

CUBRE RAILS 4.2



EL TUTORIAL DE RUBY ON RAILS

APRENDA DESARROLLO WEB CON RAILS

TERCERA EDICIÓN

LIBRO Y VIDEO EXPLICATIVOS POR

MICHAEL HARTL



Tutorial de Ruby on Rails

Aprenda Desarrollo Web con Rails

Michael Hartl

Contenido

1	Desde cero hasta el despliegue	1
1.1	Introducción	5
1.1.1	Prerequisitos	6
1.1.2	Acuerdos utilizados en este libro	8
1.2	En funcionamiento	10
1.2.1	Ambiente de desarrollo	11
1.2.2	Instalando Rails	13
1.3	La primera aplicación	16
1.3.1	Bundler	19
1.3.2	<code>rails server</code>	26
1.3.3	Modelo Vista-Controlador (MVC)	31
1.3.4	¡Hola mundo!	32
1.4	Control de versiones con Git	36
1.4.1	Instalación y preparación	37
1.4.2	¿Qué beneficios le proporciona a usted Git?	40
1.4.3	Bitbucket	41
1.4.4	Ramas, edición, confirmación e integración	45
1.5	Desplegando	51
1.5.1	Configuración de Heroku	53
1.5.2	Despliegue en Heroku, paso uno	56
1.5.3	Despliegue en Heroku, paso dos	56
1.5.4	Comandos de Heroku	58
1.6	Conclusión	58
1.6.1	Qué aprendimos en este capítulo	59

1.7	Ejercicios	60
2	Una aplicación de juguete	63
2.1	Planificando la aplicación	64
2.1.1	Un modelo de juguete para usuarios	67
2.1.2	Un modelo de juguete para microposts	67
2.2	El recurso Users	69
2.2.1	Un recorrido por User	72
2.2.2	MVC en acción	79
2.2.3	Debilidades del recurso Users	87
2.3	El recurso Microposts	88
2.3.1	Un recorrido por Micropost	88
2.3.2	Poniendo el <i>micro</i> en microposts	92
2.3.3	Un usuario tiene muchos microposts	94
2.3.4	Jerarquías de Herencia	97
2.3.5	Desplegando la aplicación de juguete	100
2.4	Conclusión	101
2.4.1	Qué aprendimos en este capítulo	103
2.5	Ejercicios	104
3	Páginas casi estáticas	107
3.1	Preparación de la aplicación de ejemplo	108
3.2	Páginas estáticas	112
3.2.1	Páginas estáticas generadas	113
3.2.2	Páginas estáticas personalizadas	123
3.3	Empezando las pruebas	124
3.3.1	Nuestra primera prueba	128
3.3.2	Rojo	130
3.3.3	Verde	132
3.3.4	Refactor	135
3.4	Páginas ligeramente dinámicas	136
3.4.1	Probando los títulos (Rojo)	137
3.4.2	Agregando títulos a las páginas (Verde)	139
3.4.3	Estructuras de diseño y Ruby embebido (Refactorización)	142

3.4.4	Configurando la ruta raíz	149
3.5	Conclusión	149
3.5.1	Qué aprendimos en este capítulo	152
3.6	Ejercicios	152
3.7	Configuración avanzada de pruebas	155
3.7.1	Reporteadores de mini-pruebas	155
3.7.2	Silenciador de traza	156
3.7.3	Pruebas automatizadas con Guard	157
4	Rails con sabor a Ruby	167
4.1	Motivación	167
4.2	Cadenas de caracteres y métodos	173
4.2.1	Comentarios	174
4.2.2	Cadenas de caracteres	175
4.2.3	Objetos y paso de mensajes	178
4.2.4	Definiciones de Métodos	181
4.2.5	Regresando al auxiliar para el título	183
4.3	Otras estructuras de datos	184
4.3.1	Arreglos y Rangos	184
4.3.2	Bloques	188
4.3.3	Arreglos Hash y Símbolos	191
4.3.4	CSS revisado	196
4.4	Clases Ruby	199
4.4.1	Constructores	199
4.4.2	Herencia de Clases	200
4.4.3	Modificando las clases incorporadas	205
4.4.4	Una clase controlador	206
4.4.5	Una clase usuario	209
4.5	Conclusión	211
4.5.1	Qué aprendimos en este capítulo	212
4.6	Ejercicios	213
5	Rellenando la estructura de diseño	215
5.1	Agregando un poco de estructura	216

5.1.1	Navegación del sitio	218
5.1.2	Bootstrap y CSS personalizado	225
5.1.3	Parciales	233
5.2	Sass y la cadena de procesos conectados	239
5.2.1	La cadena de procesos conectados	240
5.2.2	Hojas de estilo sintácticamente impresionantes	244
5.3	Enlaces de la estructura de diseño	251
5.3.1	Página de Contacto	251
5.3.2	Rutas Rails	254
5.3.3	Usando rutas nombradas	256
5.3.4	Pruebas a los enlaces de la estructura de diseño	257
5.4	Registro de usuario: primer paso	262
5.4.1	Controlador de usuarios	262
5.4.2	URL de Registro	264
5.5	Conclusión	265
5.5.1	Qué aprendimos en este capítulo	267
5.6	Ejercicios	268
6	Modelando usuarios	271
6.1	Modelo usuario	272
6.1.1	Migraciones de base de datos	274
6.1.2	El archivo modelo	281
6.1.3	Creando objetos usuario	282
6.1.4	Buscando objetos usuario	285
6.1.5	Actualizando objetos usuario	287
6.2	Validaciones de usuario	289
6.2.1	Una prueba de validez	289
6.2.2	Validación de presencia	291
6.2.3	Validación de longitud	295
6.2.4	Validación de formato	297
6.2.5	Validación de unicidad	304
6.3	Agregando una contraseña segura	312
6.3.1	Una contraseña procesada con la función hash	313
6.3.2	El usuario tiene una contraseña segura	315

6.3.3	Estándares mínimos de contraseña	317
6.3.4	Creando y autenticando un usuario	319
6.4	Conclusión	322
6.4.1	Qué aprendimos en este capítulo	324
6.5	Ejercicios	324
7	Registro	327
7.1	Mostrando usuarios	327
7.1.1	Depuración y ambientes Rails	328
7.1.2	Un recurso usuarios	336
7.1.3	Depurador	341
7.1.4	Una imagen gravatar y una barra lateral	343
7.2	El formulario de registro	349
7.2.1	Usando <code>form_for</code>	353
7.2.2	El formulario HTML de registro	356
7.3	Registros fallidos	361
7.3.1	Un formulario funcional	361
7.3.2	Parámetros fuertes	367
7.3.3	Mensajes de error de registro	369
7.3.4	Una prueba para envío de datos inválidos	375
7.4	Registros exitosos	379
7.4.1	El formulario de registro terminado	379
7.4.2	El flash	383
7.4.3	El primer registro	386
7.4.4	Una prueba para el envío de datos válidos	390
7.5	Despliegue de grado profesional	391
7.5.1	SSL en producción	392
7.5.2	Servidor web de Producción	393
7.5.3	Número de versión de Ruby	395
7.6	Conclusión	397
7.6.1	Qué aprendimos en este capítulo	397
7.7	Ejercicios	398

8 Inicio y Cierre de Sesión	403
8.1 Sesiones	404
8.1.1 Controlador de Sesiones	405
8.1.2 Formulario para inicio de sesión	408
8.1.3 Encontrando y autenticando al usuario	412
8.1.4 Mostrando un mensaje flash	417
8.1.5 Una prueba para flash	419
8.2 Entrar al sistema	422
8.2.1 El método <code>log_in</code>	423
8.2.2 Usuario actual	426
8.2.3 Actualizando los enlaces de la estructura de diseño	430
8.2.4 Probando los cambios a la estructura de diseño	435
8.2.5 Inicio de sesión al registrarse	442
8.3 Cerrando sesión	444
8.4 Recuérdame	447
8.4.1 Recordando el token y la digestión	448
8.4.2 Inicio de sesión con recuerdo	454
8.4.3 Olvidando usuarios	464
8.4.4 Dos defectos sutiles	466
8.4.5 Casilla “Recuérdame”	471
8.4.6 Pruebas para Recuérdame	478
8.5 Conclusión	486
8.5.1 Qué aprendimos en este capítulo	487
8.6 Ejercicios	488
9 Actualizando, mostrando y borrando usuarios	493
9.1 Actualizando usuarios	493
9.1.1 Formulario de edición	494
9.1.2 Ediciones fallidas	500
9.1.3 Probando las ediciones fallidas	502
9.1.4 Ediciones exitosas (con TDD)	504
9.2 Autorización	508
9.2.1 Requeriendo a los usuarios que inicien sesión	509
9.2.2 Requeriendo el usuario correcto	516

9.2.3	Redirección amigable	521
9.3	Mostrando todos los usuarios	526
9.3.1	Listado de Usuarios	526
9.3.2	Usuarios de ejemplo	531
9.3.3	Paginación	534
9.3.4	Prueba al listado de usuarios	538
9.3.5	Refactorización parcial	542
9.4	Borrando usuarios	544
9.4.1	Usuarios administrativos	546
9.4.2	La acción <code>destroy</code>	549
9.4.3	Pruebas al borrado de usuario	552
9.5	Conclusión	556
9.5.1	Qué aprendimos en este capítulo	558
9.6	Ejercicios	559
10	Activación de la cuenta y reinicio de contraseña	563
10.1	Activación de la cuenta	564
10.1.1	El recurso para la activación de la cuenta	565
10.1.2	Método de envío de correo electrónico para la activación de la cuenta	573
10.1.3	Activando la cuenta	588
10.1.4	Prueba de activación y refactorización	599
10.2	Reinicio de Contraseña	603
10.2.1	El recurso de reinicio de contraseñas	604
10.2.2	Controlador y formulario de reinicio de contraseñas	612
10.2.3	Método gestor de reinicio de contraseña	616
10.2.4	Reiniendo la contraseña	625
10.2.5	Prueba de reinicio de contraseña	631
10.3	Correo electrónico en producción	636
10.4	Conclusión	638
10.4.1	Qué aprendimos en este capítulo	641
10.5	Ejercicios	641
10.6	Comparación de la prueba de expiración	644

11 Micromensajes de usuario	647
11.1 Un modelo de micromensaje	647
11.1.1 El modelo básico	648
11.1.2 Validaciones de micromensajes	650
11.1.3 Asociaciones Usuario / Micromensaje	654
11.1.4 Refinamiento de Micromensajes	657
11.2 Mostrando los micromensajes	663
11.2.1 Desplegando micromensajes	663
11.2.2 Micromensajes de ejemplo	668
11.2.3 Pruebas a los micromensajes del perfil	673
11.3 Manipulando micromensajes	680
11.3.1 Control de acceso de micromensajes	681
11.3.2 Creando micromensajes	684
11.3.3 Un avance prototipo	693
11.3.4 Destruyendo micromensajes	702
11.3.5 Pruebas de los micromensajes	705
11.4 Micromensajes de imágenes	710
11.4.1 Carga de imágenes básica	710
11.4.2 Validación de imagen	717
11.4.3 Modificando el tamaño de la imagen	720
11.4.4 Carga de imágenes en producción	722
11.5 Conclusión	727
11.5.1 Qué aprendimos en este capítulo	730
11.6 Ejercicios	731
12 Siguiendo usuarios	735
12.1 El modelo relación	736
12.1.1 Un problema con el modelo de datos (y una solución)	736
12.1.2 Asociaciones Usuario / relación	746
12.1.3 Validaciones de relación	749
12.1.4 Usuarios seguidos	750
12.1.5 Seguidores	754
12.2 Una interfaz web para seguir usuarios	756
12.2.1 Datos de ejemplo de seguimiento	756

12.2.2 Estadísticas y el formulario de seguimiento	758
12.2.3 Páginas de seguidos y segidores	767
12.2.4 Un botón de seguimiento funcionando de forma estándar	780
12.2.5 Un botón de seguimiento funcionando con Ajax	782
12.2.6 Pruebas de seguimiento	790
12.3 El status de avance	792
12.3.1 Motivación y estrategia	792
12.3.2 Una primera implementación del avance	795
12.3.3 Subconsultas	799
12.4 Conclusión	802
12.4.1 Guía de recursos adicionales	805
12.4.2 Qué aprendimos en este capítulo	807
12.5 Ejercicios	807

Prefacio

Mi anterior compañía (CD Baby) fue una de las primeras en cambiar efusivamente a Ruby on Rails, y luego aún más efusivamente, regresar a PHP (búsqueme en Google para leer acerca de esta tragedia). Este libro escrito por Michael Hartl vino tan altamente recomendado que tuve que probarlo, y el *Tutorial de Ruby on Rails* es lo que utilicé para regresar a Rails de nuevo.

Aunque he trabajado a mi modo con varios libros de Rails, éste es el que finalmente me hizo “comprenderlo”. Todo se hace muy a “el modo Rails”—una forma que se sentía muy artificial para mí antes, pero ahora, luego de haber leído este libro, finalmente se siente natural. Éste es también el único libro de Rails que realiza desarrollo basado en pruebas todo el tiempo, un enfoque altamente recomendado por los expertos pero que nunca había sido demostrado claramente antes. Finalmente, al incluir Git, GitHub, y Heroku en los ejemplos del demo, el autor realmente le da una sensación de lo que es hacer un proyecto en el mundo real. Los códigos de ejemplo del tutorial no están aislados.

La narración lineal es un gran formato. Personalmente, me empapé del *Tutorial Rails* durante tres largos días,¹ realizando todos los ejemplos y retos al final de cada capítulo. Hágalo de principio a fin, sin saltar de un lado a otro, y obtendrá el beneficio primordial.

Disfrútelo!

Derek Sivers (sivers.org)

Fundador, CD Baby

¹¡Esto no es usual! Completar todo el libro usualmente toma *mucho* más tiempo que tres días.

Agradecimientos

El *Tutorial de Ruby on Rails* debe mucho a mi libro previo de Rails, *RailsSpace*, y por tanto a mi coautor [Aurelius Prochazka](#). Me gustaría agradecer a Aure tanto el trabajo que hizo en aquél libro, como su apoyo para éste. También me gustaría agradecer a Debra Williams Cauley, mi editora tanto en *RailsSpace* como en el *Tutorial de Ruby on Rails*; siempre que ella me siga llevando a los juegos de béisbol, yo continuaré escribiendo libros para ella.

Me gustaría expresar mi reconocimiento a una larga lista de Rubyistas que me han enseñado e inspirado a través de los años: 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, Steven 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, la buena gente de Pivotal Labs, la pandilla Heroku, los chicos de pensamiento robot, y al equipo de GitHub. Finalmente, muchos, muchos lectores—demasiados para ser nombrados—han contribuído con un gran número de reportes de errores y sugerencias durante la escritura de este libro; les agradezco su ayuda para hacerlo lo mejor posible.

Sobre el Autor

Michael Hartl es el autor del [*Ruby on Rails Tutorial*](#), uno de los tutoriales líderes en la introducción al desarrollo web, y es también el cofundador de [Softcover](#), una plataforma de auto-publicación para autores. Previo a esto Michael escribió y desarrolló *RailsSpace*, un libro y tutorial de Rails en extremo obsoleto y desarrolló Insoshi, la alguna vez popular y ahora obsoleta red social en Ruby on Rails. En el 2011, Michael recibió el reconocimiento de [Ruby Hero Award](#) por sus contribuciones a la comunidad de Ruby. Es graduado del [Harvard College](#), es [doctor en Física](#) por [Caltech](#), y es un miembro del programa de emprendedores [Y Combinator](#).

Derechos de autor y licencia

Tutorial de Ruby on Rails: Aprenda Desarrollo Web con Rails. Copyright © 2014 por Michael Hartl. Todo el código fuente en el *Tutorial Ruby on Rails* está disponible bajo la [Licencia MIT](#)² y la [Licencia Cerveza-ware](#) conjuntamente.

La Licencia MIT

TRADUCCIÓN NO OFICIAL NI AUTORIZADA PARA SU USO COMO LICENCIA LEGAL

Copyright (c) 2014 Michael Hartl

Se concede permiso por la presente, de forma gratuita, a cualquier persona que obtenga una copia de este software y de los archivos de documentación asociados (el "Software"), para utilizar el Software sin restricción, incluyendo sin limitación los derechos de usar, copiar, modificar, fusionar, publicar, distribuir, sublicenciar, y/o vender copias de este Software, y para permitir a las personas a las que se les proporcione el Software a hacer lo mismo, sujeto a las siguientes condiciones:

El aviso de copyright anterior y este aviso de permiso se incluirán en todas las copias o partes sustanciales del Software.

EL SOFTWARE SE PROPORCIONA "TAL CUAL", SIN GARANTÍA DE NINGÚN TIPO, EXPRESA O IMPLÍCITA, INCLUYENDO PERO NO LIMITADO A GARANTÍAS DE COMERCIALIZACIÓN, IDONEIDAD PARA UN PROPÓSITO PARTICULAR Y NO INFRACCIÓN. EN NINGÚN CASO LOS AUTORES O TITULARES DEL COPYRIGHT SERÁN RESPONSABLES DE NINGUNA RECLAMACIÓN, DAÑOS U OTRAS RESPONSABILIDADES, YA SEA EN UN LITIGIO, AGRAVIO O DE OTRO MODO, QUE SURJA DE O EN CONEXIÓN CON EL SOFTWARE O EL USO U OTRO TIPO DE ACCIONES EN EL SOFTWARE.

²Nota del Traductor: Dado que esta licencia la emite el MIT (Massachusetts Institute of Technology), el texto oficial sólo se encuentra disponible en inglés. La traducción que se presenta, fue tomada de la Wikipedia, donde también puede consultar la versión original.

```
/*
 * -----
 * "LA LICENCIA CERVEZA-WARE" (Revisión 43):
 * Michael Hartl escribió este código. Mientras que conserve este aviso, usted
 * puede hacer lo que quiera con este material. Si algún día nos conocemos, y si
 * usted cree que este material lo vale, puede comprarme una cerveza a cambio.
 * -----
 */
```

Capítulo 1

Desde cero hasta el despliegue

Bienvenido al *Tutorial de Ruby on Rails: Aprenda desarrollo web con Rails*. El propósito de este libro es enseñarle cómo desarrollar aplicaciones web personalizadas, y nuestra herramienta elegida es el popular marco de trabajo web [Ruby on Rails](#). Si usted es neófito en el tema, el *Tutorial de Ruby on Rails* le proporcionará una minuciosa introducción al desarrollo de aplicaciones web, incluyendo un aprendizaje básico en Ruby, Rails, HTML & CSS, bases de datos, control de versiones, pruebas, y despliegue—suficiente para lanzarlo en una carrera como desarrollador web o emprendedor en tecnología. Por otra parte, si usted ya conoce el desarrollo web, este libro le enseñará rápidamente lo esencial del marco de trabajo de Rails, incluyendo MVC y REST, generadores, migraciones, ruteos y Ruby embebido. En cualquier caso, cuando usted termine el *Tutorial de Ruby on Rails* estará en posición de beneficiarse de libros mucho más avanzados, blogs y videos que son parte del florescente ecosistema educativo en programación.¹

El *Tutorial de Ruby on Rails* adopta un enfoque integrado al desarrollo web construyendo tres aplicaciones de ejemplo con creciente sofisticación, empezando con una aplicación mínima *hola* ([Sección 1.3](#)), una aplicación ligera-

¹La versión más actualizada del *Tutorial de Ruby on Rails* puede encontrarla en el sitio web del libro <http://www.railstutorial.org/>. Si usted está leyendo este libro fuera de línea, asegúrese de revisar la [versión en línea](#) del Tutorial de Ruby on Rails en <http://www.railstutorial.org/book> para conocer las últimas actualizaciones.

mente más compleja *de juguete* ([Capítulo 2](#)), y una aplicación *de ejemplo* real ([Capítulos 3 al 12](#)). Como se deduce de sus nombres genéricos, las aplicaciones desarrolladas en el *Tutorial de Ruby on Rails* no son específicas de ningún tipo de sitio web; aunque la aplicación de ejemplo final tendrá algo más que una sutil semejanza con cierto [sitio social de micromensajes](#) muy popular (un sitio que, coincidentemente, también fue originalmente escrito en Rails), el énfasis a lo largo del tutorial está en principios generales, de forma que usted obtenga una base sólida, sin importar qué tiempo de aplicaciones web desee usted crear.

Una pregunta común es cuánto conocimiento o experiencia previa es necesaria para aprender desarrollo web mediante el *Tutorial de Ruby on Rails*. Como se comenta con mayor profundidad en la [Sección 1.1.1](#), el desarrollo web es un reto, especialmente para los novatos. Aunque el tutorial fue originalmente diseñado para lectores con alguna experiencia previa en programación o en desarrollo web, en realidad ha encontrado una audiencia significativa entre los desarrolladores principiantes. Teniendo esto en cuenta, esta tercera edición del *Tutorial de Rails Tutorial* ha dado varios pasos importantes encaminados a reducir la barrera para empezar con Rails ([Recuadro 1.1](#)).

Recuadro 1.1. Reduciendo la barrera

Esta tercera edición del *Tutorial de Ruby on Rails* tiene por objetivo reducir la barrera para empezar con Rails de varias formas:

- El uso de un ambiente de desarrollo estándar en la nube ([Sección 1.2](#)), el cual deja a un lado muchos de los problemas asociados con la instalación y configuración de un nuevo sistema
- El uso del “default stack” de Rails, incluyendo el marco de trabajo incorporado para realizar mini-pruebas
- La eliminación de muchas dependencias externas (RSpec, Cucumber, Capybara, Factory Girl)
- Un enfoque más ligero y flexible de las pruebas

- Aplazamiento o eliminación de opciones de configuración complejas (Spork, RubyTest)
- Menos énfasis en características específicas de cierta versión de Rails, con mayor énfasis en principios generales del desarrollo web

Espero que estos cambios hagan que la tercera edición del *Tutorial de Ruby on Rails* sea accesible a una audiencia aún mayor que la de las versiones anteriores.

En este primer capítulo, empezaremos con Ruby on Rails instalando todo el software necesario y configurando nuestro ambiente de desarrollo ([Sección 1.2](#)). Luego crearemos nuestra primera aplicación Rails, llamada **hello_app**. El *Tutorial de Rails* enfatiza el uso de buenas prácticas para el desarrollo de software, por lo que inmediatamente después de crear nuestro nuevo proyecto de Rails, lo pondremos bajo un control de versiones con Git ([Sección 1.4](#)). Y, créalo o no, en este capítulo llegaremos a colocar nuestra primera aplicación en internet al *desplegarla* en producción ([Sección 1.5](#)).

En el [Capítulo 2](#), crearemos un segundo proyecto, cuyo propósito es demostrar las funcionalidades básicas de una aplicación Rails. Para ponernos en marcha rápidamente, construiremos una *aplicación de juguete* (llamada **toy_app**) usando un programa generador de estructuras temporales ([Recuadro 1.2](#)) para generar código; como este código es tan feo como complejo, en el [Capítulo 2](#) nos enfocaremos en interactuar con la aplicación de juguete a través de sus *URIs* (a menudo llamadas *URLs*)² cuando lo está utilizando.

El resto del tutorial se enfoca en desarrollar una gran *aplicación real de ejemplo* (llamada **sample_app**), escribiendo todo el código desde cero. Desarrollaremos tal aplicación usando una combinación de *bosquejos, desarrollo orientado a pruebas* (TDD por sus siglas en inglés *Test Driven Development*) y *pruebas de integración*. Empezaremos en el [Capítulo 3](#) creando páginas estáticas y luego agregando un poco de contenido dinámico. Nos desviaremos

²URI por sus siglas en inglés *Uniform Resource Identifier*, mientras que la ligeramente menos genérica URL por sus siglas en inglés *Uniform Resource Locator*. En la práctica, la URL es usualmente equivalente a “lo que usted ve en la barra de direcciones de su navegador”.

ligeramente en el Capítulo 4 para aprender un poco acerca del lenguaje Ruby que es la base de Rails. Luego, en los Capítulos 5 a 10, completaremos las bases de la aplicación de ejemplo creando una estructura de diseño para el sitio, un modelo de datos de usuario y un registro completo junto con su sistema de autenticación (incluyendo la activación de la cuenta y el reinicio de contraseñas). Finalmente, en los Capítulos 11 a 12 agregaremos las características sociales y de micromensajes para hacer del sitio de ejemplo, un sitio funcional.

Recuadro 1.2. Generador de estructuras temporales: Más rápido, más fácil, más seductor

Desde el principio, Rails se ha beneficiado de un palpable sentido de la emoción, empezando con el famoso video de [un weblog en 15-minutos](#) del creador de Rails, David Heinemeier Hansson. Ese video y sus sucesores son una excelente probadita del poder de Rails; le recomiendo que los vea. Pero le advierto: para realizar la asombrosa hazaña en quince minutos utilizan una característica llamada *scaffolding*, que depende en gran medida de *código generado*, mágicamente creado por el comando **generate scaffold** de Rails.

Al escribir el tutorial de Ruby on Rails, es tentador depender de la generación de código—es [más rápido, más fácil, más seductor](#). Pero la complejidad y la cantidad de código generado de esta manera puede resultar completamente abrumador para un desarrollador principiante de Rails; usted puede ser capaz de utilizarlo, pero probablemente no lo entienda. Seguir el desarrollo usando la generación automática de código lo pone en riesgo de convertirlo en un generador virtuoso de scripts con poco (y frágil) conocimiento real de Rails.

En el *Tutorial de Ruby on Rails*, tomaremos la ruta (casi) opuesta: aunque en el Capítulo 2 desarrollaremos una pequeña aplicación de juguete usando esta técnica, la parte más importante del *Tutorial de Rails* es la aplicación de ejemplo, la cual empezaremos a escribir en el Capítulo 3. Durante el desarrollo de la aplicación de ejemplo, escribiremos *pequeños pedazos* de código—suficientemente simples de entender, pero también suficientemente nuevos como para ser desafiantes. El efecto acumulado será un conocimiento más profundo y más flexible de Rails,

proporcionándole una buena base para escribir casi cualquier tipo de aplicación web.

1.1 Introducción

Ruby on Rails (o sólo “Rails” para abreviar) es un marco de trabajo para desarrollo web escrito en el lenguaje de programación Ruby. Desde su debut en 2004, Ruby on Rails se ha convertido rápidamente en una de las herramientas más poderosas y populares para construir aplicaciones web dinámicas. Rails es utilizado en compañías tan diversas como [AirBnB](#), [Basecamp](#), [Disney](#), [GitHub](#), [Hulu](#), [Kickstarter](#), [Shopify](#), [Twitter](#) y las [Páginas Amarillas](#). Existen también muchas tiendas de desarrollo web que se especializan en Rails, tales como [ENTP](#), [Thoughtbot](#), [Pivotal Labs](#), [Hashrocket](#) y [HappyFunCorp](#), además de innumerables consultores independientes, entrenadores y contratistas.

¿Qué es lo que hace a Rails tan fantástico? Antes que nada, Ruby on Rails es 100% de código abierto, disponible bajo la permisiva [Licencia MIT](#), y como resultado no cuesta nada descargarlo ni usarlo. Rails también debe mucho de su éxito a su diseño compacto y elegante; explotando la maleabilidad del lenguaje subyacente [Ruby](#), Rails crea efectivamente un [lenguaje de dominio específico](#) para escribir aplicaciones web. Como resultado, muchas tareas de programación web comunes—tales como la generación de HTML, elaboración de modelos de datos y ruteo de URLs—son fáciles con Rails, y el código resultante de la aplicación es conciso y legible.

Rails también se adapta rápidamente a nuevos desarrollos en tecnología web y al diseño de marcos de trabajo. Por ejemplo, Rails fue uno de los primeros marcos de trabajo en implementar completamente el estilo de arquitectura REST para estructurar aplicaciones web (el cual estaremos aprendiendo a lo largo del tutorial). Y cuando otros marcos de trabajo desarrollan nuevas técnicas exitosas, el creador de Rails [David Heinemeier Hansson](#) y el [equipo principal de Rails](#) no dudan en incorporar esas ideas. Quizá el ejemplo más dramático es la fusión de Rails y Merb, un marco de trabajo web rival también en Ruby, por

lo que Rails ahora aprovecha el diseño modular y estable de la [API](#) de Merb y cuenta con un desempeño mejorado.

Finalmente, Rails se beneficia de una inusualmente entusiasta y diversa comunidad. Los resultados incluyen cientos de [contribuidores](#) de código abierto, [congresos](#) bien atendidos, y un gran número de [gemas](#) (soluciones auto-contenidas a problemas específicos tales como la paginación y la subida de imágenes a servidor), una rica variedad de blogs informativos, y una enorme cantidad de foros de discusión y canales IRC. El gran número de programadores Rails también hace más fácil de manejar los inevitables errores de aplicación: el algoritmo “busca en Google el mensaje de error” casi siempre conduce a una entrada relevante en un blog o a una discusión en un foro.

1.1.1 Prerequisitos

No hay prerequisitos formales para este libro—el *Tutorial de Ruby on Rails* contiene tutoriales integrados no sólo para Rails, sino también para el lenguaje subyacente Ruby, el marco de trabajo para pruebas por default (minitest), la línea de comandos de Unix, [HTML](#), [CSS](#), una pequeña cantidad de [JavaScript](#) y un poquito de [SQL](#). Es una gran cantidad de material para absorber, aunque yo generalmente recomiendo tener algo de bases en HTML y en programación antes de empezar este tutorial. Dicho esto, una sorprendente cantidad de principiantes han utilizado el *Tutorial de Ruby on Rails* para aprender desarrollo web desde cero, por lo que si usted tiene experiencia limitada le sugiero intentarlo. Si se siente abrumado, siempre puede empezar con uno de los recursos listados más adelante y luego regresar. Otra estrategia recomendada por múltiples lectores es simplemente hacer el tutorial dos veces; puede sorprenderle cuánto aprendió la primera vez (y cuán fácil es la segunda).

Una pregunta común al aprender Rails es si se requiere aprender Ruby primero. La respuesta depende de su estilo de aprendizaje personal y en cuánta experiencia de programación tenga. Si usted prefiere aprender todo sistemáticamente desde el principio, o si nunca ha programado antes, entonces aprender Ruby primero puede funcionarle bien, y en tal caso le recomiendo [Learn to Program](#) de Chris Pine y [Beginning Ruby](#) de Peter Cooper. Por otra parte, muchos desarrolladores principiantes en Rails están emocionados de crear aplicaciones

web, y prefieren no esperar a terminar un libro completo de Ruby antes de escribir una sola página web. En tal caso, le recomiendo seguir el pequeño tutorial interactivo de [Try Ruby](#)³ para tener una visión general antes de empezar con el *Tutorial de Rails*. Si aún así encuentra este tutorial demasiado difícil, puede intentar comenzar con [Learn Ruby on Rails](#) de Daniel Kehoe o [One Month Rails](#), dos de los cuales se enfocan en totales novatos más que el *Tutorial de Ruby on Rails*.

Al final de este tutorial, no importa dónde haya empezado, usted debería estar listo para los muchos recursos de nivel intermedio a avanzado de Rails que hay allá afuera. Aquí hay algunos que yo particularmente recomiendo:

- [Code School](#): Buenos cursos de programación interactivos en línea.
- [The Turing School of Software & Design](#): un programa de entrenamiento de tiempo completo, cuya duración es de 27 semanas en Denver, Colorado, con un [descuento de \\$500 USD](#) para los lectores de este tutorial usando el código RAILSTUTORIAL500.
- [Bloc](#): Un campamento de entrenamiento en línea con un programa estructurado, asesoría personalizada y un enfoque en aprendizaje a través de proyectos concretos. Use el cupón BLOCLOVESHARTL para obtener un descuento de \$500 USD en la inscripción (matrícula).
- [Tealeaf Academy](#): Un buen campo de entrenamiento en desarrollo Rails en línea (incluye material avanzado).
- [Thinkful](#): Una clase en línea que lo contacta con un ingeniero profesional conforme trabaja en un programa basado en proyectos.
- [Pragmatic Studio](#): Cursos de Ruby y Rails en línea, de Mike y Nicole Clark. Junto con el autor de *Programming Ruby*, Dave Thomas, Mike impartió el primer curso de Rails que tomé, allá en el año 2006.
- [RailsCasts](#) de Ryan Bates: Excelentes videos de Rails (la mayoría son gratuitos).

³<http://tryruby.org/>

- [RailsApps](#): Una gran variedad de proyectos de Rails detallados en un tema específico y tutoriales.
- [Rails Guides](#): Referencias temáticas y actualizadas de Rails.

1.1.2 Acuerdos utilizados en este libro

Los acuerdos de este libro son mayormente auto-explicativos. En esta sección, mencionaré algunos que podrían no serlo.

Muchos ejemplos del libro utilizan comandos en la línea de comandos. Por simplicidad, todos los ejemplos de línea de comandos utilizan un cursor del estilo Unix (signo de dólar), como sigue:

```
$ echo "hello, world"  
hello, world
```

Como mencionamos en la [Sección 1.2](#), le recomiendo a los usuarios de todos los sistemas operativos (especialmente Windows) que utilicen un ambiente de desarrollo en la nube ([Sección 1.2.1](#)), el cual incluye una línea de comandos Unix (Linux). Esto es particularmente útil porque Rails tiene muchos comandos que pueden ejecutarse en la línea de comandos. Por ejemplo, en la [Sección 1.3.2](#) ejecutaremos un servidor de desarrollo web local con el comando **rails server**:

```
$ rails server
```

Como con el cursor de la línea de comandos, el *Tutorial de Rails* utiliza la convención de Unix para los delimitadores de directorios (es decir, una diagonal **/**). Por ejemplo, el archivo de configuración de la aplicación de ejemplo **production.rb** se muestra como sigue:

```
config/environments/production.rb
```

Esta ruta al archivo debe entenderse como relativa al directorio raíz de la aplicación, la cual variará dependiendo del sistema; en el IDE en la nube ([Sección 1.2.1](#)), sería como ésta:

```
/home/ubuntu/workspace/sample_app/
```

Por lo que, la ruta absoluta al archivo **production.rb** sería

```
/home/ubuntu/workspace/sample_app/config/environments/production.rb
```

Por brevedad, típicamente omitiré la ruta de la aplicación y escribiré sólo **config/environments/production.rb**.

El *Tutorial de Rails* a menudo muestra la salida de varios programas (comandos de terminal, status del control de versiones, programas Ruby, etc.). Debido a innumerables pequeñas diferencias entre los sistemas de cómputo, la salida que usted visualiza no siempre coincidirá exactamente con lo que se muestra en el texto, pero esto no debe preocuparle. Adicionalmente, algunos comandos muestran errores dependiendo de su sistema; en vez de intentar la colosal tarea de documentar todos esos errores en este tutorial, se lo delego al algoritmo “busque en Google el mensaje de error”, el cual entre otras cosas es una buena práctica en el desarrollo de software en la vida real. Si usted encuentra algún problema mientras sigue el tutorial, le sugiero consultar los recursos listados en la sección [Ayuda del Tutorial de Rails](#).⁴

Como el *Tutorial de Rails* cubre las pruebas de las aplicaciones de Rails, a menudo es útil saber si un pedazo particular de código provoca que el conjunto de pruebas completo falle (indicado por el color rojo) o pase (indicado por el color verde). Por convención, el código que resulta en una prueba fallida es etiquetado como **ROJO**, mientras que el código que resulta en una prueba exitosa es etiquetado como **VERDE**.

Cada capítulo del tutorial incluye ejercicios; realizarlos es opcional pero recomendado. Con la finalidad de mantener el tema principal independiente

⁴<http://www.railstutorial.org/#help>

de los ejercicios, las soluciones generalmente no son incorporadas en listados subsecuentes. En el raro caso de que la solución a un ejercicio se utilice posteriormente, éste será solucionado explícitamente en el texto principal.

Finalmente, por conveniencia, el *Tutorial de Ruby on Rails* adopta dos convenciones diseñadas para hacer que los códigos de ejemplo sean más fáciles de entender. La primera consiste en que algunos listados de código incluyen una o más líneas resaltadas, como se muestra a continuación:

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

Tales líneas usualmente indican el código nuevo más importante del ejemplo dado, y a menudo (aunque no sucede siempre) representan la diferencia entre el nuevo listado de código y los anteriores. La segunda consiste en que, por claridad y simplicidad, muchos de los listados de código incluyen puntos suspensivos verticales, como los siguientes:

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

Estos puntos representan código omitido que no debe copiarse literalmente.

1.2 En funcionamiento

Aún para desarrolladores Rails con experiencia, instalar Ruby, Rails, y todo el software asociado puede ser un ejercicio frustrante. Las causas del problema son los múltiples escenarios: diferentes sistemas operativos, números de versión, preferencias en el editor de texto y en el ambiente de desarrollo integrado (IDE, por sus siglas en inglés *Integrated Development Environment*), etc. Los

usuarios que ya tienen instalado un IDE en su equipo local pueden utilizar sus configuraciones preferidas, pero (como se menciona en el Recuadro 1.1) se les sugiere a los nuevos usuarios dejar a un lado esta instalación y los detalles de configuración y se les invita a que utilicen un *ambiente de desarrollo integrado en la nube*. Este ambiente en la nube se ejecuta dentro de un navegador web ordinario y por tanto funciona de la misma forma en las diferentes plataformas, lo cual es especialmente útil para los sistemas operativos (tales como Windows) en los que el desarrollo con Rails ha sido históricamente difícil. Si aún a pesar de los retos que involucra, usted prefiere completar el *Tutorial de Ruby on Rails* usando un ambiente de desarrollo local, le recomiendo seguir las instrucciones que están en [InstallRails.com](#).⁵

1.2.1 Ambiente de desarrollo

Considerando las diferentes personalizaciones idiosincrásicas, existen probablemente tantos ambientes de desarrollo como programadores Rails. Para evitar esta complejidad, el *Tutorial de Ruby on Rails* se estandariza en el excelente ambiente de desarrollo en la nube [Cloud9](#). En particular, para esta tercera edición me complace tener una alianza con Cloud9 para ofrecer un ambiente de desarrollo específicamente adaptado a las necesidades de este tutorial. El espacio de trabajo resultante viene pre-configurado con la mayoría del software necesario para desarrolladores profesionales de Rails, incluyendo Ruby, RubyGems y Git. (De hecho, el único software que instalaremos por separado es Rails mismo, y esto es intencional ([Sección 1.2.2](#)).) El IDE en la nube incluye los tres componentes esenciales que se requieren para desarrollar aplicaciones web: un editor de texto, un navegador de archivos y una terminal de línea de comandos ([Figura 1.1](#)). Entre otras características, el editor de texto del IDE en la nube soporta la búsqueda global “Buscar dentro de archivos” que considero esencial para navegar dentro de un proyecto grande de Ruby o de Rails.⁶ Finalmente, aún si usted decide no utilizar el IDE en la nube de forma exclusiva en

⁵Aún así, a los usuarios de Windows se les advierte que el instalador de Rails recomendado por InstallRails a menudo está desactualizado, y es muy probable que sea incompatible con el presente tutorial.

⁶Por ejemplo, para buscar la definición de una función llamada `foo`, usted puede realizar una búsqueda global de “def foo”.

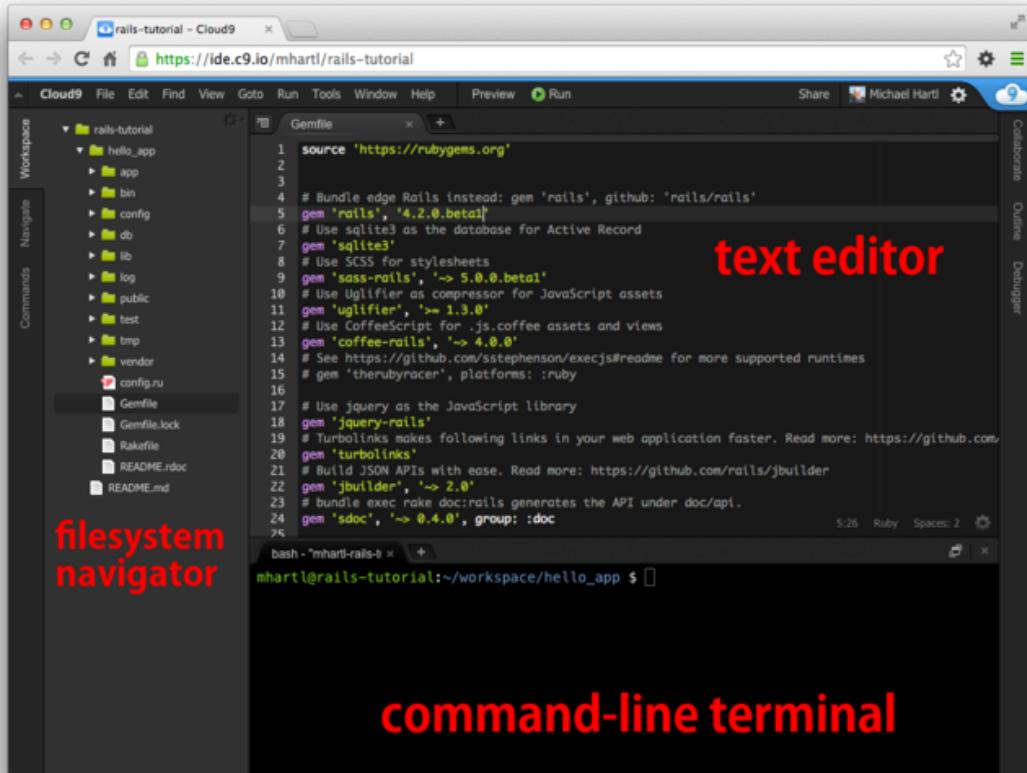


Figura 1.1: La anatomía de el IDE en la nube.

la vida real (y ciertamente le recomiendo aprender a utilizar otras herramientas también), proporciona una excelente introducción a las capacidades generales de los editores de texto y otras herramientas de desarrollo.

Estos son los pasos para empezar a utilizar el ambiente de desarrollo en la nube:

1. Regístrese para obtener una cuenta gratuita en Cloud9⁷
2. Presione el botón “Ir al panel de control”
3. Seleccionar “Crear un nuevo espacio de trabajo”

⁷<https://c9.io/web/sign-up/free>

4. Como se muestra en la [Figura 1.2](#), cree un espacio de trabajo llamado “rails-tutorial” (*no* “rails_tutorial”), seleccione la opción “Privado para la gente que yo invite”, y seleccione el ícono del Tutorial de Rails (*no* el ícono de Ruby on Rails)
5. Presione el botón “Crear”
6. Luego de que Cloud9 haya terminado de preparar su espacio de trabajo, selecciónelo y presione el botón “Empezar a editar”

Como el uso de dos espacios para indentar es una convención casi-universal en Ruby, le recomiendo configurar el editor para que utilice estos dos espacios en vez de los cuatro que usa por default. Como se muestra en la [Figura 1.3](#), usted puede hacer esto presionando el ícono de engrane que se encuentra en la esquina superior derecha y luego seleccione “Editor de Código (Ace)” para editar la configuración “Soft Tabs”. (Observe que esto toma efecto inmediatamente; por lo que no necesita presionar el botón “Guardar”).

1.2.2 Instalando Rails

El ambiente de desarrollo de la [Sección 1.2.1](#) incluye todo el software necesario para empezar excepto por el mismo Rails.⁸ Para instalar Rails, utilizaremos el comando **gem** proporcionado por el administrador de paquetes *RubyGems*, lo cual implica escribir el comando que se muestra en el [Listado 1.1](#) en su terminal de línea de comandos. (Si usted está desarrollando en su sistema local, esto significa que utilice una terminal regular; si está utilizando el IDE en la nube, esto significa que utilice el área de línea de comandos que se muestra en la [Figura 1.1](#).)

Listado 1.1: Instalando Rails con un número específico de versión.

```
$ gem install rails -v 4.2.2
```

⁸Actualmente, Cloud9 incluye una versión vieja de Rails que es incompatible con el presente tutorial, razón por la cual es importante que lo instalemos nosotros mismos.

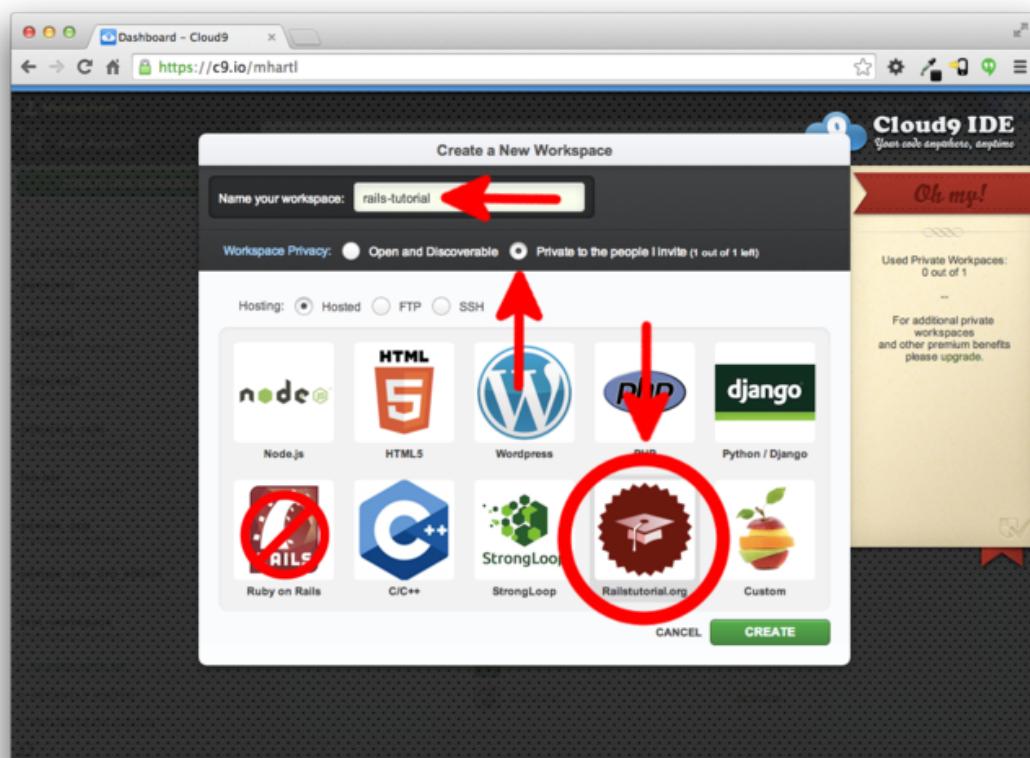


Figura 1.2: Creando un nuevo espacio de trabajo en Cloud9.

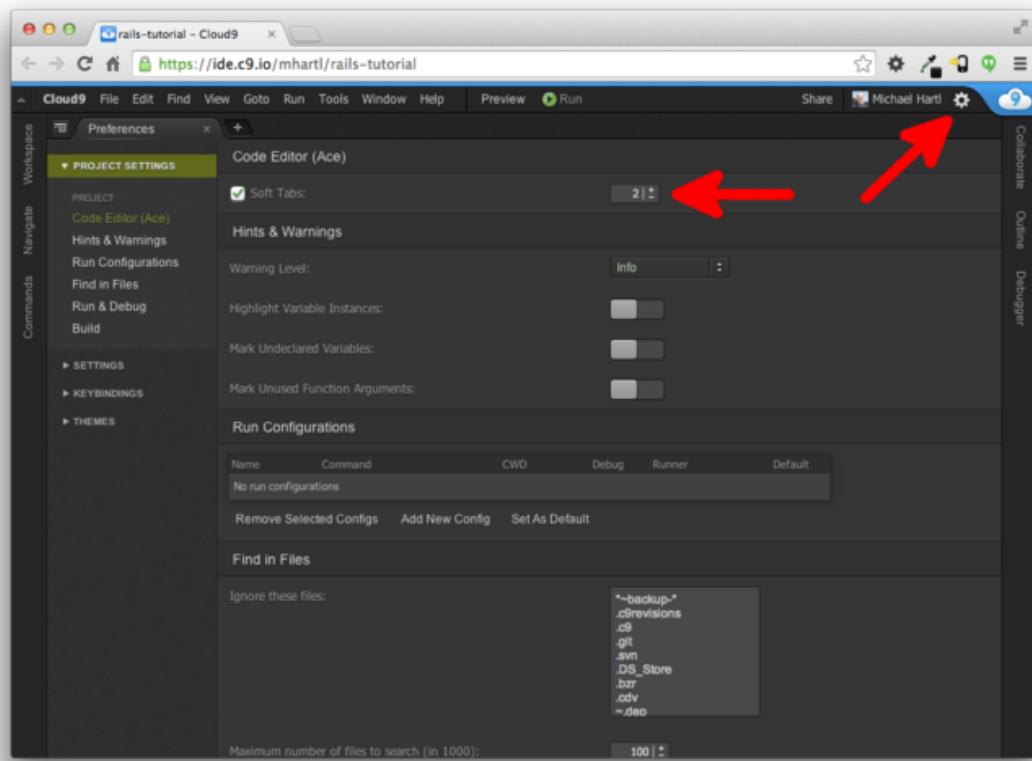


Figura 1.3: Configurando Cloud9 para que utilice dos espacios en la indentación.

Aquí la opción `-v` se encarga de que la versión de Rails especificada sea instalada, lo cual es importante para obtener resultados consistentes con este tutorial.

1.3 La primera aplicación

Siguiendo una [larga tradición](#) en la programación de computadoras, nuestro objetivo para la primera aplicación es escribir un programa “hola mundo”. En particular, crearemos una aplicación simple que muestre la cadena “hello, world!” en una página web, tanto en nuestro ambiente de desarrollo ([Sección 1.3.4](#)) como en internet ([Sección 1.5](#)).

Virtualmente todas las aplicaciones Rails empiezan de la misma forma, ejecutando el comando `rails new`. Este útil comando crea un esqueleto de la aplicación Rails en un directorio que usted elija. Para empezar, los usuarios que *no* están utilizando el IDE en la nube que recomendamos en la [Sección 1.2.1](#), deberían crear un directorio **`workspace`** para sus proyectos de Rails, si es que aún no tienen uno ([Listado 1.2](#)) y luego posicionarse en ese directorio. (El [Listado 1.2](#) utiliza los comandos de Unix `cd` y `mkdir`; vea el Recuadro 1.3 si usted no está familiarizado con estos comandos.)

Listado 1.2: Creando un directorio **`workspace`** para los proyectos de Rails (no es necesario en la nube).

```
$ cd                      # Change to the home directory.
$ mkdir workspace          # Make a workspace directory.
$ cd workspace/            # Change into the workspace directory.
```

Recuadro 1.3. Un breve curso acerca de la línea de comandos de Unix

Para los lectores que utilizan Windows o (en menor grado pero aún significativo) Macintosh OS X, la línea de comandos de Unix puede que no les resulte familiar. Afortunadamente, si usted está utilizando el ambiente en la nube recomendado, automáticamente tiene acceso a la línea de comandos de Unix (Linux) ejecutando una [terminal de línea de comandos](#) estándar conocida como **Bash**.

Descripción	Comando	Ejemplo
enlista contenidos	ls	\$ ls -l
crea directorio	mkdir <dirname>	\$ mkdir workspace
cambia de directorio	cd <dirname>	\$ cd workspace/
cambia al directorio padre		\$ cd ..
cambia al directorio <i>home</i>		\$ cd ~ o sólo \$ cd
cambia al subdirectorio de <i>home</i> indicado		\$ cd ~/workspace/
renombra un archivo	mv <source> <target>	\$ mv README.rdoc README.md
copia un archivo	cp <source> <target>	\$ cp README.rdoc README.md
borra un archivo	rm <file>	\$ rm README.rdoc
borra un directorio vacío	rmdir <directory>	\$ rmdir workspace/
borra un directorio no vacío	rm -rf <directory>	\$ rm -rf tmp/
muestra el contenido de un archivo	cat <file>	\$ cat ~/.ssh/id_rsa.pub

Tabla 1.1: Algunos comandos Unix comunes.

La idea básica de la línea de comandos es simple: al emitir comandos cortos, los usuarios realizan una gran cantidad de operaciones, tales como crear directorios (`mkdir`), mover y copiar archivos (`mv` y `cp`) y navegar en el sistema de archivos al cambiar de directorio (`cd`). Aunque la línea de comandos puede parecer primitiva para los usuarios que están familiarizados principalmente con interfaces gráficas (GUIs), las apariencias engañan: la línea de comandos es una de las herramientas más poderosas disponibles para un desarrollador. De hecho, rara vez usted verá el escritorio de un desarrollador experimentando sin varias ventanas de terminales abiertas ejecutando programas de línea de comandos.

El tema en general es profundo, pero para los propósitos de este tutorial sólo necesitamos algunos de los comandos de Unix más comunes, como se resume en la Tabla 1.1. Para conocer más detalladamente la línea de comandos de Unix, vea *Conquering the Command Line* de Mark Bates (disponible en [versión en línea gratuita, libros y videos](#)).

El siguiente paso tanto en sistemas locales como en el IDE en la nube es crear la primera aplicación usando el comando del Listado 1.3. Observe que el Listado 1.3 explícitamente incluye el número de versión de Rails (`_4.2.2_`) como parte del comando. Esto asegura que la misma versión de Rails que in-

stalamos en el Listado 1.1 es utilizada para crear la estructura de archivos de la primera aplicación. (Si el comando del Listado 1.3 regresa un error como “Could not find ‘railties’”, significa que usted no tiene instalada la versión correcta de Rails y debería verificar nuevamente que ha ejecutado el comando del Listado 1.1 exactamente como está escrito.)

Listado 1.3: Ejecutando `rails new` (con un número de versión específico).

```
$ cd ~/workspace
$ rails _4.2.2_ new hello_app
  create
  create  README.rdoc
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
  create  app/assets/javascripts/application.js
  create  app/assets/stylesheets/application.css
  create  app/controllers/application_controller.rb
  .
  .
  .
  create  test/test_helper.rb
  create  tmp/cache
  create  tmp/cache/assets
  create  vendor/assets/javascripts
  create  vendor/assets/javascripts/.keep
  create  vendor/assets/stylesheets
  create  vendor/assets/stylesheets/.keep
    run  bundle install
Fetching gem metadata from https://rubygems.org/.....
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake 10.3.2
Using i18n 0.6.11
.
.
.
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
  run bundle exec spring binstub --all
* bin/rake: spring inserted
* bin/rails: spring inserted
```

Como puede observar al final del Listado 1.3, ejecutar `rails new` automáticamente invoca el comando `bundle install` luego de haber terminado la

Archivo/Directorio	Propósito
<code>app/</code>	Código principal de la aplicación (app), incluye modelos, vistas, controladores y auxiliares
<code>app/assets</code>	Recursos de la aplicación tales como hojas de estilo en cascada (CSS), archivos JavaScript e imágenes
<code>bin/</code>	Archivos binarios ejecutables
<code>config/</code>	Configuración de la aplicación
<code>db/</code>	Archivos de base de datos
<code>doc/</code>	Documentación de la aplicación
<code>lib/</code>	Biblioteca de módulos
<code>lib/assets</code>	Biblioteca de recursos tales como hojas de estilo en cascada (CSS), archivos JavaScript e imágenes
<code>log/</code>	Archivos de bitácora de la aplicación
<code>public/</code>	Datos accesibles al público (por ejemplo, vía navegadores web), tales como páginas de error
<code>bin/rails</code>	Un programa para generar código, abrir sesiones de consola o iniciar un servidor local
<code>test/</code>	Pruebas de la aplicación
<code>tmp/</code>	Archivos temporales
<code>vendor/</code>	Código de terceros tales como complementos (plugins) y gemas
<code>vendor/assets</code>	Recursos de terceros tales como hojas de estilo en cascada (CSS), archivos JavaScript e imágenes
<code>README.rdoc</code>	Una breve descripción de la aplicación
<code>Rakefile</code>	Tareas de utilería disponibles mediante el comando <code>rake</code>
<code>Gemfile</code>	Gemas requeridas para esta aplicación
<code>Gemfile.lock</code>	Una lista de gemas utilizadas para asegurar que todas las copias de la aplicación utilicen las mismas versiones
<code>config.ru</code>	Un archivo de configuración para el software intermediario <code>Rack</code>
<code>.gitignore</code>	Patrones de nombres de archivos que deben ser ignorados por Git

Tabla 1.2: Un resumen de la estructura de directorios por default de Rails.

creación de archivos. Revisaremos con mayor detalle qué significa esto al empezar la Sección 1.3.1.

Observe cuántos archivos y directorios ha creado el comando `rails`. Esta estructura estándar de directorios y archivos (Figura 1.4) es una de las muchas ventajas de Rails; le lleva a usted de cero a una aplicación funcional (mínima). Más aún, como la estructura es común a todas las aplicaciones de Rails, usted puede orientarse inmediatamente cuando revise el código de alguien más. Un resumen de los archivos Rails creados por default, se muestra en la Tabla 1.2; aprenderemos más acerca de estos archivos y directorios a lo largo del libro. En particular, empezando la Sección 5.2.1 revisaremos el directorio `app/assets`, como parte de la *cadena de procesos conectados* que facilita más que nunca la organización y despliegue de recursos tales como hojas de estilo en cascada y archivos JavaScript.

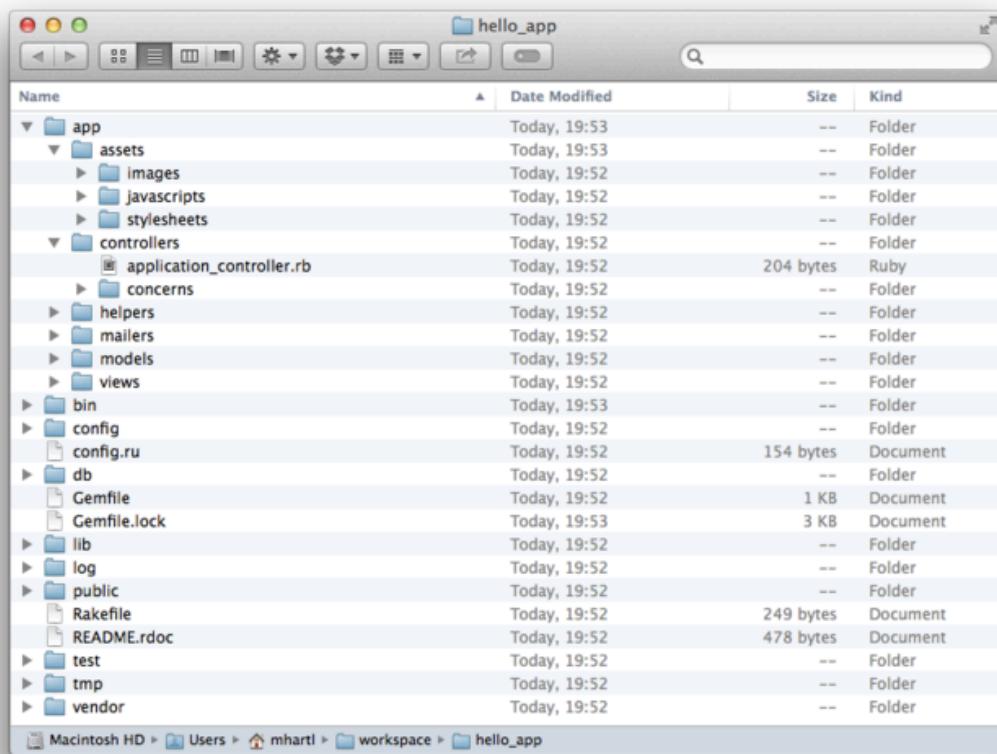


Figura 1.4: La estructura de directorios de una aplicación Rails recién creada.

