As we'll see, writing applications more robust than the toy example in this chapter takes considerable effort—we won't see the Microposts resource again until Chapter 10—and I didn't want to defer its first appearance quite

that far.)

2.3.1 A micropost microtour

As with the Users resource, we'll generate scaffold code for the Microposts resource using **`rails generate scaffold`**, in this case implementing the data model from Figure 2.3:[5]

```
$ rails generate scaffold Micropost content:string user_id:integer
      invoke  active_record
      create    db/migrate/20111123225811_create_microposts.rb
      create    app/models/micropost.rb
      invoke    test_unit
      create      test/unit/micropost_test.rb
      create      test/fixtures/microposts.yml
       route  resources :microposts
      invoke  scaffold_controller
      create    app/controllers/microposts_controller.rb
      invoke    erb
      create      app/views/microposts
      create      app/views/microposts/index.html.erb
      create      app/views/microposts/edit.html.erb
      create      app/views/microposts/show.html.erb
      create      app/views/microposts/new.html.erb
      create      app/views/microposts/_form.html.erb
      invoke    test_unit
      create      test/functional/microposts_controller_test.rb
      invoke    helper
      create      app/helpers/microposts_helper.rb
      invoke      test_unit
      create        test/unit/helpers/microposts_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/microposts.js.coffee
      invoke    scss
      create      app/assets/stylesheets/microposts.css.scss
      invoke  scss
   identical    app/assets/stylesheets/scaffolds.css.scss
```

To update our database with the new data model, we need to run a migration as in Section 2.2:

```
$ bundle exec rake db:migrate
==  CreateMicroposts: migrating ==============================================
-- create_table(:microposts)
   -> 0.0023s
==  CreateMicroposts: migrated (0.0026s) =====================================
```

Now we are in a position to create microposts in the same way we created users in Section 2.2.1. As you might guess, the scaffold generator has updated the Rails routes file with a rule for Microposts resource, as seen in Listing 2.7.[6] As with users, the `resources :microposts` routing rule maps micropost URIs to actions in the Microposts controller, as seen in Table 2.3.

Listing 2.7. The Rails routes, with a new rule for Microposts resources.

`config/routes.rb`

```
DemoApp::Application.routes.draw do
  resources :microposts
  resources :users
  .
  .
  .
end
```

HTTP request	URI	Action	Purpose
GET	/microposts	`index`	page to list all microposts
GET	/microposts/1	`show`	page to show micropost with id `1`
GET	/microposts/new	`new`	page to make a new micropost

POST	/microposts	`create`	create a new micropost
GET	/microposts/1/edit	`edit`	page to edit micropost with id `1`
PUT	/microposts/1	`update`	update micropost with id `1`
DELETE	/microposts/1	`destroy`	delete micropost with id `1`

Table 2.3: RESTful routes provided by the Microposts resource in Listing 2.7.

The Microposts controller itself appears in schematic form Listing 2.8. Note that, apart from having `MicropostsController` in place of `UsersController`, Listing 2.8 is *identical* to the code in Listing 2.3. This is a reflection of the REST architecture common to both resources.

Listing 2.8. The Microposts controller in schematic form.
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController

  def index
    .
    .
    .
  end

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
```

```
  end

  def create
    .
    .
    .
  end

  def edit
    .
    .
    .
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

To make some actual microposts, we enter information at the new microposts page, /microposts/new, as seen in Figure 2.12.

