Addison-Wesley Professional Ruby Series

RUBY ON RAILS™ TUTORIAL

SIXTH EDITION

Learn Web Development with Rails

MICHAEL HARTL

Foreword by DEREK SIVERS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for Michael Hartl's Books and Videos on Ruby on Rails

"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."

—From the Foreword by Derek Sivers (sivers.org)
Formerly: founder of CD Baby
Currently: founder of Thoughts Ltd.

"Michael Hartl's Rails Tutorial book is the #1 (and only, in my opinion) place to start when it comes to books about learning Rails. . . . It's an amazing piece of work and, unusually, walks you through building a Rails app from start to finish with testing. If you want to read just one book and feel like a Rails master by the end of it, pick the *Ruby on Rails*™ *Tutorial*."

—Peter Cooper, editor, Ruby Inside

"Michael Hartl's *Ruby on Rails*™ *Tutorial* seamlessly taught me about not only Ruby on Rails, but also the underlying Ruby language, HTML, CSS, a bit of JavaScript, and even some SQL—but most importantly it showed me how to build a web application (Twitter) in a short amount of time."

—Mattan Griffel, co-founder & CEO of One Month

"Although I'm a Python/Django developer by trade, I can't stress enough how much this book has helped me. As an undergraduate, completely detached from industry, this book showed me how to use version control, how to write tests, and, most importantly—despite the steep learning curve for setting up and getting stuff running—how the end result of perseverance is extremely gratifying. It made me fall in love with technology all over again. This is the book I direct all my friends to who want to start learning programming/building stuff. Thank you Michael!"

—Prakhar Srivastav, software engineer, Xcite.com, Kuwait

"It has to be the best-written book of its type I've ever seen, and I can't recommend it enough."

—Daniel Hollands, administrator of Birmingham.IO

"For those wanting to learn Ruby on Rails, Hartl's *Ruby on Rails™ Tutorial* is (in my opinion) the best way to do it."

—David Young, software developer and author at deepinthecode.com

"This is a great tutorial for a lot of reasons, because aside from just teaching Rails, Hartl is also teaching good development practices."

—Michael Denomy, full-stack web developer

"Without a doubt, the best way I learned Ruby on Rails was by building an actual working app. I used Michael Hartl's *Ruby on Rails™ Tutorial*, which showed me how to get a very basic Twitter-like app up and running from scratch. I cannot recommend this tutorial enough; getting something up and going fast was key; it beats memorization by a mile."

—James Fend, serial entrepreneur, JamesFend.com

"The book gives you the theory and practice, while the videos focus on showing you in person how it's done. Highly recommended combo."

—Antonio Cangiano, software engineer, IBM

"The author is clearly an expert at the Ruby language and the Rails framework, but more than that, he is a working software engineer who introduces best practices throughout the text."

—Gregory Charles, principal software developer at Fairway Technologies

RUBY ON RAILS™ TUTORIAL

Sixth Edition

This page intentionally left blank

RUBY ON RAILS™ TUTORIAL

Learn Web Development with Rails

Sixth Edition

Michael Hartl

Addison-Wesley
Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the web: informit.com/aw

Library of Congress Control Number: 2020932223

Cover image: Olga Altunina/Shutterstock

ISBN-13: 978-0-13-670265-8
ISBN-10: 0-13-670265-1

ScoutAutomatedPrintCode

Vice President and Publisher
Mark L. Taub

Executive Editor
Debra Williams Cauley

Associate Editor
Manjula Anaskar

Managing Producer
Sandra Schroeder

Sr. Content Producer
Julie B. Nahil

Project Editor
diacriTech

Copy Editor
Jill Hobbs

Indexer
diacriTech

Proofreader
diacriTech

Cover Designer
Chuti Prasertsith

Compositor
diacriTech

Contents

This page intentionally left blank

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 initially, 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 that is highly recommended by the experts but that 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. [This is not typical! Most readers take much longer to finish the tutorial. —Michael] Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

—Derek Sivers (sivers.org)
Founder, CD Baby

This page intentionally left blank

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, Mark Bates, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steph Bristol, Pratik Naik, Sarah Mei, Sarah Allen, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, Sandi Metz, Ryan Davis, Aaron Patterson, Aja Hammerly, Richard "Schneems" Schneeman, the good people at Pivotal Labs, the Heroku gang, the thoughtbot folks, and the GitHub crew.

I'd like to thank technical reviewer Andrew Thai for his careful reading of the original manuscript and for his helpful suggestions. I'd also like to thank my cofounders

at Learn Enough, Nick Merwin and Lee Donahoe, for all their help in preparing this tutorial.

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 creator of the *Ruby on Rails™ Tutorial*, one of the leading introductions to web development, and is cofounder and principal author at LearnEnough.com. Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a PhD in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

This page intentionally left blank

CHAPTER 2
A Toy App

In this chapter, we develop a toy demo 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*, which create a large amount of functionality automatically. As discussed in Box 2.1, the rest of the book will take the opposite approach, developing a full sample 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 toy app will enable us to interact with it through its URLs, 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 toy 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: We 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 meantime, 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.

Box 2.1: Scaffolding: Quicker, Easier, More Seductive

From the beginning, Rails has benefited from a palpable sense of excitement, starting with the famous 15-minute weblog video (youtu.be/Gzj723LkRJY) by Rails creator David Heinemeier Hansson. That video and its successors are a great way to get a taste of Rails' power, and I recommend watching them. But be warned: They accomplish their amazing 15-minute feat using a feature called *scaffolding*, which relies heavily on *generated code*, magically created by the Rails `generate scaffold` command.

When writing a Ruby on Rails tutorial, it is tempting to rely on the scaffolding approach—it's quicker, easier, more seductive. But the complexity and sheer amount of code in the scaffolding can be utterly overwhelming to a beginning Rails developer; you may be able to use it, but you probably won't understand it. Following the scaffolding approach risks turning you into a virtuoso script generator with little (and brittle) actual knowledge of Rails.

In the *Ruby on Rails Tutorial*, we'll take the (nearly) polar opposite approach: Although this chapter will develop a small toy app using scaffolding, the core of the Rails Tutorial is the sample app, which we'll start writing in Chapter 3. At each stage of developing the sample application, we will write *small, bite-sized* pieces of code—simple enough to understand, yet novel enough to be challenging. The cumulative effect will be a deeper, more flexible knowledge of Rails, giving you a good background for writing nearly any type of web application.

2.1 Planning the Application

In this section, we'll outline our plans for the toy application. As in Section 1.2, we'll start by generating the application skeleton using the `rails new` command with a specific Rails version number:

```
$ cd ~/environment
$ rails _6.0.2.1_ new toy_app
$ cd toy_app/
```

If you're using the cloud IDE as recommended in Section 1.1.1, note that this second app can be created in the same environment as the first. It is not necessary to create a new environment. To get the files to appear, you may need to click the gear icon in the file navigator area and select "Refresh File Tree."

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

Important note: For all the Gemfiles in this book, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed below (although they should be identical if you are reading this online).

Listing 2.1: A **Gemfile** for the toy app.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.0.2.1'
gem 'puma',       '3.12.2'
gem 'sass-rails', '5.1.0'
gem 'webpacker',  '4.0.7'
gem 'turbolinks', '5.2.0'
gem 'jbuilder',   '2.9.1'
gem 'bootsnap',   '1.4.5', require: false

group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.0.1'
  gem 'listen',                '3.1.5'
  gem 'spring',                '2.1.0'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',           '3.28.0'
  gem 'selenium-webdriver', '3.142.4'
  gem 'webdrivers',         '4.1.2'
end

group :production do
  gem 'pg', '1.1.4'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

Note that Listing 2.1 is identical to Listing 1.18.