1.3.1 Bundler

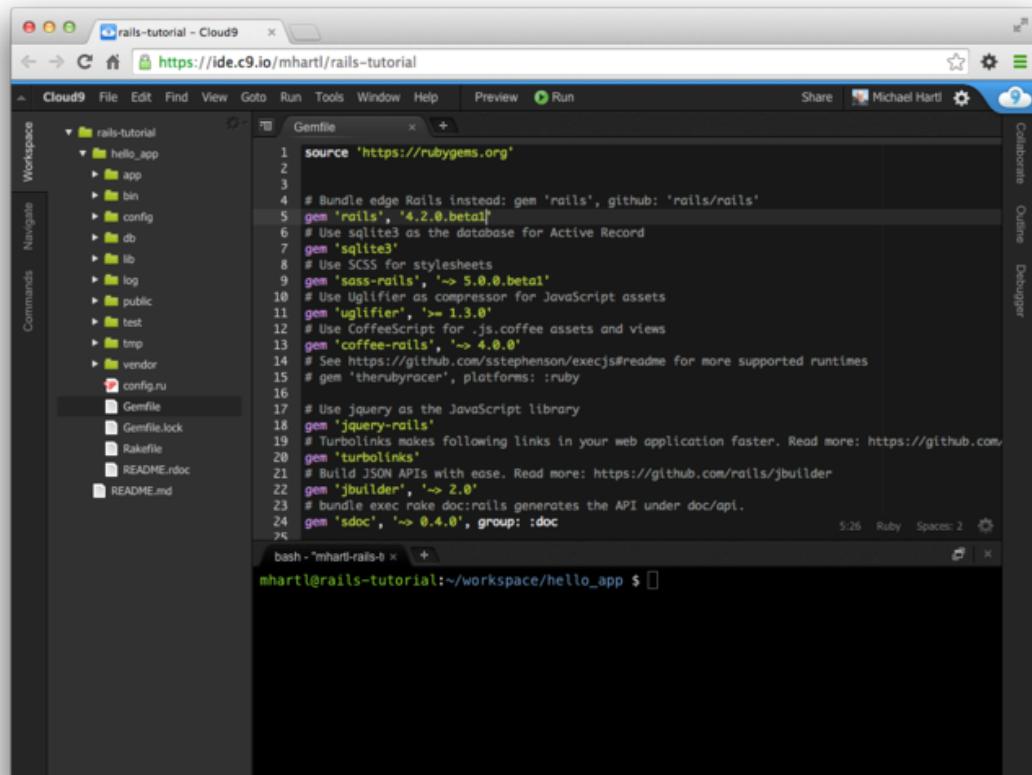
Luego de crear una aplicación nueva de Rails, el siguiente paso es utilizar *Bundler* para instalar e incluir las gemas necesarias para la aplicación. Como mencionamos brevemente en la Sección 1.3, Bundler es ejecutado automáticamente (mediante **bundle install**) por el comando **rails**, pero en esta sección haremos algunos cambios a las gemas de la aplicación indicadas por default y ejecutaremos Bundler de nuevo. Esto implica abrir el archivo **Gemfile** en un editor de texto. (Con el IDE en la nube, esto significa dar click en la flecha del navegador de archivos para abrir el directorio de la aplicación de ejemplo y dar doble click en el ícono **Gemfile**.) Aunque los números de versión exacta y los detalles pueden variar ligeramente, los resultados deben verse similares a los de la Figura 1.5 y los del Listado 1.4. (El código de este archivo es Ruby, pero en este momento no se preocupe de la sintaxis; en el Capítulo 4 revisaremos Ruby con mayor detalle.) Si los archivos y los directorios no aparecen como los de la Figura 1.5, presione el ícono de engrane en el navegador de archivos y seleccione “Refrescar árbol de archivos”. (Como regla general, deber refrescar el árbol de archivos cada vez que los directorios o archivos no aparezcan como se espera.)

Listado 1.4: El archivo **Gemfile** por default en el directorio **hello_app**.

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

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.2'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use SCSS for stylesheets
gem 'sass-rails', '→ 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '≥ 1.3.0'
# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '→ 4.0.0'
# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes following links in your web application faster. Read more:
```



The screenshot shows the Cloud9 IDE interface. The title bar reads "rails-tutorial - Cloud9" and the address bar shows "https://ide.c9.io/mhartl/rails-tutorial". The main area displays the contents of the "Gemfile" located in the "rails-tutorial/hello_app" directory. The file lists various Ruby gems and their versions, including rails (~> 4.2.0.beta1), sqlite3, sass-rails (~> 5.0.0.beta1), uglifier (~> 1.3.0), coffee-rails (~> 4.0.0), jquery-rails, turbolinks, jbuilder (~> 2.0), and sdoc (~> 0.4.0) for generating API documentation. Below the code editor is a terminal window titled "bash - mhard-rails-0" showing the command "mhartl@rails-tutorial:~/workspace/hello_app \$". The left sidebar shows the project structure with files like Gemfile, Gemfile.lock, Rakefile, README.rdoc, and README.md.

```
source 'https://rubygems.org'  
# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'  
gem 'rails', '~> 4.2.0.beta1'  
# Use sqlite3 as the database for Active Record  
gem 'sqlite3'  
# Use SCSS for stylesheets  
gem 'sass-rails', '~> 5.0.0.beta1'  
# Use Uglifier as compressor for JavaScript assets  
gem 'uglifier', '>= 1.3.0'  
# Use CoffeeScript for .js.coffee assets and views  
gem 'coffee-rails', '~> 4.0.0'  
# See https://github.com/sstephenson/execjs#readme for more supported runtimes  
# gem 'therubyracer', platforms: :ruby  
  
# Use jquery as the JavaScript library  
gem 'jquery-rails'  
# Turbolinks makes following links in your web application faster. Read more: https://github.com/turbolinks/turbolinks  
gem 'turbolinks'  
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder  
gem 'jbuilder', '~> 2.0'  
# bundle exec rake doc:rails generates the API under doc/api.  
gem 'sdoc', '~> 0.4.0', group: :doc
```

Figura 1.5: El archivo **Gemfile** por default abierto en un editor de texto.

```
# https://github.com/rails/turbolinks
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0', group: :doc

# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Unicorn as the app server
# gem 'unicorn'

# Use Capistrano for deployment
# gem 'capistrano-rails', group: :development

group :development, :test do
  # Call 'debugger' anywhere in the code to stop execution and get a
  # debugger console
  gem 'byebug'

  # Access an IRB console on exceptions page and /console in development
  gem 'web-console', '~> 2.0.0.beta2'

  # Spring speeds up development by keeping your application running in the
  # background. Read more: https://github.com/rails/spring
  gem 'spring'
end
```

Muchas de estas líneas están comentadas mediante el símbolo de número `#`; están ahí para mostrarle algunas de las gemas comúnmente utilizadas y para dar ejemplos de la sintaxis de Bundler. Por ahora, no necesitamos más que las gemas por default.

A menos que usted especifique un número de versión al comando `gem`, Bundler automáticamente instalará la versión más reciente de la gema requerida. Un ejemplo de este caso es:

```
gem 'sqlite3'
```

Existen también otras dos formas de especificar un rango de versiones para una gema, lo cual nos permite tener control, hasta cierto punto, de la versión utilizada por Rails. La primera se indica como sigue:

```
gem 'uglifier', '>= 1.3.0'
```

Esto instala la versión más reciente de la gema **uglifyer** (que se encarga de comprimir los archivos de la cadena de procesos conectados) siempre que ésta sea mayor o igual a la versión **1.3.0**—aún si ésta es, digamos, la versión **7.2**. El segundo método se muestra a continuación:

```
gem 'coffee-rails', '~> 4.0.0'
```

Esto instala la versión más reciente de la gema **coffee-rails** siempre que ésta sea mayor que la versión **4.0.0** y sea menor que la versión **4.1**. En otras palabras, la notación `>=` siempre instala la gema más reciente, mientras que la notación `~>` sólo instala gemas actualizadas en liberaciones menores (es decir, de **4.0.0** a **4.0.1**), pero no liberaciones mayores (por ejemplo, de **4.0** a **4.1**). Desafortunadamente, la experiencia muestra que aún liberaciones menores pueden causar problemas, por lo que en el *Tutorial de Ruby on Rails* tomaremos precauciones extremas al incluir el número exacto de versión para todas las gemas. Usted puede utilizar la versión más reciente de cualquier gema, incluyéndola mediante la construcción `~>` en el archivo **Gemfile** (lo cual recomiendo para los usuarios más avanzados), pero queda advertido que esto puede causar que los ejemplos y ejercicios del tutorial se comporten de forma impredecible.

Convertir el archivo **Gemfile** del Listado 1.4 para que utilice las versiones de gema exactas produce el código que se muestra en el Listado 1.5. Observe que hemos aprovechado la oportunidad para que la gema `sqlite3` sea incluida sólo en ambientes de desarrollo o pruebas (Sección 7.1.1), lo cual evita conflictos potenciales con la base de datos utilizada por Heroku (Sección 1.5).

Listado 1.5: Un archivo **Gemfile** con una versión explícita para cada gema de Ruby.

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

```

gem 'rails',                  '4.2.2'
gem 'sass-rails',             '5.0.2'
gem 'uglifier',               '2.5.3'
gem 'coffee-rails',            '4.1.0'
gem 'jquery-rails',            '4.0.3'
gem 'turbolinks',              '2.3.0'
gem 'jbuilder',                '2.2.3'
gem 'sdoc',                   '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',               '1.3.9'
  gem 'byebug',                 '3.4.0'
  gem 'web-console',           '2.0.0.beta3'
  gem 'spring',                  '1.1.3'
end

```

Una vez que ha introducido el contenido del [Listado 1.5](#) en el archivo **Gemfile** de la aplicación, instale las gemas utilizando **bundle install**.⁹

```

$ cd hello_app/
$ bundle install
Fetching source index for https://rubygems.org/
.
.
.
```

El comando **bundle install** puede tomar unos momentos en ejecutarse, pero cuando termina, nuestra aplicación estará lista para correr.

1.3.2 rails server

Gracias a la ejecución de **rails new** en la Sección 1.3 y a **bundle install** en la Sección 1.3.1, tenemos ya una aplicación que podemos ejecutar—pero ¿cómo? Afortunadamente, Rails viene con un programa de línea de comandos o *script*, que ejecuta un servidor web *local* para auxiliarnos en el desarrollo de nuestra aplicación. El comando exacto depende del ambiente en el que usted

⁹Como observamos en la [Tabla 3.1](#), usted puede omitir **install**, puesto que el comando **bundle** mismo, es un alias de **bundle install**.

lo esté utilizando: en un sistema local, ejecute solamente **rails server** (Listado 1.6), mientras que en Cloud9 necesita proporcionar una *dirección IP* adicional y un *número de puerto* para indicarle al servidor Rails la dirección que puede utilizar para hacer visible la aplicación al mundo exterior (Listado 1.7).¹⁰ (Cloud9 utiliza las *variables de ambiente* especiales: **\$IP** y **\$PORT** para asignar la dirección IP y el número de puerto dinámicamente. Si usted desea ver los valores de estas variables, escriba **echo \$IP** o **echo \$PORT** en la línea de comandos.) Si su sistema se queja de la falta de un motor para ejecutar JavaScript, visite la página [execjs](#) en GitHub para obtener una lista de candidatos. En particular le recomiendo instalar Node.js.

Listado 1.6: Ejecutando el servidor Rails en una máquina local.

```
$ cd ~/workspace/hello_app/
$ rails server
=> Booting WEBrick
=> Rails application starting on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

Listado 1.7: Ejecutando el servidor Rails en el IDE en la nube.

```
$ cd ~/workspace/hello_app/
$ rails server -b $IP -p $PORT
=> Booting WEBrick
=> Rails application starting on http://0.0.0.0:8080
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
```

Sin importar la opción que usted elija, le recomiendo que ejecute el comando **rails server** en una segunda pestaña de la terminal de forma que pueda seguir ejecutando comandos en la primera pestaña, como se muestra en las Figuras 1.6 y 1.7. (Si usted ya inició el servidor en su primera pestaña, presione Ctrl-C para deternerlo.)¹¹ En un servidor local, escriba en su navegador

¹⁰Normalmente, los sitios web corren sobre el puerto 80, pero usualmente esto requiere privilegios especiales, por lo que es una convención utilizar para el servidor de desarrollo, un número de puerto mayor que tenga menos restricciones.

¹¹Aquí “C” se refiere a la tecla del teclado, no a la letra mayúscula, por lo que no es necesario utilizar la tecla de mayúsculas.

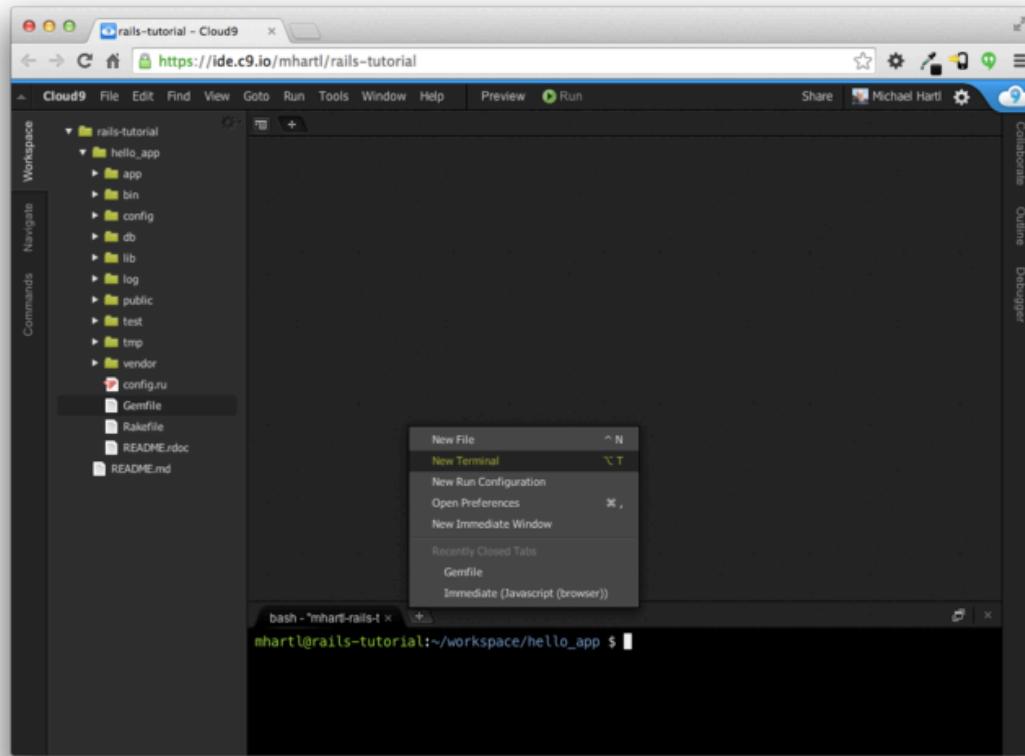


Figura 1.6: Abriendo una nueva pestaña en la terminal.

la dirección <http://localhost:3000/>; en el IDE en la nube, diríjase a Compartir y dé click en la dirección de la aplicación para abrirla ([Figura 1.8](#)). En cualquier caso, el resultado debe verse similar al de la [Figura 1.9](#).

Para ver información sobre la primera aplicación, ingrese en el enlace “ Acerca del ambiente de su aplicación”. Aunque los números de versión pueden variar, el resultado debe verse como en la [Figura 1.10](#). Por supuesto, a la larga, no necesitamos la página de Rails por default, pero es bueno verla funcionar por ahora. Eliminaremos esta página (y la reemplazaremos por una página principal personalizada) en la [Sección 1.3.4](#).

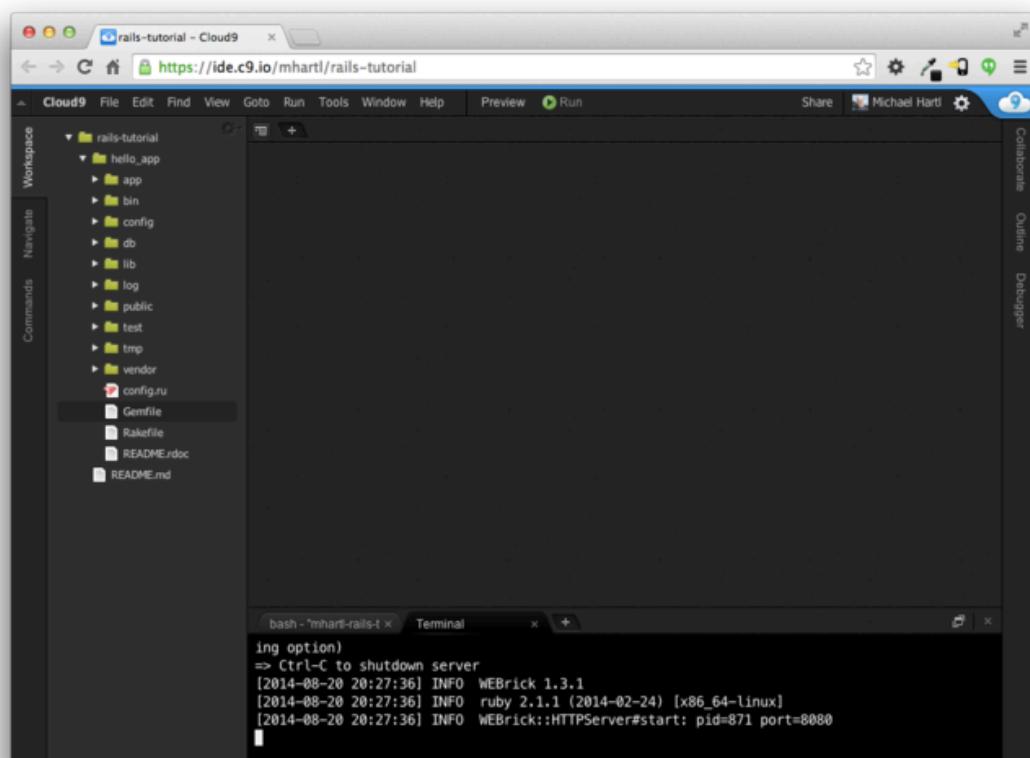


Figura 1.7: Ejecutando el servidor Rails en una pestaña separada.

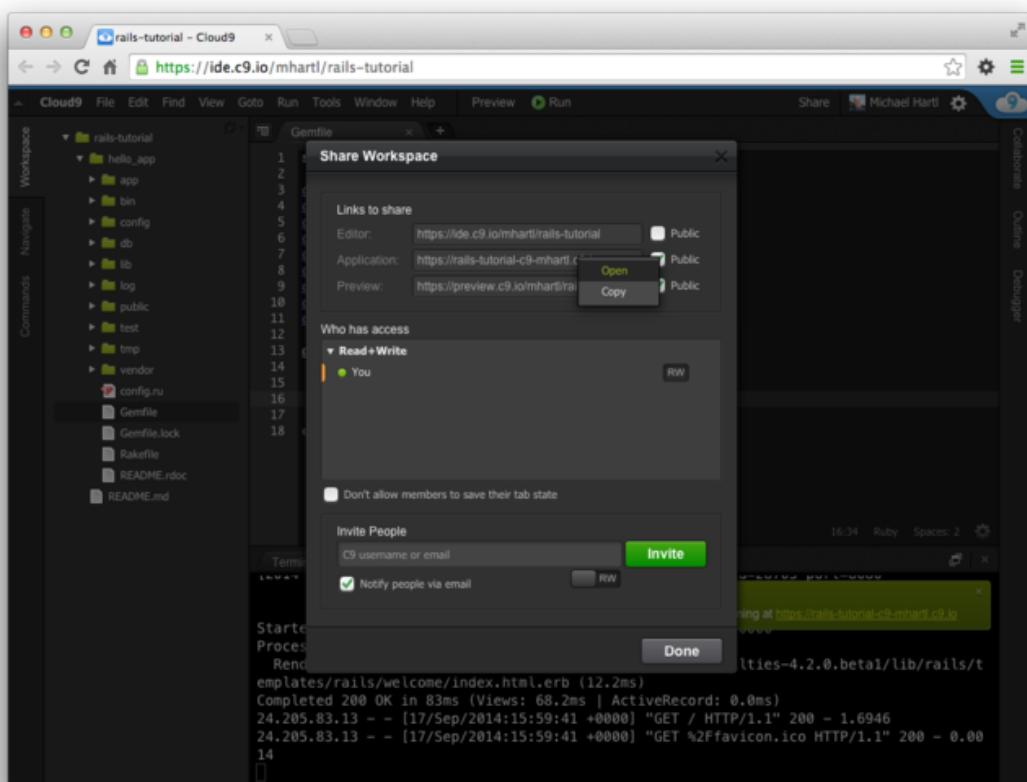


Figura 1.8: Compartiendo el servidor local que está ejecutándose en el espacio de trabajo de la nube.

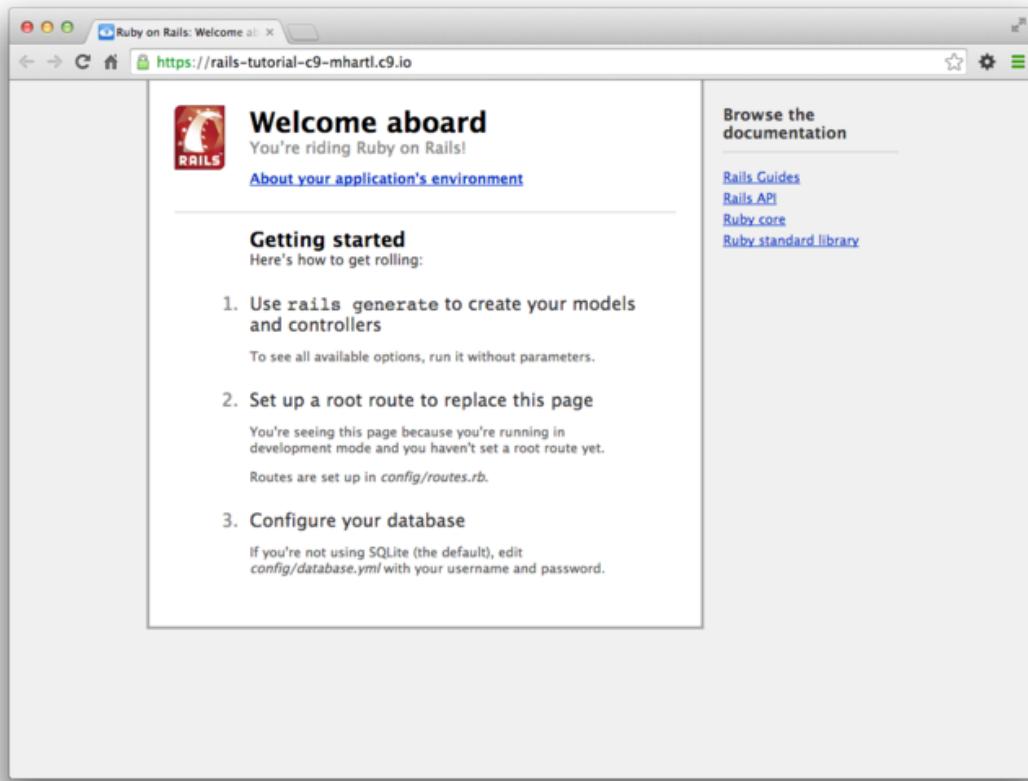


Figura 1.9: La página por default de Rails despachada por **rails server**.

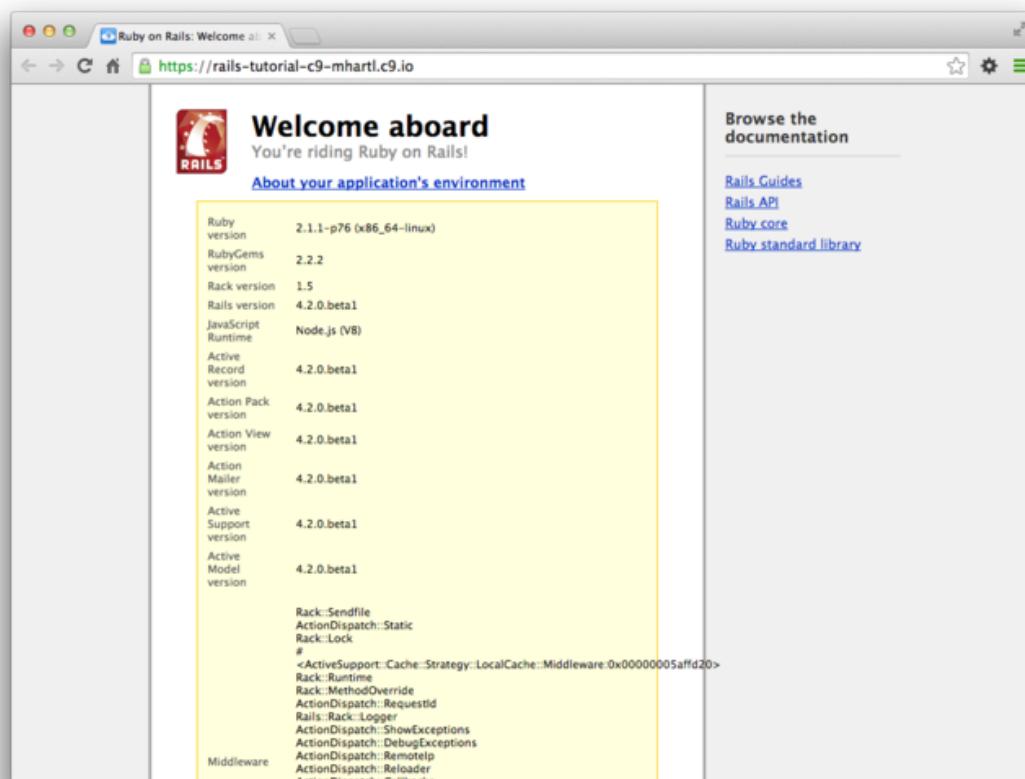


Figura 1.10: La página por default con el ambiente de la aplicación.

1.3.3 Modelo Vista-Controlador (MVC)

Aún en esta etapa temprana, es útil tener una vista general de cómo funcionan las aplicaciones de Rails (Figura 1.11). Puede que usted haya notado que la aplicación de Rails tiene una estructura estándar (Figura 1.4): un directorio de aplicación llamado `app/` con tres subdirectorios: `models`, `views`, y `controllers`. Esto es una sugerencia de que Rails sigue el patrón de arquitectura denominado **modelo vista-controlador** (MVC), el cual separa la “lógica del dominio” (también llamada “lógica de negocio”) de la lógica de presentación y entrada asociada con una interfaz de usuario gráfica (GUI). En el caso de aplicaciones web, la “lógica de dominio” típicamente consiste de modelos de datos para cosas como usuarios, artículos y productos, y la GUI es sólo una página en un navegador web.

Cuando interactuamos con una aplicación Rails, el navegador envía una *petición*, que es recibida por un servidor web y es pasada al *controlador* de Rails, el cual se encarga de realizar lo que sigue. En algunos casos, el controlador inmediatamente mostrará una *vista*, la cual es una plantilla que se convierte en HTML y es enviada de regreso al navegador. Comúnmente en los sitios dinámicos, el controlador interactúa con un *modelo*, que es un objeto Ruby que representa un elemento del sitio (tal como un usuario) y que está a cargo de comunicarse con la base de datos. Luego de invocar el modelo, el controlador genera la vista y regresa la página web completa al navegador como HTML.

Si esta discusión le parece un poco abstracta ahora, no se preocupe; nos referiremos en el futuro a esta sección frecuentemente. La Sección 1.3.4 muestra una primera aplicación tentativa de MVC, mientras que la Sección 2.2.2 incluye una discusión más detallada de MVC en el contexto de la aplicación de juguete. Finalmente, la aplicación de ejemplo utilizará todos los aspectos de MVC; revisaremos los controladores y las vistas empezando la Sección 3.2 y los modelos empezando la Sección 6.1, y veremos los tres trabajar juntos en la Sección 7.1.2.

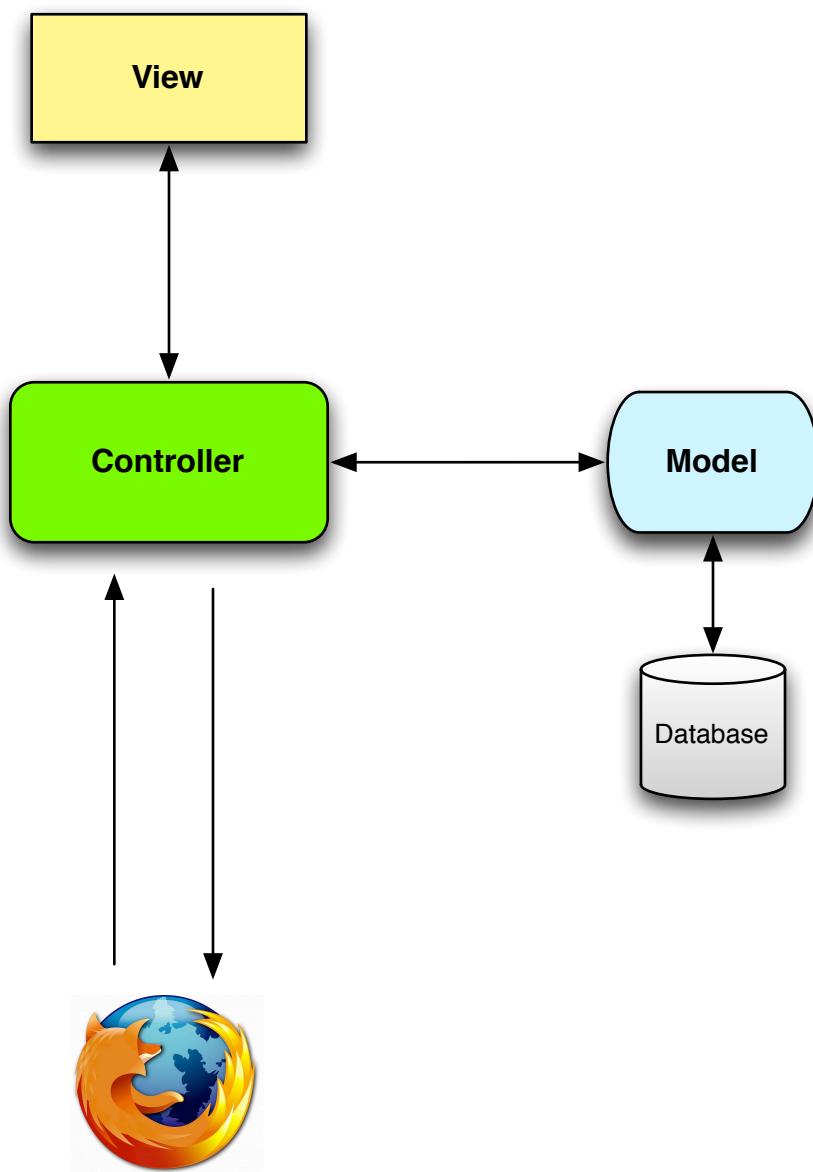


Figura 1.11: Una representación esquemática de la arquitectura del modelo vista-controlador (MVC).

1.3.4 ¡Hola mundo!

Como primera aplicación del marco de trabajo MVC, realizaremos un cambio **mínimo** a la primera aplicación agregando una *acción al controlador* para desplegar la cadena “hello, world!”. (Aprenderemos más sobre acciones de controladores al empezar la [Sección 2.2.2](#).) El resultado será reemplazar la página de Rails por default de la [Figura 1.9](#) con la página “hello, world!” que es el objetivo de esta sección.

Como se deduce de su nombre, las acciones del controlador están definidas dentro de un controlador. Nombraremos nuestra acción **hello** y la colocaremos en el controlador de la aplicación. De hecho, en este momento el controlador de la aplicación es el único controlador que tenemos, lo cual puede verificar ejecutando

```
$ ls app/controllers/*_controller.rb
```

para ver los controladores que tenemos en este momento. (Empezaremos a crear nuestros propios controladores en el [Capítulo 2](#).) El [Listado 1.8](#) muestra la definición resultante de **hello**, la cual utiliza la función **render** para regresar el texto “hello, world!”. (No se preocupe por la sintaxis de Ruby en este momento; revisaremos este tema con mayor profundidad en el [Capítulo 4](#).)

Listado 1.8: Agregando una acción **hello** al controlador de la aplicación.
app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception

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

Habiendo definido una acción que regresa la cadena deseada, necesitamos decirle a Rails que utilice esta acción en vez de la página por default que vimos en la [Figura 1.10](#). Para hacer esto, editaremos el *enrutador* de Rails, que

se encarga de determinar a dónde enviaremos las peticiones que vienen del navegador. (He omitido el enrutador de la Figura 1.11 por simplicidad, pero lo revisaremos con mayor detalle a partir de la Sección 2.2.2.) En particular, queremos cambiar la página por default, por lo que cambiaremos la *ruta raíz* que es la que determina la página que es mostrada en la *URL raíz*. Como esta es la URL para una dirección como `http://www.example.com/` (donde no viene nada luego de la primera diagonal que sigue al dominio), a menudo se hace referencia a la URL raíz como `/` (“diagonal”) por brevedad.

Como se observa en el Listado 1.9, el archivo de rutas Rails (`config/routes.rb`) incluye una línea comentada que muestra cómo estructurar la ruta raíz. Aquí “welcome” es el nombre del controlador e “index” es la acción dentro de ese controlador. Para activar la ruta raíz, descomente esta línea removiendo el carácter de signo de número y luego reemplácelo con el código del Listado 1.10, el cual le indica a Rails que envíe la ruta raíz a la acción `hello` en el controlador de la aplicación. (Como observamos en la Sección 1.1.2, los puntos suspensivos verticales indican código que ha sido omitido y que no debe copiarse literalmente.)

Listado 1.9: La ruta raíz por default (comentada).

`config/routes.rb`

```
Rails.application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  .
  .
  .
end
```

Listado 1.10: Estableciendo la ruta raíz.

`config/routes.rb`

```
Rails.application.routes.draw do
  .
  .
```

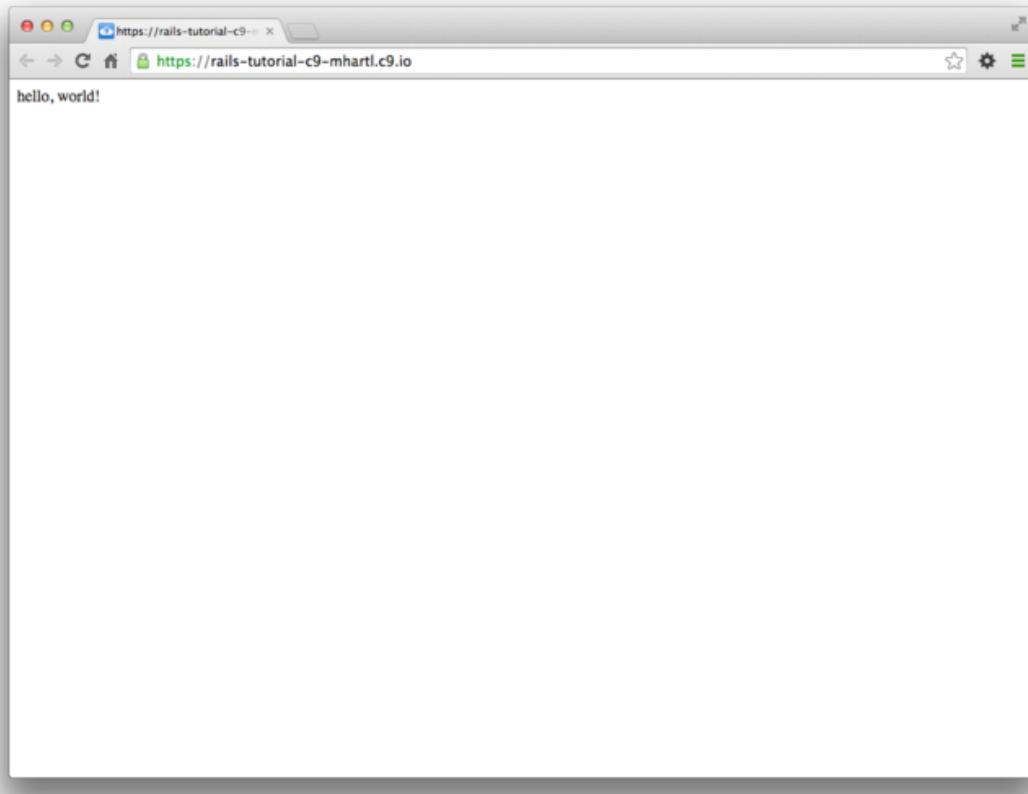


Figura 1.12: Visualizando “hello, world!” en el navegador.

```
•
# You can have the root of your site routed with "root"
root 'application#hello'
•
•
•
end
```

Con el código de los Listados 1.8 y 1.10, la ruta raíz regresa “hello, world!” como vemos en la Figura 1.12.

1.4 Control de versiones con Git

Ahora que tenemos una aplicación de Rails nueva y funcionando, dedicaremos un momento para dar un paso que, técnicamente es opcional, pero que bajo el punto de vista de los desarrolladores de software experimentados es prácticamente esencial: poner el código fuente de nuestra aplicación bajo un *control de versiones*. Los sistemas de control de versiones nos permiten dar seguimiento a los cambios que se realicen al código de nuestro proyecto, colaborar más fácilmente, y dar marcha atrás a cualquier error inadvertido (como por ejemplo borrar archivos de forma accidental). Saber cómo utilizar un sistema de control de versiones es una habilidad requerida para cualquier desarrollador de software profesional.

Existen muchas opciones para el control de versiones, pero la comunidad Rails se ha estandarizado ampliamente en [Git](#), un sistema de control de versiones distribuído originalmente desarrollado por Linus Torvalds para hospedar el kernel de Linux. Git es un tema amplio, y sólo estaremos rasgando la superficie en este libro, pero hay muchos recursos en línea buenos y gratuitos; especialmente le recomiendo [Empezando con Bitbucket](#) para un breve resumen y [Pro Git](#) de Scott Chacon como libro introductorio. Poner su código fuente bajo control de versiones con Git es *ampliamente* recomendado, no sólo porque es casi una práctica universal en el mundo Rails, sino porque le permitirá respaldar y compartir su código más fácilmente ([Sección 1.4.3](#)) así como desplegar su aplicación aquí mismo en el primer capítulo ([Sección 1.5](#)).

1.4.1 Instalación y preparación

El IDE en la nube recomendado en la [Sección 1.2.1](#) incluye Git por default, por lo que no es necesaria ninguna instalación. En cualquier otro caso, [Install-Rails.com](#) ([Sección 1.2](#)) incluye instrucciones para instalar Git en su sistema.

Preparación inicial del sistema

Antes de utilizar Git, debe realizar una serie de pasos de iniciales. Estas son configuraciones de *sistema*, lo que significa que sólo tiene que realizarlas una

vez por computadora:

```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
$ git config --global push.default matching
$ git config --global alias.co checkout
```

Observe que el nombre y la dirección de correo electrónico que usted utilice en su configuración de Git serán visibles desde cualquier repositorio que usted haga público. (Del listado anterior, únicamente las primeras dos líneas son estrictamente necesarias. La tercera línea se incluye para asegurar la compatibilidad con versiones futuras de Git. La cuarta línea, también opcional, se incluye para que pueda utilizar la abreviatura **co** en vez del comando **checkout** completo. Para mayor compatibilidad con sistemas que no tienen **co** configurado, este tutorial utilizará el comando completo **checkout** (pero en la vida real, casi siempre utilizo **git co**.)

Configuración inicial del repositorio

Ahora realizaremos algunos pasos que es necesario ejecutar cada vez que usted cree un nuevo *repositorio* (algunas veces llamado *repo* por brevedad). Para empezar, navegue al directorio raíz de la aplicación e inicialice el repositorio:

```
$ git init
Initialized empty Git repository in /home/ubuntu/workspace/hello_app/.git/
```

A continuación agregue todos los archivos del proyecto al repositorio utilizando **git add -A**:

```
$ git add -A
```

Este comando agrega todos los archivos del directorio actual excepto aquellos cuyos nombres cumplen con los patrones del archivo especial **.gitignore**. El

comando `rails new` automáticamente genera un archivo `.gitignore` apropiado para un proyecto Rails, pero usted puede agregar patrones también.¹²

Los archivos agregados son inicialmente colocados en un *área de preparación*, la cual contiene cambios pendientes en su proyecto. Usted puede ver qué archivos están en esta área utilizando el comando `status`:

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:   .gitignore
  new file:   Gemfile
  new file:   Gemfile.lock
  new file:   README.rdoc
  new file:   Rakefile
  .
  .
  .
```

(El resultado es grande, por lo que he utilizado los puntos suspensivos verticales para indicar la salida omitida.)

Para indicarle a Git que usted desea guardar los cambios, utilice el comando `commit`:

```
$ git commit -m "Initialize repository"
[master (root-commit) df0a62f] Initialize repository
:.
:.
:.
```

La opción `-m` le permite agregar un mensaje a la confirmación; si usted omite la opción `-m`, Git abrirá el editor de texto por default del sistema y le solicitará que escriba un mensaje ahí. (Todos los ejemplos de este libro utilizan la opción `-m`.)

¹²Aunque en el tutorial principal no necesitaremos editarla, un ejemplo para agregar una regla al archivo `.gitignore` se muestra en la Sección 3.7.3, la cual forma parte de la configuración de pruebas avanzadas de la Sección 3.7.

Es importante observar que las confirmaciones de Git son *locales*, guardadas únicamente en la máquina en la que se ejecuta el **commit**. Veremos cómo empujar los cambios hacia un repositorio remoto (mediante **git push**) en la Sección 1.4.4.

Por cierto, puede ver una lista de sus mensajes de confirmación usando el comando **git log**:

```
$ git log
commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Wed August 20 19:44:43 2014 +0000

    Initialize repository
```

Dependiendo de qué tan larga es la historia de confirmaciones de su repositorio, puede ser necesario que usted presione la tecla **q** para terminar la ejecución del comando.

1.4.2 ¿Qué beneficios le proporciona a usted Git?

Si usted nunca ha utilizado un sistema de control de versiones, puede ser que en este momento no quede claro el beneficio que le proporciona, por lo que le daré un ejemplo. Suponga que usted ha realizado algunos cambios accidentales, tales como (¡ah!) borrar el directorio **app/controllers/** que es indispensable en el proyecto.

```
$ ls app/controllers/
application_controller.rb  concerns/
$ rm -rf app/controllers/
$ ls app/controllers/
ls: app/controllers/: No such file or directory
```

Aquí estamos utilizando el comando **ls** de Unix para enlistar el contenido del directorio **app/controllers/** y el comando **rm** para eliminarlo (Tabla 1.1). La opción **-rf** significa “forzar recursividad”, lo cual elimina de forma recursiva todos los archivos, directorios, subdirectorios y demás, sin preguntar explícitamente por cada borrado.

Revisemos el status para ver qué ha cambiado:

```
$ git status
On branch master
Changed but not updated:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:   app/controllers/application_controller.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

Aquí vemos que un archivo ha sido borrado, pero los cambios están únicamente en el “árbol de trabajo”; no han sido confirmados aún. Esto significa que podemos deshacer estos cambios mediante el comando **checkout** con la opción **-f** para forzar la sobreescritura de los cambios actuales:

```
$ git checkout -f
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls app/controllers/
application_controller.rb  concerns/
```

Los directorios y archivos faltantes están de regreso. ¡Qué alivio!

1.4.3 Bitbucket

Ahora que hemos puesto nuestro proyecto bajo el control de versiones usando Git, es hora de subir nuestro código a [Bitbucket](#), un sitio optimizado para hospedar y compartir repositorios Git. (Ediciones previas de este tutorial utilizaban [GitHub](#); vea el Recuadro 1.4 para conocer las razones del cambio.) Poner una copia de su repositorio Git en Bitbucket tiene dos propósitos: es un respaldo completo de su código (incluyendo el historial completo de confirmaciones) y permitir cualquier colaboración futura con mucha más facilidad.

Recuadro 1.4. GitHub y Bitbucket

Por mucho los dos sitios más populares para alojar repositorios Git son GitHub y Bitbucket. Los dos servicios comparten varias similitudes: ambos permiten hospedar repositorios Git y colaborar en ellos, así como ofrecer formas convenientes de navegar y buscar en los repositorios. Las diferencias importantes (bajo la perspectiva de este tutorial) son que GitHub ofrece un número ilimitado de repositorios gratuitos (con colaboración) para proyectos “open-source”, en tanto que realiza cargos para repositorios privados, mientras que Bitbucket permite un número ilimitado de repositorios privados gratuitos, en tanto que realiza cargos si el número de colaboradores excede de cierta cantidad. Qué servicio utilice usted para un repositorio en particular, depende de sus necesidades específicas.

Ediciones previas de este libro utilizaban GitHub por su énfasis en el soporte de código open-source, pero las crecientes preocupaciones sobre seguridad hacen que recomiende que *todos* los repositorios de aplicaciones web sean privados por default. El punto es que los repositorios de aplicaciones web pueden contener información sensible, tal como llaves criptográficas y contraseñas, lo cual puede utilizarse para comprometer la seguridad de un sitio que ejecute ese código. Es posible, por supuesto, encargarnos de que esta información sea manejada de forma segura (haciendo que Git la ignore, por ejemplo), pero esto tiende a errores y requiere mayor pericia.

Sucede que, la aplicación de ejemplo creada en este tutorial es segura de exponerse en la red, pero es peligroso confiar en este hecho en general. Por lo que, para estar lo más seguros posible, seremos precavidos y utilizaremos repositorios privados por default. Puesto que GitHub realiza cargos por repositorios privados mientras que Bitbucket ofrece un número ilimitado de forma gratuita, para nuestros propósitos Bitbucket es una mejor opción que GitHub.

Empezar a utilizar Bitbucket es simple:

1. [Regístrese para obtener una cuenta de Bitbucket](#) si es que aún no tiene una.

2. Copie su *llave pública* al portapapeles. Como se indica en el Listado 1.11, los usuarios del IDE en la nube pueden ver su llave pública usando el comando **cat**, el cual pueden seleccionar y copiar. Si usted está utilizando su propio sistema y no visualiza ningún resultado cuando ejecuta el comando del Listado 1.11, siga las instrucciones de [cómo instalar una llave pública en su cuenta de Bitbucket](#).
3. Agregue su llave pública a Bitbucket dando click en la imagen del avatar en la esquina superior derecha y seleccione “Administrar cuenta” y luego “llaves SSH” (Figura 1.13).

Listado 1.11: Visualizando la llave pública mediante **cat**.

```
$ cat ~/.ssh/id_rsa.pub
```

Una vez que usted ha agregado su llave pública, presione “Crear” para [crear un nuevo repositorio](#), como se muestra en la Figura 1.14. Al completar el formulario del proyecto, recuerde habilitar la opción “Este es un repositorio privado.” Luego de presionar “Crear repositorio”, siga las instrucciones debajo de “Línea de comandos > Tengo un proyecto existente”, que debe verse similar al Listado 1.12. (Si no se parece al Listado 1.12, puede ser porque la llave pública no fue agregada correctamente, en cuyo caso le sugiero que realice nuevamente ese paso.) Cuando suba al repositorio, responda sí en caso de que se le presente la pregunta “¿Está usted seguro de que quiere continuar con la conexión (sí/no)?”

Listado 1.12: Agregando Bitbucket y subiendo al repositorio.

```
$ git remote add origin git@bitbucket.org:<username>/hello_app.git  
$ git push -u origin --all # pushes up the repo and its refs for the first time
```

Los comandos del Listado 1.12 primero le indican a Git que usted quiere agregar Bitbucket como el *origen* de su repositorio, y luego sube su repositorio al origen remoto. (No se preocupe acerca de lo que la opción **-u** hace; si tiene

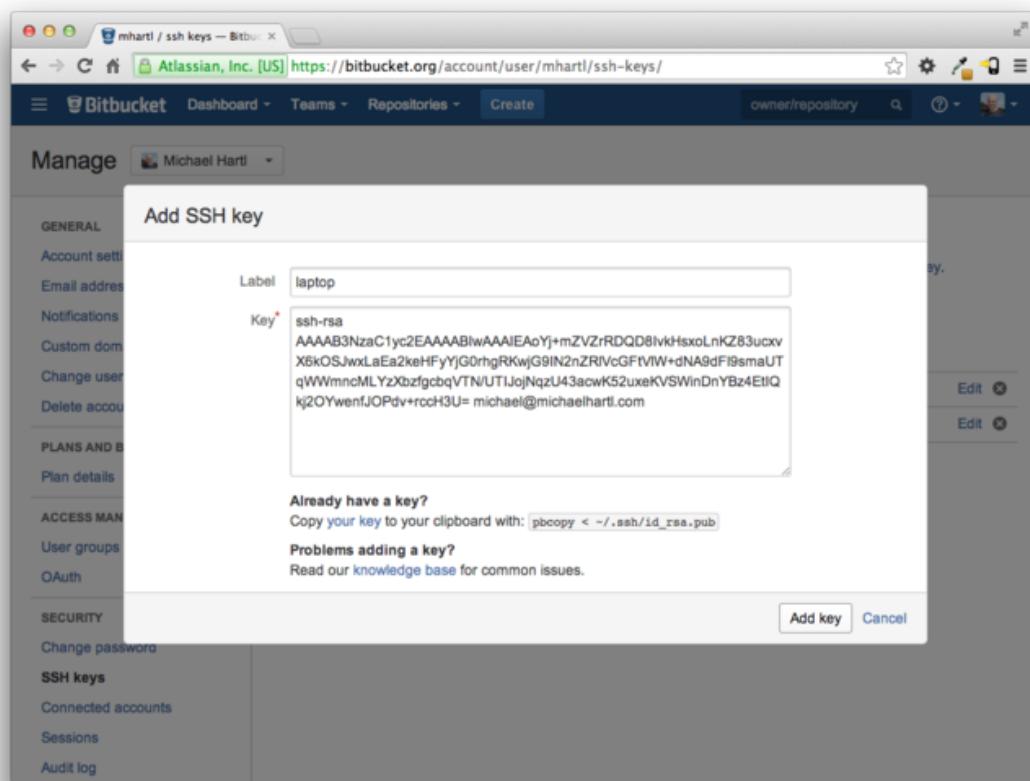


Figura 1.13: Agregando la llave pública SSH.

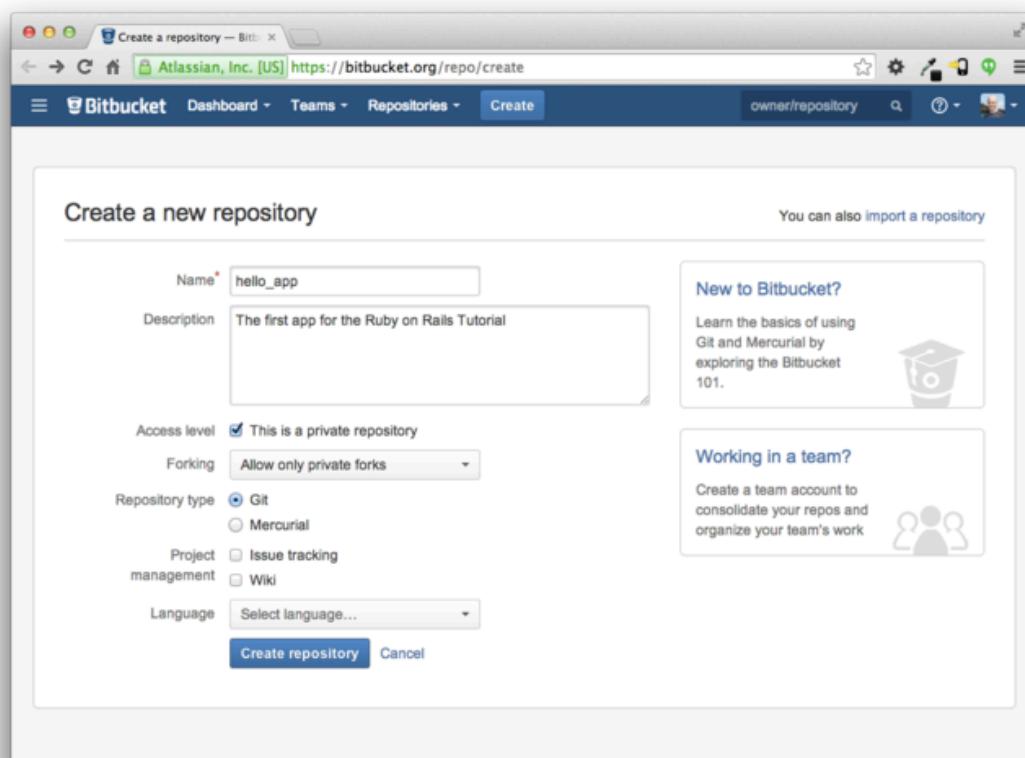


Figura 1.14: Creando el primer repositorio para una aplicación en Bitbucket.

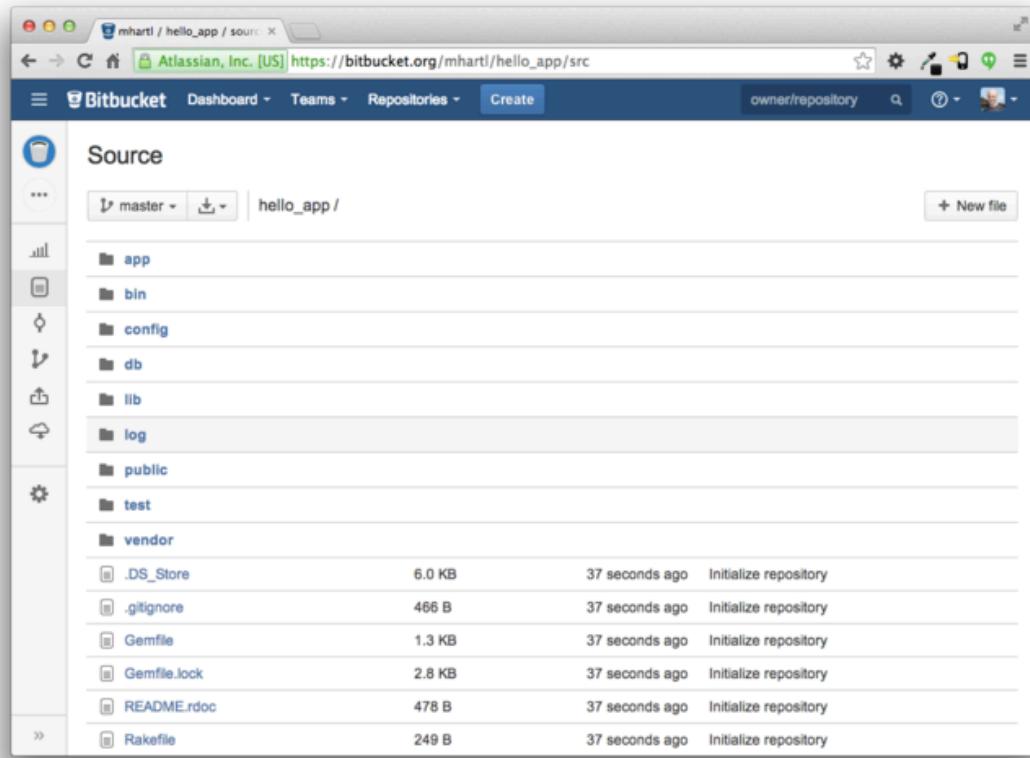


Figura 1.15: Una página de repositorio en Bitbucket.

curiosidad, realice una búsqueda en la web de “git set upstream”.) Por supuesto, debe reemplazar <username> con su nombre de usuario real. Por ejemplo, el comando que yo ejecuté fue

```
$ git remote add origin git@bitbucket.org:mhartl/hello_app.git
```

El resultado es una página en Bitbucket para el repositorio hello_app, con navegación de archivos, historial completo de confirmaciones y muchas otras bondades (Figura 1.15).

1.4.4 Ramas, edición, confirmación e integración

Si usted ha seguido los pasos indicados en la Sección 1.4.3, puede haberse percatado de que Bitbucket no detecta automáticamente el archivo **README.rdoc** de nuestro repositorio, y por tanto, se queja de que la página principal de nuestro repositorio no tiene un archivo README presente (Figura 1.16). Esto es una indicación de que el formato **rdoc** no es suficientemente común como para que Bitbucket lo entienda de forma automática, y de hecho prácticamente cualquier otro desarrollador y yo preferimos utilizar *Markdown*. En esta sección cambiaremos el archivo **README.rdoc** por **README.md**, mientras que aprovechamos la oportunidad de agregarle algo de contenido específico del Tutorial de Rails. En el proceso, veremos un primer ejemplo de los flujos de ramas, edición, confirmación e integración que recomiendo utilizar con Git.¹³

Rama

Git es increíblemente bueno creando *ramas*, las cuales son copias de un repositorio donde podemos realizar (quizá de forma experimental) cambios sin modificar los archivos originales. En la mayoría de los casos, el repositorio original es la rama *master*, y podemos crear una nueva rama para un tópico usando **checkout** junto con la opción **-b**:

```
$ git checkout -b modify-README
Switched to a new branch 'modify-README'
$ git branch
  master
* modify-README
```

Aquí el segundo comando, **git branch**, sólo enumera todas las ramas locales y el asterisco ***** indica en qué rama nos encontramos actualmente. Observe que **git checkout -b modify-README** no sólo crea una nueva rama sino que nos lleva a ella, como nos indica el asterisco junto a **modify-README**. (Si usted configuró el alias **co** en la Sección 1.4, puede utilizar el comando **git co -b modify-README** en lugar de la versión más larga.)

¹³Eche un vistazo a [Atlassian's SourceTree app](#) si desea una herramienta gráfica para visualizar los repositorios Git.