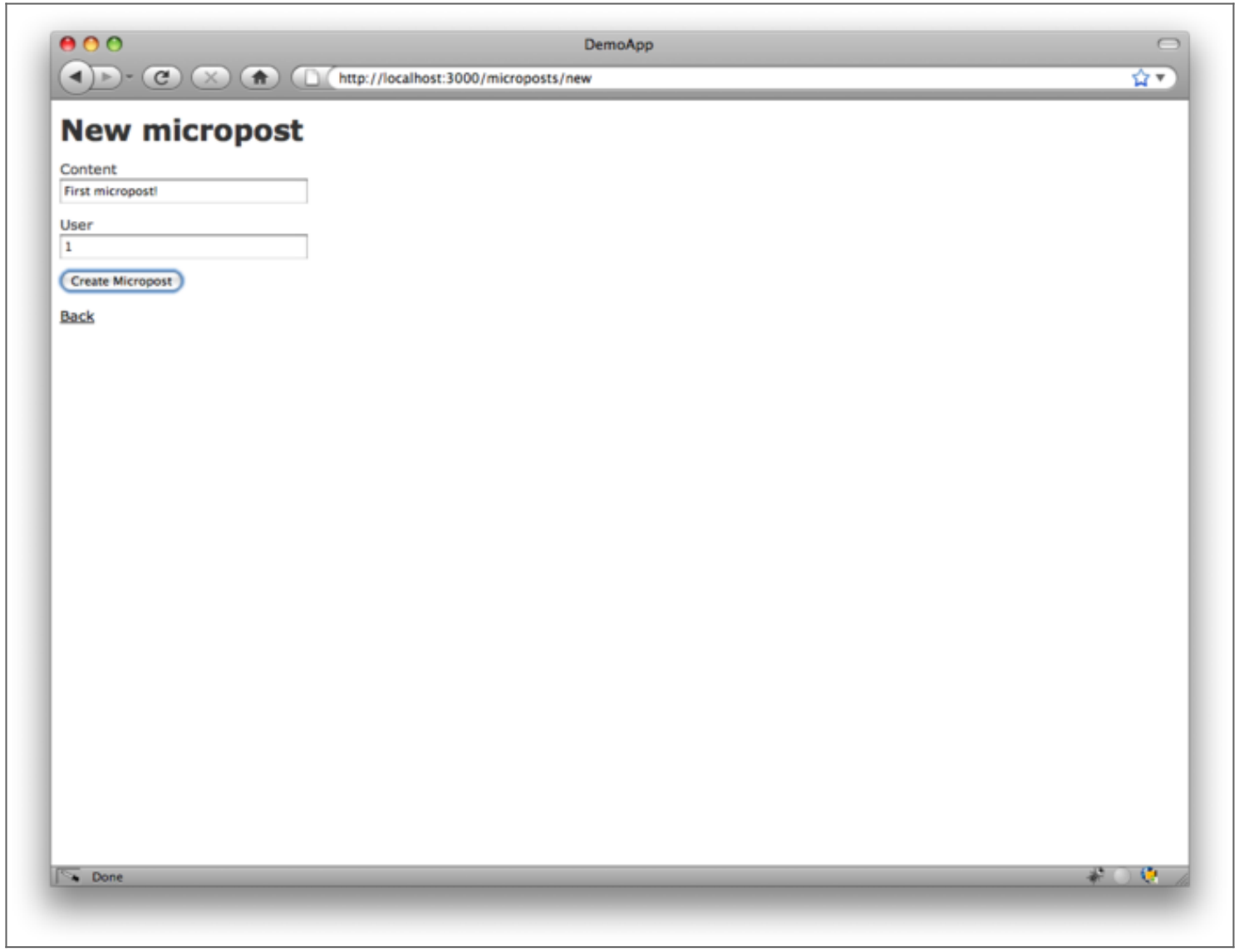

Figure 2.12: The new micropost page (/microposts/new). (full size)

At this point, go ahead and create a micropost or two, taking care to make sure that at least one has a `user_id` of `1` to match the id of the first user created in Section 2.2.1. The result should look something like Figure 2.13.