As in Section 1.4.1, we'll install the local gems while preventing the installation of production gems using the **`--without production`** option:

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

As noted in Section 1.2.1, you may need to run **`bundle update`** as well (Box 1.2).

Finally, we'll put the toy app under version control with Git:

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

You should also create a new repository at GitHub by following the same steps as in Section 1.3.3 (taking care to make it private as in Figure 2.1), and then push up to the remote repository:

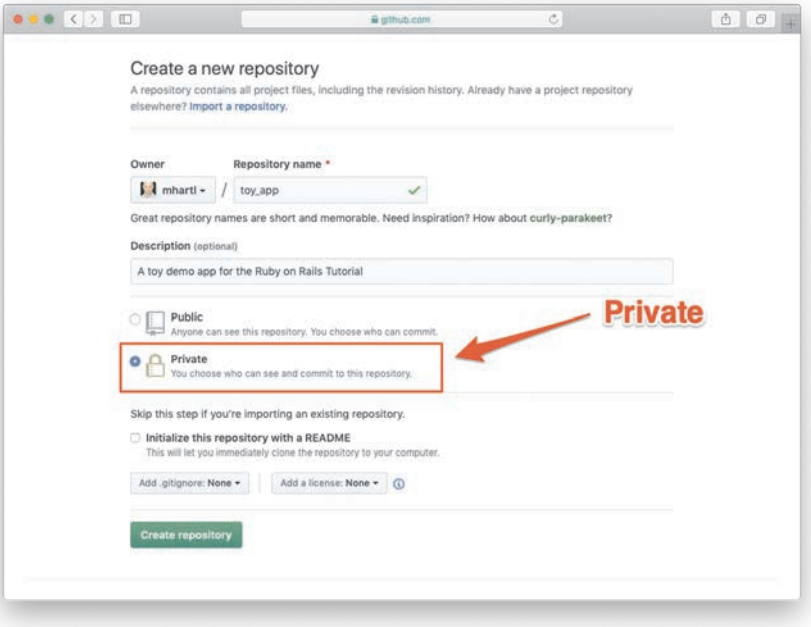


Figure 2.1: Creating the toy app repository at GitHub.

```
$ git remote add origin https://github.com/<username>/toy_app.git
$ git push -u origin master
```

Finally, it's never too early to deploy, which I suggest doing by following the same "hello, world!" steps from Section 1.2.4, as shown in Listing 2.2 and Listing 2.3.

Listing 2.2: Adding a `hello` action to the Application controller.
`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base

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

Listing 2.3: Setting the root route.
`config/routes.rb`

```
Rails.application.routes.draw do
  root 'application#hello'
end
```

Then commit the changes and push up to Heroku, and, at the same time, GitHub—it's a good idea to keep the two copies in sync:

```
$ git commit -am "Add hello"
$ heroku create
$ git push && git push heroku master
```

Here we've used the double ampersand operator `&&` (read "and") to combine the pushes to GitHub and Heroku; the second command will execute only if the first one succeeds.[1]

1. The `&&` operator is described in Chapter 4 of *Learn Enough Command Line to Be Dangerous* (www.learnenough.com/command-line).

As in Section 1.4, you may see some warning messages, which you should ignore for now. We'll deal with them in Section 7.5. Apart from the URL of the Heroku app, the result should be the same as in Figure 1.31.

2.1.1 A Toy Model for Users

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, including the relationships between them. In our case, the toy app will be a Twitter-style microblog, with only users and short (micro)posts. Thus, we'll begin with a model for *users* of the app in this section, and then we'll add a model for *microposts* (Section 2.1.2).

There are as many choices for a user data model as there are different registration forms on the web; for simplicity, we'll go with a distinctly minimalist approach. Users of our toy app will have a unique identifier called **`id`** (of type **`integer`**), a publicly viewable **`name`** (of type **`string`**), and an **`email`** address (also of type **`string`**) that will double as a unique username. (Note that there is no **`password`** attribute at this point, which is part of what makes this app a "toy." We'll cover passwords starting in Chapter 6.) A summary of the data model for users appears in Figure 2.2.

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 A Toy Model for Microposts

Recall from the introduction that a *micropost* is simply a short post, essentially a generic term for the brand-specific "tweet" (with the prefix "micro" motivated by Twitter's original description as a "micro-blog"). The core of the micropost data model is even

users	
id	integer
name	string
email	string

Figure 2.2: The data model for users.

simpler than the one for users: A micropost has only an `id` and a `content` field for the micropost's text (of type `text`).[2] 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.

We'll see in Section 2.3.3 (and more fully in Chapter 13) 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

microposts	
id	integer
content	text
user_id	integer

Figure 2.3: The data model for microposts.

2. Because microposts are short by design, the `string` type might actually be big enough to contain them, but using `text` better expresses our intent, while also giving us greater flexibility should we ever wish to relax the length constraint. Indeed, Twitter's change from allowing 140 to 280 characters in English-language tweets is a perfect example of why such flexibility is important: A `string` typically allows 255 ($2^8 - 1$) characters, which is big enough for 140-character tweets but not for 280-character ones. Using `text` allows a unified treatment of both cases.

the resource name (in this case, **User**), together with optional parameters for the data model's attributes:[3]

```
$ rails generate scaffold User name:string email:string
      invoke  active_record
      create    db/migrate/<timestamp>_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      invoke  resource_route
       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/controllers/users_controller_test.rb
      create      test/system/users_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      test_unit
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      create      app/views/users/_user.json.jbuilder
      invoke  assets
      invoke    scss
      create      app/assets/stylesheets/users.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.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**; Rails creates it automatically for use as the *primary key* in the database.)

3. 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 of **Users**.

To proceed with the toy application, we first need to *migrate* the database using **`rails db:migrate`**, as shown in Listing 2.4.

Listing 2.4: Migrating the database.

```
$ rails db:migrate
== CreateUsers: migrating ====================================
-- create_table(:users)
   -> 0.0027s
== CreateUsers: migrated (0.0036s) ===========================
```

The effect of Listing 2.4 is to update the database with our new **`users`** data model. (We'll learn more about database migrations starting in Section 6.1.1.)

Having run the migration in Listing 2.4, we can run the local webserver in a separate tab (Figure 1.15). Users of the cloud IDE should first add the same configuration as in Section 1.2.2 to allow the toy app to be served (Listing 2.5).

Listing 2.5: Allowing connections to the local web server.
`config/environments/development.rb`

```
Rails.application.configure do
  .
  .
  .
  # Allow Cloud9 connections.
  config.hosts.clear
end
```

Then run the Rails server as in Section 1.2.2:

```
$ rails server
```

Now the toy application should be available on the local server as described in Section 1.2.2. In particular, if we visit the root URL at / (read "slash", as noted in Section 1.2.4), we get the same "hello, world!" page shown in Figure 1.20.

2.2.1 A User Tour

In generating the Users resource scaffolding in Section 2.2, Rails 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 URLs.

We start with the page that shows all the users in our application, called **`index`** and located at /users. As you might expect, initially there are no users at all (Figure 2.4).

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

URL	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`**

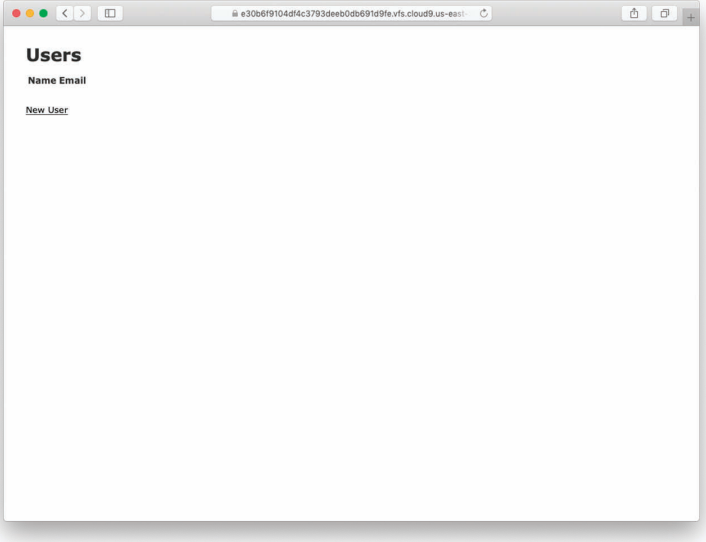


Figure 2.4: The initial index page for the Users resource (/users).

To make a new user, we can click on the New User link in Figure 2.4 to visit the `new` page at /users/new, as shown in Figure 2.5. In Chapter 7, this will become the user signup page.

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 at /users/1, 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 URL 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 page.

To change a user's information, we click the Edit link to visit the `edit` page at /users/1/edit (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 toy 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 10.1.

Now we'll create a second user by revisiting the `new` page at /users/new and submitting a second set of user information. The resulting user `index` is shown in

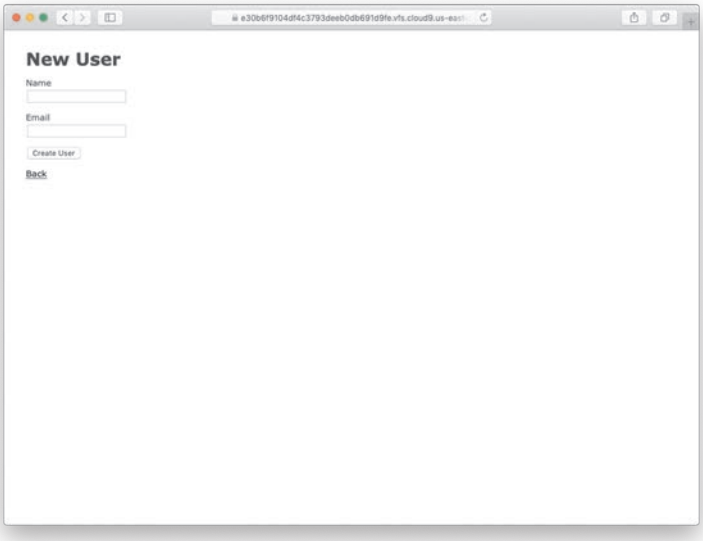


Figure 2.5: The new user page (/users/new).

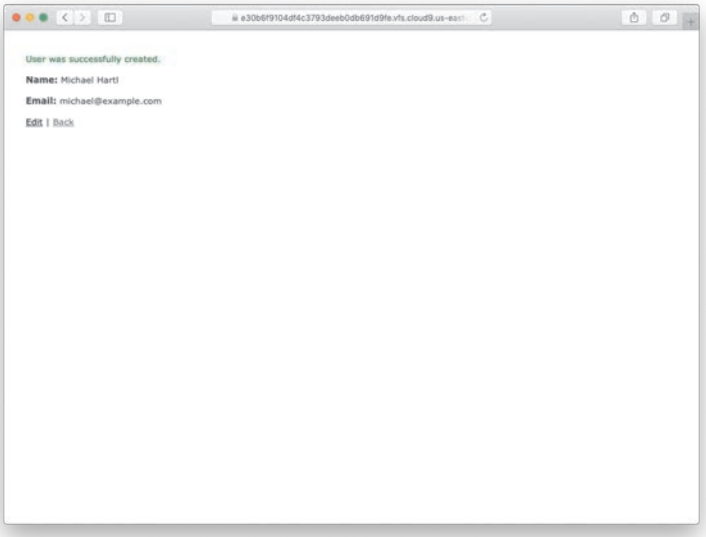


Figure 2.6: The page to show a user (/users/1).

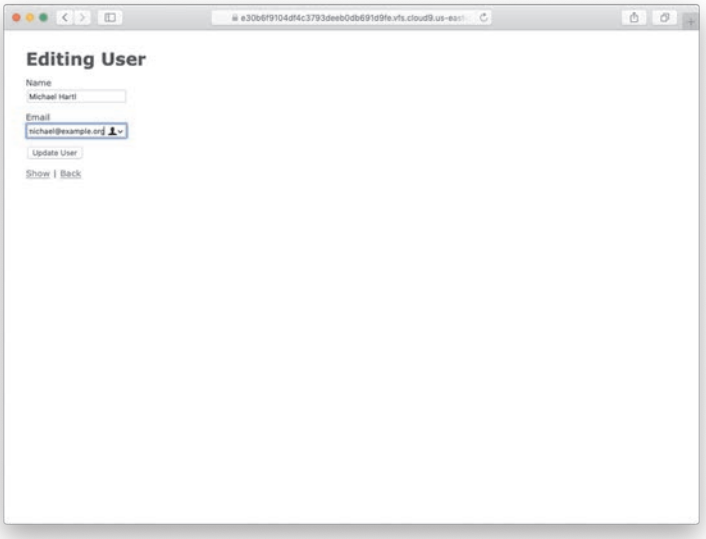


Figure 2.7: The user edit page (/users/1/edit).

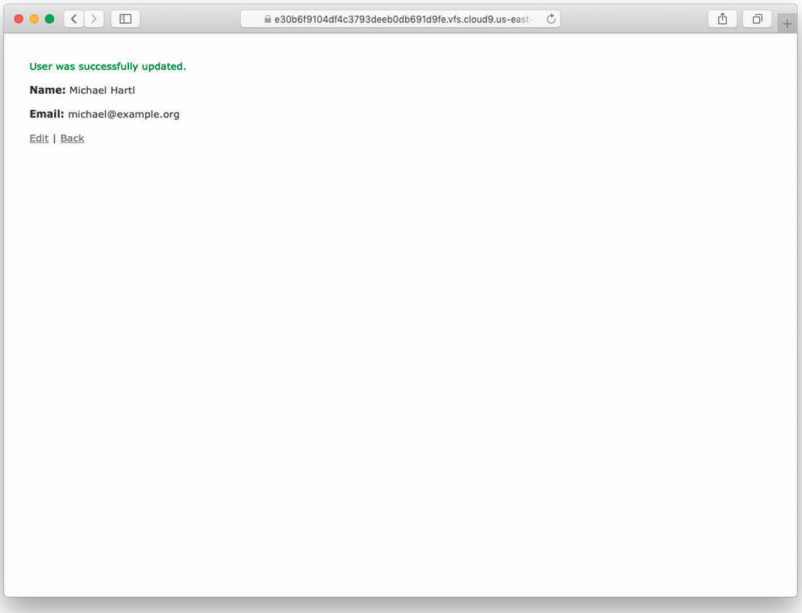


Figure 2.8: A user with updated information.

Figure 2.9. In Section 7.1, we will develop the user index into a more polished page for showing all users.

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 10.4 adds user deletion to the sample app, taking care to restrict its use to a special class of administrative users.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

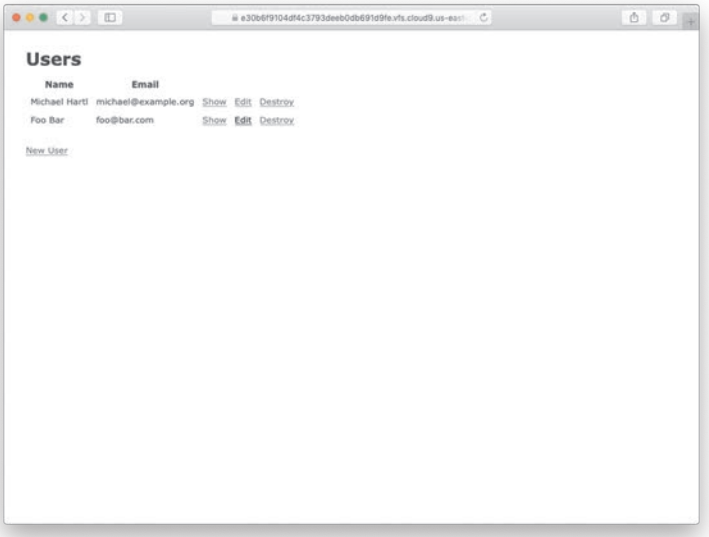


Figure 2.9: The user index page (/users) with a second user.

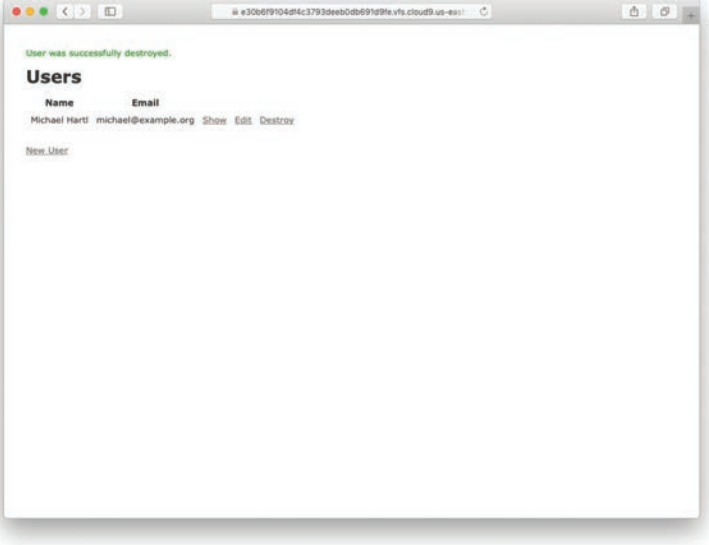


Figure 2.10: Destroying a user.

1. (For readers who know CSS) Create a new user, then use your browser's HTML inspector to determine the CSS id for the text "User was successfully created." What happens when you refresh your browser?
2. What happens if you try to create a user with a name but no email address?
3. What happens if you try create a user with an invalid email address, like "@example.com"?
4. Destroy each of the users created in the previous exercises. Does Rails display a message by default when a user is destroyed?

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 1.2.3. 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).

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

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

Now let's take a look at the these steps in more detail. We start with a request issued from the browser—that is, the result of typing a URL in the address bar or clicking on a link (Step 1 in Figure 2.11). This request hits the *Rails router* (Step 2), which

4. Some references indicate that the view returns the HTML directly to the browser (via a webserver such as Apache or Nginx). Regardless of the implementation details, I find it helpful to think of the controller as a central hub through which all the application's information flows.

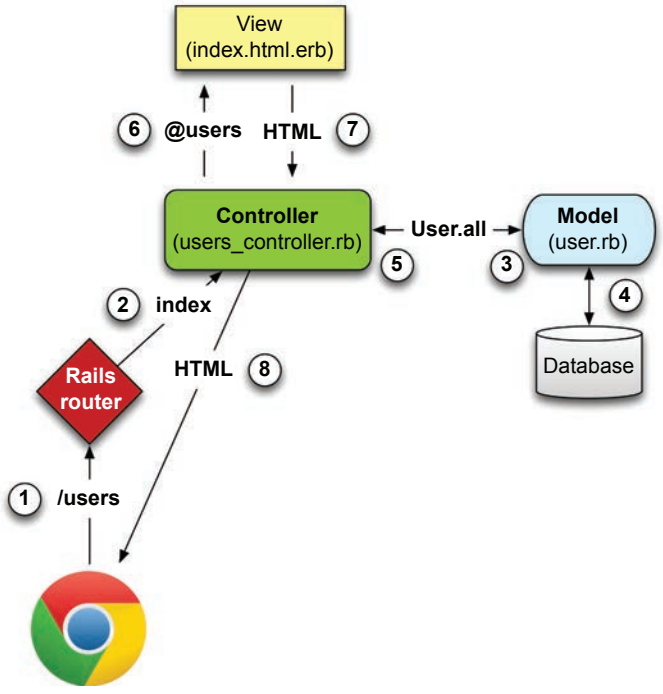


Figure 2.11: A detailed diagram of MVC in Rails.

dispatches the request to the proper *controller action* based on the URL (and, as we'll see in Box 3.2, the type of request). The code to create the mapping of user URLs to controller actions for the Users resource appears in Listing 2.6. This code effectively sets up the table of URL/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 2.6: The Rails routes, with a rule for the Users resource.
`config/routes.rb`