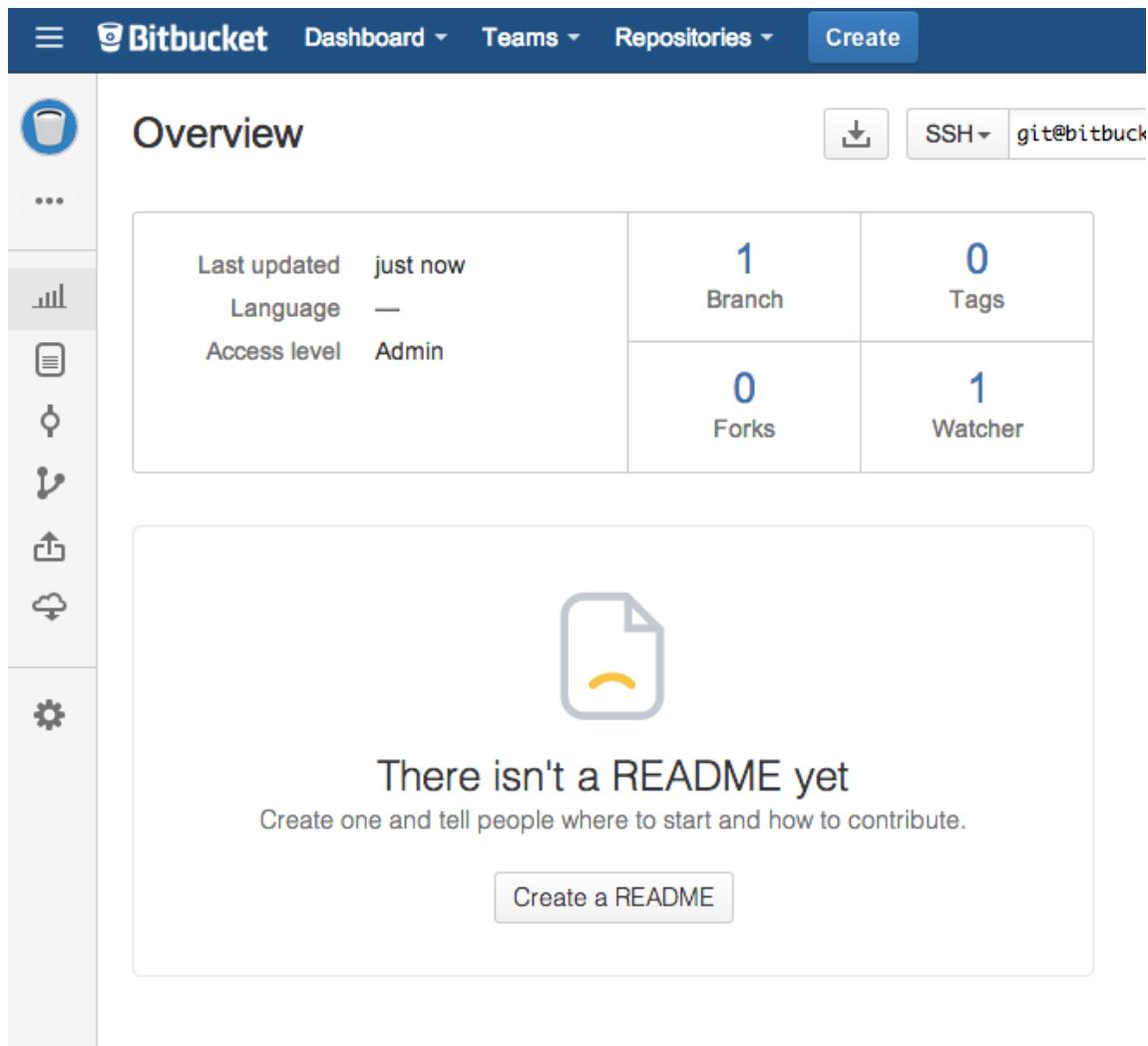


Figura 1.16: El mensaje de Bitbucket cuando falta el archivo README.

La verdadera importancia de crear ramas se hace clara cuando trabajamos en un proyecto con múltiples desarrolladores,¹⁴ pero las ramas son útiles aún para seguir un tutorial como éste, con un solo desarrollador. En particular, la rama principal está aislada de cualquier cambio que realicemos en la rama del tópico, por lo que aún si descomponemos *realmente* las cosas siempre podemos abandonar los cambios cambiándonos a la rama principal y borrando la rama del tópico. Veremos cómo realizar esto al final de la sección.

Por cierto, para un cambio tan pequeño como éste normalmente no me tomaría la molestia de crear una nueva rama, pero en el contexto actual, representa nuestra primera oportunidad para empezar a fomentar buenos hábitos.

Edición

Luego de crear la rama del tópico, la editaremos para hacerla un poco más descriptiva. En lo personal, prefiero utilizar para este propósito el [lenguaje de marcas Markdown](#) que el lenguaje por default RDoc, y si usted utiliza la extensión `.md` entonces Bitbucket automáticamente le dará un formato agradable. Por tanto, utilizaremos la versión de Git del comando Unix `mv` (move) para renombrar el archivo:

```
$ git mv README.rdoc README.md
```

Luego editaremos el archivo `README.md` agregándole el contenido del [Listado 1.13](#).

Listado 1.13: El nuevo archivo `README.md`.

```
# Ruby on Rails Tutorial: "hello, world!"  
  
This is the first application for the  
[*Ruby on Rails Tutorial*](http://www.railstutorial.org/)  
by [Michael Hartl](http://www.michaelhartl.com/).
```

¹⁴Vea el capítulo [Ramificaciones en Git](#) de *Pro Git* para mayor detalle.

Confirmación

Con los cambios realizados, podemos echar un vistazo al status de nuestra rama:

```
$ git status
On branch modify-README
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:   README.rdoc -> README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:  README.md
```

En este momento, podemos utilizar **git add -A** como en la Sección 1.4.1, pero **git commit** proporciona la opción **-a** como abreviatura para el (muy común) caso de confirmar todas las modificaciones existentes en los archivos (o archivos creados mediante **git mv**, que no cuentan como archivos nuevos para Git):

```
$ git commit -a -m "Improve the README file"
2 files changed, 5 insertions(+), 243 deletions(-)
delete mode 100644 README.rdoc
create mode 100644 README.md
```

Tenga cuidado de utilizar adecuadamente la opción **-a**; si usted ha agregado archivos nuevos al proyecto después de la última confirmación, es necesario notificarle a Git de su existencia primero, mediante el comando **git add -A**.

Observe que escribimos el mensaje de la confirmación en tiempo *presente* (y, técnicamente hablando, en *modo imperativo*). Git modela las confirmaciones como una serie de parches, y en este contexto tiene sentido describir lo que cada confirmación *hace*, en vez de lo que hizo. Más aún, este uso concuerda con los mensajes de confirmación generados por los mismos comandos de Git. Revise el artículo “[Nuevos estilos de confirmación](#)” para mayor información.

Integración

Ahora que hemos terminado de realizar nuestros cambios, estamos listos para *integrar* los resultados con nuestra rama principal:

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating 34f06b7..2c92bef
Fast forward
 README.rdoc    | 243 -----
 README.md      |   5 +
 2 files changed, 5 insertions(+), 243 deletions(-)
 delete mode 100644 README.rdoc
 create mode 100644 README.md
```

Observe que la salida de Git a menudo incluye cosas como **34f06b7**, las cuales están relacionadas con la representación interna que Git hace de los repositorios. Los resultados exactos que usted obtenga diferirán de éstos en detalle, pero en esencia coinciden con la salida que se mostró anteriormente.

Luego de que haya usted integrado sus cambios, puede depurar sus ramas borrando la rama de tópico que ya no utiliza mediante **git branch -d**:

```
$ git branch -d modify-README
Deleted branch modify-README (was 2c92bef).
```

Este paso es opcional, y de hecho es muy común dejar las ramas de tópico intactas. De esta forma usted puede ir y venir entre ellas y la rama principal, integrando cambios cada vez que alcance un punto en el que sea natural detenerse.

Como mencionamos antes, también es posible abandonar los cambios de su rama, en este caso con **git branch -D**:

```
# For illustration only; don't do this unless you mess up a branch
$ git checkout -b topic-branch
$ <really screw up the branch>
$ git add -A
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

A diferencia de la opción `-d`, la opción `-D` borrará la rama aún si no hemos integrado nuestros cambios.

Subir cambios

Ahora que hemos actualizado el archivo **README**, podemos subir nuestros cambios a Bitbucket para ver los resultados. Puesto que ya hemos realizado esta tarea por primera vez ([Sección 1.4.3](#)), en la mayoría de los sistemas podemos omitir **origin master**, y simplemente ejecutar **git push**:

```
$ git push
```

Como prometimos en la [Sección 1.4.4](#), Bitbucket le da formato al nuevo archivo que usa Markdown ([Figura 1.17](#)).

1.5 Desplegando

Aún en esta etapa temprana, vamos a realizar nuestro despliegue de la aplicación de Rails (casi vacía) en producción. Este paso es opcional, pero hacerlo tempranamente y a menudo nos permite detectar problemas de despliegue a tiempo en nuestro ciclo de desarrollo. La otra opción es desplegar luego de un esfuerzo laborioso realizado únicamente en el ambiente de desarrollo, lo cual a menudo conlleva terribles dolores de cabeza para integrar cuando llega la hora de lanzar la aplicación.¹⁵

Desplegar aplicaciones Rails solía ser doloroso, pero el ecosistema de desarrollo de Rails ha madurado rápidamente en los últimos años, y ahora hay varias opciones muy buenas. Éstas incluyen hospedaje compartido o servidores virtuales privados ejecutando **Phusion Passenger** (un módulo para servidores Apache y Nginx¹⁶), compañías que brindan servicio completo de despliegue

¹⁵Aunque esto no debería importar para las aplicaciones de ejemplo del *Tutorial de Rails*, si a usted le preocupa que accidentalmente se haga pública su aplicación cuando es demasiado pronto, existen varias opciones; revise la [Sección 1.5.4](#) para ver una de ellas.

¹⁶Pronunciado “Engine X”.

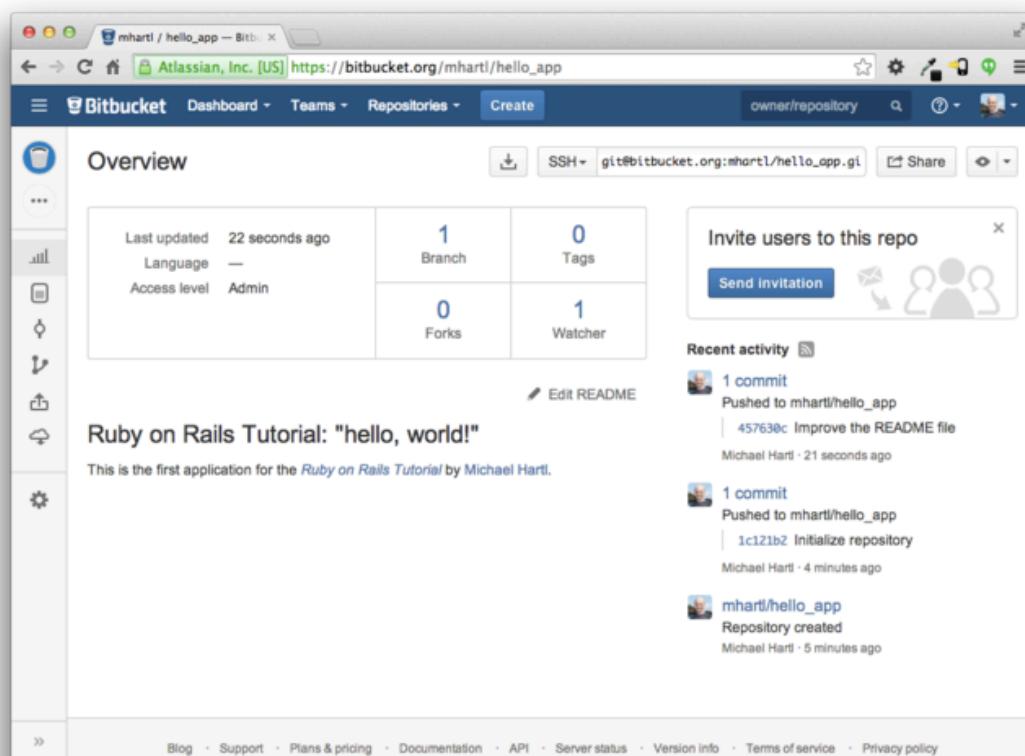


Figura 1.17: El archivo **README** mejorado con Markdown.

tales como [Engine Yard](#) y [Rails Machine](#), y servicios de desarrollo en la nube tales como [Engine Yard Cloud](#), [Ninefold](#), y [Heroku](#).

Mi opción de despliegue favorita para Rails es Heroku, que es una plataforma de hospedaje construída específicamente para desplegar Rails y otras aplicaciones web. Heroku hace que desplegar aplicaciones Rails sea extremadamente sencillo—siempre y cuando el código fuente esté bajo control de versiones con Git. (Esta es otra razón para seguir los pasos de configuración de Git indicados en la [Sección 1.4](#) si es que aún no los realiza.) Adicionalmente, la versión gratuita del servicio de Heroku es más que suficiente para realizar los ejercicios de este libro. De hecho, las primeras dos ediciones de este tutorial estuvieron hospedadas gratuitamente en Heroku, quienes atendieron varios millones de peticiones sin cobrarme un centavo.

El resto de esta sección está dedicado a desplegar nuestra primera aplicación en Heroku. Algunos conceptos son un poco avanzados, por lo que no se preocupe por entender todos los detalles; lo importante es que al final del proceso tengamos nuestra aplicación desplegada en internet.

1.5.1 Configuración de Heroku

Heroku utiliza la base de datos [PostgreSQL](#) (a menudo llamada “Postgres” como diminutivo), lo que significa que necesitamos agregar la gema pg al ambiente de producción para permitir a Rails comunicarse con Postgres.¹⁷

```
group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

Observe que también agregamos la gema `rails_12factor`, la cual es utilizada por Heroku para despachar recursos estáticos tales como imágenes y hojas de estilo. Finalmente, asegúrese de incorporar los cambios realizados en

¹⁷En general, es buena idea que los ambientes de desarrollo y producción sean lo más similares posible, lo que incluye utilizar la misma base de datos, pero para los propósitos de este tutorial siempre utilizamos SQLite localmente y PostgreSQL en producción. Revise la [Sección 3.1](#) para mayor información.

el Listado 1.5 evitando que la gema `sqlite3` sea incluída en un ambiente de producción, puesto que SQLite no tiene soporte en Heroku:

```
group :development, :test do
  gem 'sqlite3',      '1.3.9'
  gem 'byebug',       '3.4.0'
  gem 'web-console',  '2.0.0.beta3'
  gem 'spring',        '1.1.3'
end
```

El archivo **Gemfile** resultante se muestra en el Listado 1.14.

Listado 1.14: Un archivo **Gemfile** con las gemas añadidas.

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

gem 'rails',          '4.2.2'
gem 'sass-rails',     '5.0.2'
gem 'uglifier',        '2.5.3'
gem 'coffee-rails',   '4.1.0'
gem 'jquery-rails',   '4.0.3'
gem 'turbolinks',     '2.3.0'
gem 'jbuilder',        '2.2.3'
gem 'sdoc',            '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',      '1.3.9'
  gem 'byebug',       '3.4.0'
  gem 'web-console',  '2.0.0.beta3'
  gem 'spring',        '1.1.3'
end

group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

Para preparar el sistema para despliegue en producción, ejecutamos **bundle install** con una opción especial para evitar la instalación local de las gemas de producción (que en este caso son `pg` y `rails_12factor`):

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

Como las únicas gemas agregadas en el [Listado 1.14](#) están restringidas al ambiente de producción, en este momento este comando no instala realmente ninguna gema adicional en el entorno local, pero se requiere actualizar el archivo **Gemfile.lock** con las nuevas gemas **pg** y **rails_12factor**. Podemos confirmar el cambio resultante como sigue:

```
$ git commit -a -m "Update Gemfile.lock for Heroku"
```

A continuación debemos crear y configurar una cuenta en Heroku. El primer paso es [registrarse en Heroku](#). Luego revise si su sistema ya tiene el cliente de línea de comandos de Heroku instalado:

```
$ heroku version
```

Aquellos que estén utilizando el IDE en la nube deben revisar el número de versión de Heroku, lo cual indica que la línea de comandos (CLI, por sus siglas en inglés *Command Line Interface*) de **heroku** está disponible, pero en otros sistemas puede ser necesario instalarlo mediante el conjunto de [Herramientas de Heroku](#).¹⁸

Una vez que ha verificado que la línea de comandos de Heroku está instalada, utilice el comando **heroku** para iniciar sesión y agregar su llave SSH:

```
$ heroku login
$ heroku keys:add
```

Finalmente, utilice el comando **heroku create** para crear un espacio en los servidores de Heroku para colocar ahí la aplicación de ejemplo ([Listado 1.15](#)).

¹⁸<https://toolbelt.heroku.com/>

Listado 1.15: Creando una nueva aplicación en Heroku.

```
$ heroku create
Creating damp-fortress-5769... done, stack is cedar
http://damp-fortress-5769.herokuapp.com/ | git@heroku.com:damp-fortress-5769.git
Git remote heroku added
```

El comando **heroku create** crea un nuevo subdominio sólo para nuestra aplicación, disponible para ser utilizado de inmediato. Aún no hay nada ahí, así que despleguemos.

1.5.2 Despliegue en Heroku, paso uno

Para desplegar la aplicación, el primer paso es utilizar Git para subir la rama principal a Heroku:

```
$ git push heroku master
```

(Usted puede ver algunos mensajes de advertencia, los cuales puede ignorar por ahora. Hablaremos más de ellos en la [Sección 7.5](#).)

1.5.3 Despliegue en Heroku, paso dos

¡No hay paso dos! Ya terminamos. Para ver nuestra recién desplegada aplicación, visite la dirección que se indicó en la salida del comando **heroku create** (es decir, el [Listado 1.15](#)). (Si usted está trabajando en su equipo local en vez del IDE en la nube, puede utilizar también **heroku open**.) El resultado se muestra en la [Figura 1.18](#). La página es idéntica a la de la [Figura 1.12](#), sólo que ahora está ejecutándose en un ambiente de producción en internet.

1.5.4 Comandos de Heroku

Hay muchos [comandos Heroku](#), y apenas veremos algunos en este libro. Dediquemos un momento a revisar uno de ellos para renombrar la aplicación como sigue:

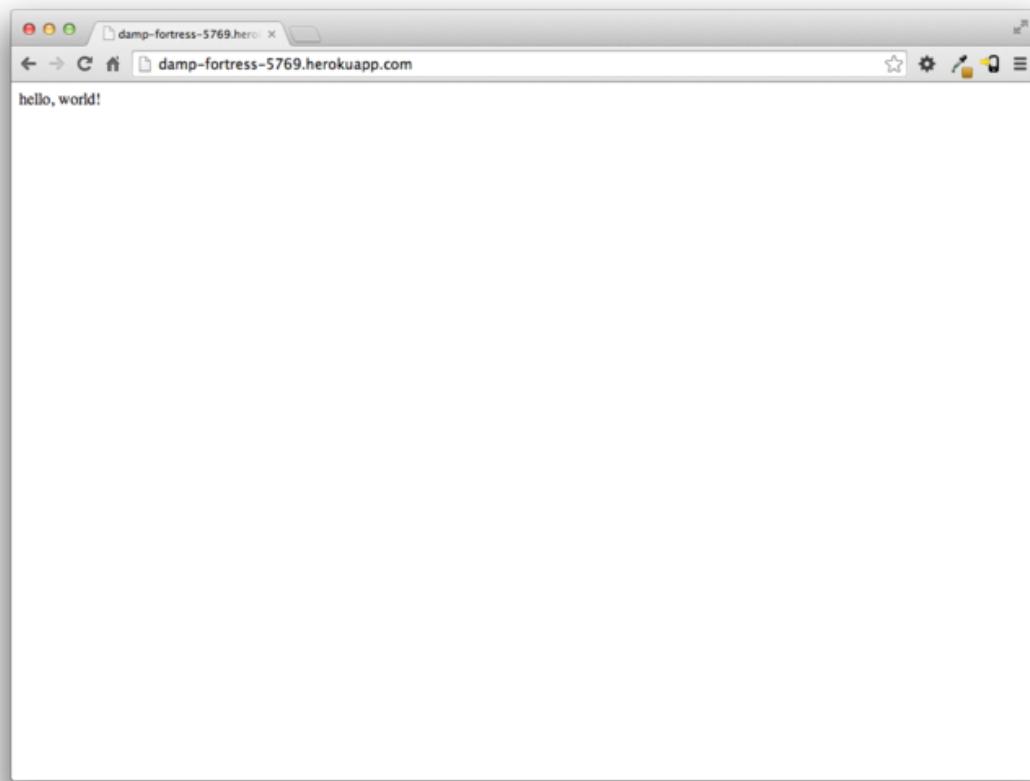


Figura 1.18: La primera aplicación del Tutorial de Rails corriendo en Heroku.

```
$ heroku rename rails-tutorial-hello
```

No utilice este nombre usted; ¡ya está ocupado por mí! De hecho, no es necesario que se tome la molestia de realizar este paso ahora; usar la dirección por default que le asignó Heroku está bien. Pero si usted desea renombrar su aplicación, puede encargarse de que sea razonablemente segura utilizando un subdominio aleatorio o poco claro, como el siguiente:

```
hwpcbmze.herokuapp.com  
seyjhflo.herokuapp.com  
jhyicevg.herokuapp.com
```

Con un dominio aleatorio como éste, alguien puede visitar su sitio sólo si usted le ha dado la dirección. (Por cierto, como un adelanto de lo grandiosa que es la compacidad de Ruby, aquí está el código que utilicé para generar los subdominios aleatorios:

```
('a'..'z').to_a.shuffle[0..7].join
```

Grandioso.)

Además de soportar subdominios, Heroku también soporta dominios personalizados. Revise la [documentación de Heroku](#) para mayor acerca de éste y otros temas de Heroku.

1.6 Conclusión

Hemos recorrido un largo camino en este capítulo: la instalación y configuración del ambiente de desarrollo, el control de versiones y el despliegue. En el próximo capítulo, construiremos sobre las bases del [Capítulo 1](#) para crear una *aplicación de juguete* que tenga respaldo de base de datos, lo cual nos dará una probadita de lo que Rails puede hacer.

Si desea compartir su progreso hasta este momento, siéntase libre de enviar un tuit o actualizar su status en Facebook con algo como esto:

Estoy aprendiendo Ruby on Rails con el @railstutorial!
<http://www.railstutorial.org/>

También le recomiendo registrarse en la [lista de correos del Tutorial de Rails](#)¹⁹, lo cual le asegura que recibirá actualizaciones prioritarias (y cupones exclusivos) relacionados con el *Tutorial de Ruby on Rails*.

1.6.1 Qué aprendimos en este capítulo

- Ruby on Rails es un marco de trabajo para desarrollo web escrito en el lenguaje de programación Ruby.
- Instalar Rails, generar una aplicación y editar los archivos resultantes es fácil utilizando un ambiente en la nube pre-configurado.
- Rails viene con una línea de comandos llamada **rails** que puede generar aplicaciones nuevas (**rails new**) y correr servidores locales (**rails server**).
- Agregamos una acción a un controlador y modificamos la ruta raíz para crear una aplicación “hola mundo”.
- Nos protegimos contra la pérdida de datos al mismo tiempo que habilitamos la colaboración al poner el código fuente de nuestra aplicación bajo un control de versiones con Git y subiendo el código resultante a un repositorio privado en Bitbucket.
- Desplegamos nuestra aplicación en un ambiente de producción usando Heroku.

1.7 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en

¹⁹<http://www.railstutorial.org/#email>

cada compra realizada en www.railstutorial.org.

1. Modifique el contenido de la acción **hello** del Listado 1.8 para leer “¡Hola, mundo!” en vez de “hello, world!”. *Punto extra:* Muestre que Rails soporta caracteres no-ASCII incluyendo el signo para abrir una admiración, como en “¡Hola, mundo!” (Figura 1.19).²⁰
2. Siguiendo el ejemplo de la acción **hello** del Listado 1.8, agregue una segunda acción llamada **goodbye** que muestre el texto “¡Adiós, mundo!”. Edite el archivo de rutas del Listado 1.10 de forma que la ruta raíz se dirija a **goodbye** en vez de a **hello** (Figura 1.20).

²⁰Su editor de texto puede mostrar un mensaje como “caracter multibyte inválido”, pero esto no debe preocu-parle. Usted puede [buscar en Google el mensaje de error](#) si quiere aprender cómo desaparecerlo.

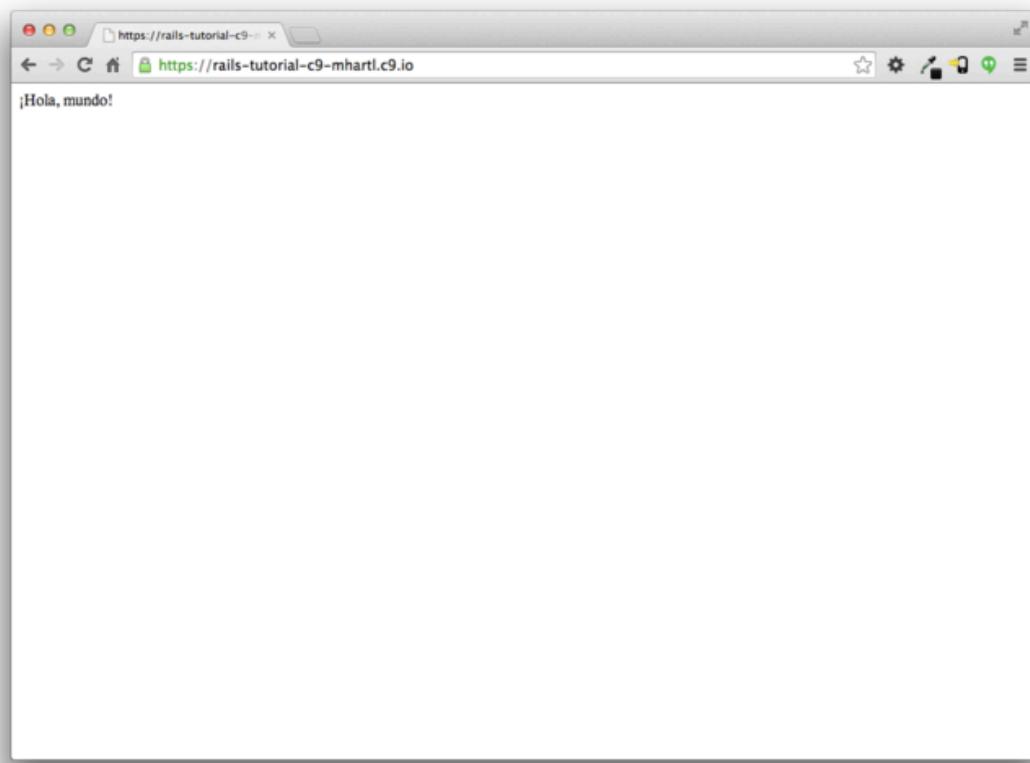


Figura 1.19: Cambiando la ruta raíz para regresar “¡Hola, mundo!”.

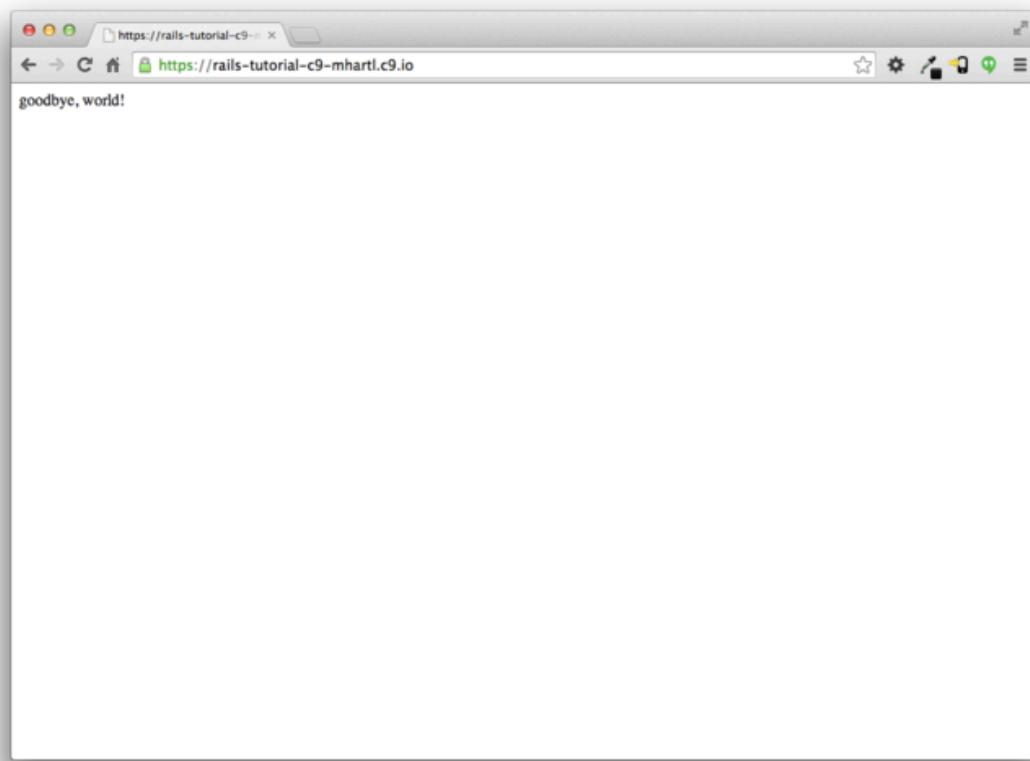


Figura 1.20: Cambiando la ruta raíz para regresar “¡Adiós, mundo!”.

Capítulo 2

Una aplicación de juguete

En este capítulo, desarrollaremos una aplicación de juguete para demostrar algunos de los poderes de Rails. El propósito es obtener una visión general de la programación con Ruby on Rails (y del desarrollo en general) al generar rápidamente una aplicación usando *generadores de estructuras*, los cuales crean una gran cantidad de funcionalidad automáticamente. Como se comentó en el recuadro 1.2, el resto del libro adoptará el enfoque opuesto, desarrollando una aplicación de ejemplo completa de forma incremental y explicando cada concepto nuevo conforme se presente, pero para una rápida visión general (y una pequeña gratificación instantánea) no hay como crear una estructura temporal. La aplicación de juguete resultante nos permitirá interactuar con ella a través de sus URLs, dándonos una idea de la estructura de una aplicación Rails, y al mismo tiempo constituye un primer ejemplo de la *arquitectura REST* favorecida por Rails.

Al igual que con la aplicación de ejemplo, la aplicación de juguete consistirá de *usuarios* y sus *microposts* (generando entonces una aplicación minimalista estilo Twitter). La funcionalidad estará totalmente subdesarrollada, y muchos de los pasos parecerán mágicos, pero no se preocupe: la aplicación de ejemplo completa desarrollará una aplicación similar desde cero empezando en el Capítulo 3, y le proporcionará abundantes referencias al material en general. Mientras tanto, tenga paciencia y un poco de fé—el gran propósito de este tutorial es llevarlo *más allá* de lo superficial; del enfoque basado en la creación de estructuras temporales para lograr un entendimiento más profundo de Rails.

2.1 Planificando la aplicación

En esta sección, bosquejaremos nuestros planes para la aplicación de juguete. Como en la [Sección 1.3](#), empezaremos generando un esqueleto de la aplicación utilizando el comando `rails new` con un número de versión específico de Rails:

```
$ cd ~/workspace
$ rails _4.2.2_ new toy_app
$ cd toy_app/
```

Si el comando anterior arroja un error como “Could not find ‘railties’”, significa que usted no tiene instalada la versión correcta de Rails, y debería verificar que ha ejecutado el comando indicado en el [Listado 1.1](#) exactamente como está escrito. (Si está utilizando el IDE en la nube como se recomendó en la [Sección 1.2.1](#), observe que esta segunda aplicación puede ser creada en el mismo espacio de trabajo que la primera. No es necesario crear un nuevo espacio de trabajo. Con la finalidad de visualizar los archivos que se generan, es posible que necesite dar click en el ícono de engrane ubicado en el área del navegador de archivos, y seleccionar “Refresh File Tree”.)

A continuación, utilizaremos un editor de texto para actualizar el archivo `Gemfile` necesario para el Bundler, con los contenidos del [Listado 2.1](#).

Listado 2.1: Un archivo `Gemfile` para la aplicación de juguete.

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

gem 'rails',          '4.2.2'
gem 'sass-rails',     '5.0.2'
gem 'uglifier',        '2.5.3'
gem 'coffee-rails',   '4.1.0'
gem 'jquery-rails',   '4.0.3'
gem 'turbolinks',     '2.3.0'
gem 'jbuilder',        '2.2.3'
gem 'sdoc',            '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',       '1.3.9'
  gem 'byebug',        '3.4.0'
```

```

gem 'web-console', '2.0.0.beta3'
gem 'spring',      '1.1.3'
end

group :production do
  gem 'pg',          '0.17.1'
  gem 'rails_12factor', '0.0.2'
end

```

Observe que el [Listado 2.1](#) es idéntico al [Listado 1.14](#).

Al igual que en la [Sección 1.5.1](#), instalaremos las gemas locales mientras evitamos la instalación de las gemas de producción utilizando la opción `--without production`:

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

Finalmente, pondremos la aplicación de juguete bajo el control de versiones con Git:

```

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

```

Usted debería también [crear un nuevo repositorio](#) dando click en el botón “Create” en Bitbucket ([Figura 2.1](#)), y luego subir sus archivos al repositorio remoto:

```

$ git remote add origin git@bitbucket.org:<username>/toy_app.git
$ git push -u origin --all # pushes up the repo and its refs for the first time

```

Finalmente, nunca es demasiado pronto para desplegar, sugiero que lo haga siguiendo los mismos pasos que para el “hello, world!” indicados en los [Listados 1.8](#) y [1.9](#).¹ Luego haga **commit** de los cambios y súbalos a Heroku:

¹La razón principal para hacer esto es que la página que se crea por defecto en Rails, usualmente genera error en Heroku, lo que hace difícil decir si el despliegue fue exitoso o no.

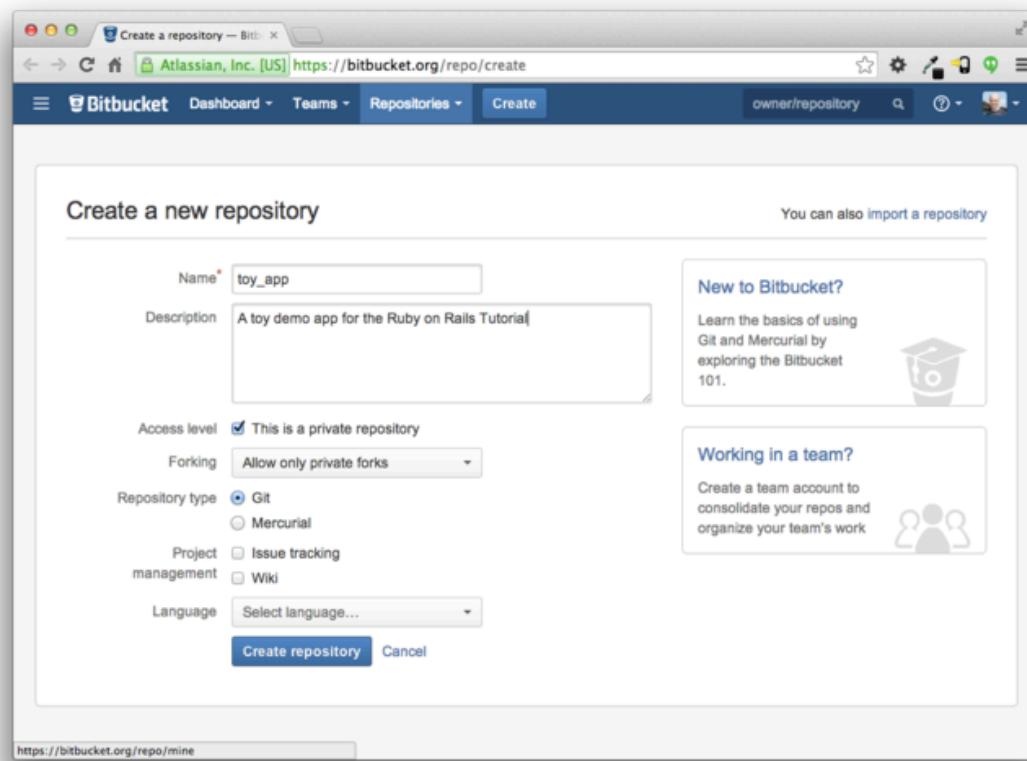


Figura 2.1: Creando el repositorio para la aplicación de juguete en Bitbucket.

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

(De igual forma que en la Sección 1.5, es probable que vea algunos mensajes de advertencia, que puede ignorar por ahora; los eliminaremos en la Sección 7.5.) Excepto por la dirección de la aplicación en Heroku, el resultado debería ser el mismo que el de la Figura 1.18.

Ahora estamos listos para iniciar el desarrollo de la aplicación. El primer paso típico cuando se desarrolla una aplicación web, es crear un *modelo de datos*, que es una representación de las estructuras necesarias para nuestra aplicación. En nuestro caso, la aplicación de juguete será un microblog, con sólo usuarios y microposts. Por lo tanto, empezaremos con un modelo para los *usuarios* de la aplicación (Sección 2.1.1), y luego añadiremos un modelo para los *microposts* (Sección 2.1.2).

2.1.1 Un modelo de juguete para usuarios

Existen tantas opciones para un modelo de datos de usuario, como diferentes formas de registro en la web; nosotros vamos a adoptar un enfoque totalmente minimalista. Los usuarios de nuestra aplicación de juguete tendrán como identificador único un número **entero** llamado **id**, un **nombre** visible públicamente (de tipo **string**), y una dirección **email** (también de tipo **string**) que además funcionará como username. Un resumen del modelo de datos para los usuarios aparece en la Figura 2.2.

Como veremos al iniciar la Sección 6.1.1, la etiqueta **users** de la Figura 2.2 corresponde a una *tabla* en la base de datos, y los atributos **id**, **email**, y **name** son *columnas* de esa tabla.

2.1.2 Un modelo de juguete para microposts

La estructura del modelo de datos para los microposts es aún más simple que la de los usuarios: un micropost sólo tiene un **id** y un campo **contenido** para

users	
id	integer
name	string
email	string

Figura 2.2: El modelo de datos para usuarios.

microposts	
id	integer
content	text
user_id	integer

Figura 2.3: El modelo de datos para microposts.

el texto del micropost (de tipo **text**).² Sin embargo, existe una consideración adicional: queremos *asociar* cada micropost con un usuario en particular. Conseguiremos esto registrando el **user_id** del dueño del mensaje. Los resultados se muestran en la Figura 2.3.

Veremos en la Sección 2.3.3 (y de forma más completa en el Capítulo 11) cómo este atributo **user_id** nos permite expresar de forma concisa, la noción de que un usuario potencialmente tiene asociados muchos microposts.

²Debido a que los microposts son pequeños por diseño, el tipo **string** es suficientemente grande para tenerlos, pero al usar **text** se expresa mejor nuestra intención, al mismo tiempo de que nos da mayor flexibilidad por si en algún momento deseamos modificar la restricción de longitud.

2.2 El recurso Users

En esta sección, implementaremos el modelo de datos para los usuarios de la Sección 2.1.1, junto con una interfaz web para ese modelo. La combinación constituirá un *recurso Users*, que nos permitirá considerar a los usuarios como objetos que pueden ser creados, leídos, actualizados y borrados vía web mediante el [protocolo HTTP](#). Como prometimos en la introducción, nuestro recurso Users será creado por un programa generador de estructuras temporales, que viene como estándar en cada proyecto Rails. Le exhorto a no mirar muy de cerca el código generado; en este punto sólo le causaría confusión.

La creación de estructuras temporales en Rails es generada al pasar el comando **scaffold** al script **rails generate**. El argumento del comando **scaffold** es el nombre del recurso en singular (en este caso, **User**), junto con los parámetros opcionales para los atributos del modelo de datos:³

```
$ rails generate scaffold User name:string email:string
  invoke  active_record
  create    db/migrate/20140821011110_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
  invoke  helper
  create    app/helpers/users_helper.rb
  invoke  test_unit
  create    test/helpers/users_helper_test.rb
  invoke  jbuilder
```

³El nombre de la estructura temporal seguida de la convención de *modelos*, que van en singular, a diferencia de los recursos y controladores, que van en plural. De esta forma, tenemos **User** en vez de **Users**.

```

create      app/views/users/index.json.jbuilder
create      app/views/users/show.json.jbuilder
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

```

Al incluir `name:string` y `email:string`, hemos organizado el modelo User para que tenga la forma mostrada en la Figura 2.2. (Observe que no hay necesidad de incluir un parámetro para el `id`; éste es creado automáticamente por Rails para emplearlo como *llave primaria* en la base de datos.)

Para continuar con la aplicación de juguete, primero necesitamos *migrar* la base de datos usando *Rake* (Recuadro 2.1):

```

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

```

Esto simplemente actualiza la base de datos con nuestro nuevo modelo de datos `user`. (Aprenderemos más sobre migraciones de base de datos al iniciar la Sección 6.1.1.) Observe que, con la finalidad de asegurar que el comando utiliza la versión de Rake correspondiente a nuestro `Gemfile`, necesitamos ejecutar `rake` utilizando `bundle exec`. En varios sistemas, incluyendo el IDE en la nube, puede omitir `bundle exec`, pero en otros sistemas es necesario, por lo que lo incluiré por completo.

Con esto, podemos ejecutar el servidor web local en una pestaña distinta (Figura 1.7):⁴

```
$ rails server -b $IP -p $PORT      # Use only `rails server` if running locally
```

⁴El script `rails` está diseñado de tal forma que usted no necesite utilizar `bundle exec`.

Ahora la aplicación de juguete debería estar disponible en el servidor local, como se describe en la Sección 1.3.2. (Si usted está utilizando el IDE en la nube, asegúrese de abrir el servidor de desarrollo resultante en una nueva pestaña del *navegador*, no dentro del mismo IDE.)

Recuadro 2.1. Rake

Según la tradición Unix, la utilería *make* ha jugado un rol importante en la construcción de programas ejecutables a partir de código fuente; muchos hackers de computadoras se han comprometido a ejercitar la memoria con la línea

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

comúnmente empleada para compilar código en los sistemas Unix (incluyendo Linux y Mac OS X).

Rake es un *make para Ruby*, un lenguaje tipo *make* escrito en Ruby. Rails utiliza Rake extensamente, especialmente para las innumerables pequeñas tareas administrativas que se necesitan cuando se desarrolla aplicaciones web respaldadas por una base de datos. El comando **rake db:migrate** es probablemente el más común, pero existen muchos otros; usted puede consultar una lista de tareas que puede realizar sobre la base de datos, utilizando **-T db**:

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

Para ver todas las tareas disponibles del comando Rake, ejecute

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

Es probable que la lista sea abrumadora, pero no se preocupe, no tiene que saber todos (o la mayoría) de estos comandos. Al final del *Tutorial de Rails*, usted conocerá los más importantes.

URL	Acción	Funcionalidad
/users	index	página para enlistar a todos los usuarios
/users/1	show	página para mostrar al usuario con id 1
/users/new	new	página para crear un nuevo usuario
/users/1/edit	edit	página para editar al usuario con id 1

Tabla 2.1: La relación entre páginas y URLs para el recurso Users.

2.2.1 Un recorrido por User

Si visitamos la URL raíz: / (se lee “diagonal”, como se señala en la Sección 1.3.4), obtenemos la misma página de Rails por defecto, mostrada en la Figura 1.9, pero al generar el recurso Users mediante la creación de estructuras temporales, también creamos varias páginas para manipular a los usuarios. Por ejemplo, la página para enlistar a todos los usuarios se encuentra en `/users`, y la página para crear un nuevo usuario está en `/users/new`. El resto de esta sección está dedicado a realizar un viaje relámpago através de estas páginas de usuario. Conforme avanzamos, puede ser útil consultar como referencia la Tabla 2.1, que muestra la relación entre páginas y URLs.

Empezaremos con la página que muestra a todos los usuarios de nuestra aplicación, llamada `index`; como es de esperarse, inicialmente no hay ningún usuario (Figura 2.4).

Para crear un nuevo usuario, visitamos la página `new`, como se muestra en la Figura 2.5. (Puesto que `http://0.0.0.0:3000` o la parte de la dirección del IDE en la nube está implícita cada vez que desarrollamos localmente, la omitiré de aquí en adelante.) En el Capítulo 7, esta página se convertirá en la página para enrolar usuarios.

Podemos crear un usuario al ingresar los valores de nombre y correo electrónico en los campos de texto y luego presionar el botón Create User. El resultado es la página del usuario `show`, como se muestra en la Figura 2.6. (El mensaje verde de bienvenida se logra usando *flash*, el cual aprenderemos en la Sección 7.4.2.) Observe que la URL es `/users/1`; como puede suponer, el número 1 es simplemente el atributo `id` del usuario, como en la Figura 2.2. En la Sección 7.1, esta página se convertirá en el perfil del usuario.

Para modificar la información de un usuario, visitamos la página `edit`

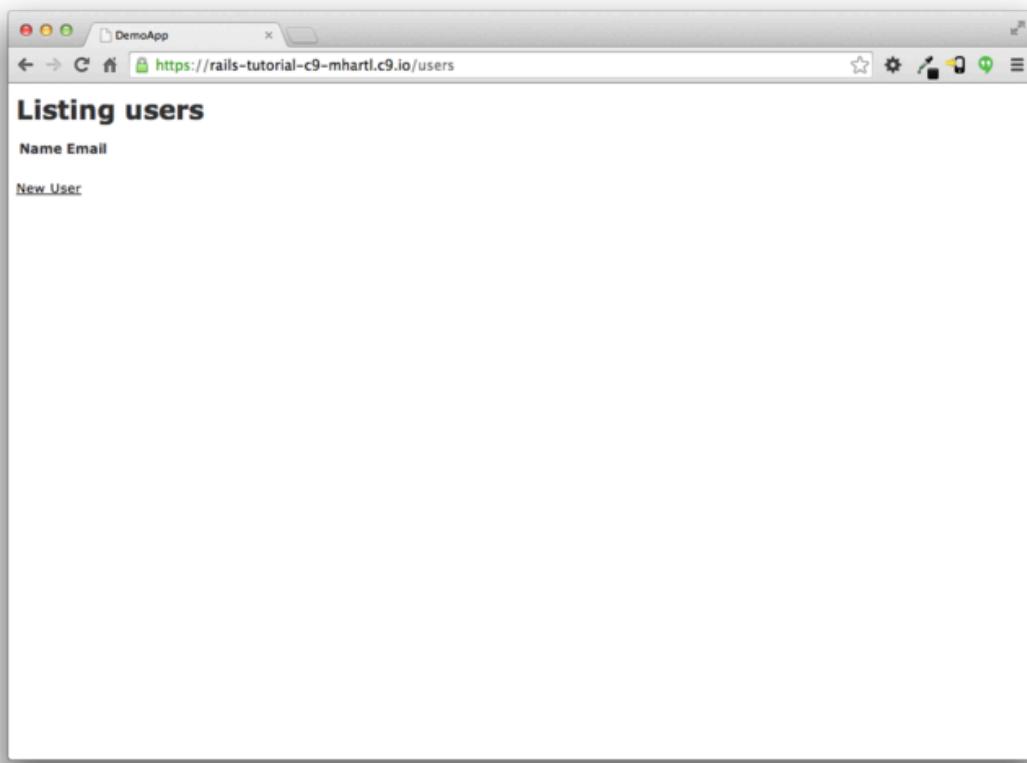


Figura 2.4: La página inicial de listado de usuarios ([/users](#)).

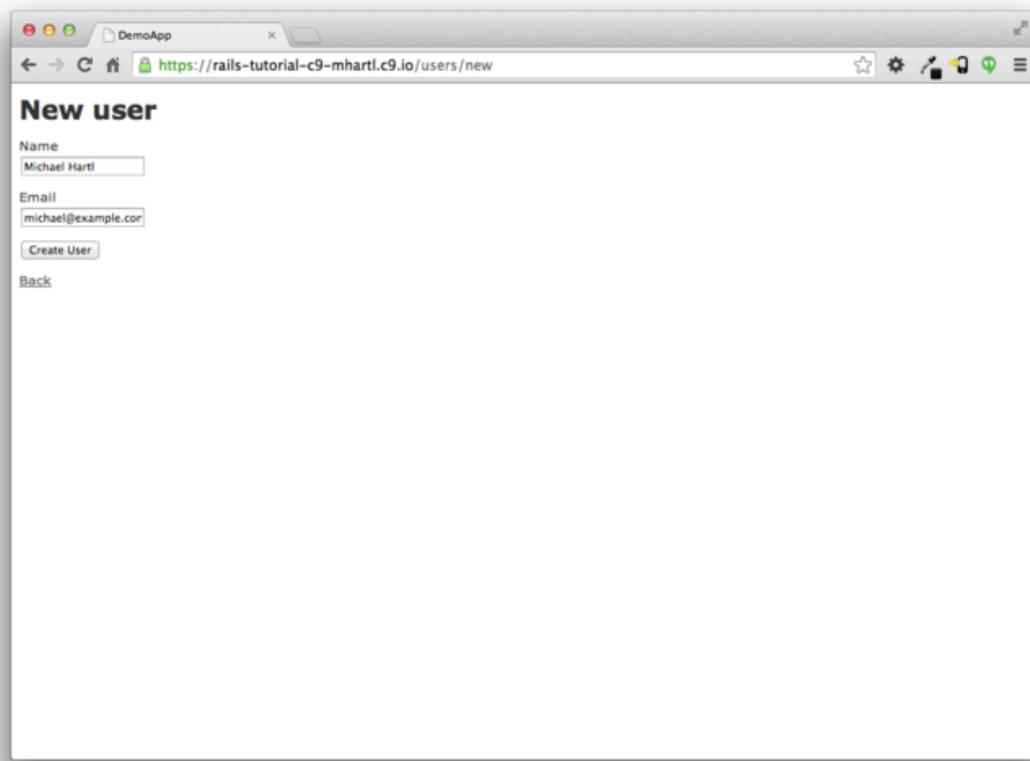


Figura 2.5: La página para crear un usuario nuevo ([/users/new](#)).

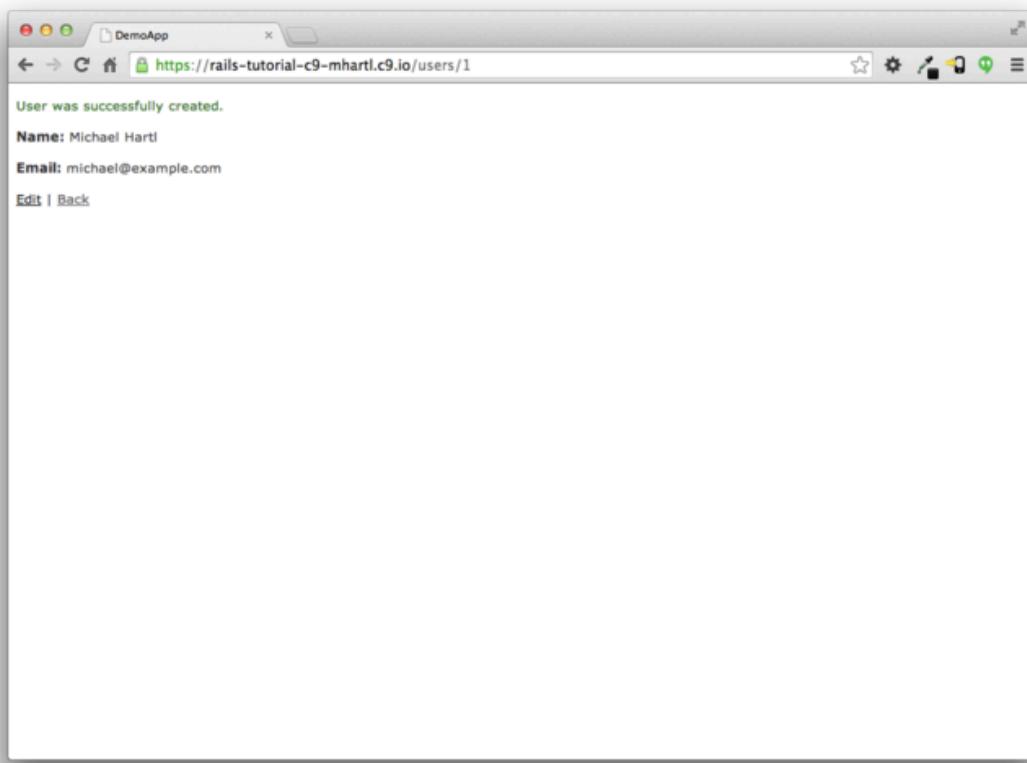


Figura 2.6: La página para mostrar un usuario ([/users/1](#)).

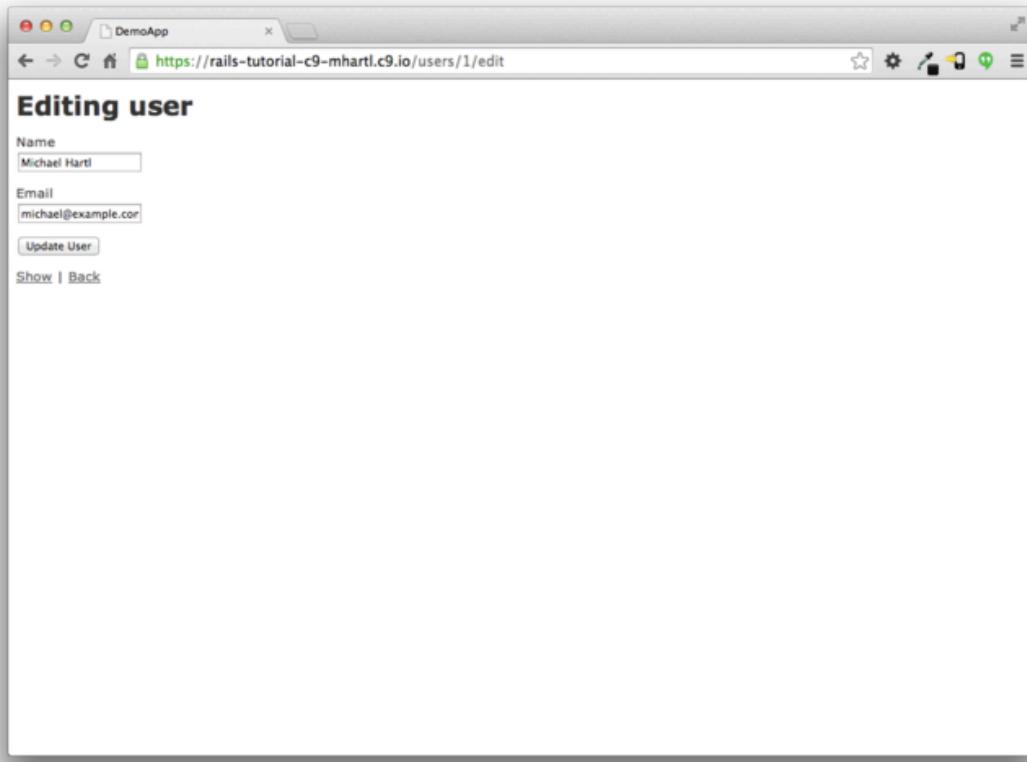


Figura 2.7: La página para editar un usuario ([/users/1/edit](#)).

(Figura 2.7). Al cambiar la información del usuario y presionar el botón Update User, conseguimos actualizar la información del usuario en la aplicación de juguete (Figura 2.8). (Como veremos en detalle al iniciar el Capítulo 6, estos datos de usuario son almacenados en una base de datos en el servidor.) Agregaremos funcionalidad para editar/actualizar usuarios en la Sección 9.1.

Ahora crearemos un segundo usuario volviendo a visitar la página [new](#) y enviando un segundo conjunto de datos de usuario; el listado de usuarios resultante [index](#) se muestra en la Figura 2.9. En la Sección 7.1 desarrollaremos una versión más refinada de este listado.

Habiendo mostrado cómo crear, mostrar y editar usuarios, finalmente llegamos a revisar cómo borrarlos (Figura 2.10). Verifique que al dar click en el

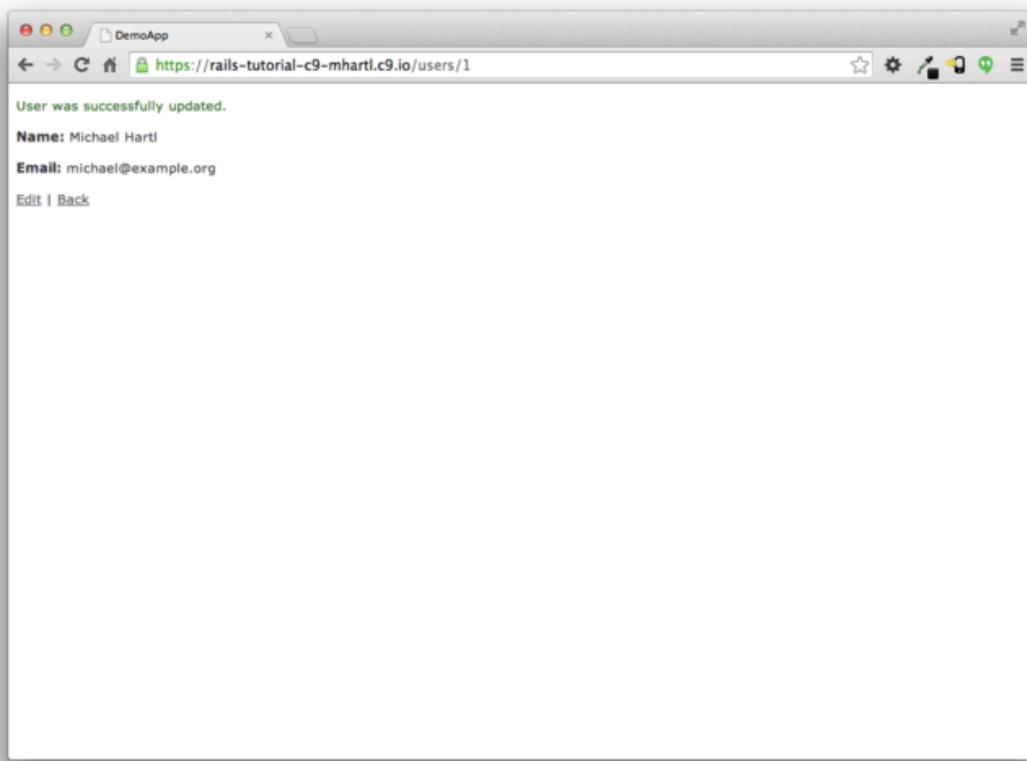


Figura 2.8: Un usuario con información actualizada.