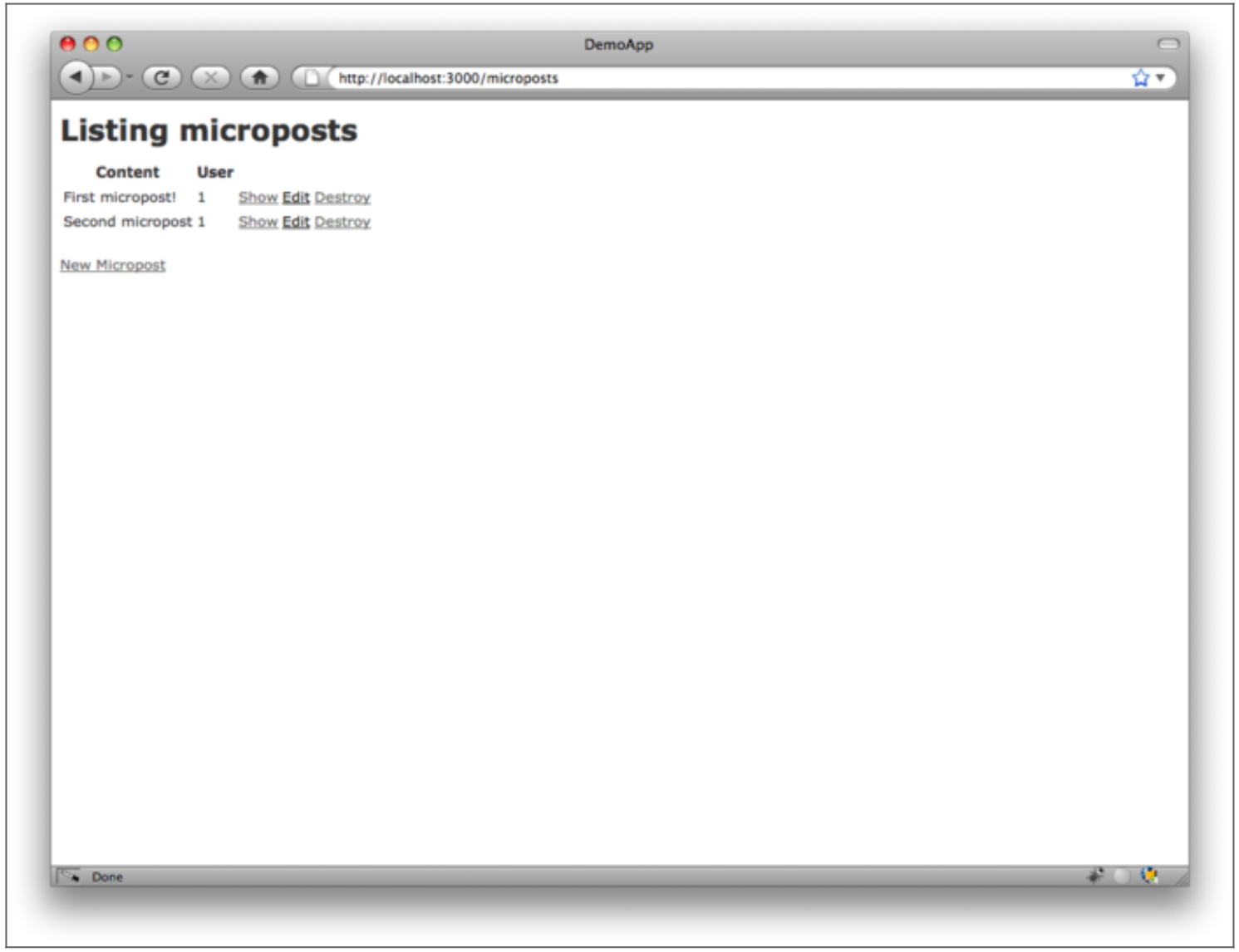

Figure 2.13: The micropost index page (/microposts). (full size)

2.3.2 Putting the *micro* in microposts

Any *micro*post worthy of the name should have some means of enforcing the length of the post. Implementing this constraint in Rails is easy with *validations*; to accept microposts with at most 140 characters (à la Twitter), we use a *length* validation. At this point, you should open the file **`app/models/micropost.rb`** in your text editor or IDE and fill it with the contents of Listing 2.9. (The use of **`validates`** in Listing 2.9 is characteristic of Rails 3; if you've previously worked with Rails 2.3, you should compare this to the use of **`validates_length_of`**.)