```
Rails.application.routes.draw do
  resources :users
  root 'application#hello'
end
```

While we're looking at the routes file, let's take a moment to associate the root route with the users index, so that "slash" goes to /users. Recall from Listing 2.3 that we added the root route

```
root 'application#hello'
```

so that the root route went to the **hello** action in the Application controller. In the present case, we want to use the **index** action in the Users controller, which we can arrange using the code shown in Listing 2.7.

Listing 2.7: Adding a root route for users.
`config/routes.rb`

```
Rails.application.routes.draw do
  resources :users
  root 'users#index'
end
```

A *controller* contains a collection of related *actions*, and the pages from the tour in Section 2.2.1 correspond to actions in the Users controller. The controller generated by the scaffolding is shown schematically in Listing 2.8. Note the code **class UsersController < ApplicationController**, which 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 2.8: The Users controller in schematic form.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def index
    .
    .
    .
  end
```

```
  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def edit
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def update
    .
    .
    .
  end

  def destroy
    .
    .
    .
  end
end
```

You might 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 can); 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 idea of *representational state transfer*, a concept identified and named by computer scientist

Table 2.2: RESTful routes provided by the Users resource in Listing 2.6.

HTTP request	URL	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**
PATCH	/users/1	**update**	update user with id **1**
DELETE	/users/1	**destroy**	delete user with id **1**

Roy Fielding.[5] Note from Table 2.2 that there is some overlap in the URLs; for example, both the user **show** action and the **update** action correspond to the URL /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 3.3.

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 to the four fundamental HTTP request methods: `POST`, `GET`, `PATCH`, and `DELETE`. (We'll learn more about HTTP requests in Section 3.3 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 14, we'll see an example where

5. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

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 the **index** action, shown in Listing 2.9. (Learning how to read code even when you don't fully understand it is an important aspect of technical sophistication (Box 1.2).)

Listing 2.9: The simplified user **index** action for the toy application.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

This **index** action includes the line **@users = User.all** (Step 3 in Figure 2.11), 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 2.10. 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 2.10 arranges for **User.all** to return all the users in the database.