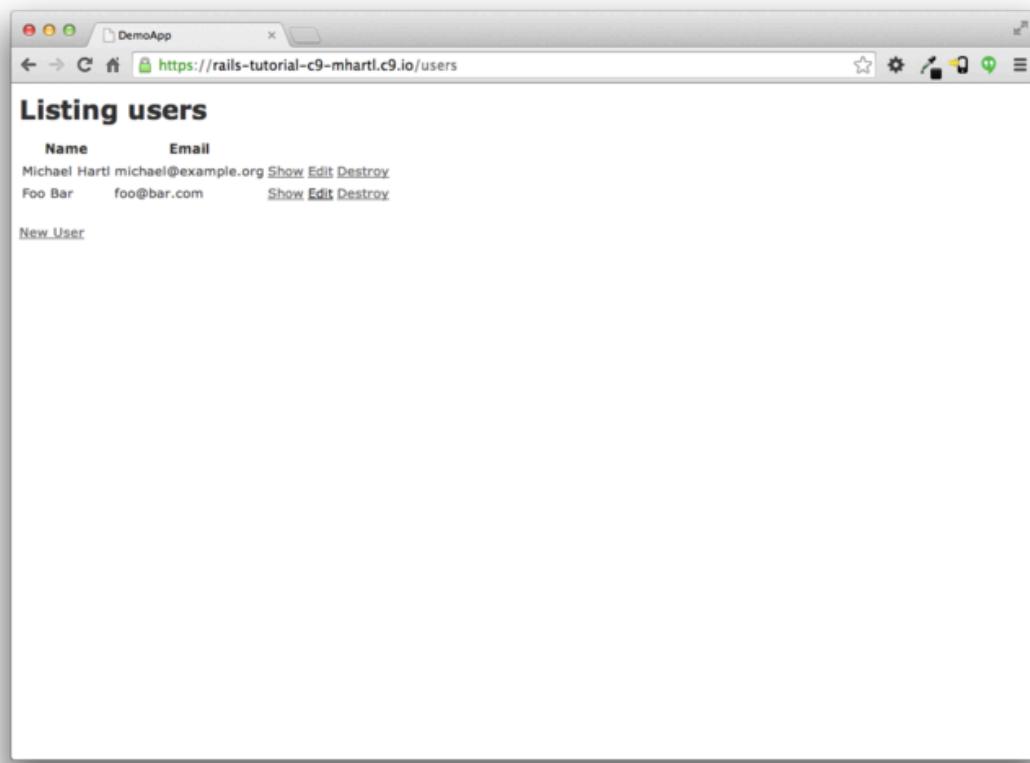


Figura 2.9: La página de listado de usuarios ([/users](#)) con un segundo usuario.

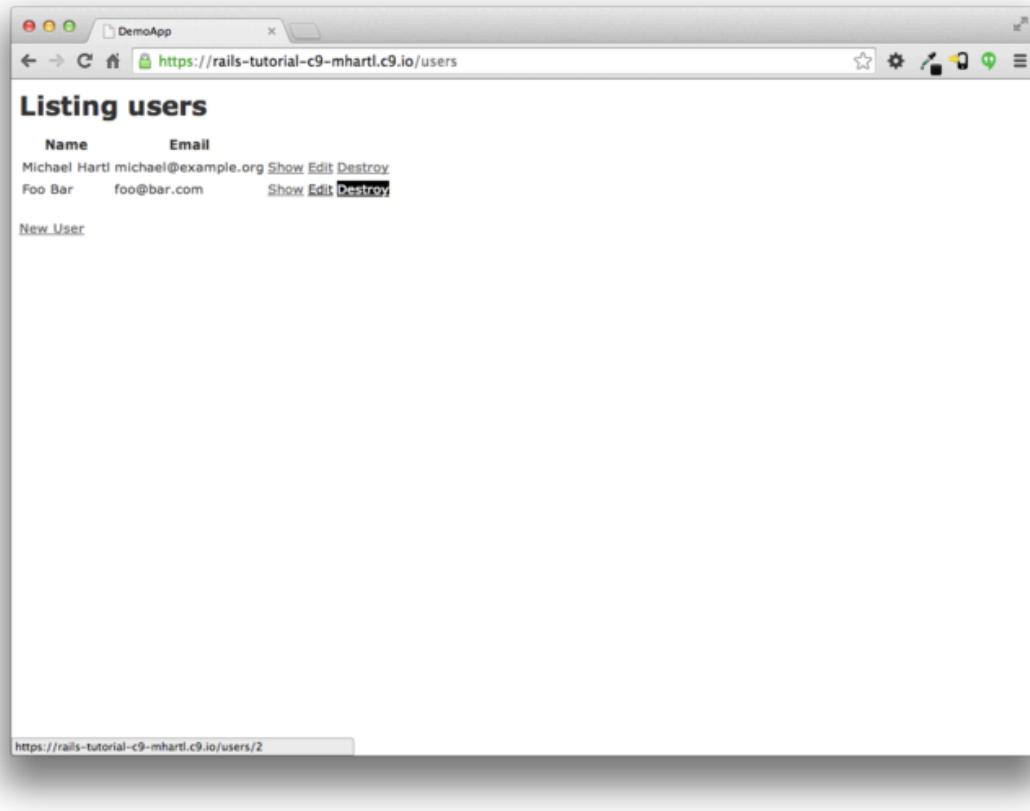


Figura 2.10: Eliminando un usuario.

enlace de la Figura 2.10 elimina al segundo usuario, produciendo una página de listado de usuarios con únicamente un elemento. (Si no funciona, asegúrese de que JavaScript está habilitado en su navegador; Rails utiliza JavaScript para emitir las peticiones necesarias para eliminar un usuario.) En la Sección 9.4 agregamos el borrado de usuario a la aplicación de ejemplo, teniendo cuidado de restringir su uso a una clase especial de usuarios administrativos.

2.2.2 MVC en acción

Ahora que hemos terminado una revisión rápida del recurso Users, examinemos una sección particular de éste en el contexto del patrón Modelo-Vista-Controlador

(MVC) mencionado en la Sección 1.3.3. Nuestra estrategia será describir los resultados de una visita típica del navegador—una visita a la página del listado que se encuentra en `/users`—en términos del MVC (Figura 2.11).

Aquí hay un resumen de los pasos mostrados en la Figura 2.11:

1. El navegador emite una petición de la URL `/users`.
2. Rails dirige `/users` a la acción `index` en el controlador de Users.
3. La acción `index` solicita al modelo User que obtenga todos los usuarios (`User.all`).
4. El modelo User consulta todos los usuarios de la base de datos.
5. El modelo User regresa la lista de usuarios al controlador.
6. El controlador almacena los usuarios en la variable `@users`, y la pasa a la vista `index`.
7. La vista utiliza código Ruby embobido para desplegar la página como HTML.
8. El controlador regresa el HTML al navegador.⁵

Ahora echemos un vistazo a los pasos anteriores con mayor detalle. Empezamos con una petición emitida por el navegador—es decir, el resultado de escribir la URL en la barra de direcciones o el resultado de dar clik en un enlace (Paso 1 de la Figura 2.11). Esta petición llega al *enrutador Rails* (Paso 2), que dispara la *acción del controlador* apropiada, de acuerdo con la URL (y, como vemos en el Recuadro 3.2, de acuerdo también con el tipo de petición). El código para crear la relación entre las URLs y las acciones del controlador para el recurso Users, aparece en el Listado 2.2; este código construye efectivamente la tabla de parejas URL/acción vista en la Tabla 2.1. (La extraña notación `:users` es un *símbolo*, de los que aprenderemos más en la Sección 4.3.3.)

⁵ Algunas referencias indican que la vista regresa el HTML directamente al navegador (por medio de un servidor web, como Apache o Nginx). Sin importar los detalles de la implementación, yo prefiero pensar al controlador como una unidad centralizada a través de la cual toda la información de la aplicación fluye.

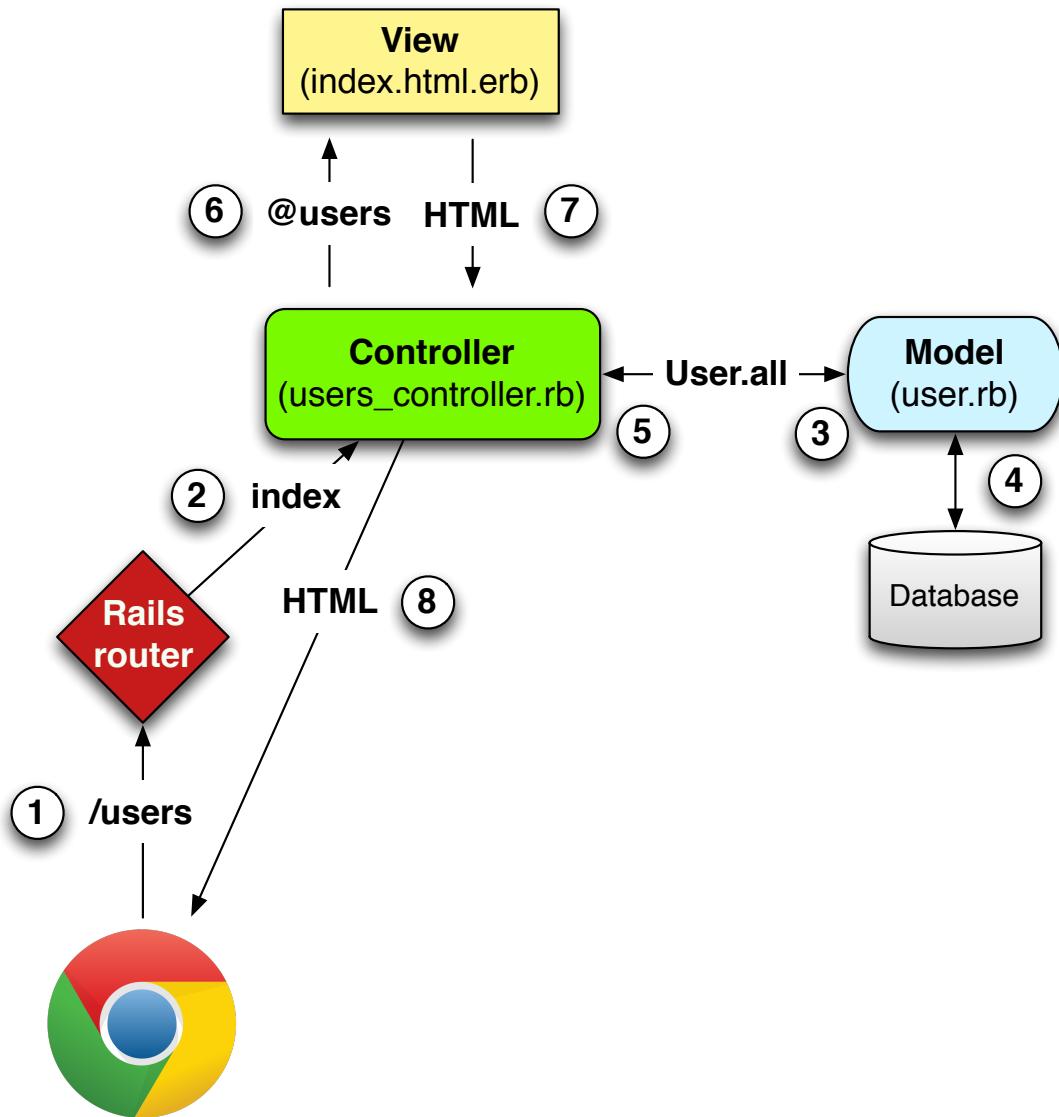


Figura 2.11: Un diagrama detallado de MVC en Rails.

Listado 2.2: Las rutas de Rails, con una regla para el recurso Users.

config/routes.rb

```
Rails.application.routes.draw do
  resources :users
  .
  .
  .
end
```

Mientras observamos el archivo de rutas, vamos a tomar un momento para asociar la ruta raíz con el listado de usuarios, de forma que “diagonal” se dirija a /users. Recuerde el [Listado 1.10](#) que cambiamos

```
# root 'welcome#index'
```

por

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

de forma que la ruta raíz se dirigiera hacia la acción **hello** en el controlador de la aplicación. En el caso que nos ocupa, queremos utilizar la acción **index** del controlador de Users, que podemos cambiar utilizando el código mostrado en el [Listado 2.3](#). (En este punto, recomiendo que también eliminemos la acción **hello** del controlador de la aplicación, si es que usted lo agregó al inicio de esta sección.)

Listado 2.3: Agregando una ruta raíz para Users.

config/routes.rb

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

Las páginas del recorrido que realizamos en la Sección 2.2.1 corresponden a *acciones* en el *controlador* Users, que es una colección de acciones relacionadas entre sí. El controlador generado cuando creamos la estructura temporal, se muestra en el esquema del Listado 2.4. Observe la notación **class UsersController < ApplicationController**; éste es un ejemplo de una *clase* Ruby con *herencia*. (Revisaremos *herencia* brevemente en la Sección 2.3.4 y cubriremos ambos temas en mayor detalle en la Sección 4.4.)

Listado 2.4: El controlador Users en forma de esquema.

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
```

petición HTTP	URL	Acción	Funcionalidad
GET	/users	index	página para enlistar a todos los usuarios
GET	/users/1	show	página para mostrar al usuario con id 1
GET	/users/new	new	página para crear un nuevo usuario
POST	/users	create	crea un nuevo usuario
GET	/users/1/edit	edit	página para actualizar al usuario con id 1
PATCH	/users/1	update	actualiza al usuario con id 1
DELETE	/users/1	destroy	borra al usuario con id 1

Tabla 2.2: Rutas RESTful proporcionadas por el recurso Users del Listado 2.2.

```

.
.
end

def destroy
.
.
end
end

```

Observe que existen más acciones que páginas; las acciones **index**, **show**, **new**, y **edit** corresponden a páginas de la Sección 2.2.1, pero existen acciones adicionales como **create**, **update**, y **destroy**. Éstas últimas, típicamente no despliegan páginas (aunque podrían hacerlo); en vez de eso, su principal propósito es modificar la información de los usuarios en la base de datos. Este conjunto completo de acciones del controlador, que se encuentra resumido en la Tabla 2.2, representa la implementación de la arquitectura REST en Rails (Recuadro 2.2), la cual está basada en la idea de *transferencia de estado representacional* identificado y nombrado por el científico en computación Roy Fielding.⁶ Observe en la Tabla 2.2 que existen algunas URLs repetidas; por ejemplo, tanto la acción **show** como la acción **update** corresponden a la URL /users/1. La diferencia entre ellas es el **método de la petición HTTP** a la que responden. Aprenderemos más sobre los métodos de las peticiones HTTP a partir de la Sección 3.3.

⁶Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Tesis doctoral, University of California, Irvine, 2000.

Recuadro 2.2. Transferencia de estado representacional (REST)

Si usted ha leído antes acerca del desarrollo web con Ruby on Rails, habrá visto muchas referencias a “REST”, que es un acrónimo de “REpresentational State Transfer”. REST es un estilo de arquitectura para desarrollar sistemas distribuidos, en red y aplicaciones web. Aunque la teoría de REST es más bien abstracta, en el contexto de aplicaciones Rails, REST significa que la mayoría de los componentes de la aplicación (tales como los usuarios y sus microposts) están modelados como *recursos* que pueden ser creados, leídos, actualizados y borrados—operaciones que corresponden tanto a las [operaciones CRUD de bases de datos relacionales](#) como a los cuatro [métodos de peticiones HTTP](#) fundamentales: POST, GET, PATCH, and DELETE.⁷ (Aprenderemos más acerca de las peticiones HTTP en la [Sección 3.3](#) y especialmente en el [Recuadro 3.2](#).)

Como desarrollador de aplicaciones Rails, el estilo de desarrollo RESTful ayuda a tomar decisiones acerca de qué controladores y acciones escribir: usted simplemente estructura la aplicación usando recursos que crean, leen, actualizan y borran. En el caso de los usuarios y sus microposts, este proceso es directo, puesto que son recursos de forma natural. En el [Capítulo 12](#), veremos un ejemplo donde los principios REST nos permiten modelar un problema más sutil, “seguir a los usuarios”, de forma natural y conveniente.

Para examinar la relación entre el controlador Users y el modelo User, enfóquémonos en una versión simplificada de la acción `index`, que se muestra en el [Listado 2.5](#). (El código generado mediante la creación de estructuras temporales, es feo y confuso, por lo que lo he omitido.)

Listado 2.5: La acción `index` de user, simplificada para la aplicación de juguete.

```
app/controllers/users_controller.rb
```

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

```

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

```

Esta acción `index` contiene la línea `@users = User.all` (Paso 3 de la Figura 2.11), que solicita al modelo User que obtenga una lista de todos los usuarios de la base de datos (Paso 4), y luego los almacena en la variable `@users` (se pronuncia “at-users”) (Paso 5). El modelo User aparece en el Listado 2.6; aunque es simple, viene equipado con una gran cantidad de funcionalidad debido a la herencia (Sección 2.3.4 y Sección 4.4). En particular, al utilizar la biblioteca de Rails llamada *Active Record*, el código del Listado 2.6 gestiona la consulta que `User.all` recupera todos los registros de usuarios de la base de datos.

Listado 2.6: El modelo User para la aplicación de juguete.

`app/models/user.rb`

```

class User < ActiveRecord::Base
end

```

Una vez que la variable `@users` es definida, el controlador llama a la *vista* (Paso 6), mostrada en el Listado 2.7. Las variables que inician con el signo `@`, son llamadas *variables de instancia*, y están disponibles en las vistas automáticamente; en este caso, la vista `index.html.erb` del Listado 2.7 itera sobre la lista `@users` y genera una línea de código HTML para cada elemento. (Recuerde, no se supone que usted entienda este código en este momento. Se muestra sólo para fines ilustrativos.)

Listado 2.7: La vista para el método index de user.

`app/views/users/index.html.erb`

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

```


| Name             | Email             |                      |                      |                                                                                |
|------------------|-------------------|----------------------|----------------------|--------------------------------------------------------------------------------|
| <%= user.name %> | <%= user.email %> | <a href="#">Show</a> | <a href="#">Edit</a> | <a 'are="" confirm:="" data:="" href="#" sure?'="" you="" {="" }="">Delete</a> |

New User

```

La vista convierte su contenido a HTML (Paso 7), el cual es regresado por el controlador al navegador para que sea mostrado (Paso 8).

2.2.3 Debilidades del recurso Users

Aunque nos sirvió para darnos una idea general de Rails, el recurso Users, generado mediante la creación de las estructuras temporales, adolesce de varias debilidades severas.

- **No realiza validación de datos.** Nuestro modelo User acepta datos tales como nombres en blanco y direcciones de correo electrónico inválidas, sin quejarse.
- **No autentica.** No tenemos la noción de iniciar o cerrar sesión, ni hay forma de evitar que cualquier usuario realice cualquier operación.

- **No realiza pruebas.** Aunque técnicamente no es cierto—la creación de estructuras temporales incluye pruebas rudimentarias— las pruebas generadas no verifican que los datos se validen, ni prueban la autenticación o algún otro requerimiento personalizado.
- **No tiene estilo ni estructura.** No hay un estilo consistente en el sitio ni navegación.
- **No se entiende realmente.** Si usted es capaz de entender el código generado cuando se crearon las estructuras temporales, probablemente no debería estar leyendo este libro.

2.3 El recurso Microposts

Habiendo generado y explorado el recurso Users, es turno ahora de su recurso asociado, Microposts. A lo largo de esta sección, recomiendo comparar los elementos del recurso Microposts con sus análogos del recurso Users de la [Sección 2.2](#); usted verá que ambos recursos son similares entre sí de varias formas. La estructura RESTful de las aplicaciones Rails es mejor comprendida mediante este tipo de repetición—en efecto, observar la semejanza entre las estructuras de Users y Microposts es una de las motivaciones principales de este capítulo.

2.3.1 Un recorrido por Micropost

Igual que hicimos con el recurso Users, generaremos código creando una estructura temporal para el recurso Microposts mediante la instrucción **rails generate scaffold**, en este caso, implementando el modelo de datos indicado en la [Figura 2.3](#):⁸

⁸De igual forma que con la generación de código para User, el generador de estructuras temporales para Microposts sigue la convención de modelos de Rails al utilizar el sustantivo en singular; por lo que escribimos **generate Micropost**.

```
$ rails generate scaffold Micropost content:text user_id:integer
  invoke  active_record
  create    db/migrate/20140821012832_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
  invoke  helper
  create    app/helpers/microposts_helper.rb
  invoke  test_unit
  create    test/helpers/microposts_helper_test.rb
  invoke  jbuilder
  create    app/views/microposts/index.json.jbuilder
  create    app/views/microposts/show.json.jbuilder
  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
```

(Si ocurre un error relacionado con Spring, simplemente ejecute el comando nuevamente.) Para actualizar nuestra base de datos con el nuevo modelo de datos, necesitamos ejecutar una migración, tal como se indica en la Sección 2.2:

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

Ahora estamos listos para crear microposts de la misma forma en que creamos

petición HTTP	URL	Acción	Funcionalidad
GET	/microposts	index	página para enlistar todos los microposts
GET	/microposts/1	show	página para mostrar el micropost con id 1
GET	/microposts/new	new	página para crear un nuevo micropost
POST	/microposts	create	crea un nuevo micropost
GET	/microposts/1/edit	edit	página para actualizar el micropost con id 1
PATCH	/microposts/1	update	actualiza el micropost con id 1
DELETE	/microposts/1	destroy	borra el micropost con id 1

Tabla 2.3: Rutas RESTful proporcionadas por el recurso Microposts del [Listado 2.8](#).

usuarios en la [Sección 2.2.1](#). Como puede imaginarse, el generador de estructuras temporales ha actualizado el archivo de rutas de Rails con una regla para el recurso Microposts, como puede observar en el [Listado 2.8](#).⁹ De igual forma que con los usuarios, la regla de enrutamiento **resources :microposts** mapea las URLs de micropost a acciones en el controlador de Microposts, como se muestra en la [Tabla 2.3](#).

Listado 2.8: Las rutas de Rails, con una nueva regla para los recursos Microposts.

config/routes.rb

```
Rails.application.routes.draw do
  resources :microposts
  resources :users
  .
  .
  .
end
```

El controlador de Microposts aparece de forma esquemática en el [Listado 2.9](#). Observe que, aparte de leer **MicropostsController** en vez de **UsersController**, el [Listado 2.9](#) es *idéntico* al código del [Listado 2.4](#). Esto es un reflejo de la arquitectura REST común a ambos recursos.

⁹El código generado puede contener algunas líneas en blanco extra comparado con el [Listado 2.8](#). Esto no es algo que deba preocuparnos, puesto que Ruby ignora las líneas en blanco.

Listado 2.9: El controlador Microposts en forma de esquema.

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
```

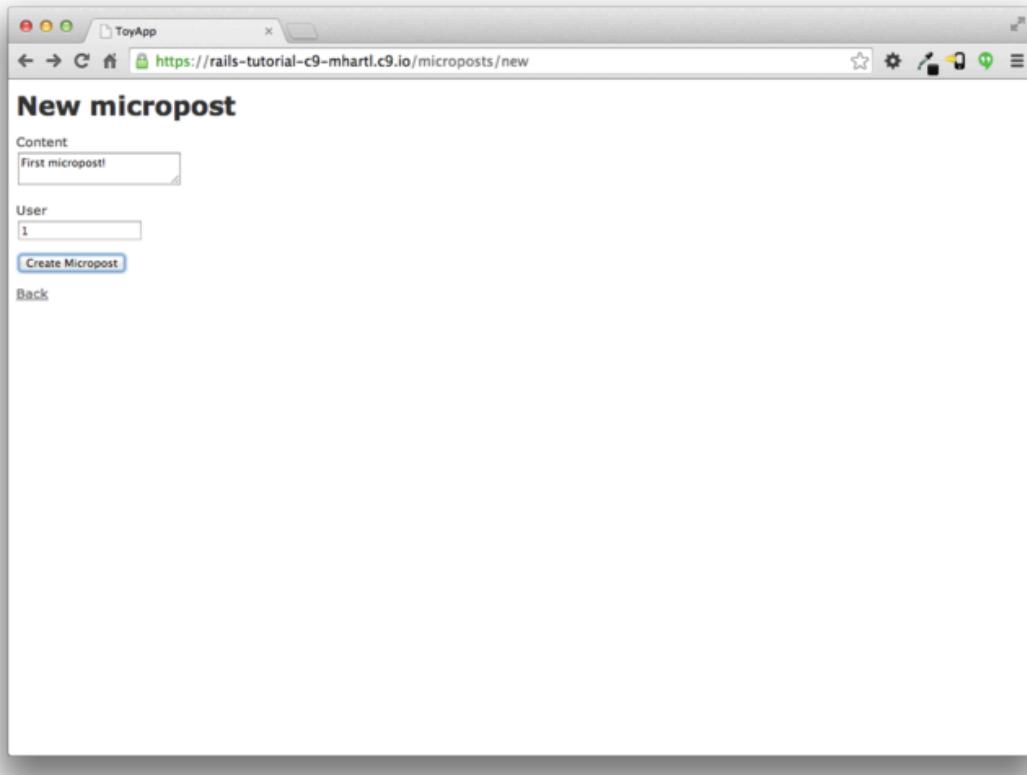


Figura 2.12: La página para crear un nuevo micropost ([/microposts/new](#)).

Para crear algunos microposts reales, ingresamos información en la página destinada para tal efecto, [/microposts/new](#), como se ve en la Figura 2.12.

En este punto, puede crear uno o dos microposts, teniendo cuidado de que al menos uno de ellos tenga el `user_id` igual a `1` para que coincida con el id del primer usuario creado en la Sección 2.2.1. El resultado debe parecerse al de la Figura 2.13.

2.3.2 Poniendo el *micro* en microposts

Cualquier *micropost* que haga honor a su nombre debe de tener alguna forma de restringir la longitud del post. La implementación de esta restricción en Rails

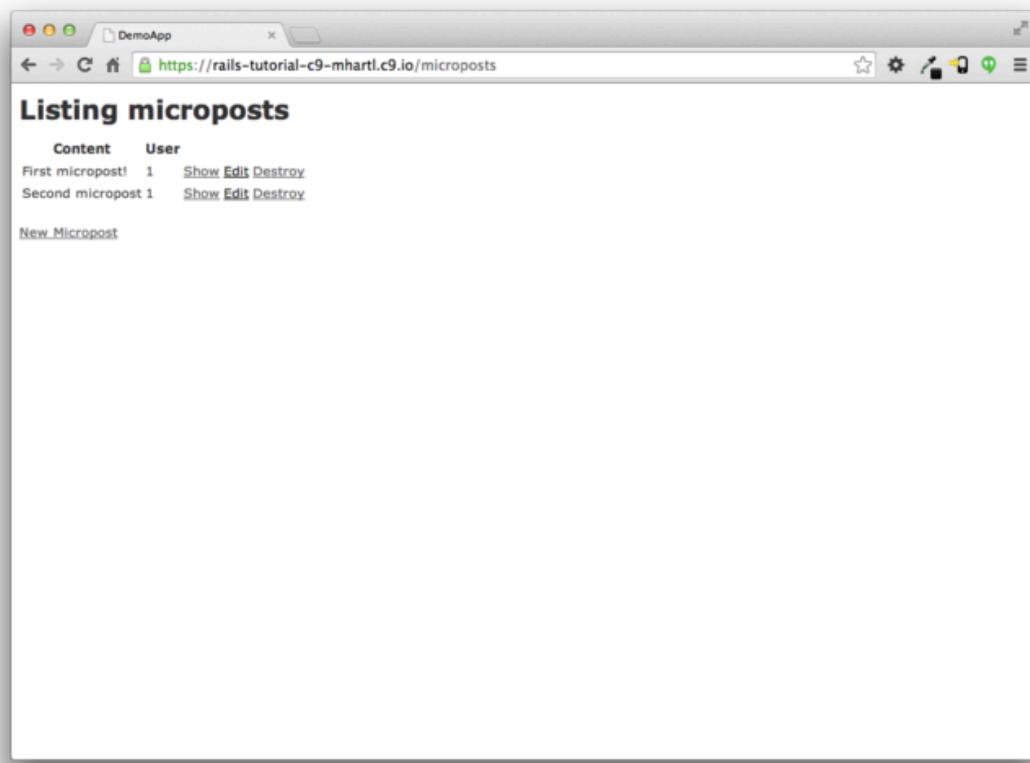


Figura 2.13: La página que enlista todos los microposts ([/microposts](#)).

es fácil con *validaciones*; para aceptar microposts con máximo 140 caracteres (al estilo Twitter), utilizamos una validación de *longitud*. En este punto, abra el archivo `app/models/micropost.rb` en su editor de texto o en su IDE y editelo con los contenidos del [Listado 2.10](#).

Listado 2.10: Restringiendo los microposts para que sean máximo de 140 caracteres.

`app/models/micropost.rb`

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

El código del [Listado 2.10](#) puede parecer algo misterioso—revisaremos más extensamente las validaciones a partir de la [Sección 6.2](#)—pero sus efectos pueden verificarse fácilmente si vamos a la página para crear un nuevo micropost e ingresamos más de 140 caracteres como contenido del post. Como se muestra en la [Figura 2.14](#), Rails muestra el *mensaje de error* indicando que el contenido del micropost es demasiado largo. (Aprenderemos más acerca de los mensajes de error en la [Sección 7.3.3](#).)

2.3.3 Un usuario tiene muchos microposts

Una de las características más poderosas de Rails es la habilidad de formar *asociaciones* entre diferentes modelos de datos. En el caso de nuestro modelo User, cada usuario tiene potencialmente muchos microposts. Podemos expresar esto en código al actualizar los modelos de User y Micropost como se muestra en los [Listados 2.11](#) y [2.12](#).

Listado 2.11: Un usuario tiene muchos microposts.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  has_many :microposts
end
```

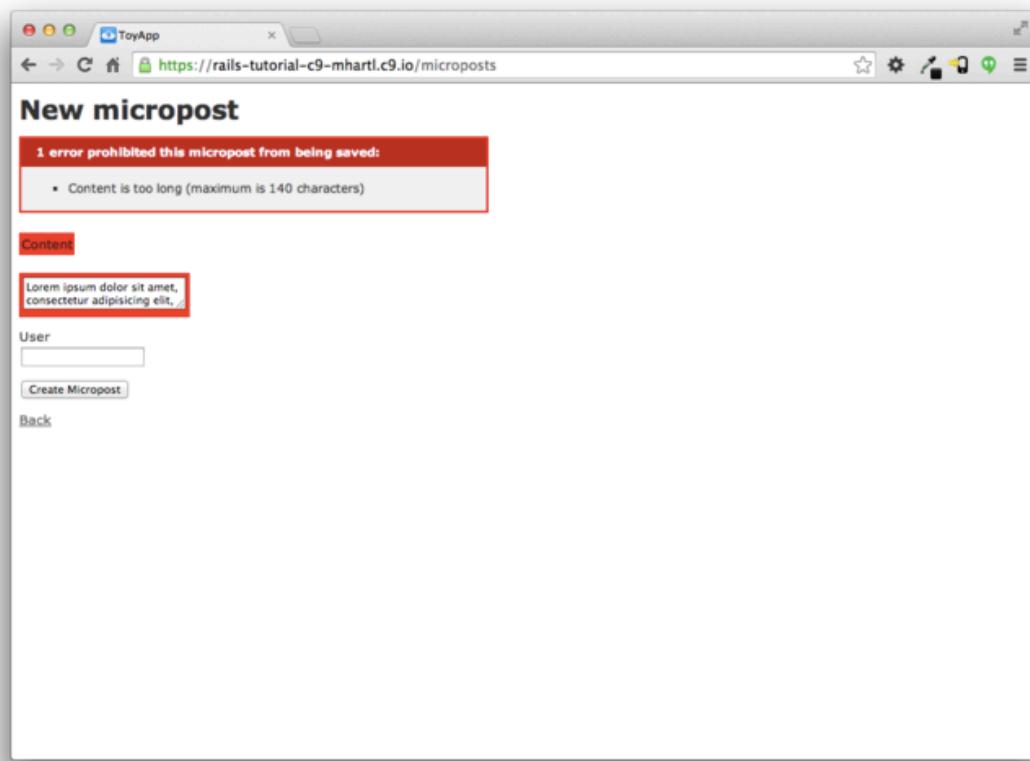


Figura 2.14: Mensajes de error para una creación fallida de un micropost.

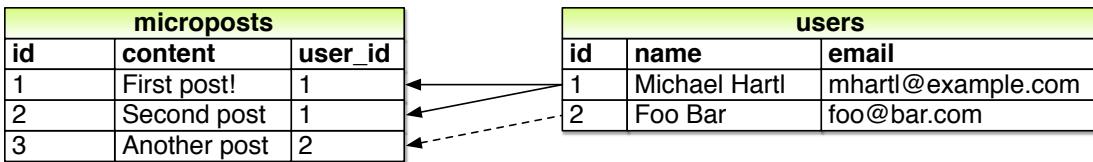


Figura 2.15: La relación entre microposts y usuarios.

Listado 2.12: Un micropost pertenece a un usuario.

app/models/micropost.rb

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

Podemos visualizar el resultado de esta relación en la Figura 2.15. Usando la columna **user_id** de la tabla **microposts**, Rails (usando Active Record) puede relacionar los microposts asociados con cada usuario.

En los capítulos 11 y 12, utilizaremos la asociación entre usuarios y microposts para mostrar todos los microposts de un usuario y construir nuevos microposts al estilo Twitter. Por ahora, examinaremos las implicaciones de la relación usuario-micropost empleando la *consola*, que es una herramienta útil para interactuar con las aplicaciones Rails. Primero invocaremos la consola con el comando **rails console**, y recuperaremos al primer usuario de la base de datos mediante la instrucción **User.first** (ponemos el resultado en la variable **first_user**):¹⁰

```
$ rails console
>> first_user = User.first
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
  created_at: "2014-07-21 02:01:31", updated_at: "2014-07-21 02:01:31">
>> first_user.microposts
=> [#<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
```

¹⁰El prompt de su consola puede verse como 2.1.1 :001 >, pero los ejemplos utilizan >> puesto que las versiones de Ruby pueden variar.

```

"2014-07-21 02:37:37", updated_at: "2014-07-21 02:37:37">>, #<Micropost id: 2,
content: "Second micropost", user_id: 1, created_at: "2014-07-21 02:38:54",
updated_at: "2014-07-21 02:38:54">]
>> micropost = first_user.microposts.first      # Micropost.first would also work.
=> #<Micropost id: 1, content: "First micropost!", user_id: 1, created_at:
"2014-07-21 02:37:37", updated_at: "2014-07-21 02:37:37">
>> micropost.user
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",
created_at: "2014-07-21 02:01:31", updated_at: "2014-07-21 02:01:31">
>> exit

```

(Incluyo la instrucción `exit` en la última línea para mostrar cómo salir de la consola. En la mayoría de los sistemas, puede utilizar Ctrl-D con el mismo propósito.)¹¹ Hasta aquí hemos accedido los microposts de un usuario utilizando el código `first_user.microposts`. Con esta instrucción, Active Record automáticamente recupera todos los microposts cuyo `user_id` es igual al id de `first_user` (en este caso, `1`). Aprenderemos más acerca de las funcionalidades disponibles en Active Record para estas relaciones, en los Capítulos 11 y 12.

2.3.4 Jerarquías de Herencia

Terminaremos nuestra discusión de la aplicación de juguete con una breve descripción de las jerarquías de clase del controlador y del modelo en Rails. Esta discusión tiene sentido sólo si usted tiene alguna experiencia con la programación orientada a objetos (OOP, por sus siglas en inglés *Object Oriented Programming*); si no ha estudiado OOP, puede omitir esta sección. En particular, si no está familiarizado con *clases* (discutidas en la Sección 4.4), le sugiero regresar a esta sección más adelante.

Empezaremos con la estructura de herencia para los modelos. Comparando los Listados 2.13 y 2.14, observe que tanto el modelo User como el modelo Micropost heredan (vía el corchete angular izquierdo `<`) de `ActiveRecord::Base`, que es la clase base para modelos proporcionada por ActiveRecord; un diagrama que muestra el resumen de esta relación aparece en la Figura 2.16. Es

¹¹Igual que con “Ctrl-C”, la letra “D” mayúscula se refiere a la tecla de su teclado, no a la letra mayúscula como tal, por lo que no es necesario presionar la tecla de mayúsculas junto con la tecla Ctrl.

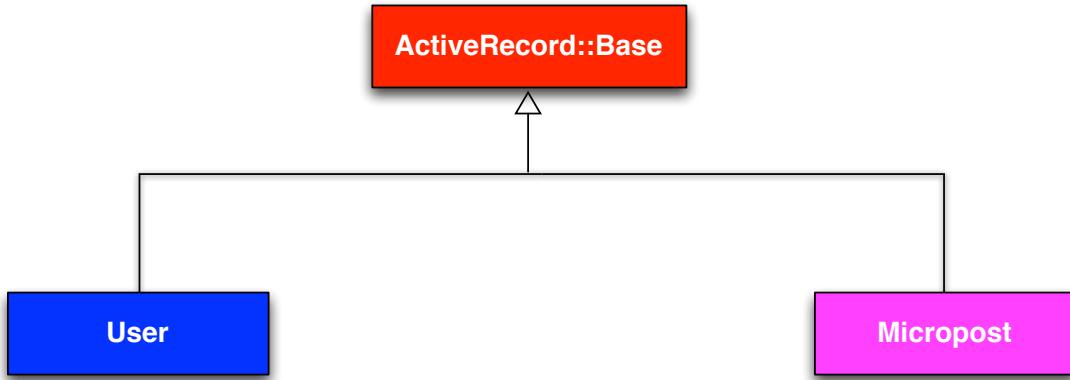


Figura 2.16: La jerarquía de herencia para los modelos User y Micropost.

por la herencia de la clase **ActiveRecord::Base** que nuestro modelo tiene la habilidad de comunicarse con la base de datos, de tratar las columnas de la tabla de la base de datos como atributos de Ruby, etcétera.

Listado 2.13: La clase **User**, resaltando la herencia.

app/models/user.rb

```

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

Listado 2.14: La clase **Micropost**, resaltando la herencia.

app/models/micropost.rb

```

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

La estructura de herencia para los controladores es ligeramente más complicada. Comparando los Listados 2.15 y 2.16, veremos que tanto el controlador

de Users como el de Microposts heredan de **ApplicationController**. Examinando el [Listado 2.17](#), veremos que el **ApplicationController** mismo, hereda de **ActionController::Base**; que es la clase base para controladores proporcionada por la biblioteca de Rails, Action Pack. Las relaciones entre estas clases están ilustradas en la [Figura 2.17](#).

Listado 2.15: La clase **UsersController**, resaltando la herencia.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController
  .
  .
  .
end
```

Listado 2.16: La clase **MicropostsController**, resaltando la herencia.

```
app/controllers/microposts_controller.rb

class MicropostsController < ApplicationController
  .
  .
  .
end
```

Listado 2.17: La clase **ApplicationController**, resaltando la herencia.

```
app/controllers/application_controller.rb

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

De igual forma que con la herencia de modelos, tanto el controlador de Users como el de Microposts obtienen una gran cantidad de funcionalidad al heredar de una clase base (en este caso, **ActionController::Base**), incluyendo la habilidad de manipular objetos de tipo modelo, filtrar peticiones HTTP de entrada, y mostrar vistas en formato HTML. Puesto que todos los controladores de

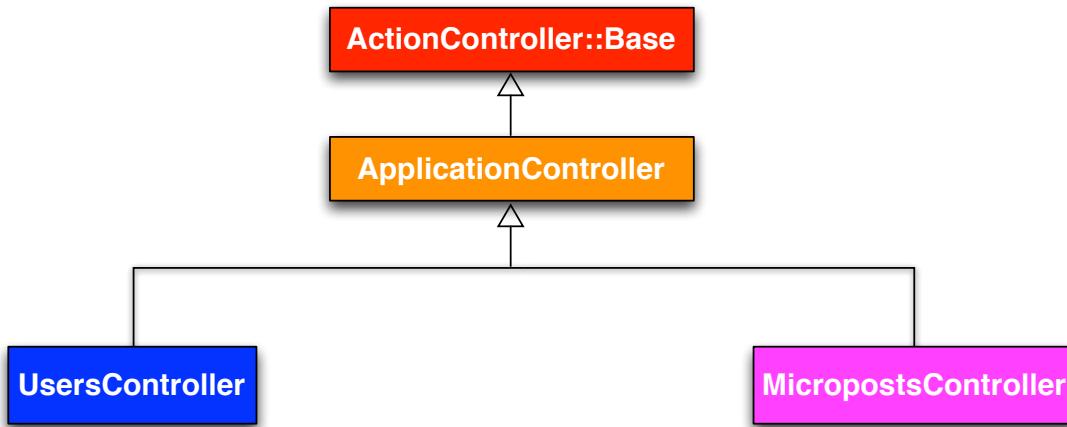


Figura 2.17: La jerarquía de herencia para los controladores de Users y Microposts.

Rails heredan de **ApplicationController**, las reglas definidas en él aplican automáticamente a cada acción en la aplicación. Por ejemplo, en la [Sección 8.4](#) veremos cómo incluir asistentes para iniciar y cerrar sesión de todos los controladores de la aplicación de ejemplo.

2.3.5 Desplegando la aplicación de juguete

Terminando de revisar el recurso Microposts, es un buen momento para subir nuestros cambios al repositorio de Bitbucket:

```

$ git status
$ git add -A
$ git commit -m "Finish toy app"
$ git push
  
```

Usualmente, se deberían hacer actualizaciones más pequeñas y frecuentes al repositorio, pero para el propósito de este capítulo, una gran actualización al finalizar es suficiente.

En este punto, también puede desplegar la aplicación de juguete en Heroku como en la [Sección 1.5](#):

```
$ git push heroku
```

(Esto supone que usted creó la aplicación para Heroku de la [Sección 2.1](#). En caso contrario, ejecute **heroku create** y luego **git push heroku master**.)

Para que la base de datos de la aplicación funcione, deberá migrar la base de datos de producción:

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

Esto actualiza la base de datos en Heroku con los modelos de datos necesarios de usuario y micropost. Luego de correr la migración, debería poder utilizar la aplicación de juguete en producción, con una base de datos PostgreSQL real en el servidor ([Figura 2.18](#)).

2.4 Conclusión

Hemos llegado al final de esta revisión general de una aplicación Rails. La aplicación de juguete desarrollada en este capítulo tiene una serie de fortalezas y debilidades.

Fortalezas

- Revisión general de Rails
- Introducción a MVC
- Primera experiencia con la arquitectura REST
- Introducción al modelado de datos
- Una aplicación web viva, respaldada por una base de datos, en producción

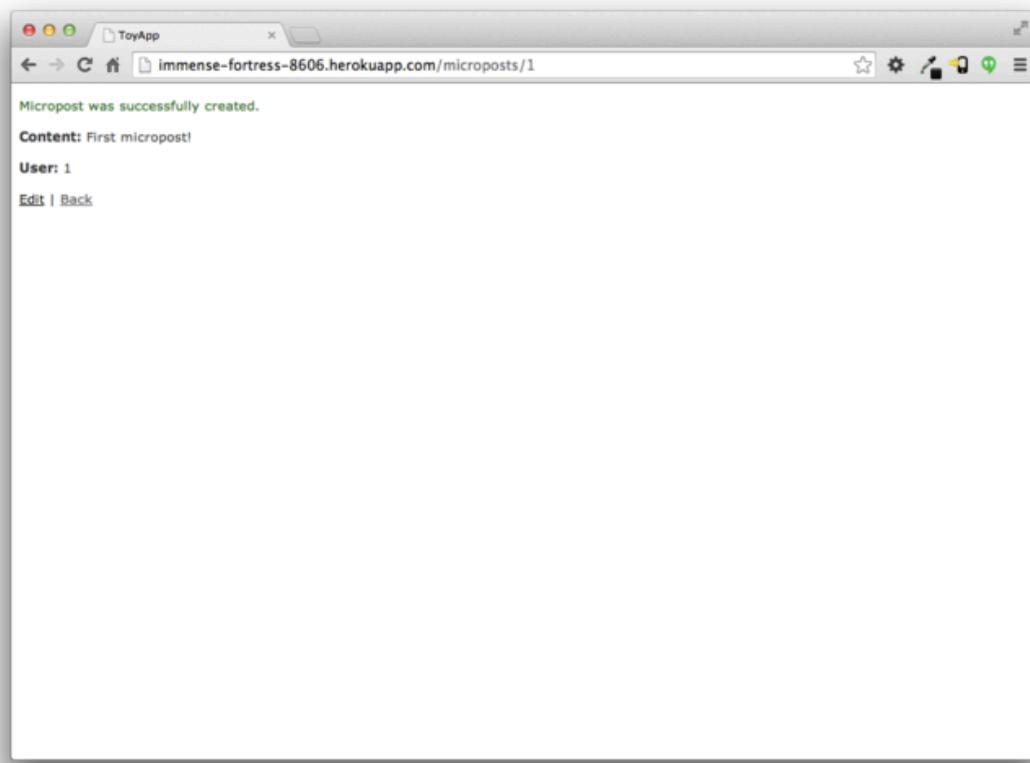


Figura 2.18: Ejecutando la aplicación de juguete en producción.

Debilidades

- No hay un diseño o estilo personalizado
- No hay páginas estáticas (como “Home” o “About”)
- No hay contraseñas de usuario
- No hay imágenes de usuario
- No hay inicio de sesión
- No hay seguridad
- No existe una asociación automática de usuario/micropost
- No existe la noción de “siguiendo” o “seguido”
- No puede alimentar microposts
- No existen pruebas significativas
- **No hay un entendimiento real**

El resto de este tutorial está dedicado a construir sobre las fortalezas y eliminar las debilidades.

2.4.1 Qué aprendimos en este capítulo

- Al crear estructuras temporales, automáticamente creamos código para modelar datos e interactuar con ellos a través de internet.
- La creación de estructuras temporales es buena para empezar rápidamente pero es mala para entender.
- Rails utiliza el patrón Modelo-Vista-Controlador (MVC) para estructurar las aplicaciones web.

- Según la interpretación de Rails, la arquitectura REST incluye un conjunto estándar de URLs y acciones del controlador para interactuar con los modelos de datos.
- Rails soporta validaciones de datos para establecer restricciones sobre los valores de los atributos del modelo de datos.
- Rails incluye funciones pre-construídas para definir relaciones entre diferentes modelos de datos.
- Podemos interactuar con las aplicaciones Rails desde línea de comandos usando la consola Rails.

2.5 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

1. El código del [Listado 2.18](#) muestra cómo agregar una validación para el contenido del micropost con la finalidad de asegurar que éstos no vayan en blanco. Verifique que puede reproducir el comportamiento mostrado en la [Figura 2.19](#).
2. Actualice el [Listado 2.19](#) reemplazando **FILL_IN** con el código apropiado para validar la presencia de los atributos `name` y `email` del modelo User ([Figura 2.20](#)).

Listado 2.18: Código para validar el contenido en un micropost.

`app/models/micropost.rb`

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

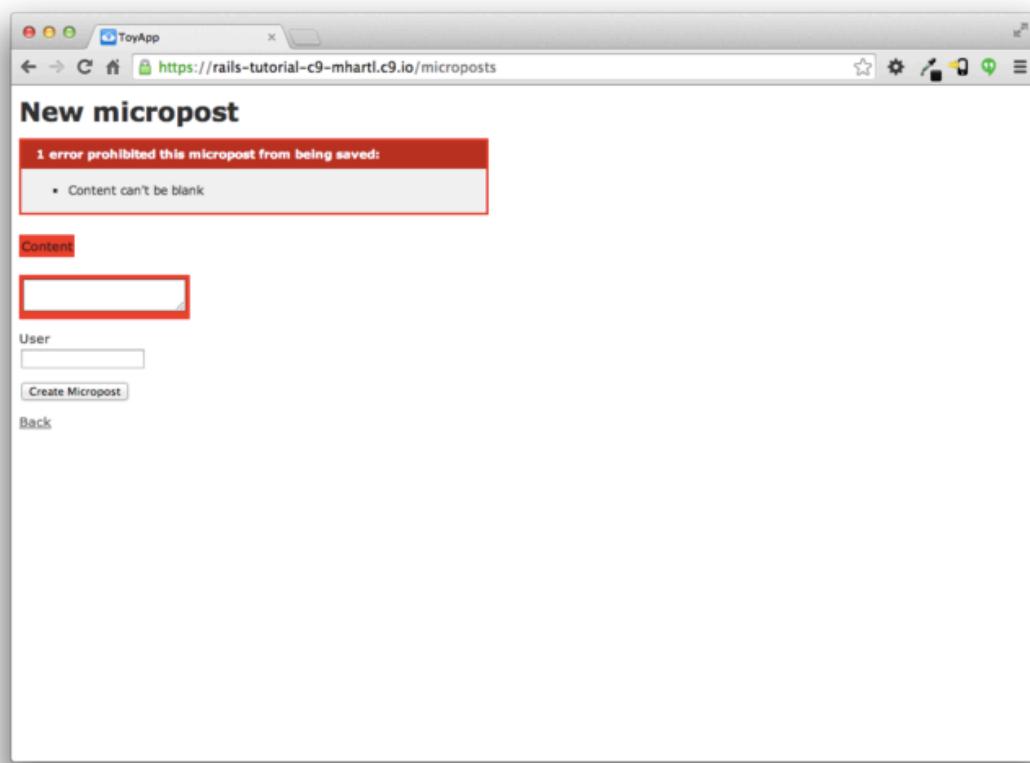


Figura 2.19: El efecto de la validación del contenido en un micropost.

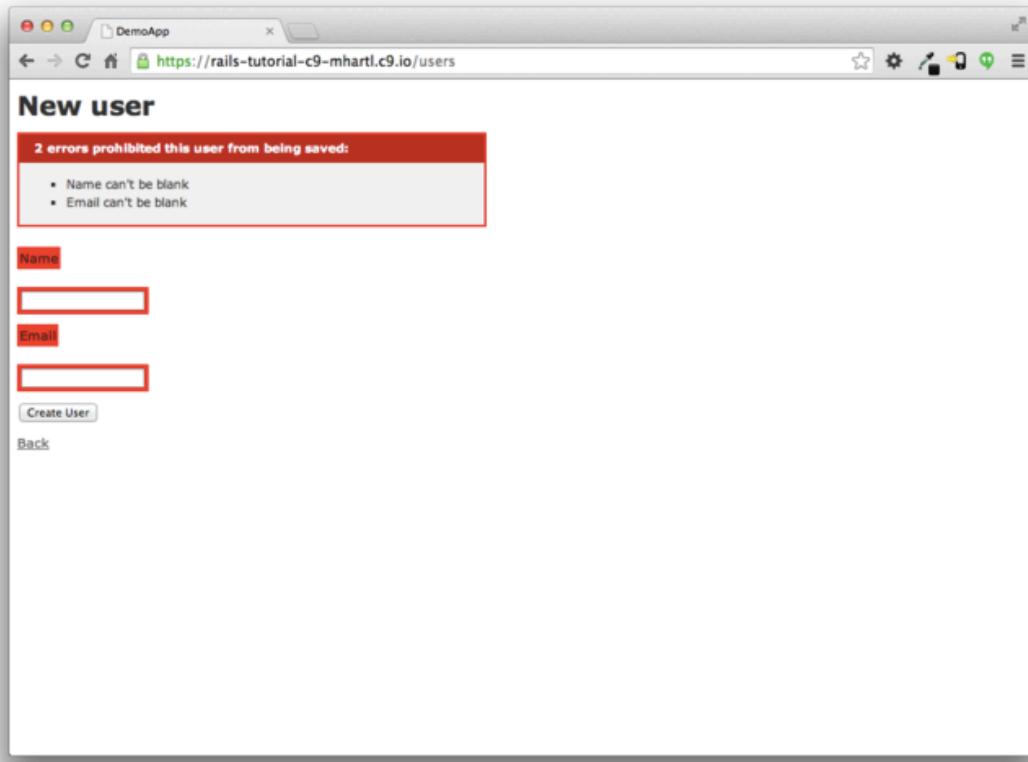


Figura 2.20: El efecto de las validaciones del contenido en el modelo User.

Listado 2.19: Agregando validaciones al modelo User.

app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts
  validates FILL_IN, presence: true
  validates FILL_IN, presence: true
end
```

Capítulo 3

Páginas casi estáticas

En este capítulo, empezaremos a desarrollar la aplicación de ejemplo de grado profesional que nos servirá como ejemplo a lo largo del resto del libro. Aunque la aplicación de ejemplo tendrá finalmente usuarios, microposts, y un marco de trabajo para el manejo de sesión y autenticación, empezaremos con un tema aparentemente limitado: la creación de páginas estáticas. A pesar de su aparente simplicidad, elaborar páginas estáticas es un ejercicio altamente educativo, rico en implicaciones—un perfecto inicio para nuestra aplicación naciente.

Aunque Rails está diseñado para crear sitios web dinámicos con bases de datos en el servidor, también sobresale en la creación de páginas estáticas del tipo que podríamos hacer únicamente con archivos HTML. De hecho, usar Rails aún para páginas estáticas nos proporciona una ventaja adicional: podemos agregar fácilmente una *pequeña* cantidad de contenido dinámico. En este capítulo aprenderemos cómo. En el camino, tendremos nuestra primera experiencia con las *pruebas automatizadas*, las cuales nos ayudarán a tener más confianza de que nuestro código es correcto. Más aún, el contar con un buen conjunto de pruebas nos permitirá *refactorizar* nuestro código con confianza, cambiando su forma sin cambiar su funcionalidad.

3.1 Preparación de la aplicación de ejemplo

De igual forma que en el Capítulo 2, antes de iniciar necesitamos crear un nuevo proyecto Rails, esta vez lo llamaremos `sample_app`, como se muestra en el Listado 3.1.¹ Si el comando del Listado 3.1 devuelve un error como “Could not find ‘railties’”, significa que usted no tiene instalada la versión correcta de Rails, y debería verificar que ha ejecutado el comando indicado en el Listado 1.1 exactamente como está escrito.

Listado 3.1: Generando una nueva aplicación de ejemplo.

```
$ cd ~/workspace
$ rails _4.2.2_ new sample_app
$ cd sample_app/
```

(Igual que en la Sección 2.1, observe que los usuarios del IDE en la nube, pueden crear este proyecto en el mismo espacio de trabajo que usaron para las aplicaciones de los dos capítulos anteriores. No es necesario crear un nuevo espacio de trabajo.)

Del mismo modo que en la Sección 2.1, nuestro siguiente paso es utilizar un editor de texto para actualizar el archivo `Gemfile` con las gemas necesarias para nuestra aplicación. El Listado 3.2 es idéntico al Listado 1.5 y al Listado 2.1 excepto por las gemas del grupo `test`, que son necesarias para la preparación de las pruebas avanzadas opcionales (Sección 3.7). *Nota:* Si desea instalar *todas* las gemas necesarias para la aplicación de ejemplo, debería utilizar el código del Listado 11.67 en este momento.

¹Si usted está utilizando el IDE en la nube, es útil usar el comando “Goto Anything” con frecuencia, pues facilita la navegación en el sistema de archivos al teclear nombres de archivo parciales. En este contexto, tener las aplicaciones `hello`, `juguete`, y `ejemplo` presentes en el mismo proyecto puede no ser conveniente debido a los archivos que tienen nombres en común. Por ejemplo, al buscar un archivo llamado “`Gemfile`”, se mostrarán seis posibilidades, puesto que cada proyecto tiene sus respectivos archivos llamados `Gemfile` y `Gemfile.lock`. Entonces, debería considerar remover las primeras dos aplicaciones antes de continuar, lo cual puede hacer navegando al directorio `workspace` y ejecutando `rm -rf hello_app/ toy_app/` (Tabla 1.1). (Siempre que haya subido el código a los repositorios de Bitbucket correspondientes, podrá recuperarlo después.)

Listado 3.2: Un archivo **Gemfile** para la aplicación de ejemplo.

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

gem 'rails',          '4.2.2'
gem 'sass-rails',     '5.0.2'
gem 'uglifier',        '2.5.3'
gem 'coffee-rails',   '4.1.0'
gem 'jquery-rails',   '4.0.3'
gem 'turbolinks',     '2.3.0'
gem 'jbuilder',        '2.2.3'
gem 'sdoc',            '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',       '1.3.9'
  gem 'byebug',        '3.4.0'
  gem 'web-console',   '2.0.0.beta3'
  gem 'spring',         '1.1.3'
end

group :test do
  gem 'minitest-reporters', '1.0.5'
  gem 'mini_backtrace',    '0.1.3'
  gem 'guard-minitest',   '2.3.1'
end

group :production do
  gem 'pg',             '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

Como en los capítulos anteriores, ejecutamos **bundle install** para instalar e incluir las gemas especificadas en el archivo **Gemfile**, podemos inhibir la instalación de las gemas de producción utilizando la opción **--without production**:²

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

Esto evita que la gema **pg** de PostgreSQL sea instalada en desarrollo, y que usemos SQLite para desarrollo y pruebas. Heroku recomienda no emplear diferentes bases de datos en desarrollo que en producción, pero para la aplicación

²Es importante mencionar que **--without production** es una “opción recordada”, lo que significa que será incluida automáticamente la próxima vez que ejecutemos **bundle install**.

de ejemplo no representa ninguna diferencia, y SQLite es *mucho* más fácil que PostgreSQL para instalar y configurar localmente.³ En caso de que haya instalado previamente una versión de la gema (tal como lo es Rails) diferente de la especificada en el archivo **Gemfile**, se recomienda *actualizar* las gemas con la instrucción **bundle update** para asegurarnos que las versiones coinciden:

```
$ bundle update
```

Hecho esto, todo lo que nos resta es inicializar el repositorio Git:

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

Como con la primera aplicación, sugiero actualizar el archivo **README** (ubicado en el directorio raíz de la aplicación) para que sea más útil y descriptivo. Empezamos cambiando el formato de RDoc a Markdown:

```
$ git mv README.rdoc README.md
```

Luego le ponemos el contenido que se muestra en el [Listado 3.3](#).

Listado 3.3: Una versión mejorada del archivo **README** para la aplicación de ejemplo.

```
# Ruby on Rails Tutorial: sample application

This is the sample application for the
[*Ruby on Rails Tutorial:
Learn Web Development with Rails*](http://www.railstutorial.org/)
by [Michael Hartl](http://www.michaelhartl.com/).
```

³Le recomiendo que en algún momento aprenda a instalar y configurar PostgreSQL en desarrollo, pero es muy probable que éste no sea ese momento. Cuando decida hacerlo, busque en Google “install configure postgresql <su sistema operativo>” y “rails postgresql setup” y prepárese para un reto. (Para el caso del IDE en la nube, <su sistema operativo> es Ubuntu.)

Por último, hacemos **commit** a los cambios:

```
$ git commit -am "Improve the README"
```

Recuerde que en la Sección 1.4.4 usamos el comando Git **git commit -a -m "Message"**, con las opciones para “todos los cambios” (**-a**) y un mensaje (**-m**). En el comando que justo acabamos de mostrar, Git nos permite juntar las dos opciones en una, utilizando **git commit -am "Message"**.