Listing 2.9. Constraining microposts to be at most 140 characters.
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  attr_accessible :content, :user_id
  validates :content, :length => { :maximum => 140 }
end
```

The code in Listing 2.9 may look rather mysterious—we'll cover validations more thoroughly starting in Section 6.2—but its effects are readily apparent if we go to the new micropost page and enter more than 140 characters for the content of the post. As seen in Figure 2.14, Rails renders *error messages* indicating that the micropost's content is too long. (We'll learn more about error messages in Section 7.3.2.)

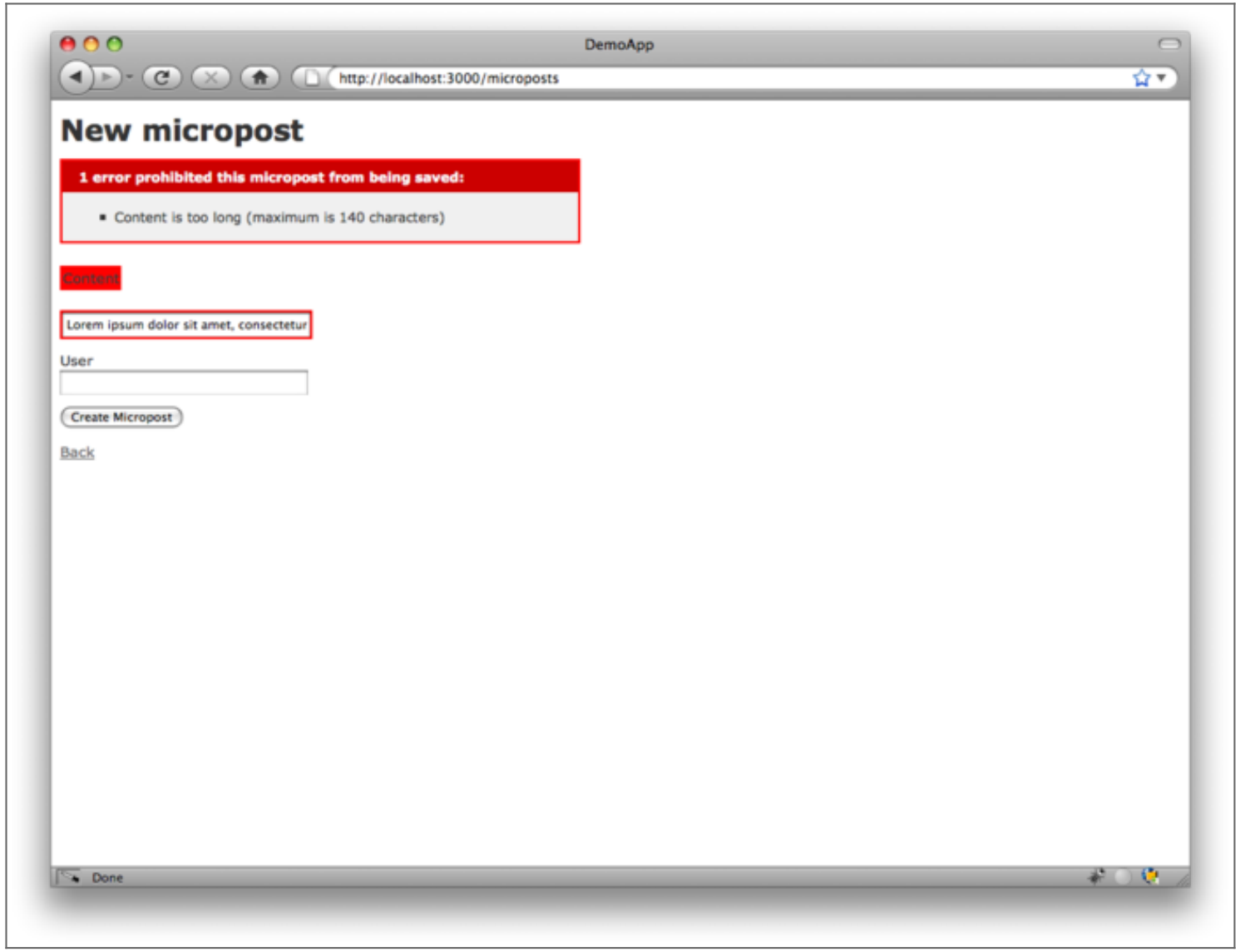

Figure 2.14: Error messages for a failed micropost creation. (full size)

2.3.3 A user `has_many` microposts

One of the most powerful features of Rails is the ability to form *associations* between different data

models. In the case of our User model, each user potentially has many microposts. We can express this in code by updating the User and Micropost models as in Listing 2.10 and Listing 2.11.

Listing 2.10. A user has many microposts.

app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
  has_many :microposts
end
```