Listing 2.10: The User model for the toy application.
`app/models/user.rb`

```
class User < ApplicationRecord
end
```

Once the **@users** variable is defined, the controller calls the *view* (Step 6), shown in Listing 2.11. Variables that start with the **@** sign, called *instance variables*, are automatically available in the views; in this case, the **index.html.erb** view in Listing 2.11 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 2.11: The view for the users index.
`app/views/users/index.html.erb`

```
<p id="notice"><%= notice %></p>

<h1>Users</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @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 %>
  </tbody>
</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).

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. By referring to Figure 2.11, write out the analogous steps for visiting the URL /users/1/edit.
2. Find the line in the scaffolding code that retrieves the user from the database in the previous exercise. *Hint*: It's in a special location called `set_user`.
3. What is the name of the view file for the user edit page?

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 of logging 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 don't test for data validation, authentication, or any other custom requirements.
- **No style or 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.

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:[6]

```
$ rails generate scaffold Micropost content:text user_id:integer
      invoke  active_record
      create    db/migrate/<timestamp>_create_microposts.rb
      create    app/models/micropost.rb
      invoke    test_unit
      create      test/models/micropost_test.rb
      create      test/fixtures/microposts.yml
      invoke  resource_route
       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/controllers/microposts_controller_test.rb
      create      test/system/microposts_test.rb
      invoke    helper
      create      app/helpers/microposts_helper.rb
      invoke      test_unit
      invoke    jbuilder
      create      app/views/microposts/index.json.jbuilder
      create      app/views/microposts/show.json.jbuilder
      create      app/views/microposts/_micropost.json.jbuilder
      invoke  assets
      invoke    scss
      create      app/assets/stylesheets/microposts.scss
      invoke  scss
   identical    app/assets/stylesheets/scaffolds.scss
```

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

6. As with the User scaffold, the scaffold generator for microposts follows the singular convention of Rails models; thus, we have **`generate Micropost`**.

```
$ rails 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 2.12.[7] As with users, the **`resources :microposts`** routing rule maps micropost URLs to actions in the Microposts controller, as seen in Table 2.3.

Listing 2.12: The Rails routes, with a new rule for Microposts resources.
`config/routes.rb`

```
Rails.application.routes.draw do
  resources :microposts
  resources :users
  root 'users#index'
end
```

Table 2.3: RESTful routes provided by the Microposts resource in Listing 2.12.

HTTP request	URL	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`**
PATCH	/microposts/1	**`update`**	update micropost with id **`1`**
DELETE	/microposts/1	**`destroy`**	delete micropost with id **`1`**

7. The scaffold code may have extra blank lines compared to Listing 2.12. This is not a cause for concern, as Ruby ignores such extra space.

The Microposts controller itself appears in schematic form in Listing 2.13. Note that, apart from having **`MicropostsController`** in place of **`UsersController`**, Listing 2.13 is *identical* to the code in Listing 2.8. This is a reflection of the REST architecture that is common to both resources.

Listing 2.13: The Microposts controller in schematic form.
`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  .
  .
  .
  def index
    .
    .
    .
  end

  def show
    .
    .
    .
  end

  def new
    .
    .
    .
  end

  def edit
    .
    .
    .
  end

  def create
    .
    .
    .
  end

  def update
    .
    .
    .
  end
```

```
  def destroy
    .
    .
    .
  end
end
```

To make some actual microposts, we click on New Micropost on the micropost index page (Figure 2.12). We then enter information at the new microposts page, /microposts/new, as seen in Figure 2.13.

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 2.14.

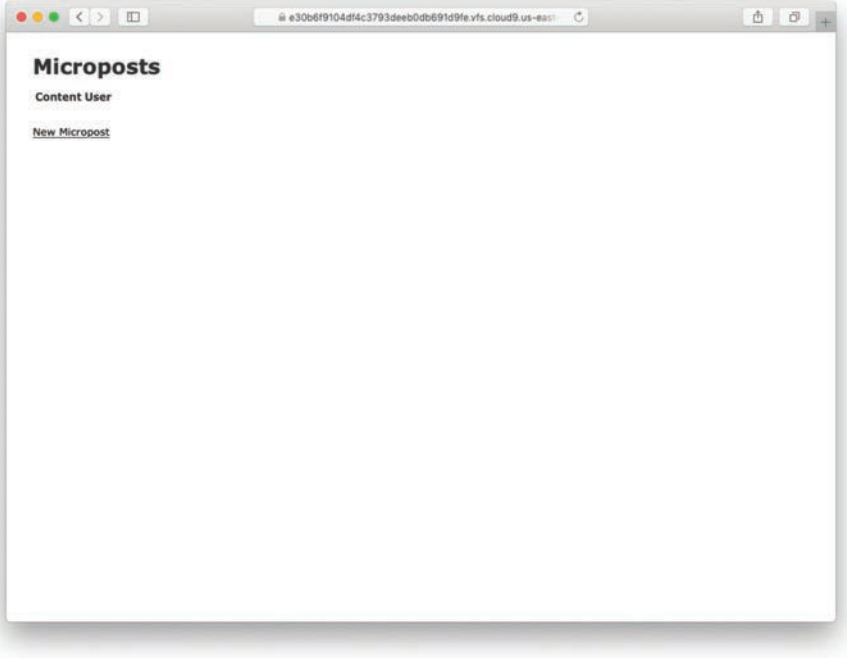


Figure 2.12: The micropost index page (/microposts).

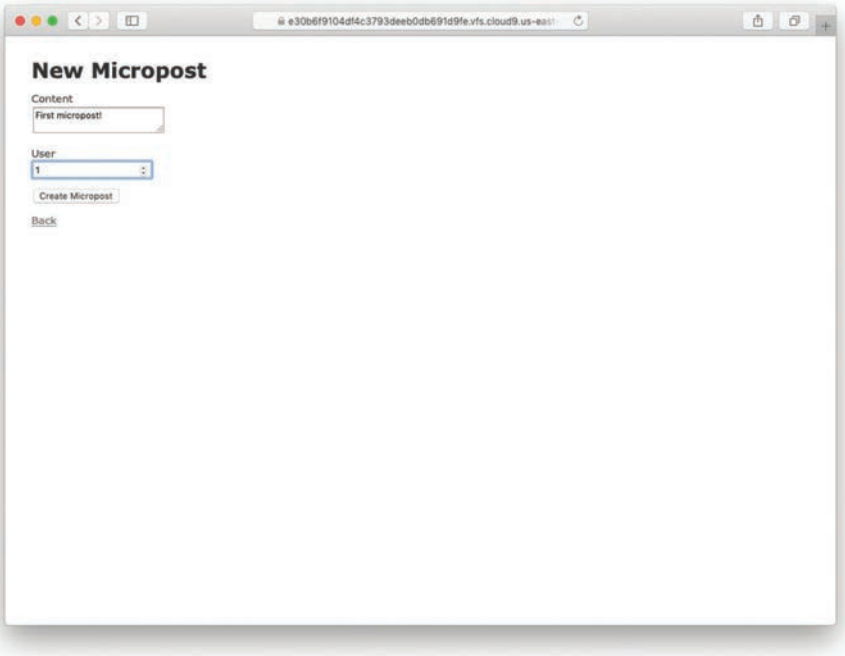


Figure 2.13: The new micropost page (/microposts/new).

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. (For readers who know CSS) Create a new micropost, then use your browser's HTML inspector to determine the CSS id for the text "Micropost was successfully created." What happens when you refresh your browser?
2. Try to create a micropost with empty content and no user id.
3. Try to create a micropost with more than 140 characters of content (say, the first paragraph from the Wikipedia article on Ruby).
4. Destroy the microposts from the previous exercises.

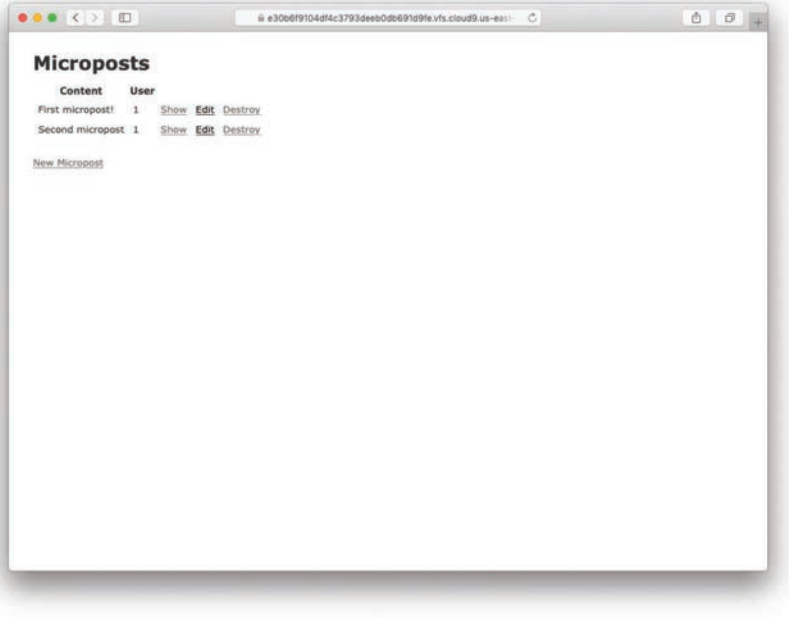


Figure 2.14: The micropost index page with a couple of posts.

2.3.2 Putting the *Micro* in Microposts

Any *micro*post worthy of the name should have some means of enforcing the rules governing the length of the post. Implementing this constraint in Rails is easy with *validations*; to accept microposts with at most 140 characters (à la the original design of 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 2.14.

Listing 2.14: Constraining microposts to be at most 140 characters.
`app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  validates :content, length: { maximum: 140 }
end
```