Puesto que estaremos utilizando esta aplicación de ejemplo a lo largo del resto del libro, se recomienda [crear un nuevo repositorio en Bitbucket](#) y agregarle nuestros cambios:

```
$ git remote add origin git@bitbucket.org:<username>/sample_app.git  
$ git push -u origin --all # pushes up the repo and its refs for the first time
```

Para evitar dolores de cabeza posteriores cuando hagamos la integración, se recomienda también desplegar la aplicación en Heroku aún cuando parezca demasiado pronto. Como en los capítulos 1 y 2, sugiero seguir los pasos del “hello, world!” mostrados en los Listados 1.8 y 1.9.⁴ Luego haga **commit** a los cambios y súbalos a Heroku:

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

(Como en la Sección 1.5, es probable que vea algunos mensajes de advertencia, los cuales puede ignorar por ahora. Los eliminaremos en la Sección 7.5.) Excepto por la dirección de la aplicación en Heroku, el resultado debe ser el mismo que el de la Figura 1.18.

Conforme avance en lo que resta del libro, le recomiendo subir sus cambios y desplegar la aplicación regularmente, esto automáticamente crea respaldos

⁴Como se hizo notar en el Capítulo 2, la principal razón para hacer esto es que la página que se crea por defecto en Rails, usualmente genera error en Heroku, lo que hace difícil decir si el despliegue fue exitoso o no.

remotos y le permite notificarse de cualquier error en producción lo más pronto posible. Si encuentra errores en Heroku, asegúrese de echar un vistazo a las bitácoras de producción para tratar de diagnosticar el problema:

```
$ heroku logs
```

Nota: Si termina utilizando Heroku para una aplicación de la vida real, asegúrese de configurar el servidor web de producción como se indica en la [Sección 7.5](#).

3.2 Páginas estáticas

Una vez terminada la preparación de la [Sección 3.1](#), estamos listos para iniciar el desarrollo de la aplicación de ejemplo. En esta sección, daremos nuestro primer paso hacia la creación de páginas dinámicas creando un conjunto de *acciones* y *vistas* de Rails que contienen únicamente HTML estático.⁵ Las acciones de Rails vienen agrupadas dentro de los *controladores* (la C en MVC de la [Sección 1.3.3](#)), los cuales contienen conjuntos de acciones relacionadas con un propósito en común. Ya echamos un vistazo a los controladores en el [Capítulo 2](#), y los entenderemos más profundamente cuando exploremos la [arquitectura REST](#) con mayor detalle (a partir del [Capítulo 6](#)). Para orientarnos, es útil recordar la estructura del directorio Rails vista en la [Sección 1.3](#) ([Figura 1.4](#)). En esta sección, estaremos trabajando principalmente en los directorios **app/controllers** y **app/views**.

Recuerde de la [Sección 1.4.4](#), que cuando usamos Git, se considera una buena práctica realizar nuestro trabajo en una rama separada en vez de realizarlo en la rama principal. Si está usted utilizando Git para el control de versiones, debería ejecutar el siguiente comando para crear una rama para las páginas estáticas:

⁵Nuestro método para crear páginas estáticas es probablemente el más sencillo, pero no el único. El método óptimo realmente depende de sus necesidades; si espera tener un *gran* número de páginas estáticas, usar un controlador para tal fin puede llegar a ser algo engorroso, pero en nuestra aplicación de ejemplo, sólo necesitaremos unas cuantas. Si necesita muchas páginas estáticas, échelle un vistazo a la gema [high_voltage](#). Para una vieja pero aún útil discusión sobre este tema, vea el [post](#) acerca de páginas simples en el blog [hasmanythrough](#).

```
$ git checkout master
$ git checkout -b static-pages
```

(La primera línea de éste código es sólo para asegurarnos de que usted se encuentra en la rama principal, de forma que la rama **static-pages** tiene como base la rama principal (**master**). Puede omitir este comando si usted está seguro de estar en la rama principal.)

3.2.1 Páginas estáticas generadas

Para empezar con las páginas estáticas, generaremos primero un controlador usando el mismo script de Rails **generate** que utilizamos en el Capítulo 2 para crear estructuras temporales. Puesto que crearemos un controlador para manejar las páginas estáticas, lo llamaremos controlador de Páginas Estáticas, designado con el nombre en inglés usando la notación **CamelCase**, **StaticPages**. También planeamos crear acciones para una página de inicio (Home), una de ayuda (Help), y una de “acerca” (About), designadas como los nombres de las acciones en minúsculas **home**, **help**, y **about**. El script **generate** acepta una lista de acciones opcional, por lo que incluiremos acciones para las páginas Home y Help directamente en la línea de comandos, e intencionalmente dejaremos fuera la acción para la página About de forma que luego veamos cómo agregarla (Sección 3.3). El comando resultante para generar el controlador de páginas estáticas aparece en el Listado 3.4.

Listado 3.4: Generando el controlador de Páginas Estáticas .

```
$ rails generate controller StaticPages home help
      create  app/controllers/static_pages_controller.rb
      route   get 'static_pages/help'
      route   get 'static_pages/home'
      invoke  erb
      create    app/views/static_pages
      create    app/views/static_pages/home.html.erb
      create    app/views/static_pages/help.html.erb
      invoke  test_unit
      create    test/controllers/static_pages_controller_test.rb
      invoke  helper
```

Comando completo	Comando Abreviado
\$ rails server	\$ rails s
\$ rails console	\$ rails c
\$ rails generate	\$ rails g
\$ bundle install	\$ bundle
\$ rake test	\$ rake

Tabla 3.1: Algunos comandos Rails abreviados.

```

create    app/helpers/static_pages_helper.rb
invoke    test_unit
create    test/helpers/static_pages_helper_test.rb
invoke    assets
invoke    coffee
create    app/assets/javascripts/static_pages.js.coffee
invoke    scss
create    app/assets/stylesheets/static_pages.css.scss

```

Por cierto, es importante observar que **rails g** es una abreviatura de **rails generate**, que es una de varias abreviaturas aceptadas por Rails (Tabla 3.1). Para mayor claridad, este tutorial siempre utiliza el comando completo, pero en la vida real, la mayoría de los desarrolladores Rails utilizan una o más de las abreviaturas que se muestran en la Tabla 3.1.

Antes de continuar, si usted está utilizando Git, le recomiendo agregar los archivos del controlador de Páginas Estáticas al repositorio remoto:

```

$ git status
$ git add -A
$ git commit -m "Add a Static Pages controller"
$ git push -u origin static-pages

```

El último comando se encarga de subir la rama **static-pages** a Bitbucket. Las siguientes actualizaciones pueden omitir los argumentos y escribir simplemente

```
$ git push
```

La secuencia de comandos `commit` y `push` que se mostró anteriormente, es comúnmente utilizada por los desarrolladores cuando trabajan en la vida real, pero por simplicidad, omitiré los *commits* intermedios de ahora en adelante.

En el [Listado 3.4](#), observe que hemos puesto el nombre del controlador en notación Camello (conocida en inglés como *CamelCase*), lo que resulta en la creación de un archivo para el controlador cuyo nombre sigue la [notación serpiente](#) (conocida en inglés como *snake case*), de modo que si el controlador se llama `StaticPages`, el archivo se llamará `static_pages_controller.rb`. Esto es únicamente una convención, y de hecho si usamos la notación serpiente en la línea de comandos, también funciona: el comando

```
$ rails generate controller static_pages ...
```

también genera un archivo para el controlador llamado `static_pages_controller.rb`. Como Ruby utiliza la notación Camello para los nombres de clases ([Sección 4.4](#)), prefiero referirme a los controladores utilizando sus nombres bajo la notación Camello, pero es cuestión de gustos. (Puesto que los nombres de archivos de Ruby siguen la notación serpiente, el generador de Rails convierte las notaciones de Camello a serpiente usando el método [guión bajo](#) (conocido en inglés como *underscore*.)

Por cierto, si usted comete un error al generar código, es útil saber cómo revertir el proceso. Revise el [Recuadro 3.1](#) para conocer algunas técnicas para deshacer cosas en Rails.

Recuadro 3.1. Deshaciendo cosas

Aún cuando usted sea muy cuidadoso, las cosas algunas veces van mal cuando se desarrollan aplicaciones. Felizmente, Rails tiene algunas herramientas para ayudarle a corregirlas.

Un escenario común es el querer deshacer la generación de código—por ejemplo, cuando usted cambia de opinión al nombrar un controlador y desea eliminar los archivos generados. Como Rails crea un número sustancial de archivos auxiliares junto con el controlador (como vimos en el [Listado 3.4](#)), no basta con

eliminar únicamente el archivo del controlador; deshacer la generación implica remover el archivo principal junto con los archivos auxiliares. (De hecho, en las Secciones 2.2 y 2.3 vimos que `rails generate` edita automáticamente el archivo `routes.rb`, y por tanto, también queremos que deshaga esos cambios automáticamente.) Podemos lograr esto mediante el comando `rails destroy` seguido del nombre del elemento generado. En particular, estos dos commandos se cancelan mutuamente:

```
$ rails generate controller StaticPages home help  
$ rails destroy controller StaticPages home help
```

Análogamente, en el [Capítulo 6](#) generaremos un *modelo* como sigue:

```
$ rails generate model User name:string email:string
```

Esta instrucción puede revertirse usando

```
$ rails destroy model User
```

(En este caso, podemos omitir los otros argumentos. Cuando lleguemos al [Capítulo 6](#), entenderá porqué.)

Otra técnica relacionada con los modelos implica deshacer *migraciones*, de las cuales hablamos brevemente en el [Capítulo 2](#) y veremos más ampliamente a partir del [Capítulo 6](#). Las migraciones modifican el estado de la base de datos mediante el comando

```
$ bundle exec rake db:migrate
```

Podemos deshacer la última migración usando

```
$ bundle exec rake db:rollback
```

Para deshacer todos los cambios desde el principio, utilizamos

```
$ bundle exec rake db:migrate VERSION=0
```

Como puede suponer, si substituimos el 0 por cualquier otro número, regresaremos a la migración que corresponde a ese número de versión; éstos números se asignan secuencialmente.

Con estas técnicas a mano, estamos bien equipados para recuperarnos de un **error** de desarrollo.

La generación del controlador de Páginas Estáticas del [Listado 3.4](#) automáticamente actualiza el archivo de rutas (**config/routes.rb**), que revisamos brevemente en la [Sección 1.3.4](#). Este archivo es responsable de implementar el enrutador (que se muestra en la [Figura 2.11](#)), el cual define las relaciones entre las URLs y las páginas web. Se encuentra ubicado en el directorio **config**, donde Rails coloca los archivos necesarios para la configuración de la aplicación ([Figura 3.1](#)).

Puesto que incluimos las acciones **home** y **help** del [Listado 3.4](#), el archivo de rutas contiene una regla para cada una de éstas acciones, como puede observarse en el [Listado 3.5](#).

Listado 3.5: Las rutas para las acciones **home** y **help** del controlador de Páginas Estáticas.

config/routes.rb

```
Rails.application.routes.draw do
  get 'static_pages/home'
  get 'static_pages/help'
  .
  .
  .
end
```

Donde la regla

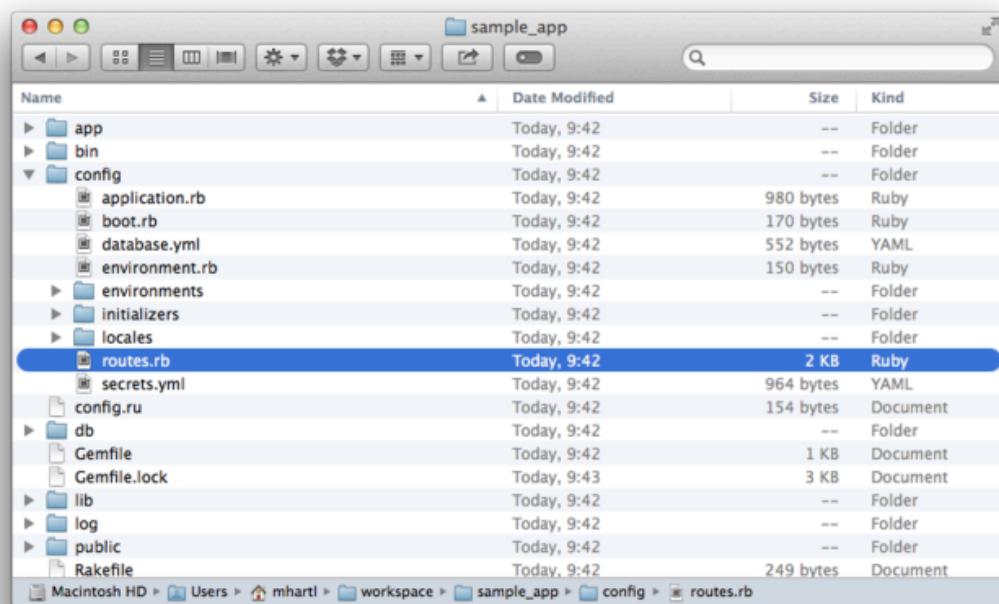


Figura 3.1: Contenido del directorio **config** de la aplicación de ejemplo.

```
get 'static_pages/home'
```

relaciona las peticiones de la URL `/static_pages/home` con la acción `home` del controlador de Páginas Estáticas. Más aún, al utilizar `get` estamos indicando que la ruta debe responder a una petición GET, que es uno de los *verbos HTTP* fundamentales soportados por el protocolo HTTP (Recuadro 3.2). En este caso, significa que cuando generamos una acción `home` dentro del controlador de Páginas Estáticas, automáticamente tenemos una página en la dirección `/static_pages/home`. Para ver el resultado, inicie el servidor de desarrollo Rails como se describió en la Sección 1.3.2:

```
$ rails server -b $IP -p $PORT      # Use only `rails server` if running locally
```

Luego navegue a la URL: [/static_pages/home](#) (Figura 3.2).

Recuadro 3.2. GET, etcétera.

El protocolo de transferencia de hipertexto ([HTTP](#)) define las operaciones básicas GET, POST, PATCH, y DELETE. Éstas se refieren a operaciones entre una computadora *cliente* (típicamente ejecutando un navegador web como Chrome, Firefox, o Safari) y un *servidor* (típicamente ejecutando un servidor web como Apache o Nginx). (Es importante entender que cuando desarrollamos aplicaciones Rails en una computadora local, el cliente y el servidor se encuentran en la misma máquina física, pero en general, son diferentes.) Las bibliotecas web (incluyendo Rails) enfatizan el uso de los verbos HTTP, influenciadas por la *arquitectura REST*, que revisamos brevemente en el Capítulo 2 y de la que aprenderemos más en el Capítulo 7.

GET es la operación HTTP más frecuente, utilizada para *leer* datos en internet; significa simplemente “obtén una página”, y cada vez que usted visita un sitio como <http://www.google.com/> o <http://www.wikipedia.org/> su navegador está enviando una petición GET. POST es la segunda operación más

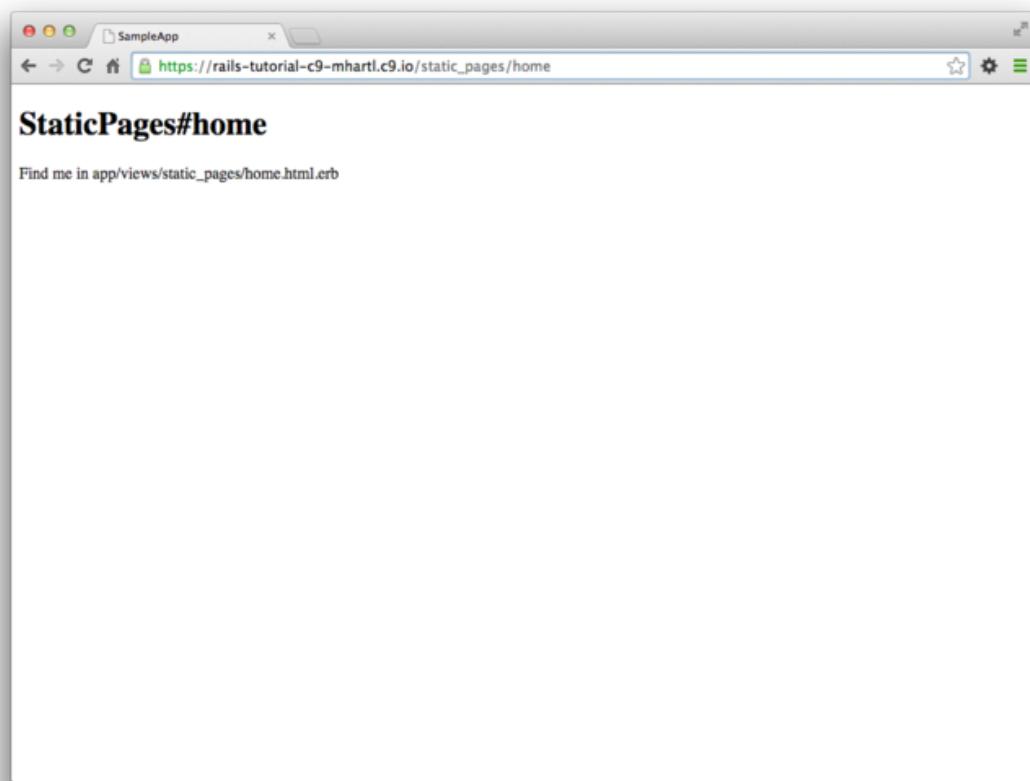


Figura 3.2: La vista home básica ([/static_pages/home](#)).

frecuente; representa la petición enviada por el navegador cuando usted envía un formulario. En las aplicaciones Rails, las peticiones POST típicamente se usan cuando *creamos* cosas (aunque HTTP también permite que POST realice actualizaciones). Por ejemplo, la petición POST se envía cuando se mandan los datos de un formulario de registro para crear un nuevo usuario en un sitio remoto. Los otros dos verbos, PATCH y DELETE, fueron diseñados para *actualizar* y *borrar* cosas en el servidor remoto. Éstas peticiones son menos comunes que GET y POST puesto que los navegadores son incapaces de enviarlas de forma nativa, pero algunas bibliotecas web (incluyendo Ruby on Rails) tienen formas ingeniosas de *simular* que los navegadores están enviando estas peticiones. Como resultado, Rails soporta los cuatro tipos de peticiones: GET, POST, PATCH, y DELETE.

Para entender de dónde viene esta página, empecemos echando un vistazo al controlador de Páginas Estáticas en un editor de texto, el cual debería verse como el del [Listado 3.6](#). Observe que, a diferencia de los controladores Users y Microposts del [Capítulo 2](#), el controlador de Páginas Estáticas no utiliza las acciones REST estándar. Esto es normal para una colección de páginas estáticas: la arquitectura REST no es la mejor solución para todos los problemas.

Listado 3.6: El controlador de Páginas Estáticas hecho por el [Listado 3.4](#).
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

Observemos que la palabra reservada **class** del [Listado 3.6](#) que **static_pages_controller.rb** define una *clase*, en este caso llamada **StaticPagesController**. Las clases son simplemente una forma conveniente de organizar *funciones* (también llamadas *métodos*) como las acciones **home** y **help**, que se definieron

usando la palabra reservada `def`. Como vimos en la Sección 2.3.4, el corchete angular izquierdo `<` indica que el `StaticPagesController` hereda de la clase Rails `ApplicationController`; como veremos en un momento, esto significa que nuestras páginas vienen equipadas con una gran cantidad de funcionalidad específica de Rails. (Aprenderemos más acerca de clases y de herencia en la Sección 4.4.)

En el caso del controlador de Páginas Estáticas, ambos métodos están inicialmente vacíos:

```
def home
end

def help
end
```

En Ruby plano, estos métodos simplemente no hacen nada. En Rails, la situación es diferente—`StaticPagesController` es una clase Ruby, pero como hereda de `ApplicationController` el comportamiento de sus métodos es específico de Rails: cuando visitamos la URL `/static_pages/home`, Rails busca el controlador de Páginas Estáticas y ejecuta el código de la acción `home`, luego despliega la *vista* (la V en MVC de la Sección 1.3.3) que corresponde a la acción. En este caso, la acción `home` está vacía, por lo que al visitar `/static_pages/home` todo lo que hace es desplegar la vista. Entonces, ¿qué aspecto tiene una vista, y cómo la encontramos?

Si usted echa otro vistazo a la salida del Listado 3.4, puede que advine la relación entre acciones y vistas: a una acción como `home` le corresponde una vista llamada `home.html.erb`. Aprenderemos en la Sección 3.4 lo que significa el sufijo `.erb`; de la parte `.html` probablemente usted no se sorprenda de que parezca básicamente HTML (Listado 3.7).

Listado 3.7: La vista generada para la página Home.

```
app/views/static_pages/home.html.erb
```

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

La vista para la acción **help** es análoga (Listado 3.8).

Listado 3.8: La vista generada para la página Help.

```
app/views/static_pages/help.html.erb
```

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

Ambas vistas son únicamente marcadores de posición: tienen un encabezado de nivel 1 (dentro de la etiqueta **h1**) y un párrafo (etiqueta **p**) con la ruta completa al archivo respectivo.

3.2.2 Páginas estáticas personalizadas

Agregaremos algo (muy poco) de contenido dinámico a partir de la Sección 3.4, pero tal y como están las vistas de los Listados 3.7 y 3.8 resaltan un punto importante: las vistas Rails pueden contener únicamente HTML estático. Esto significa que podemos empezar a personalizar las páginas Home y Help aún sin tener conocimiento de Rails, como se muestra en los Listados 3.9 y 3.10.

Listado 3.9: HTML personalizado para la página Home.

```
app/views/static_pages/home.html.erb
```

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listado 3.10: HTML personalizado para la página Help.

```
app/views/static_pages/help.html.erb
```

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.
```

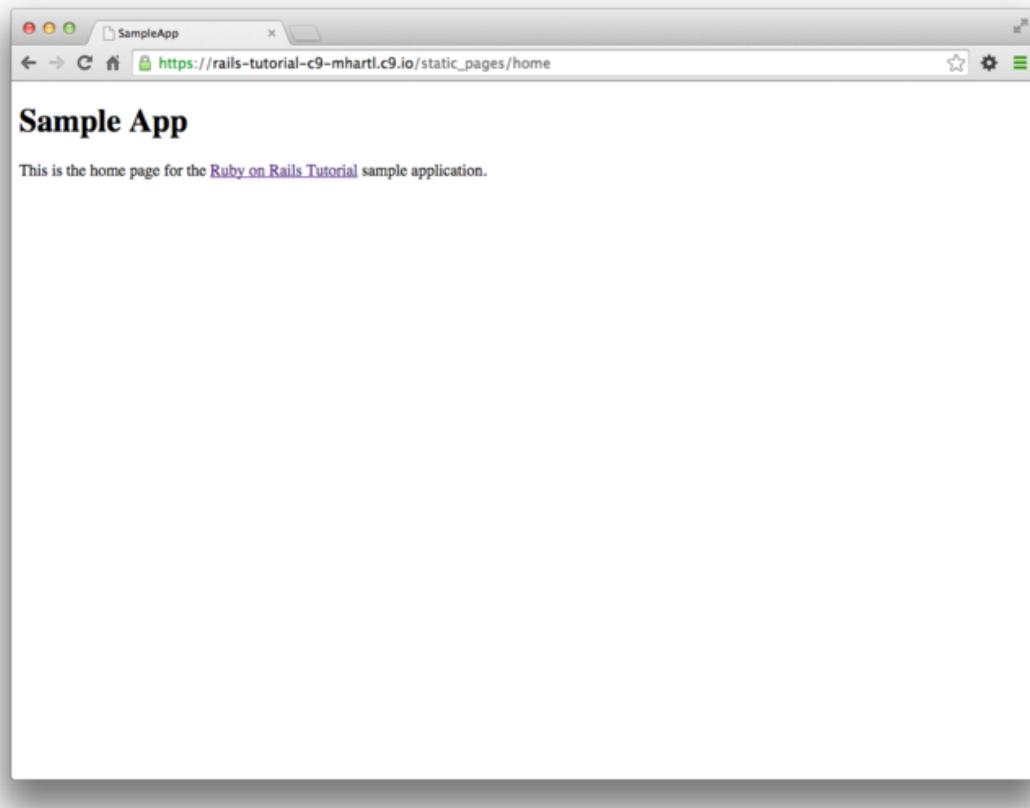


Figura 3.3: Una página Home personalizada.

```
To get help on this sample app, see the
<a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
book</a>.
</p>
```

Los resultados de los Listados 3.9 y 3.10 se muestran en las Figuras 3.3 y 3.4.

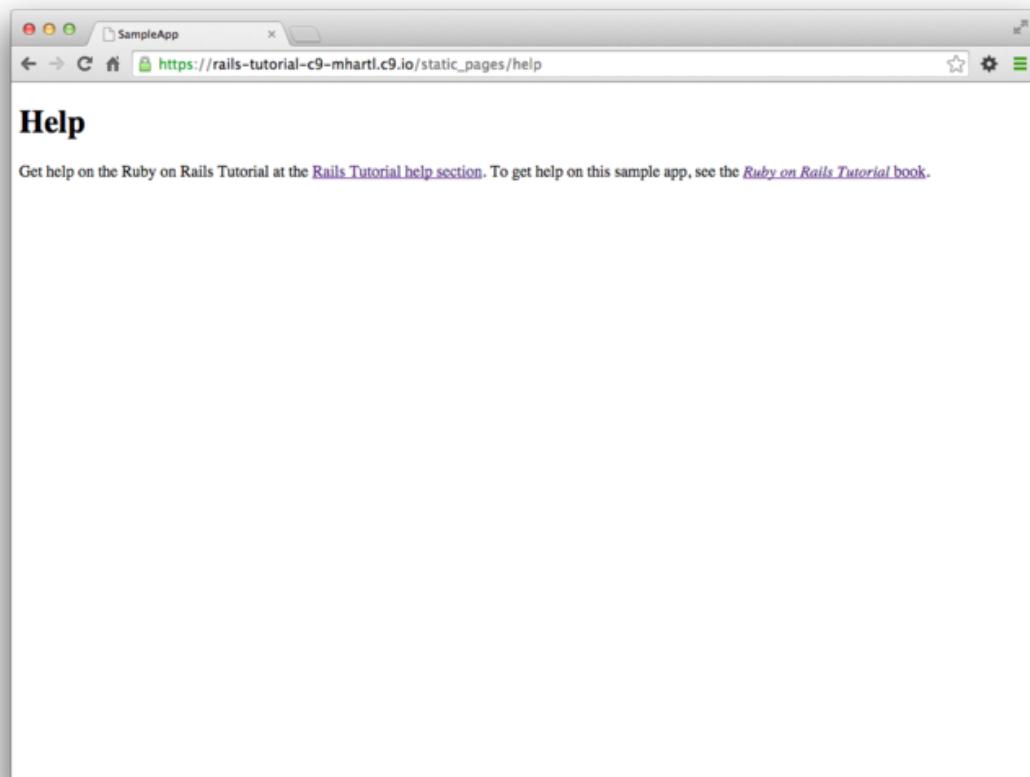


Figura 3.4: Una página Help personalizada.

3.3 Empezando las pruebas

Habiendo creado y agregado contenido a las páginas Home y Help de nuestra aplicación de ejemplo ([Sección 3.2.2](#)), ahora vamos a agregar una página About. Cuando hacemos un cambio de esta naturaleza, es una buena práctica escribir una *prueba automatizada* para verificar que la funcionalidad sea implementada correctamente. Desarrollado durante la construcción una aplicación, el *conjunto de pruebas* resultante sirve como una malla de seguridad y como una documentación ejecutable del código fuente de la aplicación. Cuando se hace correctamente, escribir las pruebas también nos permite desarrollar *más rápidamente* a pesar de requerir código extra, porque al final desperdiciamos menos tiempo tratando de rastrear defectos. Esto es cierto sólo hasta que somos buenos escribiendo pruebas, por lo que ésta es una de las razones por las que es importante empezar a practicar lo antes posible.

Aunque virtualmente todos los desarrolladores Rails están de acuerdo en que probar es una buena idea, existe una diversidad de opiniones en los detalles. Existe un debate especialmente animado sobre el uso del desarrollo orientado a pruebas (TDD, por sus siglas en inglés *Test-Driven Development*),⁶ una técnica de pruebas en la que el programador escribe las pruebas de error primero, y luego escribe el código de la aplicación para pasar esas pruebas. El *Tutorial de Ruby on Rails* utiliza un enfoque ligero e intuitivo para las pruebas, empleando TDD cuando es conveniente sin ser dogmático respecto a él ([Recuadro 3.3](#)).

Recuadro 3.3. Cuándo probar

Al decidir cuándo y cómo probar, es útil entender *porqué* probar. En mi opinión, escribir pruebas automatizadas tiene tres beneficios importantes:

1. Las pruebas nos protegen en contra de las *regresiones*, en las que una funcionalidad deja de trabajar correctamente por alguna razón.

⁶Vea por ejemplo, “[TDD está muerto. Larga vida a las pruebas.](#)” por el creador de Rails, David Heinemeier Hansson.

2. Las pruebas permiten que el código sea *refactorizado* (es decir, cambiar su forma sin cambiar su funcionalidad) con mayor confianza.
3. Las pruebas actúan como un *cliente* del código de la aplicación, ayudando así a determinar su diseño y su interfaz con otras partes del sistema.

Aunque ninguno de los beneficios anteriores *requiere* que las pruebas sean escritas primero, existen muchas circunstancias donde el desarrollo orientado a pruebas (TDD) es una herramienta con la que vale la pena contar. Decidir cuándo y cómo probar depende en parte de qué tan confortable se sienta usted escribiendo pruebas; muchos desarrolladores encuentran que, conforme mejoran sus habilidades escribiendo pruebas, están más inclinados a escribirlas primero. También depende de qué tan difícil es la prueba con respecto al código de la aplicación, qué tan precisamente se conocen las funcionalidades deseadas, y qué tan probable es que la funcionalidad se descomponga en el futuro.

En este contexto, es útil atender a las recomendaciones que nos indiquen cuándo debemos probar primero (o probar en absoluto). Aquí hay algunas sugerencias basadas en mi propia experiencia:

- Cuando una prueba es especialmente corta o simple comparada con el código de la aplicación que prueba, inclínese a escribir la prueba primero.
- Cuando el comportamiento deseado no está del todo claro, inclínese a escribir el código de la aplicación primero y luego escriba una prueba, para probar el resultado.
- Como la seguridad es de prioridad máxima, escriba primero las pruebas del modelo de seguridad.
- Cada vez que encuentre un defecto, escriba una prueba que lo reproduzca y lo proteja de las regresiones, luego escriba el código de la aplicación que lo resuelve.
- Evite escribir pruebas para código que es muy probable que cambie en el futuro (como una estructura HTML detallada).

- Escriba pruebas antes de refactorizar un código, enfocándose en probar código propenso a errores que es especialmente probable que se descomponga.

En la práctica, las recomendaciones anteriores significan que usualmente escribiremos las pruebas del controlador y del modelo primero y las pruebas de integración (las que prueban la funcionalidad entre los modelos, las vistas y los controladores) después. Y cuando estamos escribiendo el código de la aplicación que no es particularmente frágil o propenso a errores, o que es muy probable que cambie (como sucede muy a menudo con las vistas), usualmente omitimos las pruebas en conjunto.

Nuestras herramientas serán las *pruebas de los controladores* (empezando en esta sección), las *pruebas de los modelos* (a partir del Capítulo 6), y las *pruebas de integración* (a partir del Capítulo 7). Las pruebas de integración son especialmente poderosas, puesto que nos permiten simular acciones del usuario interactuando con nuestra aplicación mediante el navegador web. Las pruebas de integración serán finalmente nuestra técnica de pruebas primaria, pero las pruebas de los controladores nos proporcionan un punto de partida para empezar fácilmente.

3.3.1 Nuestra primera prueba

Ahora agregaremos la página About a nuestra aplicación. Como veremos más adelante, la prueba es pequeña y sencilla, por lo que seguiremos las recomendaciones del Recuadro 3.3 y escribiremos la prueba primero. Luego usaremos la prueba de error para dirigir la escritura del código de la aplicación.

Empezar con las pruebas puede ser desafiante, y requiere un conocimiento extenso tanto de Rails como de Ruby. En este punto, escribir pruebas puede parecernos un poco intimidante. Por suerte, Rails ha hecho la parte más difícil por nosotros, porque el comando `rails generate controller` (Listado 3.4) automáticamente generó un archivo de pruebas para que podamos

empezar:

```
$ ls test/controllers/  
static_pages_controller_test.rb
```

Echémosle un vistazo (Listado 3.11).

Listado 3.11: Las pruebas por default del controlador StaticPages. VERDE
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'  
  
class StaticPagesControllerTest < ActionController::TestCase  
  
  test "should get home" do  
    get :home  
    assert_response :success  
  end  
  
  test "should get help" do  
    get :help  
    assert_response :success  
  end  
end
```

En este punto no es importante entender la sintaxis del Listado 3.11 en detalle, pero podemos ver que existen dos pruebas, una para cada acción del controlador que incluimos en la línea de comandos del Listado 3.4. Cada prueba simplemente invoca una acción y verifica (vía una *afirmación*) que el resultado es exitoso. Aquí el uso de **get** indica que nuestra prueba espera que las páginas Home y Help sean páginas web ordinarias, que pueden accesarse mediante una petición GET (Recuadro 3.2). La respuesta **:success** es una representación abstracta del **código de status** HTTP respectivo (en este caso, **200 OK**). en otras palabras, una prueba como

```
test "should get home" do  
  get :home  
  assert_response :success  
end
```

dice “Probemos la página Home emitiendo una petición GET a la acción **home** y luego nos aseguramos de que recibimos un código de status ‘success’ en la respuesta.”

Para iniciar nuestro ciclo de pruebas, necesitamos ejecutar nuestro conjunto de pruebas para verificar que las pruebas pasan actualmente. Podemos hacer esto con la utilería **rake** (Recuadro 2.1) de la siguiente forma:⁷

Listado 3.12: VERDE

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

Como es requerido, inicialmente nuestro conjunto de pruebas pasa (VERDE). (Usted no verá el color verde a menos que agregue los *reporteadores de mini-pruebas* de la Sección 3.7.1.) Por cierto, las pruebas toman un tiempo para empezar a ejecutarse, lo cual se debe a dos factores: (1) iniciar el *servidor Spring* para pre-cargar partes del ambiente de Rails, lo cual sucede únicamente la primera vez; y (2) un tiempo extra asociado a la inicialización de Ruby. (El segundo factor mejora cuando usamos *Guard* como se sugiere en la Sección 3.7.3.)

3.3.2 Rojo

Como se observa en el Recuadro 3.3, el desarrollo orientado a pruebas implica escribir una prueba de fallo primero, escribiendo el código de la aplicación necesario para hacer que pase, y luego refactorizando conforme sea necesario. Como muchas herramientas de pruebas representan las pruebas fallidas con el color rojo y las pruebas exitosas con el color verde, esta secuencia es conocida en algunas ocasiones como el ciclo “Rojo, Verde, Refactorización”. En esta sección, completaremos el primer paso de este ciclo, al obtener **ROJO** escribiendo

⁷Como vimos en la Sección 2.2, el uso de **bundle exec** es innecesario en algunos sistemas, incluído el IDE en la nube, recomendado en la Sección 1.2.1, pero lo incluyo por completez. En la práctica, mi algoritmo usual es omitir **bundle exec** a menos de que arroje error, en cuyo caso lo reintento con **bundle exec** y verifico si funciona.

una prueba que falle. Luego obtendremos un VERDE en la Sección 3.3.3, y refactorizaremos en la Sección 3.4.3.⁸

Nuestro primer paso es escribir una prueba de fallo para la página About. Siguiendo los modelos del Listado 3.11, usted puede adivinar probablemente la prueba correcta, la cual se muestra en el Listado 3.13.

Listado 3.13: Una prueba para la página About. ROJO

```
test/controllers/static_pages_controller_test.rb

require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
  end

  test "should get help" do
    get :help
    assert_response :success
  end

  test "should get about" do
    get :about
    assert_response :success
  end
end
```

Vemos en las líneas resaltadas del Listado 3.13 que la prueba para la página About es la misma que las pruebas para las páginas Home y Help excepto por la palabra “about” en lugar de “home” o “help”.

Como es requerido, la prueba inicialmente falla:

Listado 3.14: ROJO

```
$ bundle exec rake test
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
```

⁸Por default, `rake test` muestra rojo cuando las pruebas fallan, pero no muestra verde cuando las pruebas pasan. Para implementar un ciclo Rojo–Verde verdadero, revise la Sección 3.7.1.

3.3.3 Verde

Ahora que tenemos una prueba fallando (**ROJO**), utilizaremos los mensajes de error de la prueba fallida para guiarnos hacia una prueba exitosa (**VERDE**), implementando de este modo, una página About que funciona.

Podemos empezar examinando el mensaje de error que nos muestra la prueba fallida:⁹

Listado 3.15: ROJO

```
$ bundle exec rake test
ActionController::UrlGenerationError:
No route matches {:action=>"about", :controller=>"static_pages"}
```

Este mensaje de error dice que no hay una ruta que coincida con la combinación acción/controlador deseada, lo cual es una pista de que necesitamos agregar una línea al archivo de rutas. Logramos esto siguiendo el patrón del [Listado 3.5](#), que se muestra en el [Listado 3.16](#).

Listado 3.16: Agregando la ruta **about**. ROJO

config/routes.rb

```
Rails.application.routes.draw do
  get 'static_pages/home'
  get 'static_pages/help'
  get 'static_pages/about'
  .
  .
  .
end
```

La línea resaltada del [Listado 3.16](#) le indica a Rails que dirija una petición GET de la URL /static_pages/about a la acción **about** del controlador de Páginas Estáticas.

Si ejecutamos nuestro conjunto de pruebas nuevamente, veremos que sigue **ROJO**, pero nuestro mensaje de error ha cambiado:

⁹En algunos sistemas, es posible que usted necesite desplazarse hasta la línea que dice “stack trace” o “backtrace” que rastrea la trayectoria del error a través del código fuente. Revise la [Sección 3.7.2](#) para obtener información sobre cómo filtrar esta traza y eliminar líneas indeseadas.

Listado 3.17: ROJO

```
$ bundle exec rake test
AbstractController::ActionNotFound:
The action 'about' could not be found for StaticPagesController
```

El mensaje de error ahora indica que falta una acción **about** en el controlador de Páginas Estáticas, la cual podemos agregar siguiendo el modelo proporcionado por **home** y **help** del Listado 3.6, que se muestra en el Listado 3.18.

Listado 3.18: El controlador de Páginas Estáticas con la acción **about** agregada. ROJO

app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Como antes, nuestro conjunto de pruebas sigue ROJO, pero el mensaje de error ha cambiado nuevamente:

```
$ bundle exec rake test
ActionView::MissingTemplate: Missing template static_pages/about
```

Ahora nos indica una plantilla faltante, que en el contexto de Rails es esencialmente lo mismo que una vista. Como se describe en la Sección 3.2.1, una acción llamada **home** es asociada con una vista llamada **home.html.erb** ubicada en el directorio **app/views/static_pages**, lo que significa que necesitamos crear un nuevo archivo llamado **about.html.erb** en el directorio mencionado.

La forma de crear un archivo varía de sistema a sistema, pero la mayoría de los sistemas le permitirán invocar un menú contextual dentro del directorio donde quiere crear el archivo y seleccionar la opción “Archivo Nuevo”. También puede usar el menú Archivo del editor de texto para crear un archivo nuevo y luego elegir el directorio donde desea guardarlo. Por último, puede emplear mi truco favorito utilizando el [comando Unix touch](#) como sigue:

```
$ touch app/views/static_pages/about.html.erb
```

Aunque **touch** fue diseñado para actualizar la fecha de modificación de un archivo o de un directorio sin realizar modificación alguna sobre éste, también sirve para crear archivos nuevos vacíos si el nombre que se le proporciona no existe. (Si usted está utilizando el IDE en la nube, puede ser que necesite refrescar el árbol de archivos como se indica en la [Sección 1.3.1](#).)

Una vez creado el archivo **about.html.erb** en el directorio correcto, debería editararlo y agregarle el contenido que se muestra en el [Listado 3.19](#).

Listado 3.19: Código para la página About. [VERDE](#)

```
app/views/static_pages/about.html.erb
```

```
<h1>About</h1>
<p>
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a> is a
  <a href="http://www.railstutorial.org/book">book</a> and
  <a href="http://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample application for the tutorial.
</p>
```

En este punto, si ejecutamos **rake test** deberíamos regresar a [VERDE](#):

Listado 3.20: [VERDE](#)

```
$ bundle exec rake test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

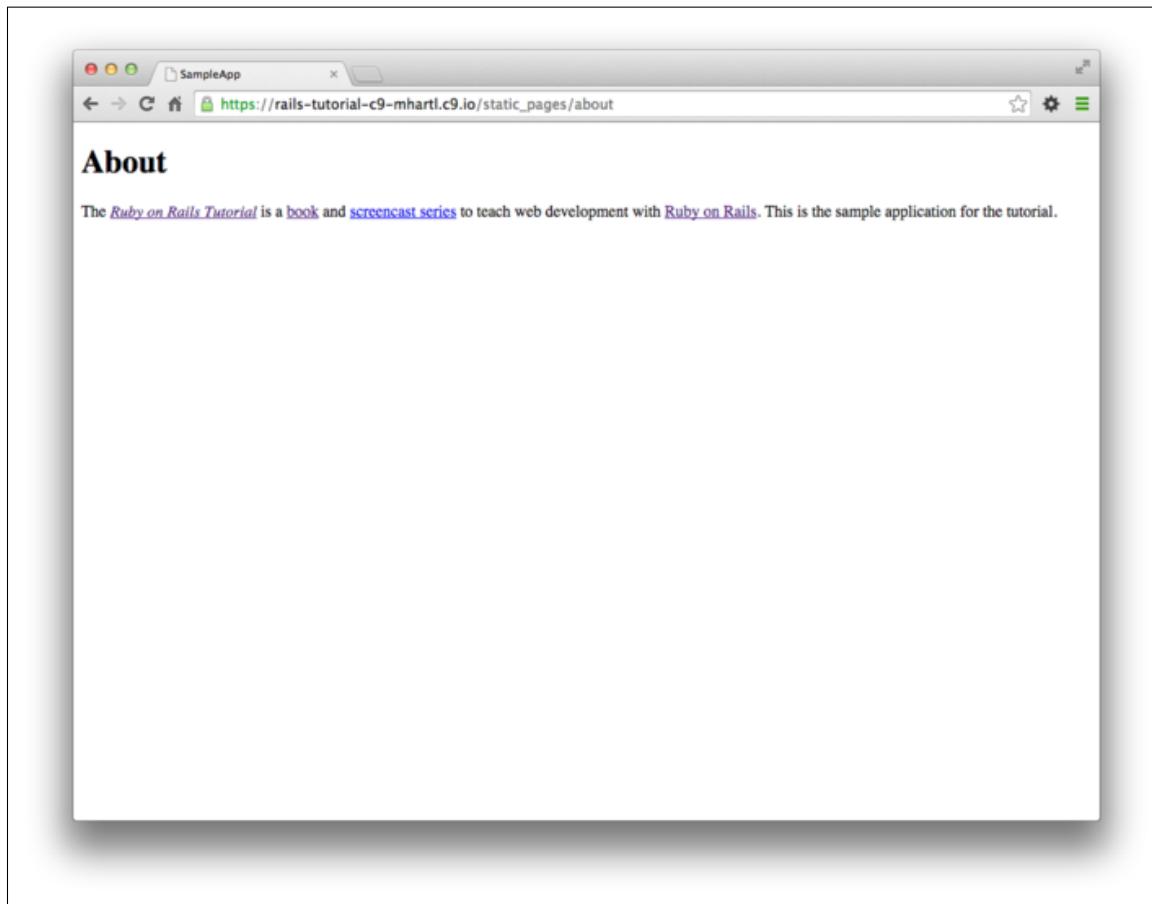


Figura 3.5: La nueva página About (/static_pages/about).

Por supuesto, nunca es mala idea echar un vistazo a la página desde un navegador para asegurarnos de que nuestras pruebas no están completamente locas (Figura 3.5).

3.3.4 Refactor

Ahora que logramos obtener el **VERDE**, podemos refactorizar nuestro código con confianza. Cuando desarrollamos una aplicación, a menudo el código empezará a “apestar”, lo que significa que se está volviendo feo, abultado, o es repetitivo. A la computadora no le importa cómo se ve el código, por supuesto, pero a los

humanos sí, por lo que es importante mantener el código limpio refactorizando frecuentemente. Aunque nuestra aplicación de ejemplo es muy pequeña para refactorizar ahora, el [código apesado](#) se filtra por cada grieta, y empezaremos a refactorizar a partir de la [Sección 3.4.3](#).

3.4 Páginas ligeramente dinámicas

Ahora que ya creamos las acciones y las vistas para algunas páginas estáticas, las haremos *ligeramente* dinámicas al agregarles un poco de contenido que cambia en cada página: haremos que el título de cada página cambie para reflejar su contenido. Es debatible si el cambio del título representa o no contenido *verdaderamente* dinámico, pero de cualquier forma sienta las bases necesarias para contenido evidentemente dinámico en el [Capítulo 7](#).

Nuestro plan es editar las páginas Home, Help y About para cambiar los títulos de cada una de ellas. Esto requerirá el uso de la etiqueta `<title>` en las vistas. La mayoría de los navegadores muestran el contenido de esta etiqueta en la parte superior de la ventana, y también es importante para la optimización de los motores de búsqueda. Estaremos empleando el ciclo completo “Rojo, Verde, Refactorización”: primero agregaremos pruebas simples para los títulos de nuestras páginas ([Rojo](#)), luego agregaremos los títulos a las mismas ([Verde](#)), y finalmente utilizaremos un archivo con la *estructura* para eliminar duplicidad de código (Refactorización). Al terminar esta sección, nuestras tres páginas estáticas tendrán títulos con el formato “<nombre de la página> | Ruby on Rails Tutorial Sample App”, donde la primera parte del título variará dependiendo de la página ([Tabla 3.2](#)).

El comando `rails new` ([Listado 3.1](#)) crea un archivo con una estructura por default, pero inicialmente lo ignoraremos, por lo que temporalmente le cambiaremos de nombre:

```
$ mv app/views/layouts/application.html.erb layout_file # temporary change
```

Normalmente no realizamos este paso en una aplicación real, pero hacerlo ahora nos ayudará a entender su propósito.

Página	URL	Título base	Título variable
Home	/static_pages/home	"Ruby on Rails Tutorial Sample App"	"Home"
Help	/static_pages/help	"Ruby on Rails Tutorial Sample App"	"Help"
About	/static_pages/about	"Ruby on Rails Tutorial Sample App"	"About"

Tabla 3.2: Las páginas (casi) estáticas de la aplicación de ejemplo.

3.4.1 Probando los títulos (Rojo)

Para agregar los títulos de las páginas, necesitamos aprender (o recordar) la estructura de una página web típica, cuyo formato se muestra en el [Listado 3.21](#).

Listado 3.21: La estructura HTML de una página web típica.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

La estructura del [Listado 3.21](#) incluye un tipo de documento *doctype*, que es una declaración al inicio para indicar a los navegadores cuál versión de HTML estamos usando (en este caso, [HTML5](#));¹⁰ una sección **head**, en este caso con “Greeting” dentro de la etiqueta **title**; y una sección **body**, en este caso con “Hello, world!” dentro de una etiqueta **p** (párrafo). (La indentación es opcional—HTML no es sensible a espacios en blanco, por lo cual ignora tanto los espacios como los tabuladores—pero indentar facilita la visualización de la estructura del documento.)

Escribiremos pruebas simples para cada uno de los títulos de la [Tabla 3.2](#) combinando las pruebas del [Listado 3.13](#) con el método **assert_select**, que

¹⁰HTML cambia con el tiempo; al indicar explícitamente el tipo de documento hacemos que sea más probable que los navegadores desplieguen correctamente nuestras páginas en el futuro. La declaración `<!DOCTYPE html>` es característica de la última versión del estándar HTML, HTML5.

nos permite probar la presencia de una etiqueta HTML en particular (algunas veces llamado “selector”, de ahí el nombre):¹¹

```
assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
```

En particular, el código anterior verifica la presencia de la etiqueta `<title>` y que contenga el texto “Home | Ruby on Rails Tutorial Sample App”. Aplicando esta idea a las tres páginas estáticas, obtenemos las pruebas que se muestran en el Listado 3.22.

Listado 3.22: La pruebas del controlador de Páginas Estáticas con pruebas para los títulos. ROJO

```
test/controllers/static_pages_controller_test.rb

require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

(Si la repetición del título base “Ruby on Rails Tutorial Sample App” le incomoda, revise los ejercicios de la Sección 3.6.)

¹¹Para ver una lista de aserciones para minipruebas, consulte la [tabla de aserciones disponibles en la Guía de Pruebas de Rails](#).

Con las pruebas del Listado 3.22 en su lugar, verifique que el conjunto de pruebas está actualmente en ROJO:

Listado 3.23: ROJO

```
$ bundle exec rake test  
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

3.4.2 Agregando títulos a las páginas (Verde)

Ahora agregaremos los títulos a cada página, para que las pruebas de la Sección 3.4.1 pasen exitosamente. Aplicando la estructura HTML básica del Listado 3.21 para personalizar la página Home del Listado 3.9 obtenemos el Listado 3.24.

Listado 3.24: La vista de la página Home con la estructura HTML completa.**ROJO**

app/views/static_pages/home.html.erb

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Home | Ruby on Rails Tutorial Sample App</title>  
  </head>  
  <body>  
    <h1>Sample App</h1>  
    <p>  
      This is the home page for the  
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>  
      sample application.  
    </p>  
  </body>  
</html>
```

La página web correspondiente aparece en la Figura 3.6.¹²

Siguiendo este modelo para las páginas Help (Listado 3.10) y About (Listado 3.19) obtenemos el código de los Listados 3.25 y 3.26.

¹²La mayoría de las capturas de pantalla del libro usan Google Chrome, pero la de la Figura 3.6 usa Safari porque Chrome no muestra el título completo de la página.

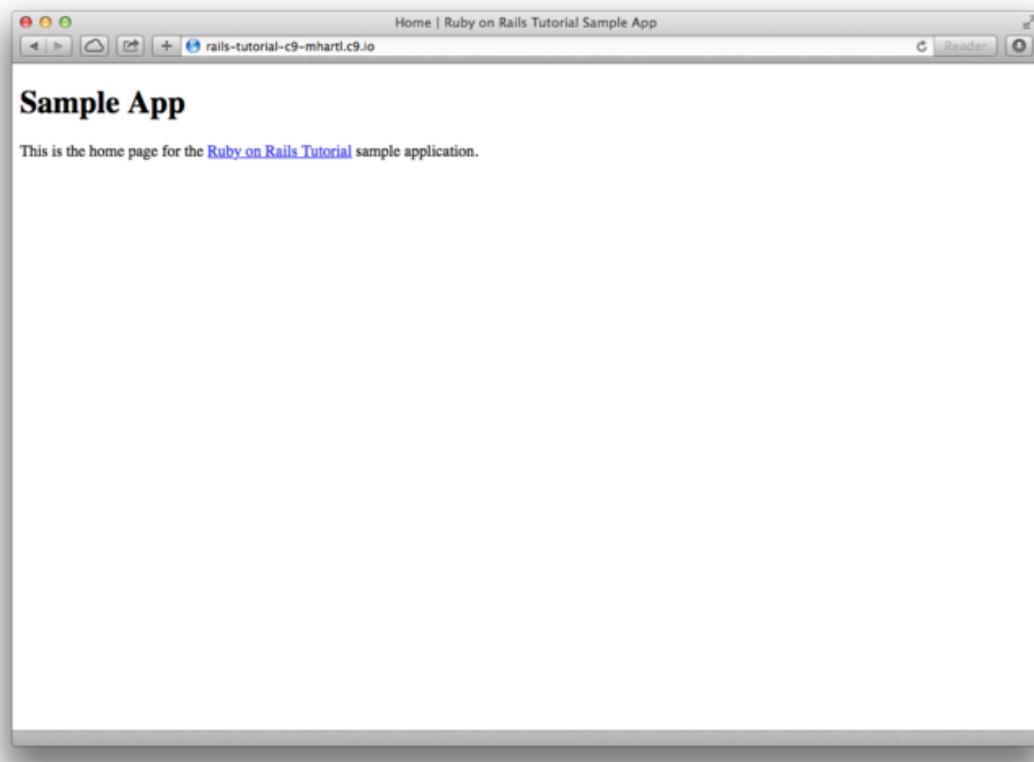


Figura 3.6: La página Home con un título.

Listado 3.25: La vista de la página Help con la estructura HTML completa.

ROJO

app/views/static_pages/help.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Help | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://www.railstutorial.org/#help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

Listado 3.26: La vista de la página About con la estructura HTML completa.

VERDE

app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>About | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>
```

En este punto, el conjunto de pruebas debería regresar a VERDE:

Listado 3.27: VERDE

```
$ bundle exec rake test  
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

3.4.3 Estructuras de diseño y Ruby embebido (Refactorización)

Hemos avanzado mucho en esta sección, generando tres páginas válidas usando controladores y acciones de Rails, pero son páginas HTML puramente estáticas que no reflejan en absoluto el poder de Rails. Más aún, padecen de duplicidad de código:

- Los títulos de las páginas son casi exactamente los mismos.
- “Ruby on Rails Tutorial Sample App” es común a los tres títulos.
- La estructura HTML completa se repite en cada página.

Esta repetición de código es una violación a un principio importante “No se repita usted mismo” (DRY, por sus siglas en inglés *Don’t Repeat Yourself*); en esta sección eliminaremos las repeticiones. Al final, volveremos a ejecutar las pruebas de la [Sección 3.4.2](#) para verificar que los títulos siguen siendo correctos.

Paradójicamente, nuestro primer paso hacia la eliminación del código duplicado es agregar más: haremos que los títulos de las páginas, que en este momento son bastante similares, sean *idénticos*. Esto hará posible que eliminemos la repetición de un solo golpe.

La técnica involucra el uso de *Ruby embebido* en nuestras vistas. Puesto que los títulos de las páginas Home, Help y About tienen una parte variable, emplearemos una función de Rails especial, llamada **provide** para indicar un título diferente a cada página. Podemos ver cómo funciona esto al reemplazar la palabra “Home” en la vista **home.html.erb** como se muestra en el [Listado 3.28](#).

Listado 3.28: La vista de la página Home con un título que usa Ruby embedido. VERDE

app/views/static_pages/home.html.erb

```
<% provide(:title, "Home") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

El [Listado 3.28](#) es nuestro primer ejemplo de Ruby embedido, también llamado *ERb*, por su nombre en inglés *Embedded Ruby*. (Ahora ya sabe porqué los archivos de las vistas HTML tienen la extensión **.html.erb**.) ERb es el sistema de plantillas primario para incluir contenido dinámico en páginas web.¹³ El código

```
<% provide(:title, "Home") %>
```

indica mediante `<% ... %>` que Rails debe invocar la función **provide** y asociar el texto `"Home"` con la etiqueta **:title**.¹⁴ Entonces, en el título, utilizamos la muy similar notación `<%= ... %>` para insertar el título en la estructura usando la función de Ruby **yield**.¹⁵

¹³Existe otro popular sistema de plantillas llamado **Haml** (observe que no es “HAML”), el cual personalmente me encanta, pero no está *suficientemente* estandarizado como para utilizarlo en un tutorial introductorio.

¹⁴Los desarrolladores Rails más experimentados podrían haber esperado que usara **content_for** en este punto, pero no funciona bien con nuestra cadena de procesos conectados. La función **provide** es su reemplazo.

¹⁵Si usted ha estudiado Ruby anteriormente, puede que sospeche que Rails está *generando* los contenidos en un bloque, y su sospecha es correcta. Pero no necesita saber esto para desarrollar aplicaciones con Rails.

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

(La diferencia entre los dos tipos de Ruby embebido es que `<% ... %>` ejecuta el código adentro, mientras que `<%= ... %>` lo ejecuta e inserta el resultado en la estructura.) La página resultante es exactamente la misma que antes, sólo que ahora la parte variable del título es generada dinámicamente con ERb.

Podemos verificar que todo esto funciona, ejecutando las pruebas de la Sección 3.4.2 y revisando que aún están en VERDE:

Listado 3.29: VERDE

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

Luego podemos hacer los reemplazos correspondientes para las páginas Help y About (Listados 3.30 y 3.31).

Listado 3.30: La vista de la página Help con un título que usa Ruby embebido.

VERDE

app/views/static_pages/help.html.erb

```
<% provide(:title, "Help") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://www.railstutorial.org/#help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

Listado 3.31: La vista de la página About con un título que usa Ruby embedido. VERDE

app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>
```

Ahora que ya reemplazamos la parte variable de los títulos de las páginas usando ERb, cada una de nuestras páginas se ven como:

```
<% provide(:title, "The Title") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    Contents
  </body>
</html>
```

En otras palabras, todas las páginas son idénticas en estructura, incluyendo los contenidos de la etiqueta **title**, con la única excepción del contenido de la etiqueta **body**.

Con el fin de extraer esta estructura en común, Rails viene con un archivo de *estructura de diseño* especial llamado **application.html.erb**, el cual

renombramos al inicio de esta sección ([Sección 3.4](#)) y al que ahora devolveremos su nombre original:

```
$ mv layout_file app/views/layouts/application.html.erb
```

Para que la estructura de diseño funcione, debemos reemplazar la siguiente línea de código que corresponde al título que usa Ruby embedido, de los ejemplos anteriores:

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

La estructura de diseño resultante se muestra en el [Listado 3.32](#).

Listado 3.32: La estructura de diseño de la aplicación de ejemplo. [VERDE](#)
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Observe aquí la línea especial

```
<%= yield %>
```

Este código es responsable de insertar el contenido de cada página en la estructura de diseño. No es relevante saber cómo lo hace exactamente; lo que importa es que utilizar esta estructura de diseño nos asegura, por ejemplo, que cuando

visitamos la página /static_pages/home convierte el contenido de `home.html.erb` a HTML y luego lo inserta en lugar de `<%= yield %>`.

Vale la pena notar que la estructura de diseño por default de Rails incluye varias líneas adicionales:

```
<%= stylesheet_link_tag ... %>
<%= javascript_include_tag "application", ... %>
<%= csrf_meta_tags %>
```

Este código se encarga de incluir la hoja de estilo de la aplicación y JavaScript, que forman parte de nuestra cadena de procesos conectados (Sección 5.2.1), junto con el método Rails `csrf_meta_tags`, el cual previene de [peticiones falsas desde otro sitio web](#) (CSRF, por sus siglas en inglés *Cross-Site Request Forgery*), un tipo de ataque web malicioso.

Por supuesto, las vistas de los Listados 3.28, 3.30 y 3.31 están aún llenas de estructura HTML que ya está incluida en la estructura de diseño, por lo que debemos removerlas, dejando únicamente el contenido interno. Las vistas resultantes, más limpias, aparecen en los Listados 3.33, 3.34 y 3.35.

Listado 3.33: La página Home con la estructura HTML removida. [VERDE](#)
`app/views/static_pages/home.html.erb`

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listado 3.34: La página Help con la estructura HTML removida. [VERDE](#)
`app/views/static_pages/help.html.erb`

```
<% provide(:title, "Help") %>
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
```

```
<a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.
To get help on this sample app, see the
<a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
book</a>.
</p>
```

Listado 3.35: La página About con la estructura HTML removida. **VERDE**
`app/views/static_pages/about.html.erb`

```
<% provide(:title, "About") %>
<h1>About</h1>
<p>
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a> is a
  <a href="http://www.railstutorial.org/book">book</a> and
  <a href="http://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample application for the tutorial.
</p>
```

Con estas vistas definidas, las páginas Home, Help y About son las mismas que antes, pero con mucho menos código duplicado.

La experiencia muestra que aún la más simple refactorización tiende a errores y puede fácilmente salir mal. Esta es una de las razones por las cuales tener un buen conjunto de pruebas es tan valioso. En vez de verificar nuevamente cada página para ver si está correcta—un procedimiento que no es muy difícil al principio pero que rápidamente se convierte en difícil de manejar conforme la aplicación crece—podemos verificar de forma sencilla que el conjunto de pruebas aún está **VERDE**:

Listado 3.36: **VERDE**

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

Esto no es una *prueba* de que nuestro código está correcto, pero incrementa enormemente la probabilidad, proporcionando de ese modo una malla de seguridad para protegernos de futuros errores.

3.4.4 Configurando la ruta raíz

Ahora que hemos personalizado las páginas y hemos comenzado a utilizar el conjunto de pruebas, configuremos la ruta raíz de la aplicación antes de continuar. Igual que en las Secciones 1.3.4 y 2.2.2, deberemos editar el archivo **routes.rb** para relacionar / con una página de nuestra elección, que en nuestro caso será la página Home. (En este punto, recomiendo que también eliminemos la acción **hello** del controlador de la aplicación, si es que usted lo agregó en la Sección 3.1.) Como vemos en el Listado 3.37, esto significa que reemplazaremos la regla **get** generada en el Listado 3.5 con el código siguiente:

```
root 'static_pages#home'
```

Esto cambia la URL **static_pages/home** a la pareja controlador/acción **static_pages#home**, que nos asegura que las peticiones GET para / sean dirigidas a la acción **home** del controlador de Páginas Estáticas. El archivo de rutas resultante se muestra en la Figura 3.7. (Observe que, con el código del Listado 3.37, la ruta anterior **static_pages/home** ya no funcionará.)

Listado 3.37: Relacionando la ruta raíz con la página Home.