Listing 2.11. A micropost belongs to a user.

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  attr_accessible :content, :user_id

  belongs_to :user

  validates :content, :length => { :maximum => 140 }
end
```

We can visualize the result of this association in Figure 2.15. Because of the `user_id` column in the `microposts` table, Rails (using Active Record) can infer the microposts associated with each user.

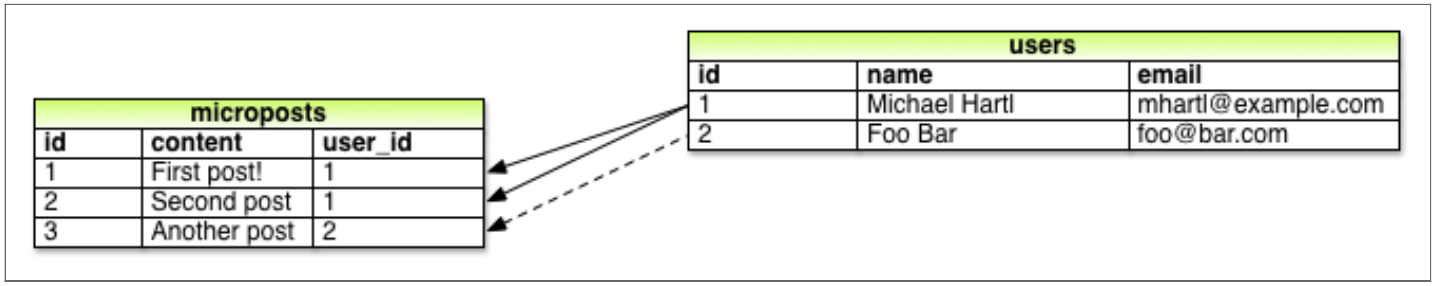

Figure 2.15: The association between microposts and users.

In Chapter 10 and Chapter 11, we will use the association of users and microposts both to display all a user's microposts and to construct a Twitter-like micropost feed. For now, we can examine the implications of the user-micropost association by using the *console*, which is a useful tool for interacting with Rails applications. We first invoke the console with `rails console` at the command line, and then retrieve the first user from the database using `User.first` (putting the results in the variable `first_user`):[7]

```
$ rails console
>> first_user = User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2011-11-03 02:01:31", updated_at: "2011-11-03 02:01:31">
>> first_user.microposts
=> [#<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2011-11-03 02:37:37", updated_at: "2011-11-03 02:37:37">, #<Micropost id: 2,
content: "Second micropost", user_id: 1, created_at: "2011-11-03 02:38:54",
updated_at: "2011-11-03 02:38:54">]
>> exit
```

(I include the last line just to demonstrate how to exit the console, and on most systems you can Ctrl-d for the same purpose.) Here we have accessed the user's microposts using the code `first_user.microposts`: with this code, Active Record automatically returns all the microposts with `user_id` equal to the id of `first_user` (in this case, `1`). We'll learn much more about the association facilities in Active Record in Chapter 10 and Chapter 11.

2.3.4 Inheritance hierarchies

We end our discussion of the demo application with a brief description of the controller and model class hierarchies in Rails. This discussion will only make much sense if you have some experience

with object-oriented programming (OOP); if you haven't studied OOP, feel free to skip this section. In particular, if you are unfamiliar with *classes* (discussed in Section 4.4), I suggest looping back to this section at a later time.

We start with the inheritance structure for models. Comparing Listing 2.12 and Listing 2.13, we see that both the User model and the Micropost model inherit (via the left angle bracket `<`) from `ActiveRecord::Base`, which is the base class for models provided by ActiveRecord; a diagram summarizing this relationship appears in Figure 2.16. It is by inheriting from `ActiveRecord::Base` that our model objects gain the ability to communicate with the database, treat the database columns as Ruby attributes, and so on.

Listing 2.12. The `User` class, with inheritance.

`app/models/user.rb`

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

Listing 2.13. The `Micropost` class, with inheritance.

`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  .
  .
  .
end
```