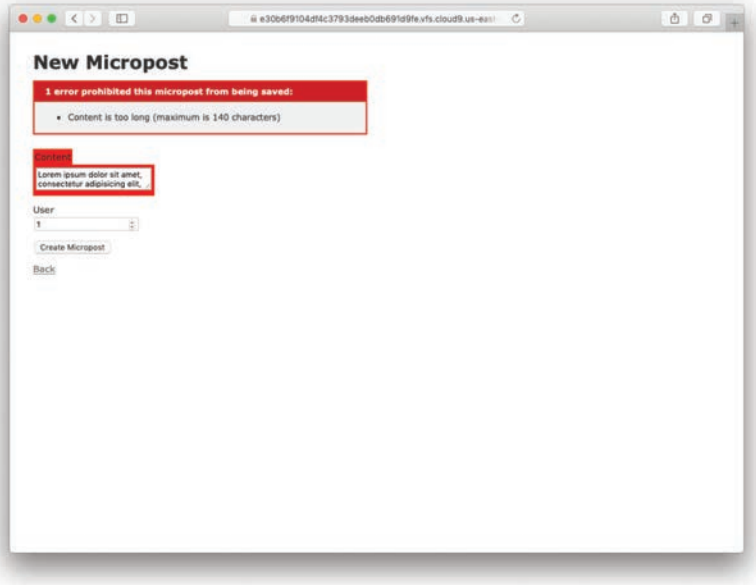


Figure 2.15: Error messages for a failed micropost creation.

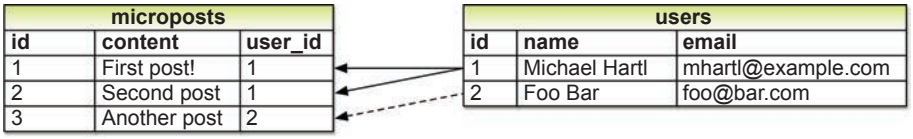
The code in Listing 2.14 may look rather mysterious—we'll cover validations more thoroughly starting in Section 6.2—but its effects are readily apparent if we go to the new micropost page and enter more than 140 characters for the content of the post. As seen in Figure 2.15, Rails renders *error messages* indicating that the micropost's content is too long. (We'll learn more about error messages in Section 7.3.3.)

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Try to create a micropost with the same long content used in a previous exercise (Section 2.3.1). How has the behavior changed?
2. (For readers who know CSS) Use your browser's HTML inspector to determine the CSS id of the error message produced by the previous exercise.



microposts		
id	content	user_id
1	First post!	1
2	Second post	1
3	Another post	2

users		
id	name	email
1	Michael Hartl	mhartl@example.com
2	Foo Bar	foo@bar.com

Figure 2.16: The association between microposts and users.

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 relationship in code by updating the User and Micropost models as in Listing 2.15 and Listing 2.16, respectively.

Listing 2.15: A user has many microposts.
`app/models/user.rb`

```
class User < ApplicationRecord
  has_many :microposts
end
```

Listing 2.16: A micropost belongs to a user.
`app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  belongs_to :user
  validates :content, length: { maximum: 140 }
end
```

We can visualize the result of this association in Figure 2.16. Because of the **`user_id`** column in the **`microposts`** table, Rails (using Active Record) can infer the microposts associated with each user.

In Chapter 13 and Chapter 14, we will use the association of users and microposts both to display all of 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`**), as shown in Listing 2.17.[8] (I include **`exit`** in the last line just to demonstrate how to exit the console. On most systems, you can also use Ctrl-D for the same purpose.)[9]

Listing 2.17: Investigating the state of the application using the Rails console.

```
$ rails console
>> first_user = User.first
   (0.5ms)  SELECT sqlite_version(*)
  User Load (0.2ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC
  LIMIT ?  [["LIMIT", 1]]
 => #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
 created_at: "2019-08-20 00:39:14", updated_at: "2019-08-20 00:41:24">
>> first_user.microposts
  Micropost Load (3.2ms)  SELECT "microposts".* FROM "microposts" WHERE
  "microposts"."user_id" = ? LIMIT ?  [["user_id", 1], ["LIMIT", 11]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Micropost id: 1, content:
 "First micropost!", user_id: 1, created_at: "2019-08-20 02:04:13", updated_at:
 "2019-08-20 02:04:13">, #<Micropost id: 2, content: "Second micropost",
 user_id: 1, created_at: "2019-08-20 02:04:30", updated_at: "2019-08-20
 02:04:30">]>
>> micropost = first_user.microposts.first
  Micropost Load (0.2ms)  SELECT "microposts".* FROM "microposts" WHERE
  "microposts"."user_id" = ? ORDER BY "microposts"."id" ASC LIMIT ?
  [["user_id", 1], ["LIMIT", 1]]
 => #<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
 "2019-08-20 02:04:13", updated_at: "2019-08-20 02:04:13">
>> micropost.user
 => #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
 created_at: "2019-08-20 00:39:14", updated_at: "2019-08-20 00:41:24"
>> exit
```

There's a lot going on in Listing 2.17, and teasing out the relevant parts is a good exercise in technical sophistication (Box 1.2). The output includes the actual return

8. Your console prompt might be something like **`2.6.3 :001 >`**, but the examples use **`>>`** since Ruby versions will vary.

9. As in the case of Ctrl-C, the capital "D" refers to the key on the keyboard, not the capital letter, so you don't have to hold down the Shift key along with the Ctrl key.

values, which are raw Ruby objects, as well as the structured query language (SQL) code that produced them.

In addition to retrieving the first user with **`User.first`**, Listing 2.17 shows two other things: (1) how to access the first user's microposts using the code **`first_user.microposts`**, which automatically returns all the microposts with **`user_id`** equal to the id of **`first_user`** (in this case, **`1`**); and (2) how to return the user corresponding to a particular post using **`micropost.user`**. We'll learn much more about the Ruby involved in Listing 2.17 in Chapter 4, and more about the association facilities in Active Record in Chapter 13 and Chapter 14.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Edit the user show page to display the content of the user's first micropost. (Use your technical sophistication (Box 1.2) to guess the syntax based on the other content in the file.) Visit /users/1 to confirm that it worked.
2. The code in Listing 2.18 shows how to add a validation for the presence of micropost content to ensure that microposts can't be blank. Verify that you get the behavior shown in Figure 2.17.
3. Update Listing 2.19 by replacing **`FILL_IN`** with the appropriate code to validate the presence of name and email attributes in the User model (Figure 2.18).

Listing 2.18: Code to validate the presence of micropost content.
`app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  belongs_to :user
  validates :content, length: { maximum: 140 },
                      presence: true
end
```