```
config/routes.rb
```

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'static_pages/help'
  get 'static_pages/about'
end
```

3.5 Conclusión

Visto desde afuera, este capítulo difícilmente logró algo: empezamos con páginas estáticas, y terminamos con páginas...*casi* estáticas. Pero las apariencias engañan: al desarrollar en términos de los controladores, las acciones y las vistas de Rails, estamos ahora en posición de agregar una cantidad arbitraria de

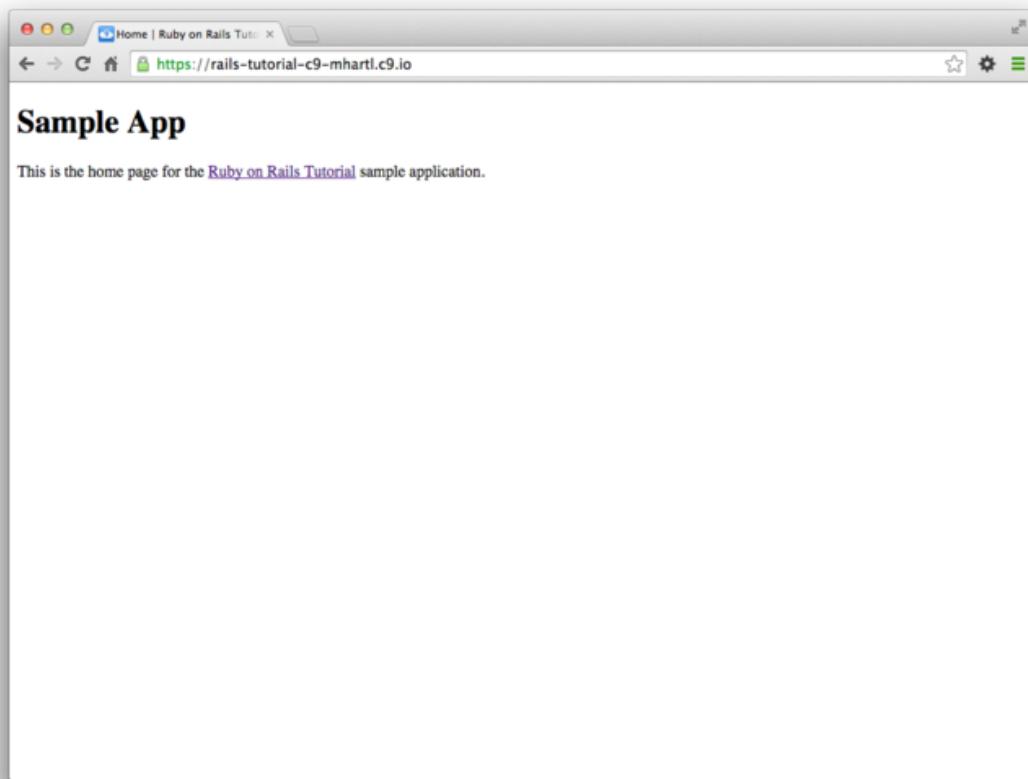


Figura 3.7: La página Home en la ruta raíz.

contenido dinámico a nuestro sitio. Ver exactamente cómo funciona esto, es la tarea que realizaremos el resto del tutorial.

Antes de continuar, tomemos un momento para subir los cambios de nuestra rama y mezclémoslos con la rama principal. Recuerde que en la [Sección 3.2](#) creamos una rama en Git para el desarrollo de páginas estáticas. Si no ha subido ningún cambio conforme hemos avanzado, primero haga un **commit** indicando que hemos llegado a este punto:

```
$ git add -A  
$ git commit -m "Finish static pages"
```

Luego mezcle los cambios en la rama principal, usando la misma técnica que en la [Sección 1.4.4](#).¹⁶

```
$ git checkout master  
$ git merge static-pages
```

Cada vez que haya alcanzado un punto como éste, en el que ha terminado una tarea, usualmente se recomienda subir su código al repositorio remoto (el cual, si usted ha seguido los pasos indicados en la [Sección 1.4.3](#), será Bitbucket):

```
$ git push
```

También le recomiendo que despliegue la aplicación en Heroku:

```
$ bundle exec rake test  
$ git push heroku
```

Hasta aquí nos hemos encargado de ejecutar el conjunto de pruebas antes de desplegar, lo cual es un buen hábito que debemos crear.

¹⁶Si obtiene un mensaje de error diciendo que el archivo de con el id del proceso Spring (pid) será sobreescrito por la mezcla, sólo borre el archivo usando **rm -f *.pid** en la línea de comandos.

3.5.1 Qué aprendimos en este capítulo

- Por tercera vez, hemos pasado por el proceso completo de crear una aplicación Rails desde cero, instalando las gemas necesarias, subiendo nuestro código a un repositorio remoto, y desplegando en producción.
- El script `rails` genera un nuevo controlador con `rails generate controller ControllerName <nombres de acciones opcionales>`.
- Definimos nuevas rutas en el archivo `config/routes.rb`.
- Las vistas Rails pueden contener HTML estático o Ruby embebido (ERb).
- Las pruebas automatizadas nos permiten escribir conjuntos de pruebas que dirigen el desarrollo de nuevas funcionalidades, permitiéndonos refactorizar con confianza y capturar regresiones.
- El desarrollo orientado a pruebas utiliza el ciclo “Rojo, Verde, Refactorización”.
- Las estructuras de diseño de Rails nos permiten utilizar una plantilla común para las páginas de nuestra aplicación, eliminando así la duplicidad de código.

3.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Desde este punto y hasta el final del tutorial, le recomiendo que resuelva los ejercicios en una rama de Git separada:

```
$ git checkout static-pages
$ git checkout -b static-pages-exercises
```

Esta práctica evitara conflictos con el tutorial principal.

Una vez que usted esté satisfecho con sus soluciones, puede subir la rama de ejercicios al repositorio remoto (si es que configuró uno):

```
<solve first exercise>
$ git commit -am "Eliminate repetition (solves exercise 3.1)"
<solve second exercise>
$ git add -A
$ git commit -m "Add a Contact page (solves exercise 3.2)"
$ git push -u origin static-pages-exercises
$ git checkout master
```

(Como preparación para futuros desarrollos, el último paso aquí se posiciona en la rama principal, pero *no* mezclamos estos cambios con la finalidad de evitar conflictos con el resto del tutorial.) En los próximos capítulos, las ramas y los mensajes de comentario serán diferentes, por supuesto, pero la idea básica es la misma.

1. Puede ser que haya notado alguna repetición en las pruebas del controlador de Páginas Estáticas ([Listado 3.22](#)). En particular, el título base, “Ruby on Rails Tutorial Sample App”, que es el mismo para cada prueba de título. Usando la función especial **setup**, que se ejecuta automáticamente antes de cada prueba, verifique que las pruebas del [Listado 3.38](#) siguen en **VERDE**. ([Listado 3.38](#) utiliza una *variable de instancia*, que revisamos brevemente en la [Sección 2.2.2](#) y que veremos más adelante en la [Sección 4.4.5](#), combinado con la *interpolación de texto*, que veremos en la [Sección 4.2.2](#).)
2. Elabore una página de Contacto para la aplicación de ejemplo.¹⁷ Siguiendo la receta del [Listado 3.13](#), primero escriba una prueba para la existencia de una página en la URL `/static_pages/contact` que verifique el título “Contact | Ruby on Rails Tutorial Sample App”. Haga que su prueba pase siguiendo los mismos pasos que cuando se hizo la página About de la [Sección 3.3.3](#), incluyendo el llenado de la página de Contacto con el contenido del [Listado 3.39](#). (Observe que, para mantener

¹⁷Este ejercicio se resuelve en la [Sección 5.3.1](#).

los ejercicios independientes, el [Listado 3.39](#) no incorpora los cambios realizados en el [Listado 3.38](#).)

Listado 3.38: Las pruebas del controlador de Páginas Estáticas con un título base. **VERDE**

```
test/controllers/static_pages_controller_test.rb

require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  def setup
    @base_title = "Ruby on Rails Tutorial Sample App"
  end

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Home | #{@base_title}"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | #{@base_title}"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | #{@base_title}"
  end
end
```

Listado 3.39: Código para una página de Contacto propuesta.

```
app/views/static_pages/contact.html.erb
```

```
<% provide(:title, "Contact") %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="http://www.railstutorial.org/#contact">contact page</a>.
</p>
```

3.7 Configuración avanzada de pruebas

Esta sección opcional describe la configuración de pruebas utilizada en la serie de videos del Tutorial de Ruby on Rails. Hay tres elementos principales: un reporteador mejorado de éxito/error (Sección 3.7.1), una utilería para filtrar la traza producida por las pruebas fallidas (Sección 3.7.2), y un ejecutor de pruebas automatizadas que detecta cambios en los archivos y automáticamente ejecuta las pruebas correspondientes (Sección 3.7.3). El código de esta sección es avanzado y se presenta únicamente por conveniencia; no se espera que usted lo entienda en este momento.

Los cambios realizados en esta sección deberán hacerse en la rama principal:

```
$ git checkout master
```

3.7.1 Reporteadores de mini-pruebas

Para que las pruebas por default de Rails muestren ROJO y VERDE en los momentos adecuados, le recomiendo agregar el código del Listado 3.40 a su archivo auxiliar de pruebas,¹⁸ haciendo uso de esta forma, de la gema `minitest-reporters` incluída en el Listado 3.2.

Listado 3.40: Configurando las pruebas para que muestren ROJO y VERDE.

`test/test_helper.rb`

```
ENV['RAILS_ENV'] ||= 'test'  
require File.expand_path('../config/environment', __FILE__)  
require 'rails/test_help'  
require "minitest/reporters"  
Minitest::Reporters.use!  
  
class ActiveSupport::TestCase
```

¹⁸El código del Listado 3.40 mezcla texto con comilla sencilla y comillas dobles. Esto es porque `rails new` genera texto con comilla sencilla, mientras que la documentación de los reporteadores de mini-pruebas usan texto con comillas dobles. Esta mezcla de los dos tipos de comillas es común en Ruby; vea la Sección 4.2.2 para más información.

```

3/3: [=====] 100% Time: 00:00:00, Time: 00:00:00
Finished in 0.06162s
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
mhartl@rails-tutorial:/home/ubuntu/workspace/sample_app (master) $ rake test
Started

3/3: [=====] 100% Time: 00:00:00, Time: 00:00:00
Finished in 0.04396s
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
mhartl@rails-tutorial:/home/ubuntu/workspace/sample_app (master) $ █

```

Figura 3.8: Pasando de ROJO a VERDE en el IDE en la nube.

```

# Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical
# order.
fixtures :all

# Add more helper methods to be used by all tests here...
end

```

La transición resultante de ROJO a VERDE en el IDE en la nube aparece en la Figura 3.8.

3.7.2 Silenciador de traza

Al encontrar un error en una prueba fallida, el ejecutor de pruebas muestra una “traza de la pila” o una “traza regresiva” que rastrea el curso de la prueba fallida a través de la aplicación. Mientras que esta traza usualmente es muy útil para rastrear el problema, en algunos sistemas (incluido el IDE en la nube) va más allá del código de la aplicación y entra en las gemas de las que depende nuestro código e incluso entra dentro del mismo Rails. La traza resultante es a menudo inconvenientemente larga, especialmente cuando la fuente del problema está usualmente en la aplicación y no en una de sus dependencias.

La solución es filtrar esta traza para eliminar las líneas indeseadas. Esto requiere de la gema `mini_backtrace` incluída en el [Listado 3.2](#), combinada con un *silenciador de traza*. En el IDE en la nube, la mayoría de las líneas indeseadas contienen el texto `rvm` (que hace referencia al Administrador de Versiones de Ruby), por lo que recomiendo utilizar el silenciador que se muestra

en el [Listado 3.41](#) para filtrarlas.

Listado 3.41: Agregando un silenciador de traza para RVM.

```
config/initializers/backtrace_silencers.rb
```

```
# Be sure to restart your server when you modify this file.

# You can add backtrace silencers for libraries that you're using but don't
# wish to see in your backtraces.
Rails.backtrace_cleaner.add_silencer { |line| line =~ /rvm/ }

# You can also remove all the silencers if you're trying to debug a problem
# that might stem from framework code.
# Rails.backtrace_cleaner.remove_silencers!
```

Como se hizo notar en el comentario del [Listado 3.41](#), usted debe reiniciar el servidor web local luego de haber añadido el silenciador.

3.7.3 Pruebas automatizadas con Guard

Una molestia asociada con el uso del comando `rake test` es tener que ir a la línea de comandos y ejecutar las pruebas manualmente. Para evitar este inconveniente, podemos usar *Guard* para automatizar la ejecución de las pruebas. Guard monitorea los cambios en el sistema de archivos, de forma que, por ejemplo, cuando cambiamos el archivo `static_pages_controller_test.rb`, sólo esas pruebas se ejecuten. Mejor aún, podemos configurar Guard de forma que cuando, digamos, el archivo `home.html.erb` sea modificado, el `static_pages_controller_test.rb` automática corra.

El archivo **Gemfile** del [Listado 3.2](#) ya incluye la gema `guard` en nuestra aplicación, por lo que para empezar, sólo necesitamos inicializarla:

```
$ bundle exec guard init
Writing new Guardfile to /home/ubuntu/workspace/sample_app/Guardfile
00:51:32 - INFO - minitest guard added to Guardfile, feel free to edit it
```

Entonces editamos el archivo **Guardfile** resultante de forma que Guard ejecute las pruebas correctas cuando las pruebas de integración y las vistas sean

actualizadas (Listado 3.42). (Dada su longitud y naturaleza avanzada, le recomiendo sólo copiar y pegar el contenido del Listado 3.42.)

Listado 3.42: Un archivo **Guardfile** personalizado.

```
# Defines the matching rules for Guard.
guard :minitest, spring: true, all_on_start: false do
  watch(%r{^test/(.*)/?(.*)_test\.rb$})
  watch('test/test_helper.rb') { 'test' }
  watch('config/routes.rb') { integration_tests }
  watch(%r{^app/models/(.*?)\.rb$}) do |matches|
    "test/models/#{matches[1]}_test.rb"
  end
  watch(%r{^app/controllers/(.*?)_controller\.rb$}) do |matches|
    resource_tests(matches[1])
  end
  watch(%r{^app/views/([^\/*]*)/.*\.\html\.erb$}) do |matches|
    ["test/controllers/#{matches[1]}_controller_test.rb"] +
    integration_tests(matches[1])
  end
  watch(%r{^app/helpers/(.*?)_helper\.rb$}) do |matches|
    integration_tests(matches[1])
  end
  watch('app/views/layouts/application.html.erb') do
    'test/integration/site_layout_test.rb'
  end
  watch('app/helpers/sessions_helper.rb') do
    integration_tests << 'test/helpers/sessions_helper_test.rb'
  end
  watch('app/controllers/sessions_controller.rb') do
    ['test/controllers/sessions_controller_test.rb',
     'test/integration/users_login_test.rb']
  end
  watch('app/controllers/account_activations_controller.rb') do
    'test/integration/users_signup_test.rb'
  end
  watch(%r{app/views/users/*}) do
    resource_tests('users') +
    ['test/integration/microposts_interface_test.rb']
  end
end

# Returns the integration tests corresponding to the given resource.
def integration_tests(resource = :all)
  if resource == :all
    Dir["test/integration/*"]
  else
    Dir["test/integration/#{resource}_*.rb"]
  end
end
```

```
# Returns the controller tests corresponding to the given resource.
def controller_test(resource)
  "test/controllers/#{resource}_controller_test.rb"
end

# Returns all tests for the given resource.
def resource_tests(resource)
  integration_tests(resource) << controller_test(resource)
end
```

Donde la línea

```
guard :minitest, spring: true, all_on_start: false do
```

hace que Guard utilice el servidor Spring proporcionado por Rails para acelerar los tiempos de carga, al mismo tiempo que previene que Guard ejecute el conjunto de pruebas completo al iniciar el servidor.

Para evitar conflictos entre Spring y Git cuando usamos Guard, agregue el directorio **spring/** al archivo **.gitignore** utilizado por Git para determinar qué ignorar cuando se agregan archivos o directorios al repositorio. La forma de hacer esto en el IDE en la nube, es como sigue:

1. Haga click en el ícono de engrane en la parte superior derecha del panel del navegador de archivos ([Figura 3.9](#)).
2. Seleccione “Mostrar archivos ocultos” para visualizar el archivo **.gitignore** en el directgorio raíz de la aplicación ([Figura 3.10](#)).
3. Dé doble-click en el archivo **.gitignore** ([Figura 3.11](#)) para abrirlo, y luego agregue el contenido del [Listado 3.43](#).

Listado 3.43: Agregando Spring al archivo **.gitignore**.

```
# See https://help.github.com/articles/ignoring-files for more about ignoring
# files.
#
# If you find yourself ignoring temporary files generated by your text editor
```

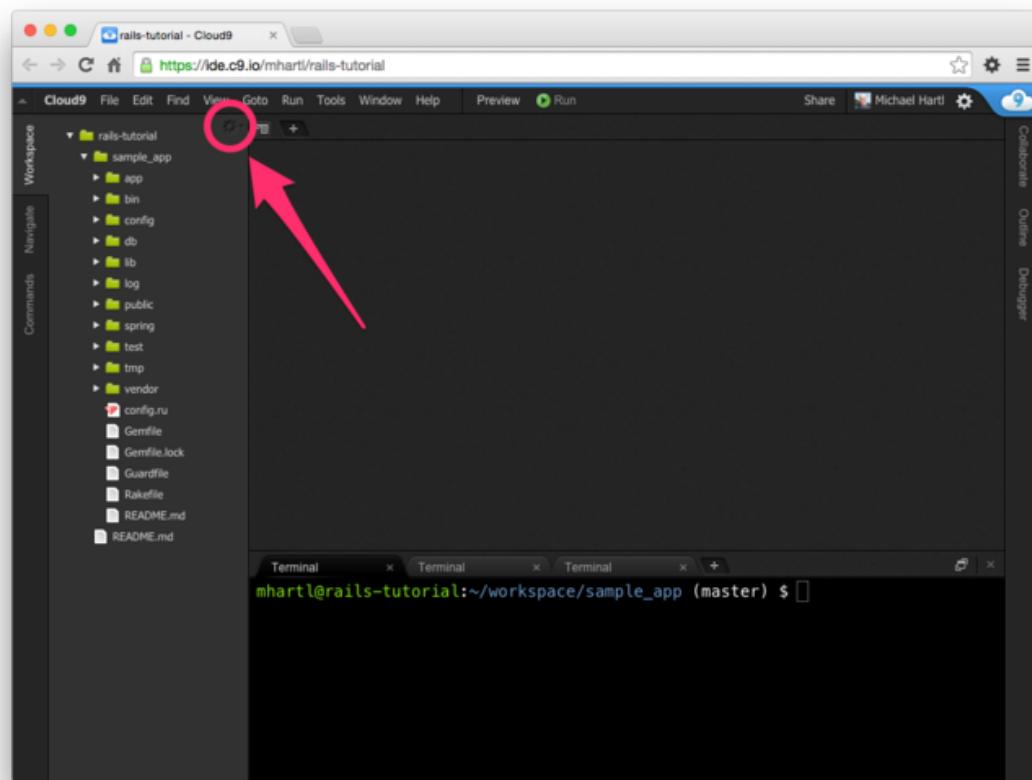


Figura 3.9: El (más bien util) ícono de engrane del panel del navegador de archivos.

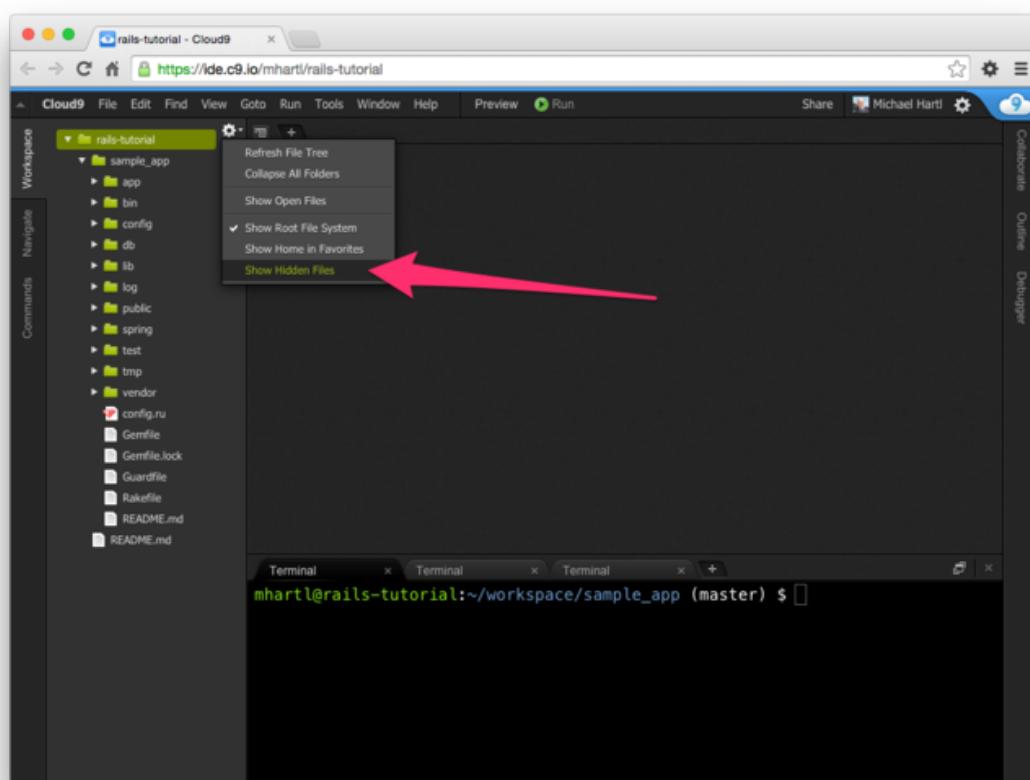


Figura 3.10: Mostrando los archivos ocultos en el navegador de archivos.

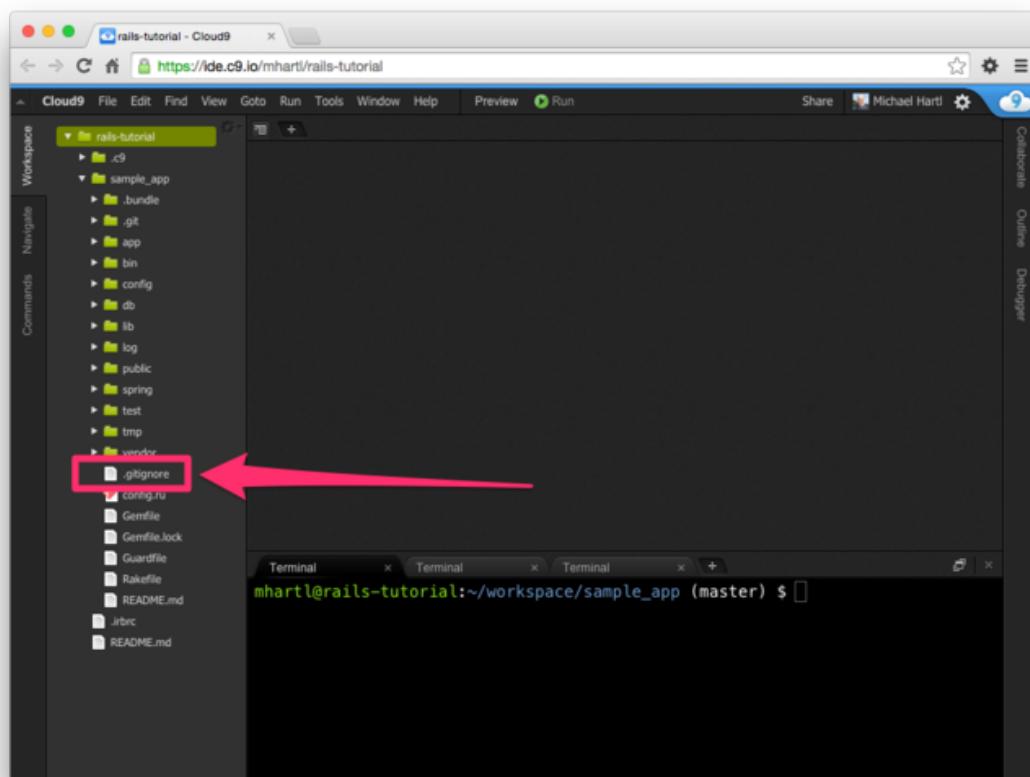


Figura 3.11: El usualmente oculto archivo **.gitignore** hecho visible.

```
# or operating system, you probably want to add a global ignore instead:  
#   git config --global core.excludesfile '~/.gitignore_global'  
  
# Ignore bundler config.  
.bundle  
  
# Ignore the default SQLite database.  
/db/*.sqlite3  
/db/*.sqlite3-journal  
  
# Ignore all logfiles and tempfiles.  
/log/*.log  
/tmp  
  
# Ignore Spring files.  
/spring/*.pid
```

El servidor Spring es un poco singular en el momento en que estoy escribiendo esto, y algunas veces los *procesos* de Spring se acumularán y provocarán una ejecución lenta de sus pruebas. Si sus pruebas parecen estarse ejecutando de forma inusualmente lenta, es aconsejable inspeccionar los procesos de sistema y matar los que sean necesarios (Recuadro 3.4).

Recuadro 3.4. Procesos Unix

En sistemas tipo Unix tales como Linux y OS X, cada tarea del usuario y del sistema se ejecutan dentro de un contenedor bien definido llamado *proceso*. Para ver todos los procesos de sus sistema, puede emplear el comando `ps` con las opciones `aux`:

```
$ ps aux
```

Para filtrar los procesos por tipo, puede utilizar el resultado del comando `ps` y pasarlo al comando `grep` mediante el carácter pipe |, de forma que se realice una búsqueda de patrones:

```
$ ps aux | grep spring  
ubuntu 12241 0.3 0.5 589960 178416 ? Ssl Sep20 1:46  
spring app | sample_app | started 7 hours ago
```

El resultado mostrado nos da algunos detalles acerca del proceso, pero el dato más importante es el primer número, que corresponde al *id del proceso*, o pid. Para eliminar un proceso indeseado, utilice el comando `kill` para invocar el código Unix para mata el proceso (que [resulta ser 9](#)) dado su pid:

```
$ kill -9 12241
```

Esta es la técnica que recomiendo para matar procesos individuales, tales como un servidor Rails colgado (cuyo pid encontramos via `ps aux | grep server`), pero algunas veces es conveniente matar todos los procesos que tienen un nombre en particular, como cuando usted desea matar todos los procesos `spring` que encuentre en su sistema. En este caso particular, debería intentar primero detener los procesos con el mismo comando `spring`:

```
$ spring stop
```

Si esto no funciona, entonces puede matar todos los procesos cuyo nombre contenga el texto `spring` usando el comando `pkill` como se indica:

```
$ pkill -9 -f spring
```

Cada vez que algo no se comporta como se espera, o que un proceso parezca congelado, se sugiere ejecutar `ps aux` para ver qué está pasando, y luego ejecutar `kill -9 <pid>` o `pkill -9 -f <name>` para resolver el problema.

Una vez que Guard ha sido configurado, debería abrir una terminal nueva y (como con el servidor Rails de la [Sección 1.3.2](#)) ejecutarlo en la línea de comandos como sigue:

```
$ bundle exec guard
```

Las reglas del [Listado 3.42](#) están optimizadas para este tutorial, ejecutando automáticamente (por ejemplo) las pruebas de integración cuando un controlador

es modificado. Para ejecutar *todas* las pruebas, presione la tecla **enter** en el cursor de **guard>** prompt. (Algunas veces esto nos arroja error, indicando que hay una falla al conectarse al servidor Spring. Para resolver el problema, sólo presione **enter** de nuevo.)

Para salir de Guard, presione Ctrl-D. Para agregar condiciones adicionales a Guard, revise los ejemplos del [Listado 3.42](#), el archivo [README de Guard](#), y la [wiki de Guard](#).

Capítulo 4

Rails con sabor a Ruby

Basado en los ejemplos del [Capítulo 3](#), este capítulo explora algunos elementos del lenguaje de programación Ruby que son importantes para Rails. Ruby es un gran lenguaje, pero afortunadamente el subconjunto necesario para ser productivo como desarrollador Rails es relativamente pequeño. También difiere de alguna forma del material que usualmente se cubre en una introducción a Ruby. Este capítulo está diseñado para darle una base sólida en Rails con sabor a Ruby, sin importar su experiencia anterior con este lenguaje. Se cubre mucho material, y está bien si no lo asimila todo la primera vez. Haré referencias frecuentes a él en los capítulos posteriores.

4.1 Motivación

Como vimos en el último capítulo, es posible desarrollar un esqueleto de una aplicación Rails, e incluso empezar a probarla, con esencialmente ningún conocimiento del lenguaje base, Ruby. Hicimos esto confiando en el código de prueba que nos proporcionó el tutorial y revisando cada mensaje de error hasta que el conjunto de pruebas pasó exitosamente. Esta situación no puede durar para siempre, por lo que empezaremos este capítulo agregando al sitio funcionalidad que nos permita enfrentarnos cara a cara con nuestras limitaciones en Ruby.

Cuando vimos nuestra aplicación por última vez, habíamos actualizado nues-

tras casi estáticas páginas web para usar la estructura de diseño de Rails y eliminar el código duplicado de nuestras vistas, como se muestra en el [Listado 4.1](#) (que es el mismo que el [Listado 3.32](#)).

Listado 4.1: La estructura de diseño del sitio de la aplicación de ejemplo.

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

Enfoquémonos en una línea particular del [Listado 4.1](#):

```
<%= stylesheet_link_tag 'application', media: 'all',
                       'data-turbolinks-track' => true %>
```

Esto utiliza una función incorporada en Rails `stylesheet_link_tag` (de la que puede leer más en [la documentación de Rails](#)) para incluir `application.css` para todos [los tipos de medios](#) (incluyendo pantallas de computadoras e impresoras). Para un desarrollador experimentado de Rails, esta línea parece simple, pero contiene al menos cuatro conceptos potencialmente confusos de Ruby: métodos incorporados de Rails, invocación de un método sin paréntesis, símbolos y arreglos hash. Estudiaremos todos estos conceptos en este capítulo.

Además de venir equipado con un gran número de funciones incorporadas para ser usadas en las vistas, Rails también permite la creación de nuevas funciones. Tales funciones son llamadas *auxiliares*; para ver cómo hacer un auxiliar personalizado, empecemos examinando la línea del título en el [Listado 4.1](#):

```
<%= yield(:title) %> | Ruby on Rails Tutorial Sample App
```

Esto se basa en la definición de un título de página (usando **provide**) en cada vista, como en

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Pero, ¿qué pasa si no proporcionamos un título? Es una buena costumbre tener un *título de base* que podemos usar en cada página, con una parte opcional de ese título si queremos ser más específicos. *Casi* hemos logrado esto con nuestra estructura de diseño actual, salvo por un detalle: como puede observar, si borramos la llamada a **provide** en una de nuestras vistas, en ausencia de un título específico de la página, el título completo se muestra como sigue:

```
| Ruby on Rails Tutorial Sample App
```

En otras palabras, hay un título base adecuado, pero también hay un carácter, que es una línea vertical | al inicio.

Para resolver el problema cuando falta el título específico de la página, definiremos un auxiliar personalizado llamado **full_title**. El auxiliar **full_title** regresará el título base, “Ruby on Rails Tutorial Sample App”, si ningún título de página es definido, y en caso contrario, agregará el carácter de línea vertical precedido por el título de la página ([Listado 4.2](#)).¹

¹Si un auxiliar es específico de un controlador particular, debería ponerlo en el archivo auxiliar correspondiente; por ejemplo, los auxiliares para el controlador de Páginas Estáticas generalmente van en **app/helpers/static_pages_helper.rb**. En nuestro caso, esperamos que el auxiliar **full_title** sea utilizado en todas las páginas del sitio, y Rails tiene un archivo auxiliar especial para este caso: **app/helpers/application_helper.rb**.

Listado 4.2: Definiendo un auxiliar `full_title`.

app/helpers/application_helper.rb

```
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      page_title + " | " + base_title
    end
  end
end
```

Ahora que tenemos un auxiliar, podemos utilizarlo para simplificar nuestra estructura de diseño al reemplazar

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

con

```
<title><%= full_title(yield(:title)) %></title>
```

como se muestra en el [Listado 4.3](#).

Listado 4.3: La estructura de diseño del sitio con el auxiliar `full_title`.

VERDE

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
```

```
</head>
<body>
  <%= yield %>
</body>
</html>
```

Para poner nuestro auxiliar a trabajar, podemos eliminar la palabra innecesaria “Home” de la página Home, permitiéndole regresar al título base. Hacemos esto actualizando nuestra prueba con el código del [Listado 4.4](#), que modifica la prueba del título anterior y agrega una prueba para la ausencia del texto personalizado “**Home**” en el título.

Listado 4.4: Una prueba actualizada para el título de la página Home. **ROJO**
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase
  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

Ejecutemos nuestro conjunto de pruebas para verificar que al menos una de ellas falla:

Listado 4.5: ROJO

```
$ bundle exec rake test  
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
```

Para hacer que pase, removeremos la línea **provide** de la vista de la página Home, como se muestra en el [Listado 4.6](#).

Listado 4.6: La página Home sin un título personalizado de página. **VERDE**
app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>  
<p>  
  This is the home page for the  
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>  
  sample application.  
</p>
```

En este momento las pruebas deben pasar exitosamente:

Listado 4.7: VERDE

```
$ bundle exec rake test
```

(*Nota:* Ejemplos previos han incluído parcialmente la salida de la ejecución de **rake test**, incluyendo el número de pruebas exitosas y fallidas, pero para abreviar, usualmente serán omitidos de aquí en adelante.)

Al igual que con la línea para incluir la hoja de estilo de la aplicación, el código del [Listado 4.2](#) puede parecer simple ante los ojos de un desarrollador Rails experimentado, pero está *lleno* de conceptos de Ruby importantes: módulos, definición de métodos, argumentos de método opcionales, comentarios, asignación de variables locales, booleanos, control de flujo, concatenación de cadenas de caracteres, y valores de retorno. En este capítulo se revisarán todos estos conceptos también.

4.2 Cadenas de caracteres y métodos

Nuestra principal herramienta para aprender Ruby será la *consola de Rails*, una herramienta de línea de comandos para interactuar con las aplicaciones Rails que vimos por primera vez en la Sección 2.3.3. La consola en sí misma está construída sobre Ruby interactivo (**irb**), y por tanto, tiene acceso a todo el poder del lenguaje Ruby. (Como veremos en la Sección 4.4.4, la consola también tiene acceso al ambiente Rails.)

Si usted está utilizando el IDE en la nube, hay un par de parámetros de configuración del irb que le recomiendo que incluya. Usando el simple editor de texto **nano**, edite el archivo llamado **.irbrc** en el directorio home con los contenidos del Listado 4.8:

```
$ nano ~/.irbrc
```

El Listado 4.8 se encarga de simplificar el *prompt* del irb y eliminar el comportamiento de auto-indentado que resulta algo fastidioso.

Listado 4.8: Configurando el irb.

```
~/.irbrc
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

Sin importar si ha incluido o no la configuración del Listado 4.8, usted puede iniciar la consola en la línea de comandos como sigue:

```
$ rails console
Loading development environment
>>
```

Por default, la consola inicia en un *ambiente de desarrollo*, que es uno de los tres ambientes separados definidos por Rails (los otros son *pruebas* y *producción*). Esta distinción no es importante en este capítulo, pero aprenderemos más sobre los ambientes en la Sección 7.1.1.

La consola es una gran herramienta de aprendizaje, siéntase libre de explorarla—no se preocupe, usted (probablemente) no descompondrá nada. Cuando usamos la consola, tecleamos Ctrl-C si nos atoramos, o Ctrl-D para salir completamente de la consola. En lo que resta del capítulo, puede ser útil consultar la [documentación de Ruby](#). Está repleta (quizá *demasiado*) de información; por ejemplo, para aprender más sobre el manejo de cadenas de caracteres en Ruby puede echar un vistazo a la sección de la clase **String** en la mencionada documentación.

4.2.1 Comentarios

Los *comentarios* en Ruby empiezan con el signo de número `#` (también llamado “gatito”) y se extiende hasta el final de la línea. Ruby ignora los comentarios, pero son útiles para los humanos que lo leen (incluyendo a menudo, ¡al autor original!). En el código

```
# Returns the full title on a per-page basis.
def full_title(page_title = '')
  .
  .
  .
end
```

la primera línea es un comentario, indicando el propósito de la definición de la función que le sigue.

Usualmente no se incluyen comentarios en las sesiones de consola, pero por motivos didácticos incluiré algunos comentarios en lo que sigue, como éste:

```
$ rails console
>> 17 + 42    # Integer addition
=> 59
```

Si usted sigue esta sección escribiendo o copiando-y-pegando comandos en su propia consola, puede omitir los comentarios si lo desea; la consola los ignorará de todas formas.

4.2.2 Cadenas de caracteres

Las *Cadenas de caracteres* son probablemente la estructura de datos más importante para las aplicaciones web, puesto que las páginas web, a últimas, consisten de cadenas de caracteres enviados desde un servidor hacia un navegador. Empecemos explorando las cadenas de caracteres con la consola:

```
$ rails console
>> ""           # An empty string
=> ""
>> "foo"         # A nonempty string
=> "foo"
```

Estas son *cadenas de caracteres literales*, creadas usando el carácter de comillas dobles ". La consola imprime el resultado de evaluar cada línea, que en el caso de cadenas de caracteres literales, es únicamente la misma cadena de caracteres.

También podemos concatenar cadenas de caracteres usando el operador +:

```
>> "foo" + "bar"    # String concatenation
=> "foobar"
```

En este ejemplo, el resultado de evaluar "foo" más "bar" es la cadena "foobar".²

Otra forma de construir cadenas de caracteres es mediante *interpolación* usando la sintaxis especial #{ }:³

```
>> first_name = "Michael"      # Variable assignment
=> "Michael"
>> "#{first_name} Hartl"      # String interpolation
=> "Michael Hartl"
```

Aquí hemos *asignado* el valor "Michael" a la variable `first_name` y luego lo interpolamos con la cadena "#{first_name} Hartl". También pudimos haber asignado ambas cadenas a una variable respectivamente:

²Para saber más sobre los orígenes de "foo" y "bar"—y, en particular, de la posible *no*-relación de "foobar" con "FUBAR"—vea [Archivo de Jerga sobre "foo"](#).

³Los programadores que han trabajado con Perl o PHP pueden comparar esto con la interpolación automática que hace el signo de dólar con las variables en expresiones como "foo \$bar".

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name      # Concatenation, with a space in between
=> "Michael Hartl"
>> "#{first_name} #{last_name}"      # The equivalent interpolation
=> "Michael Hartl"
```

Observe que las dos expresiones finales son equivalentes, pero yo prefiero la versión interpolada; porque el hecho de tener que agregar un espacio en blanco " " parece un poco incómodo.

Imprimiendo

Para *imprimir* una cadena, la función Ruby más comúnmente usada es **puts**:

```
>> puts "foo"      # put string
foo
=> nil
```

El método **puts** opera como un *efecto-secundario*: la expresión **puts "foo"** imprime la cadena en la pantalla y luego regresa **literalmente nulo**: **nil** que es un valor especial de Ruby para “nada en absoluto”. (En lo que resta, algunas veces omitiré la parte **=> nil** por simplicidad.)

Como vimos en los ejemplos anteriores, cuando usamos **puts** automáticamente agrega al final de la línea un carácter de retorno de línea \n a la salida. El método similar **print** no lo hace:

```
>> print "foo"      # print string (same as puts, but without the newline)
foo=> nil
>> print "foo\n"    # Same as puts "foo"
foo
=> nil
```

Cadenas de caracteres con comillas simples

Todo los ejemplos que hemos visto emplean *cadenas de caracteres con comillas dobles*, pero Ruby también le da soporte a las cadenas de caracteres con *comillas simples*. Para varios usos, los dos tipos de cadenas son, en la práctica, idénticos:

```
>> 'foo'          # A single-quoted string
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

Aunque existe una diferencia importante; Ruby no interpola en cadenas con comillas simples:

```
>> '#{foo} bar'      # Single-quoted strings don't allow interpolation
=> "\#{foo} bar"
```

Observe cómo la consola regresa valores si usamos cadenas con comillas dobles, pero require una diagonal inversa como carácter especial que llamamos *escape* para combinaciones tales como `#\{`.

Si las cadenas con comillas dobles pueden hacer todo lo que hacen las cadenas con comillas simples, y además pueden interpolar, entonces ¿cuál es el objetivo de tener cadenas de caracteres con comillas simples? Son útiles a menudo porque son verdaderamente literales, y contienen exactamente los caracteres que usted escribe. Por ejemplo, el carácter “diagonal inversa” es un carácter especial en la mayoría de los sistemas, como en el retorno de línea `\n`. Si usted desea que una variable contenga una diagonal, es más fácil hacerlo con comillas sencillas:

```
>> '\n'          # A literal 'backslash n' combination
=> "\\n"
```

De igual forma que con la combinación `#\{` del ejemplo anterior, Ruby necesitaría el carácter de escape adicional a la diagonal inversa que desea que se muestre; cuando va dentro de comillas dobles, es decir, necesitamos de *dos*

diagonales inversas. Para un ejemplo pequeño como éste, no hay mucha diferencia, pero si se requiere utilizar muchos caracteres de escape, puede haber una gran diferencia:

```
>> 'Newlines (\n) and tabs (\t) both use the backslash character \.'
=> "Newlines (\\\n) and tabs (\\\t) both use the backslash character \\\."
```

Finalmente, vale la pena observar que, en el caso más común, en el que utilizar comillas simples o dobles da lo mismo, a menudo encontrará que el código cambia de una a otra sin ninguna razón aparente. No hay nada que pueda hacerse al respecto, excepto decir, “Bienvenido a Ruby!”

4.2.3 Objetos y paso de mensajes

Todo en Ruby, incluyendo las cadenas de caracteres y aún **nil**, es un *objeto*. Veremos el significado técnico de esto en la [Sección 4.4.2](#), pero no creo que nadie alguna vez haya entendido a los objetos sólo por leer la definición en un libro; puede imaginarlos mejor viendo muchos ejemplos.

Es más fácil describir lo que los objetos *hacen*, que es responder a mensajes. Un objeto como una cadena de caracteres, por ejemplo, puede responder al mensaje **length**, que regresa el número de caracteres de la cadena, es decir, su longitud:

```
>> "foobar".length          # Passing the "length" message to a string
=> 6
```

Típicamente, los mensajes que son pasados a los objetos son *métodos*, que son funciones definidas sobre los objetos.⁴ Las cadenas de caracteres también responden al método **empty?**

⁴Mis disculpas por adelantado por intercambiar abruptamente entre *función* y *método* a lo largo de este capítulo; en Ruby, son la misma cosa: todos los métodos son funciones, y todas las funciones son métodos, porque todo es un objeto.

```
>> "foobar".empty?
=> false
>> "".empty?
=> true
```

Observe el signo de interrogación al final del método `empty?`. Esta es una convención en Ruby para indicar que el valor de retorno es un *booleano*: **verdadero** o **falso**. Los booleanos son especialmente útiles para el *control de flujo*:

```
>> s = "foobar"
>> if s.empty?
>>   "The string is empty"
>> else
>>   "The string is nonempty"
>> end
=> "The string is nonempty"
```

Para incluir más de una cláusula, podemos utilizar **elsif** (**else** + **if**):

```
>> if s.nil?
>>   "The variable is nil"
>> elsif s.empty?
>>   "The string is empty"
>> elsif s.include?("foo")
>>   "The string includes 'foo'"
>> end
=> "The string includes 'foo'"
```

Los booleanos también pueden combinarse utilizando los operadores **&&** (“and”), **||** (“or”), y **!** (“not”):

```
>> x = "foo"
=> "foo"
>> y = ""
=> ""
>> puts "Both strings are empty" if x.empty? && y.empty?
=> nil
>> puts "One of the strings is empty" if x.empty? || y.empty?
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil
```

Puesto que todo en Ruby es un objeto, se deduce que `nil` es un objeto, por lo que puede responder a métodos. Por ejemplo, el método `to_s` puede convertir casi cualquier objeto en una cadena de caracteres:

```
>> nil.to_s
=> ""
```

Esto ciertamente parece ser una cadena vacía, como podemos verificar si *encadenamos* los mensajes que le pasamos a `nil`:

```
>> nil.empty?
NoMethodError: undefined method `empty?' for nil:NilClass
>> nil.to_s.empty?      # Message chaining
=> true
```

Observemos aquí que el objeto `nil` por sí mismo no responde al método `empty?`, pero `nil.to_s` sí lo hace.

Existe un método especial para probar la *nulidad*, que es posible que usted adivine:

```
>> "foo".nil?
=> false
>> "".nil?
=> false
>> nil.nil?
=> true
```

El código

```
puts "x is not empty" if !x.empty?
```

también muestra un uso alternativo de la palabra reservada `if`: Ruby le permite escribir una sentencia que es evaluada sólo si la sentencia que le sigue es verdadera. Existe una palabra reservada complementaria `unless` que funciona de la misma forma:

```
>> string = "foobar"
>> puts "The string '#{string}' is nonempty." unless string.empty?
The string 'foobar' is nonempty.
=> nil
```

Es importante observar que el objeto `nil` es especial, en el sentido de que es el *único* objeto Ruby que es falso en un contexto booleano, además del mismo objeto `false`. Podemos ver esto si usamos `!!`, que niega un objeto dos veces, forzándolo de esta forma a que muestre su valor booleano:

```
>> !!nil
=> false
```

En particular, todos los demás objetos Ruby son *verdaderos*, incluyendo el 0:

```
>> !!0
=> true
```

4.2.4 Definiciones de Métodos

La consola nos permite definir métodos de la misma forma en la que hicimos la acción `home` del Listado 3.6 o el auxiliar `full_title` del Listado 4.2. (Definir métodos en la consola es algo incómodo, y usualmente usaríamos un archivo, pero es conveniente si nuestros propósitos son demostrativos.) Por ejemplo, definamos una función `string_message` que toma un solo *argumento* y regresa un mensaje dependiendo de si el argumento es vacío o no:

```
>> def string_message(str = '')
>>   if str.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> :string_message
>> puts string_message("foobar")
```

```
The string is nonempty.
>> puts string_message("")
It's an empty string!
>> puts string_message
It's an empty string!
```

Como puede verse en el ejemplo final, es posible omitir por completo los argumentos (en cuyo caso también podemos omitir los paréntesis). Esto es porque el código

```
def string_message(str = '')
```

contiene un argumento por *default*, que en este caso es una cadena vacía. Esto hace que el argumento **str** sea opcional, y si lo omitimos, automáticamente toma el valor por default.

Observe que las funciones en Ruby tienen un valor de *retorno implícito*, lo que significa que regresan el valor de la última expresión evaluada—en este caso, una de las dos cadenas; dependiendo de si el argumento del método **str** está vacío o no. Ruby también cuenta con una opción de retorno explícito: la función siguiente es equivalente a la anterior:

```
>> def string_message(str = '')
>>   return "It's an empty string!" if str.empty?
>>   return "The string is nonempty."
>> end
```

(El lector que ha prestado atención puede notar en este punto que el segundo **return** es en realidad, innecesario—siendo la última expresión de la función, la cadena "**The string is nonempty.**" será regresada, sin importar la palabra reservada **return**, pero utilizar **return** en ambos lugares le aporta una simetría agradable al código.)

También es importante entender que el nombre del argumento de la función es irrelevante en lo que concierne al que invoca la función. En otras palabras, el ejemplo anterior puede reemplazar **str** con cualquier otro nombre de variable válido, como podría ser **the_function_argument**, y funcionaría de la misma forma:

```

>> def string_message(the_function_argument = '')
>>   if the_function_argument.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> nil
>> puts string_message("")
It's an empty string!
>> puts string_message("foobar")
The string is nonempty.

```

4.2.5 Regresando al auxiliar para el título

Ahora estamos en posición de entender el auxiliar para el título `full_title` del Listado 4.2,⁵ que aparece comentado en el Listado 4.9.

Listado 4.9: Un auxiliar `title_helper` comentado.

`app/helpers/application_helper.rb`

```

module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      page_title + " | " + base_title
    end
  end
end

```

Estos elementos—la definición de una función (con un argumento opcional), la asignación de variables, las pruebas booleanas, el control de flujo, y la con-

⁵Bueno, aún nos va a faltar *una* cosa que no entenderemos, que es cómo Rails amarra todo: el mapeo de las URLs a las acciones, cómo hace disponible el auxiliar `full_title` a las vistas, etc. Este es un tema interesante, y le invito a investigarlo más a fondo, pero saber exactamente *cómo* trabaja Rails no es necesario cuando *usamos* Rails. (Para una mayor comprensión, le recomiendo *The Rails 4 Way* escrito por Obie Fernández.)

catenación de cadenas de caracteres⁶—hagamos un método auxiliar compacto que usaremos en nuestra estructura de diseño del sitio. El elemento final es el **módulo ApplicationHelper**: éste modulo nos proporciona un medio para agrupar los métodos relacionados, los cuales pueden ser *agregados* a clases Ruby usando **include**. Cuando escribimos Ruby ordinario, a menudo se escriben módulos y se incluyen explícitamente por usted, pero en el caso del módulo auxiliar, Rails maneja la inclusión por nosotros. El resultado es que el método **full_title** está **automáticamente** disponible en todas nuestras vistas.

4.3 Otras estructuras de datos

Aunque las aplicaciones web tratan a últimas sobre cadenas de caracteres, *crear* esas cadenas en la realidad, requiere utilizar también otras estructuras de datos. En esta sección, aprenderemos acerca de algunas estructuras de datos de Ruby importantes para escribir aplicaciones Rails.

4.3.1 Arreglos y Rangos

Un arreglo es sólo una lista de elementos en un orden particular. Aún no hemos hablado de arreglos en el *Tutorial de Rails*, pero entenderlos nos proporciona una buena base para entender los arreglos hash (Sección 4.3.3) y para entender algunos aspectos del modelado de datos en Rails (tales como la asociación **has_many** que vimos en la Sección 2.3.3 y de la que hablaremos más en la Sección 11.1.3).

Hasta aquí hemos dedicado mucho tiempo a la comprensión de las cadenas de caracteres, y existe una forma natural de convertir cadenas a arreglos, mediante el método **split**:

⁶Es tentador utilizar interpolación de cadenas en vez de concatenación—ciertamente, esta es la técnica usada en todas las versiones previas de este tutorial—pero de hecho, la llamada a **provide** convierte la cadena en el tan nombrado objeto SafeBuffer en vez de convertirlo en una cadena de caracteres ordinaria. Interpolar y luego insertar en una plantilla de una vista sobre-escapa cualquier código HTML insertado, de forma que una cadena como “Help’s on the way” se convertiría en “Help's on the way”. (Agradezco al lector Jeremy Fleischman por señalar este pequeño detalle.)

```
>> "foo bar      baz".split      # Split a string into a three-element array.
=> ["foo", "bar", "baz"]
```

El resultado de esta operación es un arreglo de tres cadenas. Por default, **split** divide una cadena en un arreglo dividiéndola en los espacios en blanco, pero usted puede dividirla en casi cualquier otra cosa también:

```
>> "fooxbarxbazx".split('x')
=> ["foo", "bar", "baz"]
```

Como es convención en la mayoría de los lenguajes de programación, los arreglos Ruby tienen *índice cero*, lo que significa que el primer elemento del arreglo tiene índice cero, el segundo índice uno y así sucesivamente:

```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]                  # Ruby uses square brackets for array access.
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]                 # Indices can even be negative!
=> 17
```

Observe que Ruby utiliza corchetes cuadrados para accesar a los elementos del arreglo. Adicionalmente a esta notación de corchetes, Ruby ofrece sinónimos para algunos de los elementos más comúnmente utilizados:⁷

```
>> a                      # Just a reminder of what 'a' is
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
```

⁷El método **second** mencionado aquí no es realmente parte de Ruby, pero es añadido por Rails. Esto funciona porque la consola Rails automáticamente incluye las extensiones de Rails.

```
=> 17
>> a.last == a[-1]      # Comparison using ==
=> true
```

Ésta última línea introduce el operador de comparación de igualdad `==`, que Ruby comparte con muchos otros lenguajes, junto con su operador asociado `!=` (“desigual”), etc.:

```
>> x = a.length          # Like strings, arrays respond to the 'length' method.
=> 3
>> x == 3
=> true
>> x == 1
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
```

Además de `length` (vea la primera línea del código anterior), los arreglos responden a muchos otros métodos:

```
>> a
=> [42, 8, 17]
>> a.empty?
=> false
>> a.include?(42)
=> true
>> a.sort
=> [8, 17, 42]
>> a.reverse
=> [17, 8, 42]
>> a.shuffle
=> [17, 42, 8]
>> a
=> [42, 8, 17]
```

Observe que ninguno de los métodos anteriores realiza cambios sobre el arreglo `a`. Para *modificar* el arreglo, utilice los métodos “bang” correspondientes (se llaman así porque el signo de exclamación usualmente se pronuncia “bang” en este contexto):

```
>> a
=> [42, 8, 17]
>> a.sort!
=> [8, 17, 42]
>> a
=> [8, 17, 42]
```

También puede agregar elementos al arreglo usando el método **push** o su operador equivalente, <<:

```
>> a.push(6)                      # Pushing 6 onto an array
=> [42, 8, 17, 6]
>> a << 7                      # Pushing 7 onto an array
=> [42, 8, 17, 6, 7]
>> a << "foo" << "bar"        # Chaining array pushes
=> [42, 8, 17, 6, 7, "foo", "bar"]
```

Este último ejemplo muestra que puede encadenar varias operaciones juntas, y que, a diferencia de otros lenguajes, los arreglos en Ruby pueden contener elementos de diferentes tipos (en este caso, números enteros y cadenas).

Ya vimos cómo **split** convierte una cadena en un arreglo. También podemos convertir un arreglo en una cadena, mediante el método **join**:

```
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join                         # Join on nothing.
=> "428177foobar"
>> a.join(',')                     # Join on comma-space.
=> "42, 8, 17, 7, foo, bar"
```

Muy relacionados con los arreglos están los *rangos*, que probablemente sean más fáciles de entender si los convertimos a arreglos mediante el método **to_a**:

```
>> 0..9
=> 0..9
>> 0..9.to_a                      # Oops, call to_a on 9.
NoMethodError: undefined method `to_a' for 9:Fixnum
>> (0..9).to_a                   # Use parentheses to call to_a on the range.
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Aunque **0..9** es un rango válido, la segunda expresión del código anterior nos muestra que necesitamos agregar paréntesis para invocar un método sobre el rango.

Los rangos son útiles para extraer elementos de un arreglo:

```
>> a = %w[foo bar baz quux]           # Use %w to make a string array.
=> ["foo", "bar", "baz", "quux"]
>> a[0..2]
=> ["foo", "bar", "baz"]
```

Un truco particularmente útil es utilizar el índice -1 al final del rango para seleccionar todos los elementos desde el inicio hasta el final del arreglo *sin* tener que usar explícitamente la longitud del arreglo:

```
>> a = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..(a.length-1)]                # Explicitly use the array's length.
=> [2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..-1]                          # Use the index -1 trick.
=> [2, 3, 4, 5, 6, 7, 8, 9]
```

Los rangos también funcionan con caracteres:

```
>> ('a'..'e').to_a
=> ["a", "b", "c", "d", "e"]
```

4.3.2 Bloques

Tanto los arreglos como los rangos responden a un conjunto de métodos que aceptan *bloques*, que son al mismo tiempo una de las características de Ruby más poderosas y más confusas:

```
>> (1..5).each { |i| puts 2 * i }
2
4
6
```

```
8  
10  
=> 1..5
```

Este código invoca el método `each` sobre el rango `(1..5)` y lo pasa al bloque `{ |i| puts 2 * i }`. Las barras verticales que rodean al nombre de la variable `|i|` constituyen una sintaxis de Ruby para una variable de bloque, y es decisión del método saber qué hacer con el bloque. En este caso, el rango del método `each` puede manejar un bloque con una sola variable local, que hemos llamado `i`, y que sólo ejecuta el bloque para cada valor en el rango.