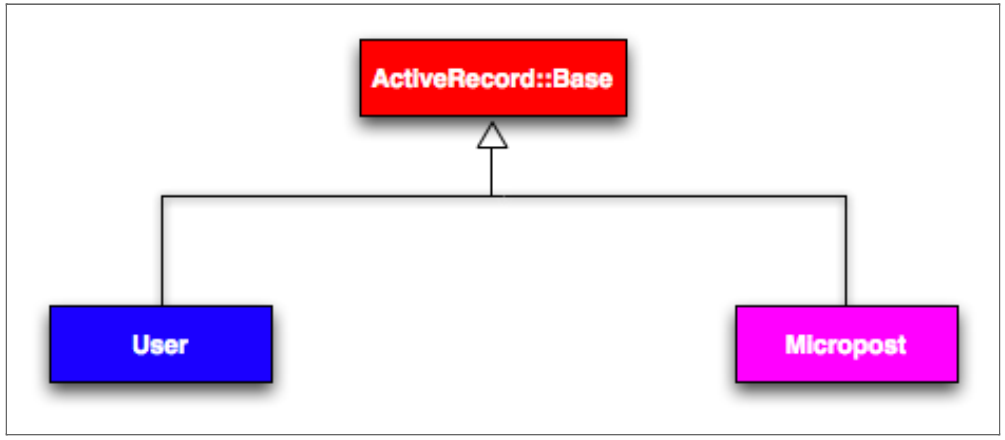

Figure 2.16: The inheritance hierarchy for the User and Micropost models.

The inheritance structure for controllers is only slightly more complicated. Comparing Listing 2.14 and Listing 2.15, we see that both the Users controller and the Microposts controller inherit from the Application controller. Examining Listing 2.16, we see that `ApplicationController` itself inherits from `ActionController::Base`; this is the base class for controllers provided by the Rails library Action Pack. The relationships between these classes is illustrated in Figure 2.17.

Listing 2.14. The `UsersController` class, with inheritance.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
end
```

Listing 2.15. The `MicropostsController` class, with inheritance.
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  .
  .
  .
end
```

Listing 2.16. The `ApplicationController` class, with inheritance.