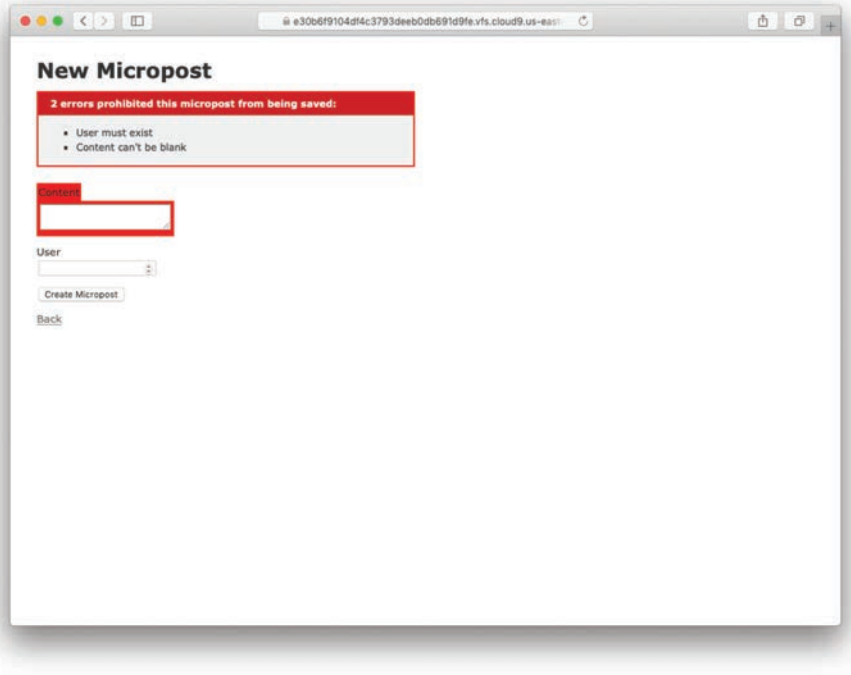


Figure 2.17: The effect of a micropost presence validation.

Listing 2.19: Adding presence validations to the User model.
`app/models/user.rb`

```
class User < ApplicationRecord
  has_many :microposts
  validates FILL_IN, presence: true    # Replace FILL_IN with the right code.
  validates FILL_IN, presence: true    # Replace FILL_IN with the right code.
end
```

2.3.4 Inheritance Hierarchies

We end our discussion of the toy application with a brief description of the controller and model class hierarchies in Rails. This discussion will make more sense if you

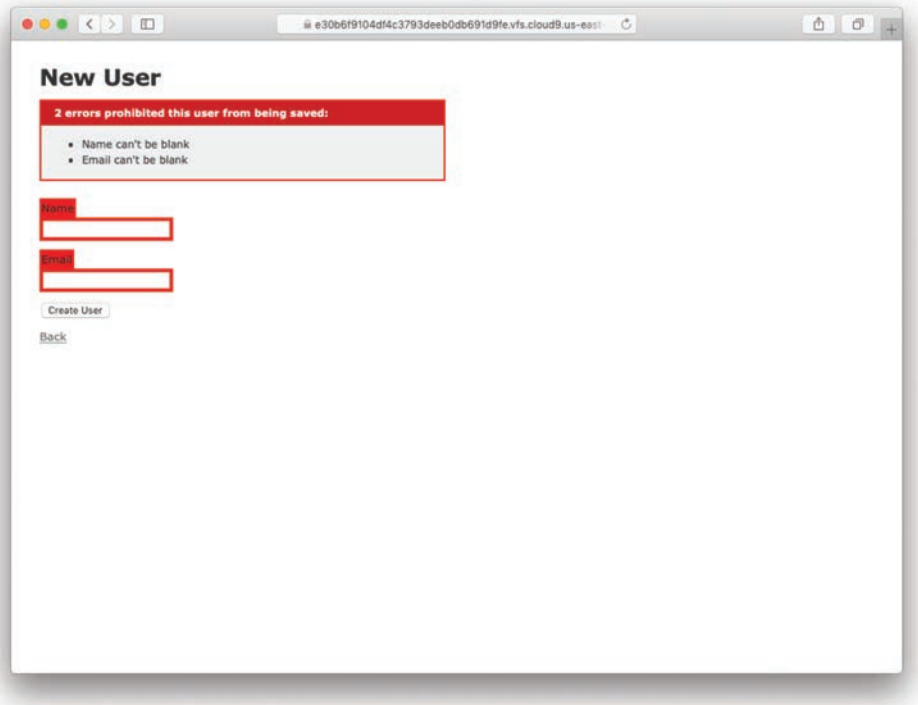


Figure 2.18: The effect of presence validations on the User model.

have some experience with object-oriented programming (OOP), particularly *classes*. Don't worry if it's confusing for now; we'll discuss these ideas more thoroughly in Section 4.4.

We start with the inheritance structure for models. Comparing Listing 2.20 and Listing 2.21, we see that both the User model and the Micropost model inherit (via the left angle bracket `<`) from `ApplicationRecord`, which in turn inherits from `ActiveRecord::Base`, which is the base class for models provided by Active Record; a diagram summarizing this relationship appears in Figure 2.19. By inheriting from `ActiveRecord::Base`, our model objects gain the ability to communicate with the database, treat the database columns as Ruby attributes, and so on.

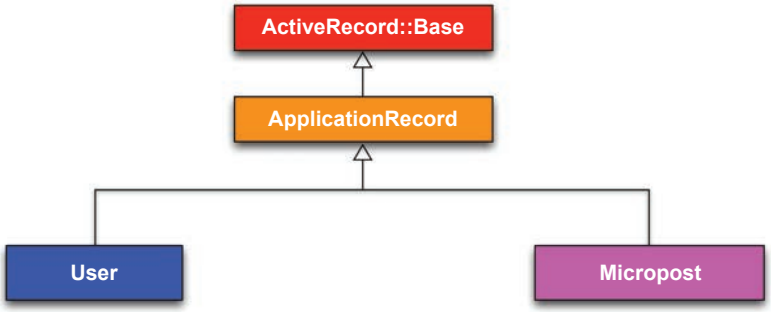


Figure 2.19: The inheritance hierarchy for the User and Micropost models.

Listing 2.20: The `User` class, highlighting inheritance.
`app/models/user.rb`

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

Listing 2.21: The `Micropost` class, highlighting inheritance.
`app/models/micropost.rb`

```
class Micropost < ApplicationRecord
  .
  .
  .
end
```

The inheritance structure for controllers is essentially the same as that for models. Comparing Listing 2.22 and Listing 2.23, we see that both the Users controller and the Microposts controller inherit from the Application controller. Examining Listing 2.24, we see that **`ApplicationController`** itself inherits from **`ActionController::Base`**, which is the base class for controllers provided by the Rails library Action Pack. The relationships between these classes are illustrated in Figure 2.20.