Las llaves son una forma de delimitar un bloque, pero hay otra forma de hacerlo:

```
>> (1..5).each do |i|  
?>   puts 2 * i  
=> end  
2  
4  
6  
8  
10  
=> 1..5
```

Los bloques pueden contener más de una línea, y a menudo las tienen. En el *Tutorial de Rails* seguiremos la convención común de usar llaves sólo para bloques de una sola línea y la sintaxis `do..end` para bloques multilínea:

```
>> (1..5).each do |number|  
?>   puts 2 * number  
?>   puts '--'  
=> end  
2  
--  
4  
--  
6  
--  
8  
--  
10  
--  
=> 1..5
```

Aquí he usado **number** en vez de **i** sólo para enfatizar que cualquier nombre de variable funciona.

A menos que usted ya cuente con amplia experiencia programando, no hay atajos para entender los bloques; sólo debe trabajarlos mucho, y en algún momento se acostumbrará a ellos.⁸ Afortunadamente, los humanos somos bastante buenos haciendo generalizaciones a partir de ejemplos concretos; por lo que aquí le muestro algunos más, incluyendo un par que utilizan el método **map**:

```
>> 3.times { puts "Betelgeuse!" }      # 3.times takes a block with no variables.
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
=> 3
>> (1..5).map { |i| i**2 }           # The ** notation is for 'power'.
=> [1, 4, 9, 16, 25]
>> %w[a b c]                      # Recall that %w makes string arrays.
=> ["a", "b", "c"]
>> %w[a b c].map { |char| char.upcase }
=> ["A", "B", "C"]
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
```

Como puede observar, el método **map** regresa el resultado de aplicar el bloque a cada elemento del arreglo o rango. En los dos ejemplos finales, el bloque dentro de **map** involucra el llamado a un método particular en la variable de bloque, y en este caso hay una abreviatura comúnmente usada:

```
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
>> %w[A B C].map(&:downcase)
=> ["a", "b", "c"]
```

(Este código de apariencia extraña pero compacta utiliza un *símbolo*, que discutiremos en la Sección 4.3.3.) Una cosa interesante acerca de esta construcción es que originalmente formaba parte de Ruby on Rails, y a la gente le gustó tanto que ahora se ha incorporado a Ruby.

⁸Los expertos en programación, por otra parte, pueden beneficiarse de saber que los bloques son *cerraduras*, que son funciones anónimas desechables con datos adjuntos.

Como ejemplo final de bloques, echemos un vistazo a una prueba individual del archivo que aparece en el [Listado 4.4](#):

```
test "should get home" do
  get :home
  assert_response :success
  assert_select "title", "Ruby on Rails Tutorial Sample App"
end
```

No es importante entender los detalles (y de hecho, a priori, yo desconozco los detalles), pero podemos deducir de la presencia de la palabra reservada `do` que el cuerpo de la prueba es un bloque. El método `test` toma como argumento una cadena (la descripción) y un bloque, y luego ejecuta el cuerpo del bloque cuando ejecuta todo el conjunto de pruebas.

Por cierto, ahora estamos en buen momento para entender la línea de Ruby que mostré en la [Sección 1.5.4](#) cuando generamos subdominios aleatorios:

```
('a'..'z').to_a.shuffle[0..7].join
```

Construyámoslo paso a paso:

```
>> ('a'..'z').to_a                                # An alphabet array
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
    "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
>> ('a'..'z').to_a.shuffle                         # Shuffle it.
=> ["c", "g", "l", "k", "h", "z", "s", "i", "n", "d", "y", "u", "t", "j", "q",
    "b", "r", "o", "f", "e", "w", "v", "m", "a", "x", "p"]
>> ('a'..'z').to_a.shuffle[0..7]                  # Pull out the first eight elements.
=> ["f", "w", "i", "a", "h", "p", "c", "x"]
>> ('a'..'z').to_a.shuffle[0..7].join             # Join them together to make one string.
=> "mznpwybj"
```

4.3.3 Arreglos Hash y Símbolos

Los arreglos hash son arreglos que esencialmente no están limitados a índices numéricos. (De hecho, algunos lenguajes, especialmente Perl, algunas veces llaman hashes a *arreglos asociativos* por esta razón.) En su lugar, los índices

hash, o *llaves*, pueden ser casi cualquier objeto. Por ejemplo, podemos utilizar cadenas como llaves:

```
>> user = {}                                # {} is an empty hash.
=> {}
>> user["first_name"] = "Michael"          # Key "first_name", value "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"              # Key "last_name", value "Hartl"
=> "Hartl"
>> user["first_name"]                      # Element access is like arrays.
=> "Michael"
>> user                                    # A literal representation of the hash
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Los arreglos hash están indicados con llaves que contienen parejas de llave-valor; una pareja de llaves sin estas parejas—i.e., `{}`—es un arreglo hash vacío. Es importante observar que las llaves para los arreglos hash no tienen nada qué ver con las llaves para los bloques. (Aunque esto pueda parecer confuso.) Aunque los arreglos hash se parecen a los arreglos, una diferencia importante es que los arreglos hash generalmente no garantizan mantener sus elementos en un orden en particular.⁹ Si el orden importa, utilice un arreglo.

En vez de definir arreglos hash elemento por elemento usando corchetes cuadrados, es más fácil usar la representación literal con llaves y valores separados por `=>`, llamado “hashrocket”:

```
>> user = { "first_name" => "Michael", "last_name" => "Hartl" }
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Aquí he empleado la convención usual de Ruby de poner un espacio extra en los dos extremos del arreglo hash—una convención ignorada por la salida de la consola. (No me pregunte porqué estos espacios son una convención; probablemente algún programador influyente de las primeras versiones de Ruby le gustaba cómo se veían los espacios extra, y la convención se arraigó.)

Hasta ahora hemos usado cadenas como llaves del arreglo hash, pero en Rails es mucho más común utilizar *símbolos* en vez de cadenas. Los símbolos

⁹Las versiones de Ruby 1.9 y posteriores de hecho garantizan que los elementos de los arreglos hash conserven el mismo orden en que fueron agregados, pero sería imprudente contar con un orden en particular.

se parecen a las cadenas, pero van precedidos por el signo de dos puntos en vez de rodeada por comillas. Por ejemplo, `:name` es un símbolo. Puede pensar en los símbolos como cadenas sin bagaje extra:¹⁰

```
>> "name".split('')
=> ["n", "a", "m", "e"]
>> :name.split('')
NoMethodError: undefined method `split' for :name:Symbol
>> "foobar".reverse
=> "raboof"
>> :foobar.reverse
NoMethodError: undefined method `reverse' for :foobar:Symbol
```

Los símbolos son un tipo de dato de Ruby especial, compartido con muy pocos lenguajes, por lo que pueden parecer extraños al principio, pero Rails los utiliza bastante, por lo que rápidamente se acostumbrará a ellos. A diferencia de las cadenas, no todos los caracteres son válidos:

```
>> :foo-bar
NameError: undefined local variable or method `bar' for main:Object
>> :2foo
SyntaxError
```

Mientras que empiecen con una letra y se mantengan dentro del alfabeto del idioma inglés, serán aceptados.

En términos de símbolos como llaves de arreglos hash, podemos definir un arreglo hash, `user` como se indica a continuación:

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> { :name=>"Michael Hartl", :email=>"michael@example.com" }
>> user[:name]           # Access the value corresponding to :name.
=> "Michael Hartl"
>> user[:password]       # Access the value of an undefined key.
=> nil
```

¹⁰Como resultado de un menor bajaje, los símbolos son más fáciles de comparar entre sí; las cadenas necesitan ser comparadas carácter por carácter, mientras que los símbolos pueden compararse completamente de una sola vez. Esto los hace ideales para ser usados como llaves de un arreglo hash.

Vemos en el ejemplo anterior que el valor del arreglo hash para una llave no definida es simplemente **nil**.

Puesto que es tan común para los arreglos hash usar símbolos como llaves, la versión 1.9 de Ruby soporta una nueva sintaxis, sólo para este caso especial:

```
>> h1 = { :name => "Michael Hartl", :email => "michael@example.com" }
=> { :name=>"Michael Hartl", :email=>"michael@example.com" }
>> h2 = { name: "Michael Hartl", email: "michael@example.com" }
=> { :name=>"Michael Hartl", :email=>"michael@example.com" }
>> h1 == h2
=> true
```

La segunda sintaxis reemplaza la combinación símbolo / hashrocket con el nombre de la llave seguido por el signo de dos puntos y un valor:

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

Esta construcción sigue más de cerca la notación de arreglos hash presente en otros lenguajes (tales como JavaScript) y disfruta de una creciente popularidad entre la comunidad Rails. Como ambas sintaxis de arreglos hash se usan comúnmente, es primordial ser capaz de reconocerlas. Desafortunadamente, esto puede ser confuso, especialmente porque **:name** es válido por sí mismo (usado como símbolo independiente) pero **name:** no tiene significado por sí mismo. La última línea indica que **:name =>** y **name:** significan lo mismo *únicamente dentro de arreglos hash literales*, de modo que

```
{ :name => "Michael Hartl" }
```

y

```
{ name: "Michael Hartl" }
```

son equivalentes, pero en otro contexto, debe usar **:name** (con el signo de dos puntos al inicio) para denotar un símbolo.

Los valores de los arreglos hash pueden ser virtualmente cualquier cosa, incluso otros arreglos hash, como se muestra en el Listado 4.10.

Listado 4.10: Arreglos hash anidados.

```
>> params = {}           # Define a hash called 'params' (short for 'parameters').
=> {}
>> params[:user] = { name: "Michael Hartl", email: "mhartl@example.com" }
=> { :name=>"Michael Hartl", :email=>"mhartl@example.com" }
>> params
=> { :user=>{ :name=>"Michael Hartl", :email=>"mhartl@example.com" } }
>> params[:user][:email]
=> "mhartl@example.com"
```

Estos tipos de arreglos hash de arreglos hash, o *arreglos hash anidados*, son muy usados en Rails, como veremos cuando empecemos con la Sección 7.3.

De la misma forma que con los arreglos y los rangos, los arreglos hash responden al método **each**. Por ejemplo, considere un arreglo hash llamado **flash** con llaves para dos condiciones, **:success** y **:danger**:

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

Observe que, mientras el método **each** para arreglos toma un bloque con sólo una variable, el método **each** para arreglos hash toma dos, una *llave* y un *valor*. Por lo que, el método **each** para un arreglo hash itera a través de una *pareja* llave-valor a la vez.

El último ejemplo utiliza el método **inspect**, que regresa una cadena con una representación literal del objeto sobre el cual es invocado:

```
>> puts (1..5).to_a           # Put an array as a string.
1
2
3
```

```

4
5
>> puts (1..5).to_a.inspect      # Put a literal array.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"

```

Por cierto, usar `inspect` para imprimir un objeto es tan común que existe una abreviatura, la función `p`:¹¹

```

>> p :name                      # Same output as 'puts :name.inspect'
:name

```

4.3.4 CSS revisado

Es tiempo ahora de revisar la línea del [Listado 4.1](#) utilizada en la estructura de diseño para incluir las hojas de estilo en cascada:

```

<%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>

```

Estamos ahora casi a punto de entender esto. Como mencionamos brevemente en la [Sección 4.1](#), Rails define una función especial para incluir hojas de estilo, y

```

stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true

```

es una llamada a esta función. Pero hay varios misterios. Primero, ¿dónde están los paréntesis? En Ruby, son opcionales, por lo que estas dos líneas son equivalentes:

¹¹De hecho existe una diferencia sutil, que es que `p` regresa el objeto que se ha impreso, mientras que `puts` siempre regresa `nil`. (Agradezco al lector Katarzyna Siwek por señalar esto.)

Segundo, el argumento **media** parece un arreglo hash, pero ¿dónde están las llaves que los delimitan? Cuando los arreglos hash son el *último* argumento en una llamada a una función, las llaves que los delimitan son opcionales, por lo que estas dos líneas son equivalentes:

```
# Curly braces on final hash arguments are optional.  
stylesheet_link_tag 'application', { media: 'all',  
                                'data-turbolinks-track' => true }  
stylesheet_link_tag 'application', media: 'all',  
                    'data-turbolinks-track' => true
```

Luego, ¿porqué la pareja de llave-valor **data-turbolinks-track** utiliza la vieja sintaxis hashrocket? Esto es porque usar la nueva sintaxis para escribir

data-turbolinks-track: true

resulta en código no válido debido a los guiones. (Recuerde de la [Sección 4.3.3](#) que los guiones no pueden ser usados en los símbolos.) Esto nos obliga a utilizar la sintaxis vieja, produciendo

```
'data-turbolinks-track' => true
```

Finalmente, ¿porqué Ruby interpreta correctamente las líneas

```
stylesheet_link_tag 'application', media: 'all',
                    'data-turbolinks-track' => true
```

aún con un salto de línea entre los elementos finales? La respuesta es que Ruby no distingue entre saltos de línea y espacios en blanco en este contexto.¹² La razón por la que elegí dividir en pedazos el código es que prefiero mantener las líneas de código debajo de los 80 caracteres de longitud, por legibilidad.¹³

Entonces, vemos ahora que la línea

```
stylesheet_link_tag 'application', media: 'all',
                    'data-turbolinks-track' => true
```

invoca la función `stylesheet_link_tag` con dos argumentos: una cadena, indicando la ruta de la hoja de estilo, y un arreglo hash con dos elementos, indicando el tipo de medios e indicándole a Rails que utilice la característica `turbolinks` agregada en la versión 4.0 de Rails. Debido a los corchetes `<%= %>`, los resultados son insertados en la plantilla por ERb, y si usted observa el código de la página en su navegador, verá el código HTML necesario para incluir una hoja de estilo ([Listado 4.11](#)). (Puede observar algunas otras cosas, como `?body=1`, después de los nombres de los archivos CSS. Éstos son insertados por Rails para asegurarse de que los navegadores vuelvan a cargar el archivo CSS cuando éste cambie en el servidor.)

Listado 4.11: El código HTML producido por las inclusiones CSS.

```
<link data-turbolinks-track="true" href="/assets/application.css" media="all"
      rel="stylesheet" />
```

Si usted ve realmente el archivo CSS navegando a <http://localhost:3000/assets/-application.css>, verá que (aparte de los comentarios) está vacío. Nos encargaremos de cambiar esto en el [Capítulo 5](#).

¹²Un salto de línea es el carácter que viene al final de una línea, iniciando de este modo una nueva línea. En código, es representado por el carácter `\n`.

¹³En la actualidad, *contar* las columnas puede enloquecerlo, es por esto que muchos editores de texto tienen una ayuda visual. Por ejemplo, si echa un vistazo a la [Figura 1.5](#), podrá ver una pequeña línea vertical a la derecha para ayudarlo a no sobrepasar los 80 caracteres por línea. El IDE en la nube ([Sección 1.2.1](#)) incluye esta línea por default. Si usted utiliza TextMate, puede encontrar esta opción bajo el menú `View > Wrap Column > 78`. En Sublime Text, puede utilizar `View > Ruler > 78` o `View > Ruler > 80`.

4.4 Clases Ruby

Hemos dicho antes que todo en Ruby es un objeto, y en esta sección finalmente definiremos uno por nuestra cuenta. Ruby, como muchos otros lenguajes orientados a objetos, utiliza *clases* para organizar los métodos; estas clases luego son *instanciadas* para crear objetos. Si usted no está familiarizado con la programación orientada a objetos, esto puede sonarle poco inteligible, por lo cual veremos algunos ejemplos concretos.

4.4.1 Constructores

Hemos visto muchos ejemplos de clases que instancian objetos, pero nosotros no lo hemos hecho explícitamente. Por ejemplo, creamos una instancia de una cadena utilizando las comillas dobles, lo cual representa un *constructor literal* para cadenas:

```
>> s = "foobar"      # A literal constructor for strings using double quotes
=> "foobar"
>> s.class
=> String
```

Vemos aquí que las cadenas responden al método **class**, y éste regresa simplemente el nombre de la clase a la que pertenecen.

En vez de usar un constructor literal, podemos utilizar el equivalente *constructor nombrado*, que involucra la llamada al método **new** sobre el nombre de la clase:¹⁴

```
>> s = String.new("foobar")    # A named constructor for a string
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

¹⁴Estos resultados varían dependiendo de la versión de Ruby que esté usando. Este ejemplo supone que usted está utilizando Ruby 1.9.3 o superior.

Esto es equivalente al constructor literal, pero es más explícito acerca de lo que estamos haciendo.

Los arreglos funcionan de la misma forma que las cadenas:

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

Los arreglos hash, por el contrario, son diferentes. Mientras que el constructor de arreglos `Array.new` toma un valor inicial para el arreglo, `Hash.new` toma un valor por *default* para el arreglo hash, que es el valor del hash para una llave no existente:

```
>> h = Hash.new
=> {}
>> h[:foo]           # Try to access the value for the nonexistent key :foo.
=> nil
>> h = Hash.new(0)   # Arrange for nonexistent keys to return 0 instead of nil.
=> {}
>> h[:foo]
=> 0
```

Cuando un método es invocado sobre la clase misma, como en el caso de `new`, se denomina *método de clase*. El resultado de llamar a `new` sobre la clase, es un objeto de esa clase, también llamado *instancia* de la clase. Un método invocado sobre una instancia, tal como `length`, es denominado *método de instancia*.

4.4.2 Herencia de Clases

Cuando aprendemos acerca de clases, es útil encontrar la *jerarquía de la clase* usando el método `superclass`:

```
>> s = String.new("foobar")
=> "foobar"
>> s.class           # Find the class of s.
=> String
>> s.class.superclass # Find the superclass of String.
```

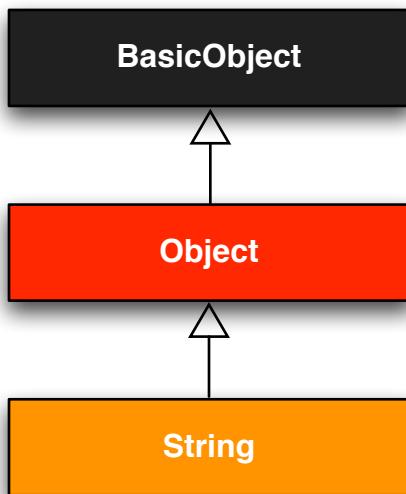


Figura 4.1: La jerarquía de herencia de la clase `String`.

```
=> Object
=> s.class.superclass.superclass # Ruby 1.9 uses a new BasicObject base class
=> BasicObject
=> s.class.superclass.superclass.superclass
=> nil
```

Un diagrama de esta jerarquía de herencia aparece en la Figura 4.1. Aquí vemos que la superclase de `String` es `Object` y la superclase de `Object` es `BasicObject`, pero `BasicObject` no tiene superclase. Este patrón es cierto para cada objeto Ruby: averiguar la jerarquía completa de cualquier clase en Ruby indica que todas heredan de `BasicObject`, que ya no tiene superclase. Éste es el significado técnico de “todo en Ruby es un objeto”.

Para entender las clases un poco más a fondo, no hay como crear una por nuestra cuenta. Hagamos una clase `Word` con un método `palindrome?` que regrese `verdadero` si la palabra se lee igual al derecho que al revés:

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

Podemos utilizarlo como sigue:

```
>> w = Word.new          # Make a new Word object.
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

Si este ejemplo le parece un poco superficial, bien—es por diseño. Es extraño crear una nueva clase sólo para crear un método que toma una cadena como argumento. Puesto que una palabra es una cadena de caracteres, es más natural que nuestra clase **Word** herede de la clase **String**, como se muestra en el [Listado 4.12](#). (Debe salir de la consola y volver a entrar para limpiar de la memoria la definición anterior de **Word**.)

Listado 4.12: Definiendo una clase **Word** en la consola.

```
>> class Word < String           # Word inherits from String.
>>   # Returns true if the string is its own reverse.
>>   def palindrome?
>>     self == self.reverse        # self is the string itself.
>>   end
>> end
=> nil
```

Aquí **Word < String** es la sintaxis de Ruby para herencia (discutida brevemente en la [Sección 3.2](#)), lo cual asegura que, adicionalmente al nuevo método **palindrome?**, las palabras también tendrán los mismos métodos que las cadenas:

```
>> s = Word.new("level")      # Make a new Word, initialized with "level".
=> "level"
>> s.palindrome?           # Words have the palindrome? method.
=> true
>> s.length                # Words also inherit all the normal string methods.
=> 5
```

Puesto que la clase **Word** hereda de **String**, podemos usar la consola para ver la jerarquía de clase explícitamente:

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

Esta jerarquía está ilustrada en la [Figura 4.2](#).

En el [Listado 4.12](#), observe que para verificar que la palabra es su propia inversa, necesitamos tener acceso a la palabra dentro de la clase **Word**. Ruby nos permite hacer esto mediante la palabra reservada **self**: dentro de la clase **Word**, **self** es el objeto mismo, lo que significa que podemos utilizar

```
self == self.reverse
```

para verificar si la palabra es un palíndromo.¹⁵ De hecho, dentro de la clase **String** el uso de **self** es opcional sobre un método o atributo (a menos que estemos haciendo una asignación), por lo que

```
self == reverse
```

debe funcionar también.

¹⁵Para saber más sobre las clases Ruby y la palabra reservada **self**, vea el post [RailsTips](#) “Variables de Clase e Instancia en Ruby”.

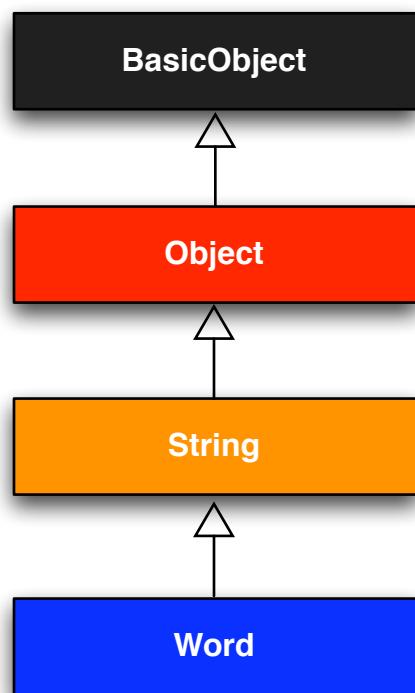


Figura 4.2: La jerarquía de herencia para la (no incorporada) clase `Word` del Listado 4.12.

4.4.3 Modificando las clases incorporadas

Mientras que la idea de herencia es poderosa, en el caso de palíndromos puede ser más natural agregar el método `palindrome?` a la misma clase `String`, por lo que (entre otras cosas) podríamos invocar `palindrome?` en una cadena literal, lo cual no podemos hacer en este momento:

```
>> "level".palindrome?  
NoMethodError: undefined method `palindrome?' for "level":String
```

Sorpresivamente, Ruby le permite hacer esto; las clases Ruby pueden ser *abiertas* y modificadas, permitiendo a los mortales ordinarios como nosotros, agregarles métodos:

```
>> class String  
>>   # Returns true if the string is its own reverse.  
>>   def palindrome?  
>>     self == self.reverse  
>>   end  
>> end  
=> nil  
>> "deified".palindrome?  
=> true
```

(No sé qué es mejor: que Ruby le permita agregar métodos a sus clases incorporadas, o que "`reconocer`" es un palíndromo.)

Modificar las clases incorporadas es una técnica poderosa, pero con gran poder viene también una gran responsabilidad, y es considerada una mala práctica agregar métodos a las clases incorporadas sin tener una *muy* buena razón para hacerlo. Rails tiene algunas buenas razones; por ejemplo, en aplicaciones web, a menudo queremos evitar que las variables estén *en blanco*—digamos, el nombre de un usuario debería ser algo diferente de `espacios en blanco`—por lo que Rails agrega el método `blank?` a Ruby. Puesto que la consola de Rails automáticamente incluye las extensiones de Rails, podemos ver un ejemplo aquí (esto no funciona en `irb` solo):

```
>> "".blank?
=> true
>> ""      .empty?
=> false
>> ""      .blank?
=> true
>> nil.blank?
=> true
```

Vemos que una cadena de espacios no está *vacía*, pero está en *blanco*. Observe también que **nil** se considera blanco; puesto que **nil** no es una cadena, éste es un indicio de porqué Rails agrega el método **blank?** a la clase base de **String**, que (como vimos al inicio de esta sección) es la clase **Object**. Veremos otros ejemplos de adiciones de Rails a clases de Ruby en la Sección 8.4.

4.4.4 Una clase controlador

Toda la plática sobre clases y herencia pudo haber emitido un destello de familiaridad, porque las hemos visto antes, en el controlador de Páginas Estáticas ([Listado 3.18](#)):

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Ahora usted está en posición de apreciar, aunque sea vagamente, lo que significa este código: **StaticPagesController** es una clase que hereda de la clase **ApplicationController**, y viene equipada con los métodos **home**, **help** y **about**. Puesto que cada sesión de la consola Rails carga el ambiente de Rails

local, podemos crear un controlador explícitamente y examinar su jerarquía de clases:¹⁶

```
>> controller = StaticPagesController.new
=> #<StaticPagesController:0x22855d0>
>> controller.class
=> StaticPagesController
>> controller.class.superclass
=> ApplicationController
>> controller.class.superclass.superclass
=> ActionController::Base
>> controller.class.superclass.superclass.superclass
=> ActionController::Metal
>> controller.class.superclass.superclass.superclass.superclass
=> AbstractController::Base
>> controller.class.superclass.superclass.superclass.superclass.superclass
=> Object
```

Un diagrama de esta jerarquía aparece en la [Figura 4.3](#).

Incluso podemos invocar las acciones del controlador desde la consola, puesto que a últimas, son sólo métodos:

```
>> controller.home
=> nil
```

Aquí el valor de retorno es `nil` porque la acción `home` está en blanco.

Pero espere—las acciones no tienen valores de retorno, al menos no alguno que importe. El objetivo de la acción `home`, como vimos en el [Capítulo 3](#), es desplegar una página web, no regresar un valor. Y no recuerdo haber invocado `StaticPagesController.new` en ningún lugar. ¿Qué está pasando?

Lo que sucede es que Rails está *escrito en Ruby*, pero Rails no es Ruby. Algunas clases Rails son utilizadas como objetos Ruby ordinarios, pero algunas son solamente *materia prima* para el molino mágico de Rails. Rails es *sui generis*, y debería ser estudiado y comprendido de forma independiente de Ruby.

¹⁶No es necesario que sepa qué hace cada una de las clases de esta jerarquía. Yo no sé qué hacen todas ellas, y he estado programando con Ruby on Rails desde el 2005. Lo cual significa que (a) Soy sumamente incompetente, o que (b) usted puede ser un desarrollador de Rails hábil sin conocer todas sus entrañas. Espero por el bien de ambos, que sea la segunda.

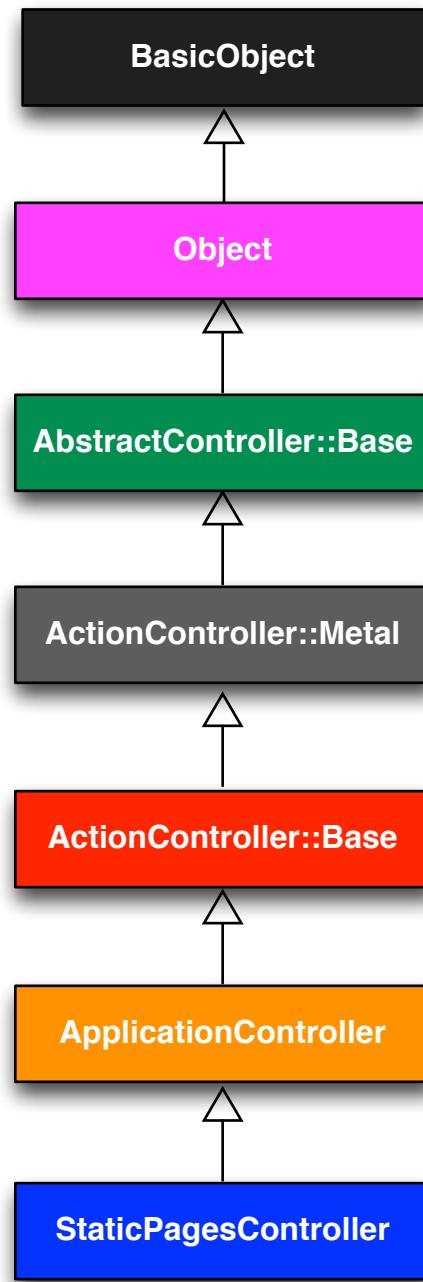


Figura 4.3: La jerarquía de herencia para las Páginas Estáticas.

4.4.5 Una clase usuario

Terminaremos nuestro paseo por Ruby con una clase completa de nuestra propiedad, una clase **User** que anticipa el modelo User que viene en el [Capítulo 6](#).

Hasta aquí hemos realizado definiciones de clases en la consola, pero este procedimiento rápidamente se vuelve cansado; en su lugar, crearemos el archivo **example_user.rb** en el directorio raíz de su aplicación y le agregaremos los contenidos del [Listado 4.13](#).

Listado 4.13: Código para un ejemplo de usuario.

`example_user.rb`

```
class User
  attr_accessor :name, :email

  def initialize(attributes = {})
    @name = attributes[:name]
    @email = attributes[:email]
  end

  def formatted_email
    "#{@name} <#{@email}>"
  end
end
```

Revisemos paso a paso lo que está sucediendo aquí. La primera línea,

`attr_accessor :name, :email`

crea los *accesos a los atributos* correspondientes al nombre del usuario y su dirección electrónica. Esto crea los métodos “getter” y “setter” que nos permiten obtener (get) y asignar (set) valores a las *variables de instancia* `@name` y `@email`, las cuales mencionamos brevemente en las Secciones [2.2.2](#) y [3.6](#). En Rails, la mayor importancia de las variables de instancia es que están disponibles de forma automática en las vistas, pero en general son variables que están disponibles dentro de una clase Ruby. (Mencionaremos un poco más sobre esto en un momento.) Las variables de instancia siempre comienzan con el signo `@`, y su valor es `nil` si no se le define uno.

El primer método, `initialize`, es especial en Ruby: este método es invocado cuando ejecutamos `User.new`. En particular `initialize` recibe un argumento, `attributes`:

```
def initialize(attributes = {})
  @name  = attributes[:name]
  @email = attributes[:email]
end
```

Aquí la variable `attributes` tiene un *valor por default* igual a un arreglo hash vacío, por lo que podemos definir un usuario sin nombre o dirección electrónica. (Recuerde de la Sección 4.3.3 que los arreglos hash regresan `nil` para llaves no existentes, por lo que `attributes[:name]` será `nil` si no existe una llave `:name`, y lo mismo sucede con `attributes[:email]`.)

Finalmente, nuestra clase define un método llamado `formatted_email` que utiliza los valores asignados a las variables `@name` y `@email` para construir una versión de la dirección electrónica del usuario, con un formato elegante, utilizando interpolación de cadenas (Sección 4.2.2):

```
def formatted_email
  "#{@name} <#{@email}>"
end
```

Como `@name` y `@email` son variables de instancia (como nos indica el signo `@`), están disponibles de forma automática en el método `formatted_email`.

Iniciemos una consola, ejecute una instrucción `require` para el código del ejemplo de usuario, y juegue un poco con nuestra clase User:

```
>> require './example_user'      # This is how you load the example_user code.
=> true
>> example = User.new
=> #<User:0x224ceec @email=nil, @name=nil>
>> example.name                # nil since attributes[:name] is nil
=> nil
>> example.name = "Example User"        # Assign a non-nil name
=> "Example User"
>> example.email = "user@example.com"    # and a non-nil email address
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

Aquí el '.' significa en Unix “directorio actual”, y '`./example_user`' le dice a Ruby que busque un archivo con ese nombre ubicado en una ruta relativa a la ubicación indicada. El código que sigue, crea un usuario de ejemplo vacío y luego asigna valores al nombre y a la dirección electrónica accediendo directamente a los atributos respectivos (también se pueden hacer las asignaciones usando la línea `attr_accessor` del Listado 4.13). Luego escribimos

```
example.name = "Example User"
```

Ruby asigna a la variable `@name` el valor "`Example User`" (y de forma análoga asigna valor al atributo `email`), que luego usaremos en el método `formatted_email`.

Recuerde de la Sección 4.3.4 que podemos omitir las llaves cuando el último argumento de la función es un arreglo hash. Podemos crear otro usuario si pasamos un arreglo hash al método `initialize` para crear un usuario con atributos pre-definidos:

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User:0x225167c @email="mhartl@example.com", @name="Michael Hartl">
>> user.formatted_email
=> "Michael Hartl <mhartl@example.com>"
```

Al empezar el Capítulo 7 veremos que inicializar objetos usando como argumento un arreglo hash, es una técnica conocida como *asignación masiva*, común en las aplicaciones Rails.

4.5 Conclusión

Esto concluye nuestro repaso del lenguaje Ruby. En el Capítulo 5, empezaremos a darle buen uso mientras desarrollamos la aplicación de ejemplo.

No volveremos a hacer uso del archivo `example_user.rb` de la Sección 4.4.5, por lo que le sugiero borrarlo:

```
$ rm example_user.rb
```

Haga **commit** a los otros cambios, súbalos a Bitbucket, y despliegue en Heroku:

```
$ git status
$ git commit -am "Add a full_title helper"
$ git push
$ bundle exec rake test
$ git push heroku
```

4.5.1 Qué aprendimos en este capítulo

- Ruby tiene una gran cantidad de métodos para manipular cadenas de caracteres.
- Todo en Ruby es un objeto.
- Ruby soporta la definición de un método mediante la palabra reservada **def**.
- Ruby soporta la definición de una clase mediante la palabra reservada **class**.
- Las vistas de Rails pueden contener HTML estático ó Ruby Embbebido (ERb).
- Las estructuras de datos incorporadas en Ruby incluyen arreglos, rangos y arreglos hash.
- Los bloques de Ruby son construcciones flexibles que (entre otras cosas) permiten una iteración natural sobre estructuras de datos enumerables.
- Los símbolos son como etiquetas, como cadenas sin estructura adicional.
- Ruby soporta herencia de objetos.

- Es posible abrir y modificar las clases Ruby incorporadas.
- La palabra “reconocer” es un palíndromo.

4.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

1. Al reemplazar los signos de interrogación del Listado 4.14 con los métodos apropiados, combine **split**, **shuffle** y **join** para escribir una función que revuelva las letras de una cadena dada.
2. Usando el Listado 4.15 como guía, agregue un método **shuffle** a la clase **String**.
3. Cree tres arreglos hash llamados **person1**, **person2** y **person3**, con nombres y apellidos almacenados en las llaves **:first** y **:last**. Luego cree un arreglo hash **params** de forma que **params[:father]** sea **person1**, **params[:mother]** sea **person2** y **params[:child]** sea **person3**. Verifique que, por ejemplo, **params[:father][:first]** tiene el valor correcto.
4. Busque una versión en línea de la documentación del API de Ruby y lea acerca del método **merge** para arreglos hash. ¿Cuál es el valor de la expresión siguiente?

```
{ "a" => 100, "b" => 200 }.merge({ "b" => 300 })
```

Listado 4.14: Esqueleto de una función que revuelve cadenas.

```
>> def string_shuffle(s)
>>   s.?('').?.
>> end
>> string_shuffle("foobar")
=> "oobfra"
```

Listado 4.15: Esqueleto de un método **shuffle** adjuntado a una clase **String**.

```
>> class String
>>   def shuffle
>>     self.?('').?.
>>   end
>> end
>> "foobar".shuffle
=> "borafo"
```

Capítulo 5

Rellenando la estructura de diseño

En el proceso de haber realizado el breve recorrido por Ruby en el [Capítulo 4](#), aprendimos acerca de cómo incluir la hoja de estilo en la aplicación de ejemplo ([Sección 4.1](#)), pero (como observamos en la [Sección 4.3.4](#)) la hoja de estilo aún no contiene nada de CSS. En este capítulo, empezaremos a llenar la hoja de estilo personalizada al incorporar una biblioteca de CSS a nuestra aplicación, y luego agregaremos algunos estilos personalizados por nuestra cuenta.¹ También empezaremos a llenar la estructura de diseño con enlaces a las páginas (tales como Home y About) que hemos creado hasta ahora ([Sección 5.1](#)). En el camino, aprenderemos sobre parciales, rutas Rails, y sobre la cadena de procesos conectados, incluyendo una introducción a Sass ([Sección 5.2](#)). Terminaremos dando un paso importante al permitir a los usuarios registrarse en nuestro sitio ([Sección 5.4](#)).

La mayoría de los cambios en este capítulo involucran agregar y editar la estructura de diseño de la aplicación de ejemplo, lo cual (en base a nuestras directrices del Recuadro 3.3) es exactamente la clase de trabajo que ordinariamente no hay que probar. Como resultado, la mayor parte del tiempo estaremos utilizando el editor de texto y el navegador; usaremos TDD únicamente para agre-

¹Agradezco al lector [Colm Tuite](#) por su excelente trabajo ayudando a convertir la aplicación de ejemplo para que utilice la biblioteca CSS de Bootstrap.

gar una página de contacto (Sección 5.3.1). Aunque agregaremos una prueba importante, escribiendo nuestra primera *prueba de integración* para verificar que los enlaces en la estructura de diseño final están correctos (Sección 5.3.4).

5.1 Agregando un poco de estructura

El *Tutorial de Ruby on Rails* es un libro acerca de desarrollo web, no de diseño web, pero sería deprimente trabajar en una aplicación cuya apariencia es *deplorable*, por lo que en esta sección agregaremos un poco de estructura al diseño y le daremos algo de estilo con CSS. Además de emplear algunas reglas CSS personalizadas, utilizaremos *Bootstrap*, una biblioteca de diseño web de código abierto elaborada por Twitter. También le daremos a nuestro *código* algo de estilo, por así decirlo, usando *parciales* para arreglar un poco la estructura cuando ésta parezca un poco desordenada.

Cuando construimos aplicaciones web, a menudo es útil contar con una descripción general de la interfaz de usuario lo más pronto posible. Por esta razón, en lo que resta del libro, incluiré a menudo *bocetos* (en un contexto web a menudo se les conoce como *esquemas de páginas*), que son bosquejos de cómo podría verse una eventual aplicación.² En este capítulo, estaremos desarrollando principalmente las páginas estáticas introducidas en la Sección 3.2, incluyendo un logo para el sitio, un encabezado de navegación, y un pie de página. Un boceto para la más importante de estas páginas, la página Home, aparece en la Figura 5.1. Puede ver el resultado final en la Figura 5.7. Observará que existen diferencias en algunos detalles—por ejemplo, al final agregaremos un logo de Rails a la página—pero está bien, puesto que un boceto no requiere ser exacto.

Como es usual, si usted está usando Git para el control de versiones, ahora es un buen momento para crear una nueva rama:

²Los bocetos del *Tutorial de Ruby on Rails* fueron hechos con una excelente aplicación en línea llamada [Mockingbird](#).

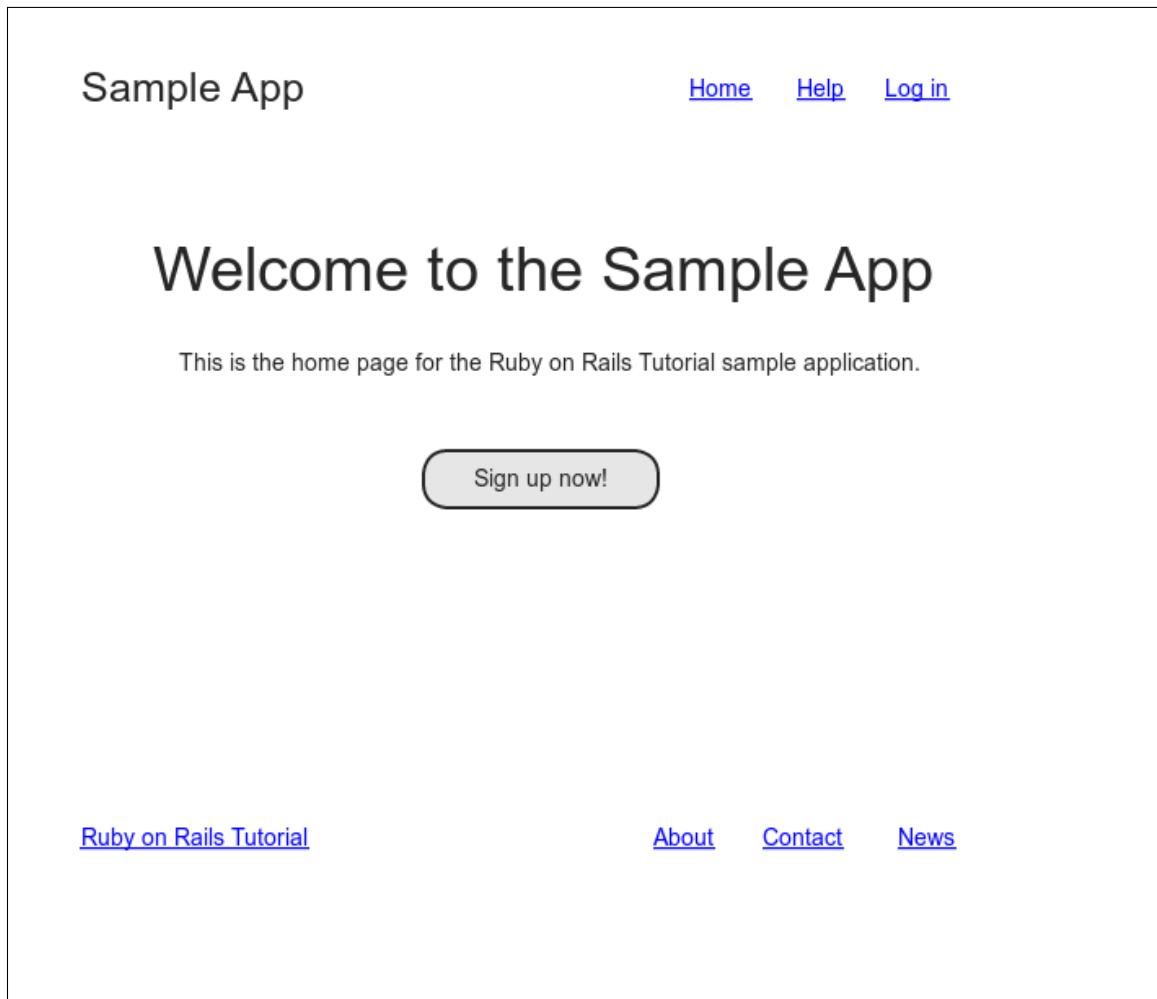


Figura 5.1: Un boceto para la página Home de la aplicación de ejemplo.

```
$ git checkout master
$ git checkout -b filling-in-layout
```

5.1.1 Navegación del sitio

Nuestro primer paso para agregar enlaces y estilo a la aplicación de ejemplo será actualizar el archivo que contiene la estructura de diseño del sitio **application.html.erb** (que vimos por última vez en el [Listado 4.3](#)) con elementos HTML nuevos. Esto incluye algunas divisiones, clases CSS y el inicio de la navegación por el sitio. El archivo completo se muestra en el [Listado 5.1](#); las explicaciones para las diversas partes se encuentran justo a continuación de las mismas. Si usted no desea retardar la gratificación, puede ver los resultados en la [Figura 5.2](#). (*Nota:* no es muy gratificante (aún).)

Listado 5.1: La estructura de diseño del sitio con elementos nuevos.

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
    <!--[if lt IE 9]>
      <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
      </script>
    <![endif]-->
  </head>
  <body>
    <header class="navbar navbar-fixed-top navbar-inverse">
      <div class="container">
        <%= link_to "sample app", '#', id: "logo" %>
        <nav>
          <ul class="nav navbar-nav navbar-right">
            <li><%= link_to "Home", '#' %></li>
            <li><%= link_to "Help", '#' %></li>
            <li><%= link_to "Log in", '#' %></li>
          </ul>
        </nav>
      </div>
    </header>
```

```
</header>
<div class="container">
  <%= yield %>
</div>
</body>
</html>
```

Revisemos los elementos nuevos del Listado 5.1 de arriba hacia abajo. Como se mencionó brevemente en la Sección 3.4.1, Rails emplea HTML5 por default (como se indica en la línea `<!DOCTYPE html>`); puesto que el estándar HTML5 es relativamente nuevo, algunos navegadores (especialmente versiones viejas de Internet Explorer) no lo soportan completamente, por lo que incluimos un poco de código JavaScript (conocido como “HTML5 shim (o shiv)”)³ para mitigar el problema:

```
<!--[if lt IE 9]>
<script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
</script>
<![endif]-->
```

La sintaxis algo extraña

```
<!--[if lt IE 9]>
```

incluye las líneas dentro del bloque sólo si la versión de Microsoft Internet Explorer (IE) es inferior a 9 (`if lt IE 9`). La singular sintaxis `[if lt IE 9]` no es parte de Rails; en realidad es un [comentario condicional](#) soportado por los navegadores Internet Explorer para estos casos. Esto es bueno, porque significa que podemos incluir el HTML5 shim *únicamente* para los navegadores IE cuya versión es menor que 9, evitando afectar otros navegadores como Firefox, Chrome y Safari.

³Las palabras *shim* y *shiv* se usan indistintamente en este contexto; la primera es el término apropiado, basado en la palabra del inglés cuyo significado es “una arandela o tira de material delgado utilizado para hacer coincidir partes, o reducir el desgaste”, mientras que la segunda (significa “un cuchillo o navaja utilizada como arma”) es aparentemente un juego de palabras junto con el nombre del autor, Sjoerd Visscher.

La siguiente sección incluye un **encabezado** para el logo del sitio (en texto plano), un par de divisiones (utilizando la etiqueta **div**), y una lista de elementos con enlaces de navegación:

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", '#', id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", '#' %></li>
        <li><%= link_to "Help", '#' %></li>
        <li><%= link_to "Log in", '#' %></li>
      </ul>
    </nav>
  </div>
</header>
```

Aquí la etiqueta **header** comprende los elementos que deben ir en la parte superior de la página. Le hemos asociado a la etiqueta **header** tres *clases CSS*,⁴ llamadas **navbar**, **navbar-fixed-top**, y **navbar-inverse**, separadas por espacios:

```
<header class="navbar navbar-fixed-top navbar-inverse">
```

A todos los elementos HTML se les puede asociar tanto clases como *ids* (identificadores); éstos son meramente etiquetas, y son útiles para agregar estilo con CSS ([Sección 5.1.2](#)). La principal diferencia entre clases y *ids* es que las clases pueden usarse varias veces en una página, pero los *ids* deben utilizarse sólo una vez. En el caso que nos ocupa, todas las clases “navbar” tienen significado especial para la biblioteca Bootstrap, que instalaremos y utilizaremos en la [Sección 5.1.2](#).

Dentro de la etiqueta **header**, vemos una etiqueta **div**:

```
<div class="container">
```

⁴Estas no están relacionadas en absoluto con las clases Ruby.

La etiqueta **div** es una división genérica; no hace nada además de dividir el documento en distintas partes. En el estilo antiguo de HTML, las etiquetas **div** se utilizaban para casi todas las divisiones del sitio, pero HTML5 agregó los elementos **header**, **nav**, y **section** para divisiones comunes a muchas aplicaciones. En nuestro caso, la etiqueta **div** también tiene una clase CSS: (**container**). De igual forma que con las clases de la etiqueta **header**, ésta clase tiene significado especial para Bootstrap.

Después del div, encontramos algo de Ruby embebido:

```
<%= link_to "sample app", '#', id: "logo" %>
<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><%= link_to "Home", '#' %></li>
    <li><%= link_to "Help", '#' %></li>
    <li><%= link_to "Log in", '#' %></li>
  </ul>
</nav>
```

Esto usa la función auxiliar de Rails **link_to** para crear enlaces (los cuales creamos directamente con la etiqueta ancla **a** en la Sección 3.2.2); el primer argumento de **link_to** es el texto del enlace, mientras que el segundo, es la URL. Rellenaremos las URLs con *rutas nombradas* en la Sección 5.3.3, pero por ahora utilizaremos las URLs neutras '`#`' comúnmente usadas en diseño web. El tercer argumento es un arreglo hash de opciones, en este caso agrega el *id* **logo** al enlace de la aplicación de ejemplo. (Los otros tres enlaces no tienen arreglo hash de opciones, lo cual está bien, puesto que éste argumento es opcional.) Las funciones auxiliares Rails a menudo aceptan arreglos hash para las opciones de esta forma, dandonos la flexibilidad de agregar opciones HTML arbitrarias sin dejar Rails.

El segundo elemento dentro de los *divs* es una lista de enlaces de navegación, que creamos mediante la etiqueta para *listas sin ordenamiento* **ul**, junto con la etiqueta para *elementos de una lista* **li**:

```
<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><%= link_to "Home", '#' %></li>
    <li><%= link_to "Help", '#' %></li>
```

```

<li><%= link_to "Log in", '#' %></li>
</ul>
</nav>

```

La etiqueta `<nav>`, aunque formalmente es innecesaria aquí, se utiliza para comunicar de forma más clara el propósito de las ligas de navegación. Mientras tanto, las clases `nav`, `navbar-nav`, y `navbar-right` de la etiqueta `ul` tienen significado especial para Bootstrap y adquirirán de forma automática su estilo cuando incluyamos el CSS de Bootstrap en la [Sección 5.1.2](#). Como puede verificar si examina el código en su navegador, una vez que Rails ha procesado la estructura de diseño y evaluado el código de Ruby embebido, la lista aparece como sigue:⁵

```

<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Home</a></li>
    <li><a href="#">Help</a></li>
    <li><a href="#">Log in</a></li>
  </ul>
</nav>

```

Este es el texto que será enviado a los navegadores.

La parte final de la estructura de diseño es un `div` para el contenido principal:

```

<div class="container">
  <%= yield %>
</div>

```

De igual forma que en el caso anterior, la clase `container` tiene significado especial para Bootstrap. Como aprendimos en la [Sección 3.4.3](#), el método `yield` inserta los contenidos de cada página en la estructura de diseño del sitio.

Aparte del pie de página que agregaremos en la [Sección 5.1.3](#), nuestra estructura de diseño está completa ahora, y podemos visualizar los resultados

⁵Los espacios pueden mostrarse ligeramente diferentes, lo cual está bien porque (como mencionamos en la [Sección 3.4.1](#)) HTML no es sensible a espacios en blanco.

visitando la página Home. Para aprovechar los elementos de estilo que vienen, agregaremos algunos elementos extra a la vista `home.html.erb` (Listado 5.2).

Listado 5.2: La página Home con un enlace a la página de registro.

`app/views/static_pages/home.html.erb`

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

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

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

<%= link_to image_tag("rails.png", alt: "Rails logo"),
           'http://rubyonrails.org/' %>
```

Preparándonos para agregar usuarios a nuestro sitio en el Capítulo 7, el primer `link_to` crea un enlace neutro de la forma

```
<a href="#" class="btn btn-lg btn-primary">Sign up now!</a>
```

En la etiqueta `div`, la clase CSS `jumbotron` tiene significado especial para Bootstrap, al igual que las clases `btn`, `btn-lg`, y `btn-primary` del botón de registro.

El segundo `link_to` utiliza la función auxiliar `image_tag`, que toma como argumentos la ruta a una imagen y, opcionalmente, un arreglo hash de opciones. En este caso asocia el atributo `alt` de la etiqueta imagen usando símbolos. Para que esto funcione, necesita haber una imagen llamada `rails.png`, que usted debería descargar de la página principal de Ruby on Rails <http://rubyonrails.org/images/rails.png> y colocarla en el directorio `app/assets/images/`. Si usted está utilizando el IDE en la nube u otro sistema tipo Unix, puede realizar esta tarea con ayuda de la utilería `curl` como sigue:⁶

⁶Si usted tiene Homebrew en OS X, puede instalar `curl` mediante `brew install curl`.

```
$ curl -O http://rubyonrails.org/images/rails.png  
$ mv rails.png app/assets/images/
```

Si el segundo comando falla, lo cual sucede a veces en el IDE en la nube por razones que no entiendo, intente ejecutar nuevamente el primer comando (`curl`) hasta que el archivo se descargue correctamente. (Para más información sobre `curl`, vea Capítulo 3 de *Conquering the Command Line*.) Al utilizar la función auxiliar `image_tag` en el Listado 5.2, Rails automáticamente encontrará las imágenes en el directorio `app/assets/images/` usando la cadena de procesos conectados (Sección 5.2).

Para hacer más claros los efectos de `image_tag`, echemos un vistazo al código HTML que produce:⁷

```

```

Aquí la cadena `9308b8f92fea4c19a3a0d8385b494526` (que puede diferir en su sistema) es añadida por Rails para asegurar que el nombre del archivo es único, lo que motiva a los navegadores a descargar las imágenes cuando éstas hayan sido actualizadas (en vez de cargarlas de su propio caché). Observe que el atributo `src` no incluye el directorio `images`, pero en su lugar utiliza el directorio `assets` que es común a todos los recursos (imágenes, JavaScript, CSS, etc.). En el servidor, Rails asocia las imágenes del directorio `assets` con el directorio correcto `app/assets/images`, pero en lo que concierne al navegador, todos los recursos parecen estar ubicados en el mismo directorio, lo que permite que sean despachados más rápidamente. Mientras tanto, el atributo `alt` es lo que será mostrado si la página es visitada por un programa que no puede desplegar imágenes (como los lectores de pantalla utilizados por las personas con discapacidad visual).

Ahora estamos listos por fin para ver los frutos de nuestra labor, como se muestra en la Figura 5.2. ¿Bastante decepcionante, diría usted? Quizá. Aunque por fortuna, hemos hecho un buen trabajo dando a nuestros elementos HTML

⁷Puede observar que la etiqueta `img`, en vez de verse como `...`, se muestra así: ``. Las etiquetas que siguen este patrón se conocen como etiquetas de *cierre automático*.

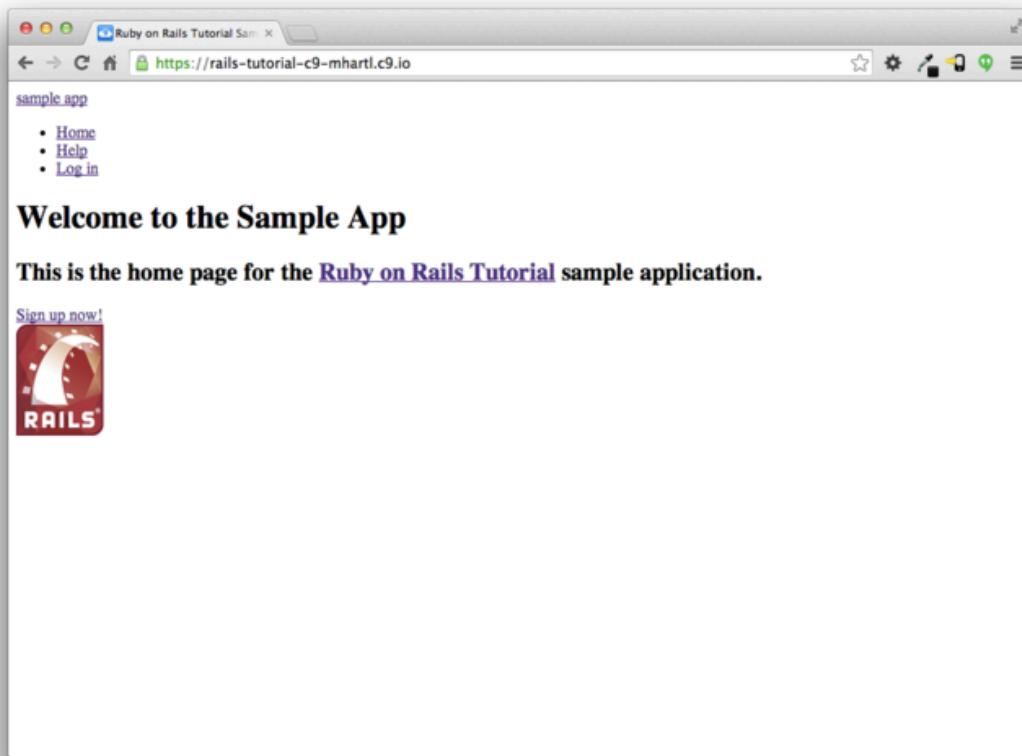


Figura 5.2: La página Home sin CSS personalizado.

clases adecuadas, lo que nos coloca en buena posición para agregar estilo al sitio con CSS.

5.1.2 Bootstrap y CSS personalizado

En la [Sección 5.1.1](#), asociamos muchos de los elementos HTML con clases CSS, lo que nos proporciona bastante flexibilidad para construir una estructura de diseño basada en CSS. Como mencionamos en la [Sección 5.1.1](#), muchas de estas clases son específicas de [Bootstrap](#), una biblioteca de Twitter que facilita agregar un diseño web agradable y elementos de interfaz de usuario a una aplicación HTML5. En esta sección, combinaremos Bootstrap con algunas reglas

CSS personalizadas para empezar a agregar algo de estilo a la aplicación de ejemplo. Vale la pena resaltar que al utilizar Bootstrap, el diseño de nuestra aplicación se convierte automáticamente en *responsivo*, asegurando así, que se despliega correctamente en un amplio rango de dispositivos.

Nuestro primer paso es añadir Bootstrap, lo cual puede lograrse en aplicaciones Rails mediante la gema `bootstrap-sass`, como se muestra en el Listado 5.3. La biblioteca Bootstrap utiliza de forma nativa el lenguaje `Less CSS` para crear hojas de estilo dinámicas, pero la cadena de procesos conectados soporta el (muy similar) lenguaje Sass por default (Sección 5.2), de forma que `bootstrap-sass` convierte Less a Sass y hace todos los archivos de Bootstrap que sean necesarios, disponibles a la aplicación actual.⁸

Listado 5.3: Agregando la gema `bootstrap-sass` al `Gemfile`.