`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  .
  .
  .
end
```

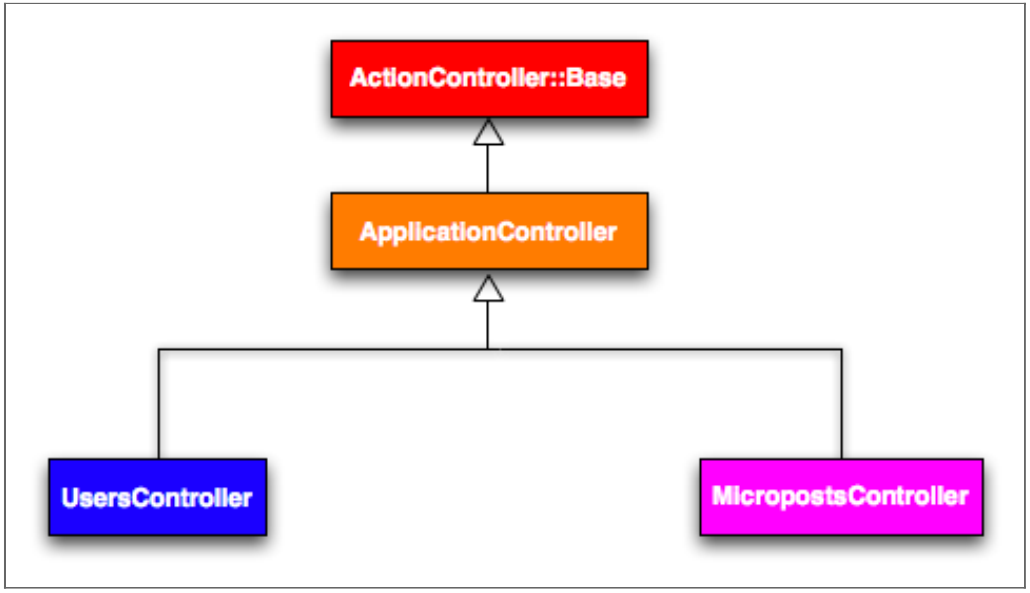

Figure 2.17: The inheritance hierarchy for the Users and Microposts controllers.

As with model inheritance, by inheriting ultimately from `ActionController::Base` both the

Users and Microposts controllers gain a large amount of functionality, such as the ability to manipulate model objects, filter inbound HTTP requests, and render views as HTML. Since all Rails controllers inherit from `ApplicationController`, rules defined in the Application controller automatically apply to every action in the application. For example, in Section 8.2.1 we'll see how to include helpers for signing in and signing out of all of the sample application's controllers.

2.3.5 Deploying the demo app

With the completion of the Microposts resource, now is a good time to push the repository up to GitHub:

```
$ git add .
$ git commit -m "Finish demo app"
$ git push
```

Ordinarily, you should make smaller, more frequent commits, but for the purposes of this chapter a single big commit at the end is fine.

At this point, you can also deploy the demo app to Heroku as in Section 1.4:

```
$ heroku create --stack cedar
$ git push heroku master
```

Finally, migrate the production database (see below if you get a deprecation warning):

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

This updates the database at Heroku with the necessary user/micropost data model. You may get a deprecation warning regarding assets in `vendor/plugins`, which you should ignore since there aren't any plugins in that directory.

2.4 Conclusion

We've come now to the end of the 30,000-foot view of a Rails application. The demo app developed in this chapter has several strengths and a host of weaknesses.

Strengths

- High-level overview of Rails
- Introduction to MVC
- First taste of the REST architecture
- Beginning data modeling
- A live, database-backed web application in production

Weaknesses

- No custom layout or styling
- No static pages (like "Home" or "About")
- No user passwords
- No user images
- No signing in
- No security
- No automatic user/micropost association
- No notion of "following" or "followed"
- No micropost feed
- No test-driven development
- **No real understanding**

The rest of this tutorial is dedicated to building on the strengths and eliminating the weaknesses.

« Chapter 1 From zero to deploy

Chapter 3 Mostly static pages »

---

1. When modeling longer posts, such as those for a normal (non-micro) blog, you should use the `text` type in place of `string`. ↑
2. The name of the scaffold follows the convention of *models*, which are singular, rather than resources and controllers, which are plural. Thus, we have `User` instead `Users`. ↑
3. Some references indicate that the view returns the HTML directly to the browser (via a web server such as Apache or Nginx). Regardless of the implementation details, I prefer to think of the controller as a central hub through which all the application's information flows. ↑
4. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. ↑
5. As with the User scaffold, the scaffold generator for microposts follows the singular convention of Rails models; thus, we have `generate Micropost`. ↑
6. The scaffold code may have extra newlines compared to Listing 2.7. This is not a cause for concern, as Ruby ignores extra newlines. ↑
7. Your console prompt might be something like `ruby-1.9.3-head >`, but the examples use >> since Ruby versions will vary. ↑