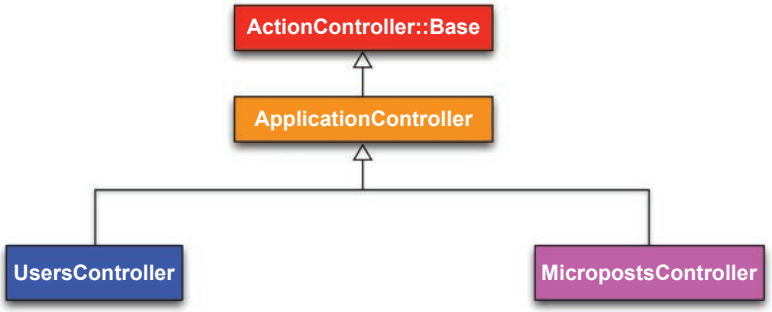


Figure 2.20: The inheritance hierarchy for the Users and Microposts controllers.

Listing 2.22: The `UsersController` class, highlighting inheritance.
`app/controllers/users_controller.rb`

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

Listing 2.23: The `MicropostsController` class, highlighting inheritance.
`app/controllers/microposts_controller.rb`

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

Listing 2.24: The `ApplicationController` class, highlighting inheritance.
`app/controllers/application_controller.rb`

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

As with model inheritance, both the Users and Microposts controllers gain a large amount of functionality by inheriting from a base class (in this case, **`ActionController::Base`**), including the capability to manipulate model objects, filter inbound HTTP requests, and render views as HTML. Since all Rails controllers inherit from **`ApplicationController`**, all rules defined in the Application controller automatically apply to every action in the application. For example, in Section 9.1 we'll see how to include helpers for logging in and logging out of all of the sample application's controllers.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. By examining the contents of the Application controller file, find the line that causes **`ApplicationController`** to inherit from **`ActionController::Base`**.
2. Is there an analogous file containing a line where **`ApplicationRecord`** inherits from **`ActiveRecord::Base`**? *Hint*: It would probably be a file called something like **`application_record.rb`** in the **`app/models`** directory.

2.3.5 Deploying the Toy App

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

```
$ git status    # It's a good habit to check the status before adding
$ git add -A
$ git commit -m "Finish toy 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 toy app to Heroku as is described in Section 1.4:

```
$ git push heroku
```

(This assumes you created the Heroku app in Section 2.1. Otherwise, you should run **`heroku create`** and then **`git push heroku master`**.)

At this point, visiting the page at Heroku yields an error message, as shown in Figure 2.21. We can track down the problem by inspecting the Heroku logs:

```
$ heroku logs
```

Scrolling up in the logs, you should see a line that includes something like this:

```
ActionView::Template::Error (PG::UndefinedTable: ERROR:  relation "users" does
not exist
```

This line is a big hint that there is a missing **`users`** table. Luckily, we learned how to handle that problem way back in Listing 2.4: All we need to do is run the database migrations (which will create the **`microposts`** table as well).

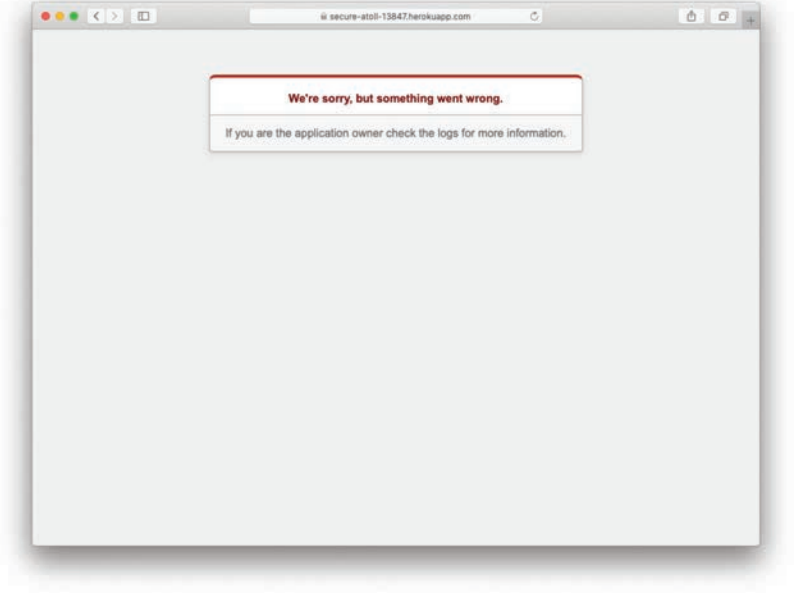


Figure 2.21: An error page at Heroku.

The way to execute this sort of command at Heroku is to prefix the usual Rails command with **`heroku run`**, like this:

```
$ heroku run rails db:migrate
```

This updates the database at Heroku with the user and micropost data models as required. After running the migration, you should be able to use the toy app in production, with a real PostgreSQL database back end (Figure 2.22).[10]

Finally, if you completed the exercises in Section 2.3.3, you will have to remove the code to display the first user's micropost to get the app to load properly. In this case, simply delete the offending code, make another commit, and push again to Heroku.

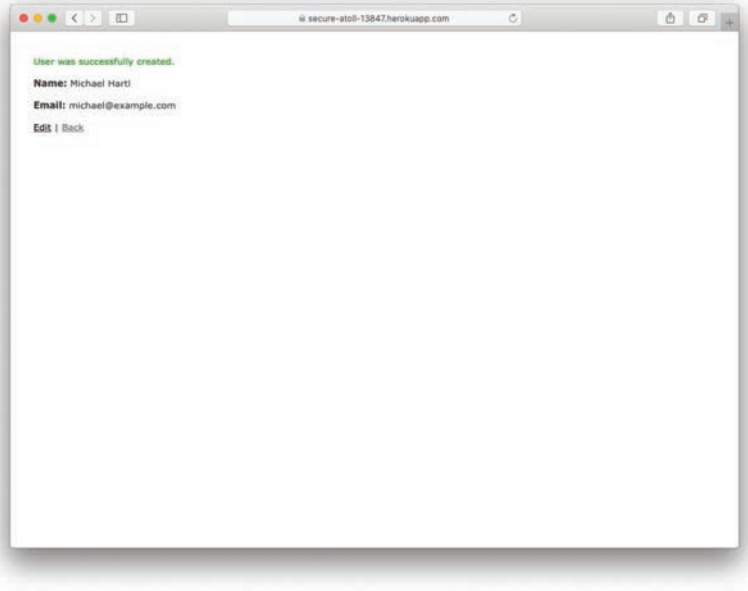


Figure 2.22: Running the toy app in production.

10. The production database should work without any additional configuration, but in fact some configuration is recommended by the official Heroku documentation. We'll take care of this detail in Section 7.5.3.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers at https://www.railstutorial.org/aw-solutions.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Create a few users on the production app.
2. Create a few production microposts for the first user.
3. By trying to create a micropost with content exceeding 140 characters, confirm that the validation from Listing 2.14 works on the production app.

2.4 Conclusion

We've come now to the end of the high-level overview of a Rails application. The toy 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 (such as "Home" or "About")
- No user passwords
- No user images
- No logging in
- No security
- No automatic user/micropost association

- No notion of "following" or "followed"
- No micropost feed
- No meaningful tests
- **No real understanding**

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

2.4.1 What We Learned in this Chapter

- Scaffolding automatically creates code to model data and interact with it through the web.
- Scaffolding is good for getting started quickly but is bad for understanding.
- Rails uses the model-view-controller (MVC) pattern for structuring web applications.
- As interpreted by Rails, the REST architecture includes a standard set of URLs and controller actions for interacting with data models.
- Rails supports data validations to place constraints on the values of data model attributes.
- Rails comes with built-in functions for defining associations between different data models.
- We can interact with Rails applications at the command line using the Rails console.

This page intentionally left blank

Index

C

D

E

F

G

H

I

J

K

L

N

Q

R

S

T

U

V

W

X

Y

Z