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

gem 'rails',                  '4.2.2'
gem 'bootstrap-sass',          '3.2.0.0'
.
.
.
```

Para instalar Bootstrap, ejecute `bundle install` como es usual:

```
$ bundle install
```

Aunque `rails generate` crea automáticamente un archivo CSS para cada controlador, es sorprendentemente difícil incluirlos todos de forma apropiada y en el orden correcto, por lo que por simplicidad pondremos todo el CSS necesario para este tutorial en un solo archivo. El primer paso para poner CSS personalizado a funcionar, es crear el archivo CSS respectivo:

⁸También es posible utilizar Less con la cadena de procesos conectados; revise la gema `less-rails-bootstrap` para mayor detalle.

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

(Esto emplea el truco con el comando **touch** que mostramos en la Sección 3.3.3 con una ruta, pero puede crear el archivo de cualquier forma que guste.) Aquí tanto el nombre del directorio, como la extensión del archivo son importantes. El directorio

```
app/assets/stylesheets/
```

es parte de la cadena de procesos conectados (Sección 5.2), y cualquier hoja de estilo dentro de este directorio será incluída automáticamente como parte del archivo **application.css** incluido en la estructura de diseño del sitio. Más aún, el nombre de archivo **custom.css.scss** incluye la extensión **.css**, lo cual indica que es un archivo CSS, y la extensión **.scss**, que indica que es un archivo “Sass CSS” por lo que lo incluye dentro del flujo de información y lo procesa usando Sass. (No emplearemos Sass antes de la Sección 5.2.2, pero lo necesitamos ahora para que la gema **bootstrap-sass** realice su magia.)

Dentro del archivo para el CSS personalizado, podemos utilizar la función **@import** para incluir Bootstrap (junto con la utilería asociada Sprockets), como se muestra en el Listado 5.4.⁹

Listado 5.4: Agregando el CSS de Bootstrap.

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

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

Las dos líneas del Listado 5.4 incluyen la biblioteca CSS de Bootstrap completa. Luego de reiniciar el servidor web para incorporar los cambios en la aplicación de desarrollo (tecleando Ctrl-C y luego ejecutando **rails server** como en la Sección 1.3.2), los resultados aparecen como en la Figura 5.3. El lugar donde

⁹Si estos pasos parecen misteriosos, tenga confianza: Sólo estoy siguiendo las instrucciones del archivo **README** de **bootstrap-sass**.

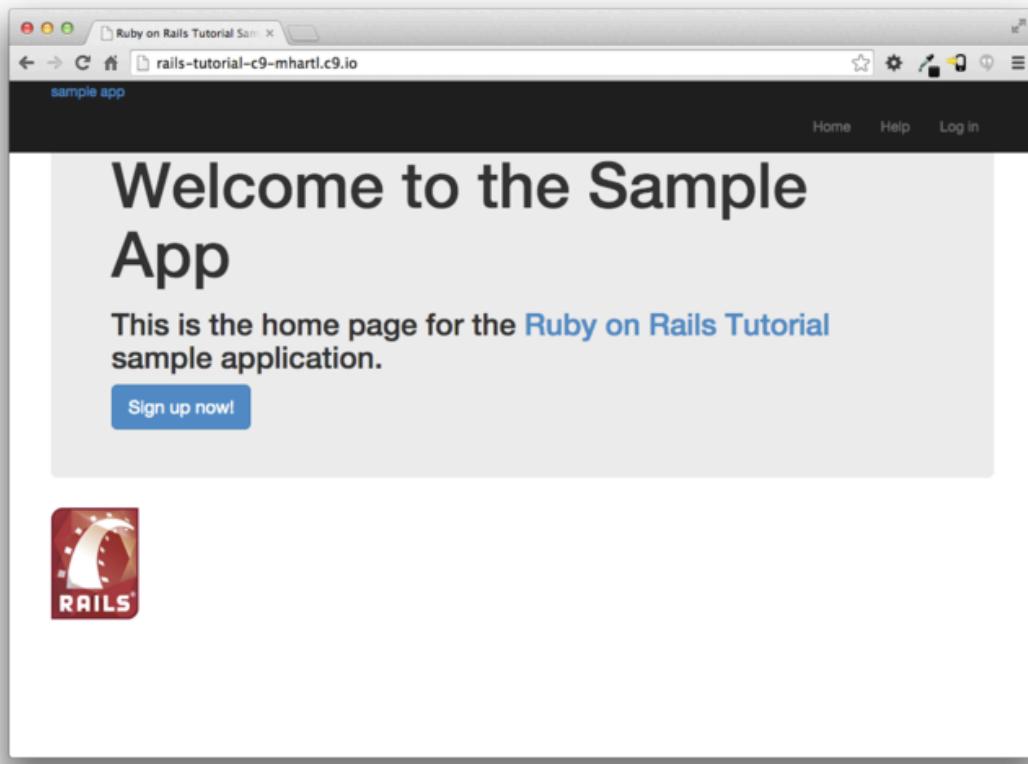


Figura 5.3: La aplicación de ejemplo con el CSS de Bootstrap.

aparece el texto no es bueno, y el logo no tiene ningún estilo, pero los colores y el botón de registro lucen prometedores.

A continuación agregaremos algo de CSS que será empleado en todo el sitio para dar estilo a la estructura general y a cada página individual, como se muestra en el [Listado 5.5](#). El resultado se muestra en la [Figura 5.4](#). (Hay algunas reglas en el [Listado 5.5](#); para tener idea de lo que una regla CSS hace, a menudo es útil comentarla usando los comentarios CSS, es decir, poniéndola dentro de `/* ... */`, y observar qué es lo que cambia.)

Listado 5.5: Agregando CSS para un estilo universal que se aplica a todas las páginas.

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

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

/* universal */

body
  padding-top: 60px;

section
  overflow: auto;

textarea
  resize: vertical;

.center
  text-align: center;

.center h1
  margin-bottom: 10px;
```

Observe que el CSS del [Listado 5.5](#) tiene una forma consistente. En general, las reglas CSS se refieren ya sea a una clase, a un “id”, a una etiqueta HTML, o a alguna combinación de éstos, seguida de una lista de comandos de estilo. Por ejemplo,

```
body {
  padding-top: 60px;
}
```

agrega 60 pixeles de relleno en la parte superior de la página. Gracias a la clase **navbar-fixed-top** de la etiqueta **header**, Bootstrap fija la barra de navegación en la parte superior de la página, de forma que el relleno sirve para separar el texto principal de la navegación. (Como el color por default de navbar cambió a partir de Bootstrap 2.0, necesitamos emplear la clase **navbar-inverse** para hacerla obscura en vez de clara.) Mientras tanto, el CSS de la regla

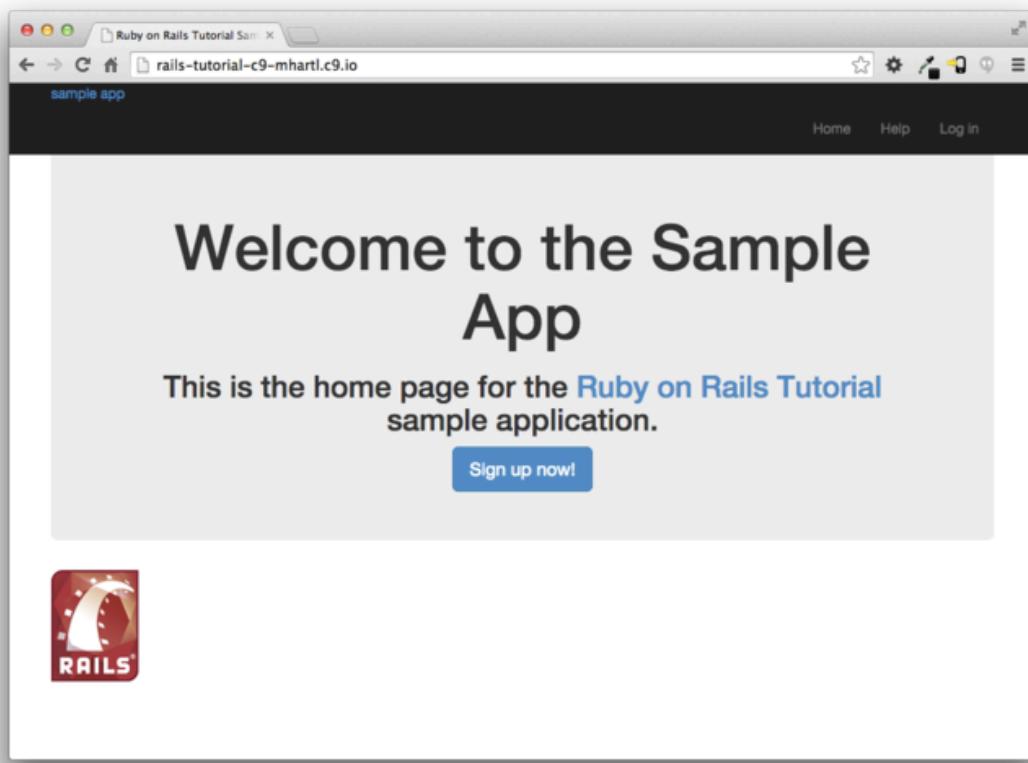


Figura 5.4: Agregando algo de espacio y otros estilos universales.

```
.center {  
    text-align: center;  
}
```

asocia la clase **center** con la propiedad **text-align: center**. En otras palabras, el punto `.` en `.center` indica que la regla se define para una clase. (Como veremos en el [Listado 5.7](#), el signo de número `#` identifica a una regla CSS que se define para un *id*.) Esto significa que los elementos dentro de cualquier etiqueta (como por ejemplo `div`) con clase **center** se mostrarán centrados en la página. (Vimos un ejemplo de esta clase en el [Listado 5.2](#).)

Aunque Bootstrap viene con reglas CSS para una tipografía agradable, agregaremos algunas reglas personalizadas para la apariencia del texto en nuestro sitio, como se muestra en el [Listado 5.6](#). (No todas estas reglas aplican para la página Home, pero cada regla será utilizada en algún momento en la aplicación de ejemplo.) El resultado del [Listado 5.6](#) se muestra en la [Figura 5.5](#).

Listado 5.6: Agregando CSS para una tipografía agradable.

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

```
@import "bootstrap-sprockets";  
@import "bootstrap";  
.  
. .  
./* typography */  
  
h1, h2, h3, h4, h5, h6 {  
    line-height: 1;  
}  
  
h1 {  
    font-size: 3em;  
    letter-spacing: -2px;  
    margin-bottom: 30px;  
    text-align: center;  
}  
  
h2 {  
    font-size: 1.2em;  
    letter-spacing: -1px;  
    margin-bottom: 30px;  
    text-align: center;
```

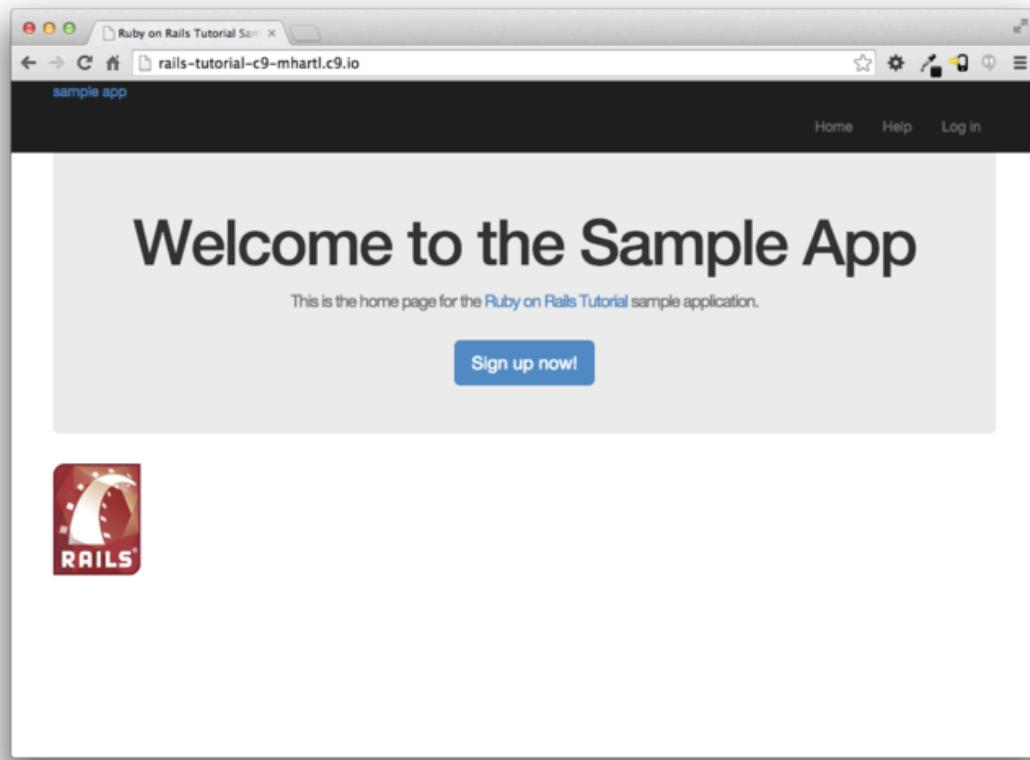


Figura 5.5: Agregando algo de estilo tipográfico.

```
font-weight: normal;
color: #777;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}
```

Finalmente, agregaremos algunas reglas para estilizar el logo del sitio, que por ahora consiste simplemente en el texto “sample app”. El CSS del [Listado 5.7](#) convierte el texto a mayúsculas y modifica su tamaño, color y ubicación. (Hemos utilizado un id para el CSS porque esperamos que el logo del

sitio aparezca sólo una vez por página, pero también se puede utilizar una clase si lo prefiere.)

Listado 5.7: Agregando CSS para el logo del sitio.*app/assets/stylesheets/custom.css.scss*

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

.

.

/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

Aquí **color: #fff** cambia el color del logo a blanco. Los colores HTML pueden codificarse con tres parejas de números en base-16 (hexadecimales), uno por cada color primario: rojo, verde y azul (en ese orden). El código **#ffffff** maximiza los tres colores, dando como resultado el blanco puro, y el código **#fff** es una abreviatura de **#ffffff**. El estándar CSS también define una gran cantidad de sinónimos para **colores HTML** comunes, incluyendo **white** para **#fff**. El resultado del Listado 5.7 se muestra en la Figura 5.6.

5.1.3 Parciales

Aunque la estructura de diseño del Listado 5.1 cumple su propósito, se está volviendo un poco desordenada. El HTML shim ocupa tres líneas y utiliza una

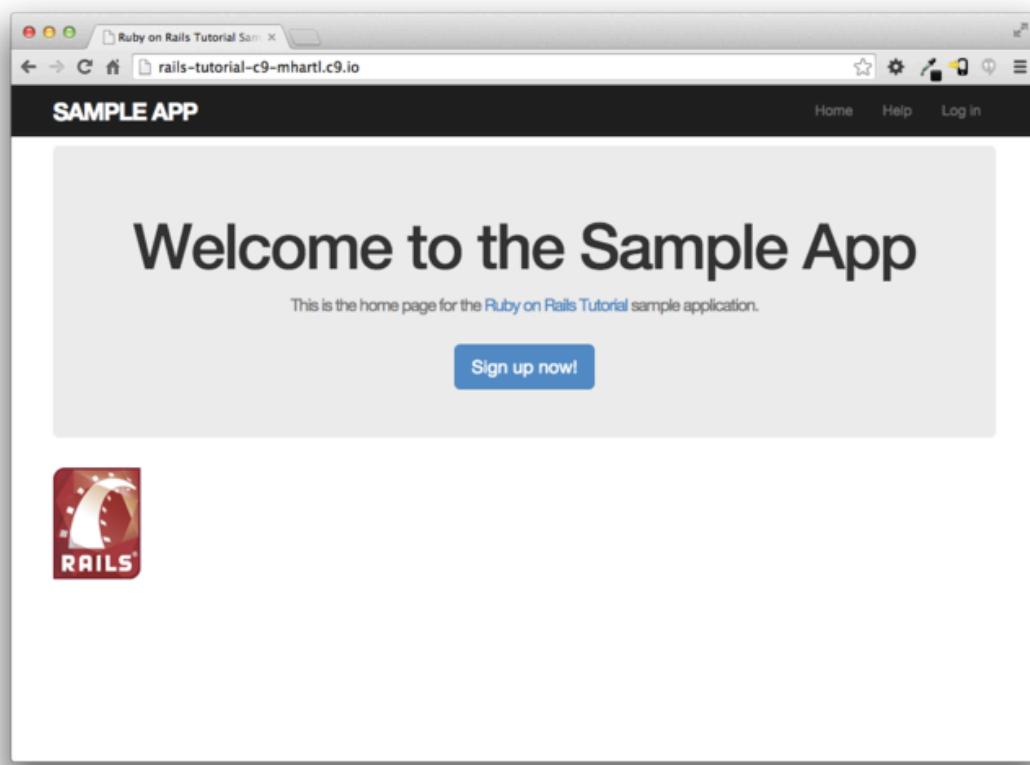


Figura 5.6: La aplicación de ejemplo con un logo estilizado.

sintaxis extraña específica de IE, por lo que sería conveniente aislarlo. Adicionalmente, el encabezado HTML conforma una unidad lógica, por lo que podemos agruparla en un solo lugar. La forma de lograr esto con Rails es mediante una característica llamada *parciales*. Primero echemos un vistazo a cómo se vería la estructura de diseño luego de definir los parciales (Listado 5.8).

Listado 5.8: La estructura de diseño del sitio con parciales para las hojas de estilo y el encabezado.

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag 'application', media: 'all',
                           'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

En el Listado 5.8, hemos reemplazado las líneas del HTML shim con una simple llamada a una función auxiliar de Rails llamado **render**:

```
<%= render 'layouts/shim' %>
```

El efecto de esta línea es buscar un archivo llamado **app/views/layouts/-_shim.html.erb**, evaluar su contenido, e insertar los resultados en la vista.¹⁰

¹⁰Muchos desarrolladores Rails utilizan un directorio **shared** para los parciales que se utilizan en diferentes vistas. Yo prefiero utilizar el directorio **shared** para los parciales de utilería que son utilizados en múltiples vistas, mientras que los parciales que se usan literalmente en todas las páginas (como parte de la estructura de diseño del sitio) los pongo en el directorio **layouts**. (Crearemos el directorio **shared** al inicio del Capítulo 7.) Me parece que esta es una división lógica, pero ubicarlos todos en el directorio **shared** ciertamente funciona también.

(Recuerde que `<%= ... %>` es la sintaxis de Ruby embebido necesaria para evaluar una expresión Ruby y luego insertar los resultados en la plantilla.) Observe el guión bajo al inicio del nombre de archivo `_shim.html.erb`; este guión bajo es la convención universal para nombrar parciales, y entre otras cosas permite identificar a todos los parciales de un directorio de un vistazo.

Por supuesto, para que el parcial funcione, tenemos que crear el archivo correspondiente y agregarle su contenido. En el caso del parcial shim, éste contiene sólo las tres líneas del [Listado 5.1](#). El resultado se muestra en el [Listado 5.9](#).

Listado 5.9: Un parcial para el HTML shim.

`app/views/layouts/_shim.html.erb`

```
<!--[if lt IE 9]>
<script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
</script>
<![endif]-->
```

Análogamente, podemos mover el código del encabezado al parcial que se muestra en el [Listado 5.10](#) e insertarlo en la estructura de diseño con otra llamada a `render`. (Como es usual con los parciales, usted debe crear el archivo manualmente usando su editor de texto.)

Listado 5.10: Un parcial para el encabezado del sitio.

`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", '#', id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", '#' %></li>
        <li><%= link_to "Help", '#' %></li>
        <li><%= link_to "Log in", '#' %></li>
      </ul>
    </nav>
  </div>
</header>
```

Ahora que sabemos cómo crear parciales, agreguemos un pie de página al sitio que acompañe al encabezado. En este punto usted probablemente adivine como lo llamaremos `_footer.html.erb` y lo pondremos en el directorio de `layouts` (Listado 5.11).¹¹

Listado 5.11: Un parcial para el pie de página del sitio.

`app/views/layouts/_footer.html.erb`

```
<footer class="footer">
  <small>
    The <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    by <a href="http://www.michaelhartl.com/">Michael Hartl</a>
  </small>
  <nav>
    <ul>
      <li><%= link_to "About", '#' %></li>
      <li><%= link_to "Contact", '#' %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

Como con el encabezado, en el pie de página hemos empleado `link_to` para los enlaces internos a las páginas About y Contact y hemos neutralizado las URLs con `'#'` por ahora. (De igual forma que `header`, la etiqueta `footer` es nueva en HTML5.)

Podemos desplegar el parcial del pie de página en la estructura de diseño siguiendo el mismo patrón que con las hojas de estilo y los parciales del encabezado (Listado 5.12).

Listado 5.12: La estructura de diseño del sitio con un parcial de pie de página.

`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
```

¹¹Puede que usted se pregunte porqué usamos tanto la etiqueta `footer` como la clase `.footer`. La respuesta es que la etiqueta tiene un significado claro para los lectores humanos, y la clase es utilizada por Bootstrap. Usar la etiqueta `div` en vez de `footer` también funciona.

```
<%= stylesheet_link_tag "application", media: "all",
                        "data-turbolinks-track" => true %>
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>
<%= csrf_meta_tags %>
<%= render 'layouts/shim' %>
</head>
<body>
  <%= render 'layouts/header' %>
  <div class="container">
    <%= yield %>
    <%= render 'layouts/footer' %>
  </div>
</body>
</html>
```

Por supuesto, el pie de página se verá feo sin algo de estilo (Listado 5.13). El resultado se muestra en la Figura 5.7.

Listado 5.13: Agregando el CSS para el pie de página del sitio.*app/assets/stylesheets/custom.css.scss*

```
.
.
.

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #777;
}

footer a {
  color: #555;
}

footer a:hover {
  color: #222;
}

footer small {
  float: left;
}

footer ul {
  float: right;
  list-style: none;
```

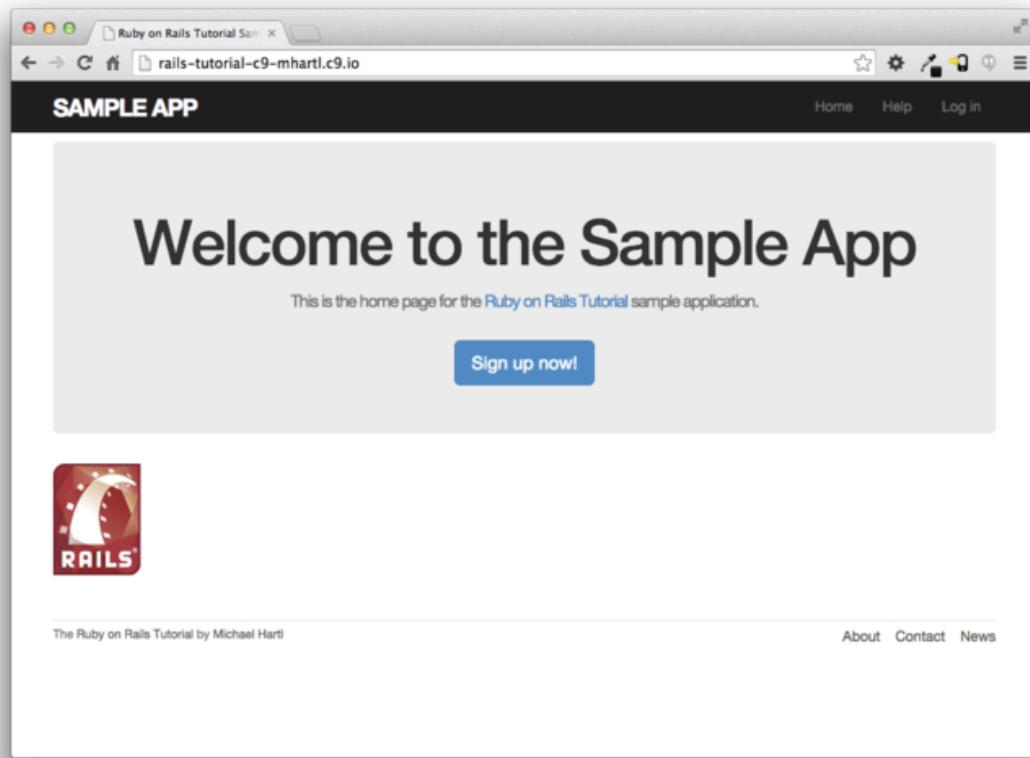


Figura 5.7: La página Home con el pie de página agregado.

```
}
```

```
footer ul li {
```

```
    float: left;
```

```
    margin-left: 15px;
```

```
}
```

5.2 Sass y la cadena de procesos conectados

Una de las adiciones más importantes en versiones recientes de Rails es la *cadena de procesos conectados*, que mejora significativamente la producción y

manejo de recursos estáticos tales como imágenes, CSS y JavaScript. Esta sección primero muestra una revisión general de la cadena de procesos conectados, y luego muestra cómo utilizar *Sass*, una poderosa herramienta para escribir CSS.

5.2.1 La cadena de procesos conectados

La cadena de procesos conectados involucra muchos cambios internos en Rails, pero desde la perspectiva de un desarrollador de Rails típico, existen tres características principales a entender: los directorios de recursos, los archivos manifiesto y los motores de preprocesamiento.¹² Revisémoslas una por una.

Directorio de recursos

En las versiones 3.0 de Rails y previas, los recursos estáticos vivían en el directorio **public/**, como se muestra:

- **public/stylesheets**
- **public/javascripts**
- **public/images**

Los archivos de estos directorios son despachados (aún luego de la versión 3.0) automáticamente mediante peticiones a `http://www.example.com/stylesheets`, etc.

En la última versión de Rails, hay *tres* directorios canónicos para recursos estáticos, cada uno con su propio propósito:

- **app/assets**: recursos específicos a la aplicación actual
- **lib/assets**: recursos de bibliotecas escritas por su propio equipo de desarrollo

¹²La estructura de esta sección está basada en el excelente post “La cadena de procesos conectados de Rails 3 en (casi) 5 Minutos” del blog de Michael Erasmus. Para más detalle, consulte [Guía de Rails sobre la cadena de procesos conectados](#).

- **vendor/assets**: recursos de proveedores terceros

Como puede intuir, cada uno de estos directorios tiene un subdirectorio para cada tipo de recurso, por ejemplo,

```
$ ls app/assets/
images/  javascripts/  stylesheets/
```

En este momento, podemos entender la motivación detrás de la ubicación del archivo CSS personalizado de la Sección 5.1.2: `custom.css.scss` es específico a la aplicación de ejemplo, por lo tanto debe ubicarse en `app/assets/stylesheets`.

Archivos manifiesto

Una vez que usted ha colocado sus recursos en sus ubicaciones correctas, puede emplear los *archivos manifiesto* para indicarle a Rails (via la gema [Sprockets](#)) cómo combinarlos para formar un solo archivo. (Esto aplica para CSS y JavaScript pero no para imágenes.) Por ejemplo, echemos un vistazo al archivo manifiesto para las hojas de estilo de la aplicación ([Listado 5.14](#)).

Listado 5.14: El archivo manifiesto CSS específico para la aplicación.

`app/assets/stylesheets/application.css`

```
/*
 * This is a manifest file that'll be compiled into application.css, which
 * will include all the files listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets,
 * vendor/assets/stylesheets, or vendor/assets/stylesheets of plugins, if any,
 * can be referenced here using a relative path.
 *
 * You're free to add application-wide styles to this file and they'll appear
 * at the bottom of the compiled file so the styles you add here take
 * precedence over styles defined in any styles defined in the other CSS/SCSS
 * files in this directory. It is generally better to create a new file per
 * style scope.
 *
 *= require_tree .
 *= require_self
 */
```

Las líneas clave están comentadas como CSS, pero son utilizadas por Sprockets para incluir los archivos como tal:

```
/*
.
.
.
*= require_tree .
*= require_self
*/
```

Aquí

```
*= require_tree .
```

asegura que todos los archivos CSS del directorio **app/assets/stylesheets** (incluyendo sus subdirectorios) serán incluidos en el CSS de la aplicación. La línea

```
*= require_self
```

especifica el lugar en la secuencia de carga, en que el archivo CSS **application.css** será considerado.

Rails viene con archivos manifiesto por default, y en el *Tutorial de Rails* no necesitaremos realizar ningún cambio, pero en la [Guía de Rails sobre la cadena de procesos conectados](#) encontrará mayor detalle, si usted lo requiere.

Motores de preprocesamiento

Luego de que usted haya terminado de elaborar sus recursos, Rails los prepara para la plantilla del sitio ejecutando sobre ellos varios motores de preprocesamiento y utilizando los archivos manifiesto para combinarlos antes de entregarlos al navegador. Le indicamos a Rails qué procesador usar a través de la extensión del nombre del archivo; los tres casos más comunes son **.scss** para Sass, **.coffee** para CoffeeScript, y **.erb** para Ruby embebido (ERb).

Revisamos ERb por primera vez en la [Sección 3.4.3](#), y revisaremos Sass en la [Sección 5.2.2](#). No necesitaremos CoffeeScript en este tutorial, pero es un pequeño lenguaje elegante que compila para generar JavaScript. (El [RailsCast sobre los fundamentos de CoffeeScript](#) es un buen lugar para comenzar.)

Los motores de preprocesamiento pueden estar encadenados, de forma que **foobar.js.coffee** es procesado por el procesador de CoffeeScript, y **foobar.js.erb.coffee** es procesado por CoffeeScript y ERb (el orden va de derecha a izquierda, por lo que primero pasa por CoffeeScript).

Eficiencia en Producción

Una de las mejores cosas acerca de la cadena de procesos conectados es que automáticamente produce recursos optimizados para que sean eficientes en una aplicación de producción. Los métodos tradicionales para organizar CSS y JavaScript involucran dividir la funcionalidad en archivos separados y usar un buen formato en el código (con mucha indentación). Mientras que esto es conveniente para el programador, es ineficiente en producción. En particular, incluir muchos archivos puede decrementar significativamente los tiempos de carga de una página, que es uno de los factores más importantes que afectan la calidad de la experiencia de usuario. Con la cadena de procesos conectados, no necesitamos escoger entre velocidad y conveniencia: podemos trabajar con múltiples archivos con un formato elegante para desarrollo, y luego utilizarlos en la cadena de procesos conectados para hacerlos eficientes en producción. En particular, la cadena de procesos conectados combina todas las hojas de estilo de la aplicación en un solo archivo CSS (**application.css**), y combina todo el JavaScript de la aplicación en un solo archivo JavaScript (**application.js**) y luego los *minimiza* para remover todo el espacio e indentaciones innecesarias que abultan el tamaño del archivo. El resultado es lo mejor de los dos mundos: conveniencia en desarrollo y eficiencia en producción.

5.2.2 Hojas de estilo sintácticamente impresionantes

Sass es un lenguaje para escribir hojas de estilo que mejora CSS en varias formas. En esta sección, revisaremos dos de las mejoras más importantes, *anidado* y *variables*. (Una tercera técnica, *mezclas*, es ligeramente revisada en la Sección 7.1.1.)

Como observamos brevemente en la Sección 5.1.2, Sass soporta el formato llamado SCSS (indicado con la extensión de archivo `.scss`), que es estrictamente un super-conjunto de CSS; es decir, SCSS únicamente *agrega* características a CSS, más que definir por completo una nueva sintaxis.¹³ Esto significa que todo archivo CSS válido también es un archivo SCSS válido, lo cual es conveniente para proyectos que ya tienen reglas de estilo definidas. En nuestro caso, utilizamos SCSS desde el principio con la finalidad de aprovechar Bootstrap. Puesto que la cadena de procesos conectados de Rails automáticamente utiliza Sass para procesar archivos con la extensión `.scss`, el archivo `custom.css.scss` será procesado por el preprocesador Sass antes de ser empacado para su entrega al navegador.

Anidado

Un patrón común en las hojas de estilo es tener reglas que apliquen a elementos anidados. Por ejemplo, en el Listado 5.5 tenemos reglas tanto para `.center` como para `.center h1`:

```
.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

Podemos reemplazar esto en Sass con

¹³El viejo formato `.sass`, también soportado por Sass, define un nuevo lenguaje simplificado (y tiene menos llaves) pero es menos conveniente para los proyectos ya existentes y es más difícil de aprender para los que ya están familiarizados con CSS.

```
.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}
```

Aquí la regla **h1** anidada, automáticamente hereda el contexto **.center**.

Hay un segundo candidato para anidado que requiere una sintaxis ligeramente diferente. En el [Listado 5.7](#), tenemos el código

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

Aquí el id del logo: **#logo** aparece dos veces, una por sí solo y otra con el atributo **hover** (que controla su apariencia cuando el puntero del ratón se encuentra sobre el elemento en cuestión). Con la finalidad de anidar la segunda regla, necesitaremos una referencia al elemento padre: **#logo**; en SCSS, esto se logra usando el carácter ampersand **&** como sigue:

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}
```

```
&:hover {
  color: #fff;
  text-decoration: none;
}
```

Sass cambia `&:hover` a `#logo:hover` como parte de su conversión de SCSS a CSS.

Estas dos técnicas aplican para el CSS del pie de página del Listado 5.13, que puede transformarse en lo siguiente:

```
footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #777;
  a {
    color: #555;
    &:hover {
      color: #222;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 15px;
    }
  }
}
```

Convertir el Listado 5.13 manualmente es un buen ejercicio, y debería verificar que el CSS aún funciona adecuadamente luego de la conversión.

Variables

Sass nos permite definir *variables* para eliminar duplicados y escribir un código más expresivo. Por ejemplo, si observa los Listados 5.6 y 5.13, veremos que hay repetidas referencias a un mismo color:

```
h2 {  
  .  
  .  
  .  
  color: #777;  
}  
.  
.  
.  
footer {  
  .  
  .  
  .  
  color: #777;  
}
```

En este caso, **#777** es un gris claro, y le podemos dar nombre definiendo una variable como se indica:

```
$light-gray: #777;
```

Esto nos permite re-escribir nuestro SCSS como sigue:

```
$light-gray: #777;  
.  
.  
.  
h2 {  
  .  
  .  
  .  
  color: $light-gray;  
}  
.  
.  
.  
footer {  
  .  
  .  
  .  
  color: $light-gray;  
}
```

Como los nombres de variables tales como `$light-gray` son más descriptivos que `#777`, a menudo es útil definir variables aún para valores que no se repiten. De hecho, la biblioteca Bootstrap define una gran cantidad de variables para los colores, puede consultarla en línea en la [página de Bootstrap para variables Less](#). Esa página define variables usando Less, no Sass, pero la gema `bootstrap-sass` proporciona los equivalentes en Sass. No es difícil adivinar la correspondencia; donde Less usa un signo arroba `@`, Sass usa un signo de dólar `$`. Echando un vistazo a la página de variables de Bootstrap, veremos que existe una variable para el gris claro:

```
@gray-light: #777;
```

Esto significa que, mediante la gema `bootstrap-sass`, debería haber una variable SCSS correspondiente a `$gray-light`. Podemos usar esto para reemplazar nuestra variable personalizada, `$light-gray`, lo que produce

```
h2 {
  .
  .
  .
  color: $gray-light;
}

.
.
.
footer {
  .
  .
  .
  color: $gray-light;
}
```

Aplicando el anidado de Sass y las características para definir variables al archivo SCSS obtenemos el archivo del [Listado 5.15](#). Esto emplea tanto variables Sass (como se infiere de la página de variables Less de Bootstrap) como nombres de colores preconstruídos (por ejemplo, `white` para `#fff`). Observe en particular la dramática mejoría en las reglas para la etiqueta `footer`.

Listado 5.15: El archivo SCSS ya convertido utilizando anidado y variables.

app/assets/stylesheets/custom.css.scss

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

/* mixins, variables, etc. */

$gray-medium-light: #eaeaea;

/* universal */

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}

/* typography */

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: $gray-light;
```

```
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}

/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: white;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  &:hover {
    color: white;
    text-decoration: none;
  }
}

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid $gray-medium-light;
  color: $gray-light;
  a {
    color: $gray;
    &:hover {
      color: $gray-darker;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 15px;
    }
  }
}
```

Sass nos proporciona más formas de simplificar nuestras hojas de estilo, pero el código del [Listado 5.15](#) usa las características más importantes y nos proporciona un buen comienzo. Consulte el [sitio web Sass](#) para más detalles.

5.3 Enlaces de la estructura de diseño

Ahora que hemos terminado la estructura de diseño del sitio con un estilo decente, es tiempo de empezar a llenar los enlaces que neutralizamos con '#'. Por supuesto, podemos poner en el código enlaces como

```
<a href="/static_pages/about">About</a>
```

pero ésta no es la forma en que Rails funciona. Sería bueno si la URL de la página about fuera /about en vez de /static_pages/about. Más aún, Rails utiliza por convención *rutas nombradas*, que involucran código como

```
<%= link_to "About", about_path %>
```

De esta forma el código tiene un significado más transparente, y también es más flexible puesto que podemos cambiar la definición de `about_path` y tener el cambio disponible en todas las lugares donde `about_path` es utilizado.

La lista completa de nuestros enlaces aparece en la [Tabla 5.1](#), junto con sus correspondientes URLs y rutas. Nos encargamos de la primer ruta en la [Sección 3.4.4](#), y para el final de este capítulo habremos implementado todas excepto la última. (la cual elaboraremos en el [Capítulo 8](#).)

5.3.1 Página de Contacto

Por completez, agregaremos la página de Contacto, que se dejó como ejercicio del [Capítulo 3](#). La prueba aparece en el [Listado 5.16](#), la cual simplemente sigue el modelo que vimos en el [Listado 3.22](#).

Page	URL	Named route
Home	/	<code>root_path</code>
About	/about	<code>about_path</code>
Help	/help	<code>help_path</code>
Contact	/contact	<code>contact_path</code>
Sign up	/signup	<code>signup_path</code>
Log in	/login	<code>login_path</code>

Tabla 5.1: Correspondencia entre rutas y URLs para los enlaces del sitio.

Listado 5.16: Una prueba para la página de Contacto. ROJO

```
test/controllers/static_pages_controller_test.rb

require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end

  test "should get contact" do
    get :contact
    assert_response :success
    assert_select "title", "Contact | Ruby on Rails Tutorial Sample App"
  end
end
```

En este momento, las pruebas del [Listado 5.16](#) deberían estar en ROJO:

Listado 5.17: ROJO

```
$ bundle exec rake test
```

Los pasos a seguir son similares a los que dimos para agregar la página About de la Sección 3.3: primero actualizamos las rutas (Listado 5.18), luego agregamos una acción **contact** al controlador de Páginas Estáticas (Listado 5.19), y finalmente creamos una vista (Listado 5.20).

Listado 5.18: Agregando una ruta para la página de Contacto. ROJO

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'static_pages/help'
  get 'static_pages/about'
  get 'static_pages/contact'
end
```

Listado 5.19: Agregando una acción para la página de Contacto. ROJO

app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController
  .
  .
  .
  def contact
  end
end
```

Listado 5.20: La vista para la página de Contacto. VERDE

app/views/static_pages/contact.html.erb

```
<% provide(:title, 'Contact') %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="http://www.railstutorial.org/#contact">contact page</a>.
</p>
```

Ahora asegúrese de que las pruebas están en **VERDE**:

Listado 5.21: **VERDE**

```
$ bundle exec rake test
```

5.3.2 Rutas Rails

Para agregar las rutas nombradas a las páginas estáticas de la aplicación de ejemplo, editaremos el archivo de rutas, **config/routes.rb**, que Rails utiliza para definir las asignaciones de las URLs. Empezaremos por revisar la ruta para la página Home (definida en la [Sección 3.4.4](#)), que es un caso especial, y luego definiremos un conjunto de rutas para las páginas estáticas restantes.

Hasta ahora, hemos visto tres ejemplos de cómo definir una ruta raíz, empezando con el código

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

de la aplicación “hello” ([Listado 1.10](#)), el código

```
root 'users#index'
```

de la aplicación de juguete ([Listado 2.3](#)), y el código

```
root 'static_pages#home'
```

de la aplicación de ejemplo ([Listado 3.37](#)). En cada caso, el método **root** se encarga de que la ruta raíz / sea dirigida al controlador y la acción de nuestra elección. Definir la ruta raíz de esta forma tiene un segundo efecto importante, que es crear rutas nombradas que nos permiten referirnos a ellas por nombre en vez de la URL concreta. En este caso, estas rutas son **root_path** y **root_url**, con la única diferencia de que la última incluye la URL completa:

```
root_path -> '/'
root_url  -> 'http://www.example.com/'
```

En el *Tutorial de Rails*, seguiremos la convención común de utilizar el formato `_path` excepto cuando hagamos redireccionamiento, en la que utilizaremos el formato `_url`. (Esto es porque el estándar HTTP standard técnicamente requiere una URL completa luego de redireccionar, aunque en la mayoría de los navegadores funcionará de ambas formas.)

Para definir las rutas nombradas para las páginas Help, About y Contact, necesitamos hacer cambios a las reglas `get` del [Listado 5.18](#), transformándolas en líneas como

```
get 'static_pages/help'
```

a

```
get 'help' => 'static_pages#help'
```

El segundo de estos patrones dirige una petición GET para la URL /help a la acción `help` en el controlador de páginas estáticas, de forma que podemos utilizar la URL /help en vez de /static_pages/help, que es más larga de escribir. Como con la regla para la ruta raíz, esto genera dos rutas nombradas, `help_path` y `help_url`:

```
help_path -> '/help'
help_url  -> 'http://www.example.com/help'
```

Aplicando este cambio de regla a lo que resta de las rutas de las páginas estáticas del [Listado 5.18](#) obtenemos el [Listado 5.22](#).

Listado 5.22: Rutas para las páginas estáticas.

```
config/routes.rb
```

```
Rails.application.routes.draw do
  root      'static_pages#home'
  get 'help'   => 'static_pages#help'
  get 'about'  => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
end
```

5.3.3 Usando rutas nombradas

Con las rutas definidas en el [Listado 5.22](#), estamos ahora en posición de usar las rutas nombradas resultantes en la estructura de diseño del sitio. Esto simplemente involucra el llenar los segundos argumentos de las funciones `link_to` con las rutas nombradas correspondientes. Por ejemplo, convertiremos

```
<%= link_to "About", '#' %>
```

a

```
<%= link_to "About", about_path %>
```

y así sucesivamente.

Empezaremos con el parcial del encabezado, `_header.html.erb` ([Listado 5.23](#)), que tiene enlaces a las páginas Home y Help. Mientras nos ocupamos de esto, seguiremos una convención web común y pondremos un enlace a la página Home en el logo también.

Listado 5.23: El parcial del encabezado con enlaces.

```
app/views/layouts/_header.html.erb
```

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
```

```

<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><%= link_to "Home", root_path %></li>
    <li><%= link_to "Help", help_path %></li>
    <li><%= link_to "Log in", '#' %></li>
  </ul>
</nav>
</div>
</header>

```

No tendremos una ruta nombrada para el enlace “Log in” sino hasta el Capítulo 8, por lo que lo dejaremos como ‘#’ por ahora.

El otro lugar con enlaces es el parcial del pie de página, `_footer.html.erb`, que tiene enlaces para las páginas About y Contact (Listado 5.24).

Listado 5.24: El parcial del pie de página con enlaces.

`app/views/layouts/_footer.html.erb`

```

<footer class="footer">
  <small>
    The <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    by <a href="http://www.michaelhartl.com/">Michael Hartl</a>
  </small>
  <nav>
    <ul>
      <li><%= link_to "About", about_path %></li>
      <li><%= link_to "Contact", contact_path %></li>
      <li><a href="http://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>

```

Con esto, nuestra estructura de diseño tiene ligas para todas las páginas estáticas creadas en el Capítulo 3, por lo que, por ejemplo, `/about` se dirige a la página About (Figura 5.8).

5.3.4 Pruebas a los enlaces de la estructura de diseño

Ahora que hemos rellenado varios de los enlaces en la estructura de diseño, es prudente probarlos para asegurarnos de que están funcionando correctamente. Podríamos hacerlo manualmente con un navegador, primero visitando la ruta

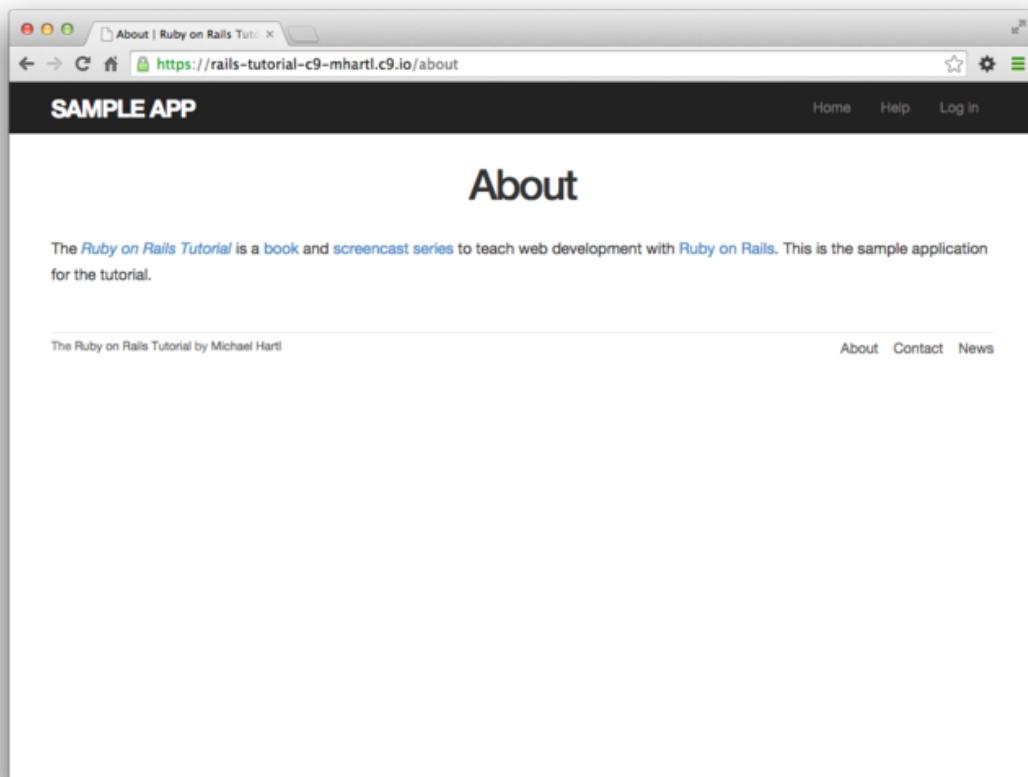


Figura 5.8: La página About en [/about](#).

raíz y luego verificando cada uno de los los enlaces, pero esto se vuelve incómodo rápidamente. En vez de esto, simularemos la misma serie de pasos utilizando una *prueba de integración*, que nos permite escribir una prueba de principio a fin del comportamiento de nuestra aplicación. Podemos empezar generando una prueba de la plantilla, que llamaremos `site_layout`:

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

Observe que el generador de Rails automáticamente agrega el sufijo `_test` al nombre del archivo de prueba.

Nuestro plan para probar los enlaces de la estructura de diseño involucran verificar la estructura HTML del sitio:

1. Visitar la ruta raíz (Home page)
2. Verificar que se muestra la página correcta
3. Verificar que los enlaces a las páginas Home, Help, About y Contact están correctos

El [Listado 5.25](#) muestra cómo podemos utilizar las pruebas de integración de Rails para traducir estos pasos a código, empezando con el método `assert_template` para verificar que la página Home se muestra utilizando la vista correcta.¹⁴

¹⁴ Algunos desarrolladores insisten en que una sola prueba no debe contener múltiples aserciones. Yo creo que esta práctica puede ser innecesariamente complicada, al mismo tiempo que generamos una sobrecarga si existen tareas comunes al inicio de cada test. Adicionalmente, una prueba bien escrita cuenta una historia coherente, y si la dividimos en piezas, se interrumpe la narrativa. Es por ello que tengo una fuerte preferencia por incluir múltiples aserciones en una prueba, confiando en que Ruby (via minitest) me indique exactamente en qué línea han fallado mis aserciones, si es que sucede.

Listado 5.25: Una prueba para los enlaces en la estructura de diseño. VERDE
test/integration/site_layout_test.rb

```
require 'test_helper'

class SiteLayoutTest < ActionDispatch::IntegrationTest

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

El [Listado 5.25](#) utiliza algunas de las opciones más avanzadas del método `assert_select`, que vimos en los Listados [3.22](#) y [5.16](#). En este caso, utilizamos una sintaxis que nos permite probar la presencia de un enlace particular—una combinación que especifica el nombre de la etiqueta `a` y su atributo `href`, como en

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

Aquí Rails automáticamente inserta el valor de `about_path` en lugar del signo de interrogación (escapando cualquier carácter especial si es necesario), verificando de esta forma la etiqueta HTML de la forma

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

Observe que la aserción para la ruta raíz verifica que hay *dos* de tales enlaces (una para el logo y otra para el elemento del menú de navegación):

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

Esto asegura que ambas ligas a la página Home definidas en el [Listado 5.23](#) están presentes.

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

Tabla 5.2: Algunos usos de `assert_select`.

Algunos otros usos de `assert_select` aparecen en la Tabla 5.2. Mientras que `assert_select` es flexible y poderoso (tiene muchas más opciones que las que mostramos aquí), la experiencia muestra que es prudente adoptar una estrategia que únicamente pruebe los elementos HTML (tales como los enlaces de la estructura de diseño del sitio) que son más improbables que cambien con el tiempo.

Para verificar que la nueva prueba del Listado 5.25 pasa, podemos ejecutar sólo las pruebas de integración mediante la siguiente tarea Rake:

Listado 5.26: VERDE

```
$ bundle exec rake test:integration
```

Si todo salió bien, debería ejecutar el conjunto de pruebas completo para verificar que todas las pruebas están en VERDE:

Listado 5.27: VERDE

```
$ bundle exec rake test
```

Con la prueba de integración que agregamos para los enlaces de la estructura de diseño, estamos en buena posición para atrapar regresiones rápidamente usando nuestro conjunto de pruebas.

5.4 Registro de usuario: primer paso

Como toque final a nuestro trabajo en la estructura de diseño y enrutamiento, en esta sección crearemos una ruta para la página de registro, lo que significa que crearemos un segundo controlador en el camino. Este es un primer paso importante encaminado a permitir que los usuarios se registren en nuestro sitio; daremos un siguiente paso, modelando usuarios, en el [Capítulo 6](#), y terminaremos el trabajo en el [Capítulo 7](#).

5.4.1 Controlador de usuarios

Creamos nuestro primer controlador, el controlador de páginas estáticas, en la [Sección 3.2](#). Es hora de crear el segundo, el controlador de usuarios. De igual forma que antes, utilizaremos **generate** para crear el controlador más simple que cumpla con nuestras necesidades actuales, digamos, uno con una página de registro simple para los usuarios nuevos. Siguiendo la [arquitectura REST](#) convencional favorecida por Rails, llamaremos la acción para usuarios nuevos **new**, que podemos crear automáticamente pasando **new** como argumento de **generate**. El resultado se muestra en el [Listado 5.28](#).

Listado 5.28: Generando un controlador Users (con una acción **new**).

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

Como es requerido, el Listado 5.28 crea un controlador Users con una acción **new** (Listado 5.30) y una vista de usuario simple (Listado 5.31). También genera una prueba mínima para la nueva página de usuario (Listado 5.32), que en este momento debería pasar:

Listado 5.29: VERDE

```
$ bundle exec rake test
```

Listado 5.30: El controlador Users inicial, con una acción **new**.

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController

  def new
  end
end
```

Listado 5.31: La vista **new** inicial para Users.

```
app/views/users/new.html.erb
```

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

Listado 5.32: Una prueba para la nueva página de usuario. VERDE

```
test/controllers/users_controller_test.rb
```

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase

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

5.4.2 URL de Registro

Con el código de la Sección 5.4.1, ya tenemos una página funcional para los nuevos usuarios en /users/new, pero recuerde de la Tabla 5.1 que queremos que la URL sea /signup en vez de la otra. Seguiremos los ejemplos del Listado 5.22 y agregaremos una regla `get '/signup'` para la URL de registro, como se muestra en el Listado 5.33.

Listado 5.33: Una ruta para la página de registro.

`config/routes.rb`

```
Rails.application.routes.draw do
  root      'static_pages#home'
  get 'help'   => 'static_pages#help'
  get 'about'  => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup'  => 'users#new'
end
```

A continuación, usaremos la recién definida ruta nombrada para agregar el enlace adecuado al botón de la página Home. De igual forma que con las otras rutas, `get 'signup'` automáticamente nos proporciona una ruta nombrada `signup_path`, que podemos utilizar en el Listado 5.34. Agregar una prueba para la página de registro se deja como ejercicio (Sección 5.6.)

Listado 5.34: Enlazando el botón con la página de registro.

`app/views/static_pages/home.html.erb`

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

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

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

<%= link_to image_tag("rails.png", alt: "Rails logo"),
           'http://rubyonrails.org/' %>
```

Finalmente, agregaremos una vista personalizada simple para la página de registro ([Listado 5.35](#)).

Listado 5.35: La página inicial (simple) de registro.

app/views/users/new.html.erb

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

Hecho esto, hemos terminado con los enlaces y las rutas nombradas, al menos hasta que agreguemos una ruta para entrar al sistema ([Capítulo 8](#)). La nueva página de usuario resultante (ubicada en la URL /signup) aparece en la [Figura 5.9](#).

5.5 Conclusión

En este capítulo, hemos trabajado nuestra estructura de diseño para darle forma y pulir las rutas. El resto del libro se dedica a profundizar en la aplicación de ejemplo: primero, agregando usuarios que puedan registrarse, entrar y salir del sistema; y luego, agregando microposts de los usuarios; y finalmente, agregando la funcionalidad para seguir a otros usuarios.

En este punto, si usted está usando Git, debería integrar sus cambios en la rama principal:

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

Y luego subirlos a Bitbucket:

```
$ git push
```

Finalmente, despliegue en Heroku:

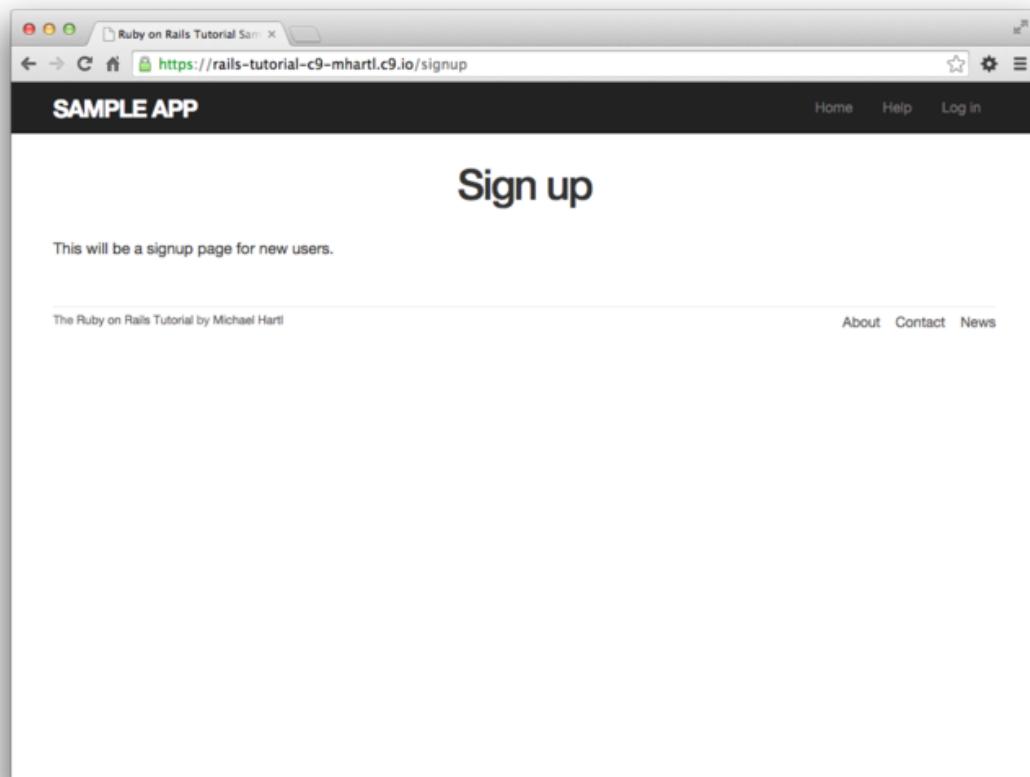


Figura 5.9: La nueva página de registro en [/signup](#).

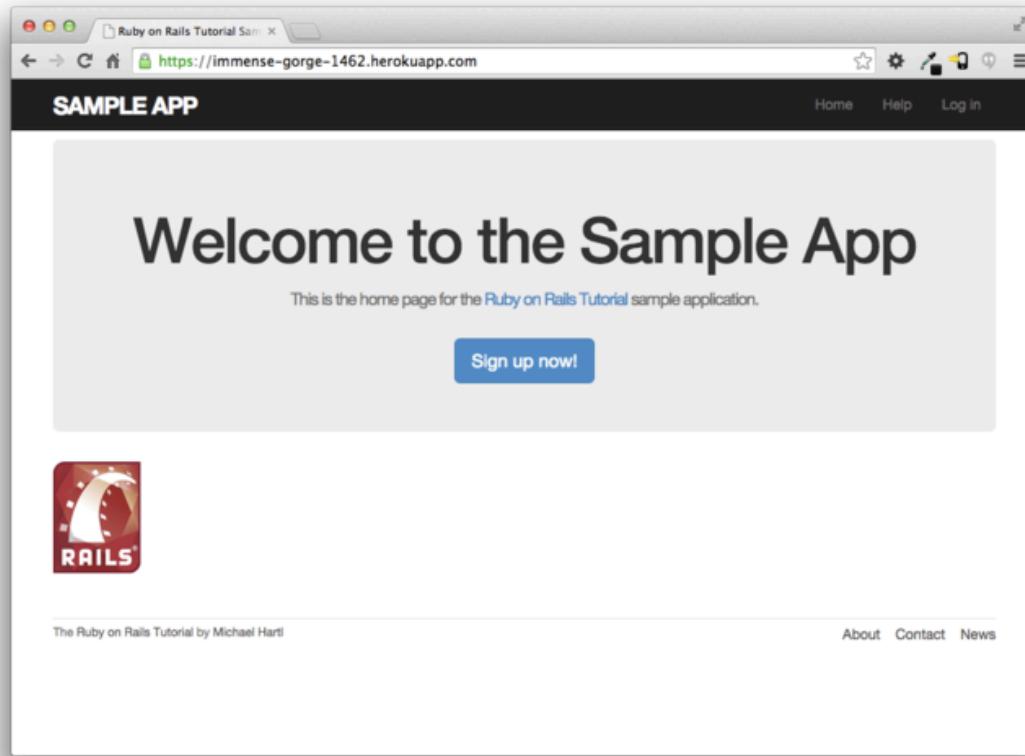


Figura 5.10: La aplicación de ejemplo en producción.

```
$ git push heroku
```

El resultado del despliegue debería ser una aplicación de ejemplo funcionando en el servidor de producción ([Figura 5.10](#)).

5.5.1 Qué aprendimos en este capítulo

- Usando HTML5, podemos definir una estructura de diseño para un sitio con logo, encabezado, pie de página y contenido principal en el cuerpo.

- Los parciales de Rails son utilizados para poner código de la vista en un archivo separado por conveniencia.
- CSS nos permite agregar estilo a la estructura de diseño del sitio basado en clases CSS y *ids*.
- La biblioteca Bootstrap nos facilita crear un sitio con un diseño agradable de forma rápida.
- Sass y la cadena de procesos conectados nos permiten eliminar repetición en nuestro CSS al mismo tiempo que empaqueta los resultados para hacerlos eficientes en producción.
- Rails nos permite definir reglas de enrutamiento personalizadas, y de esta forma nos proporciona rutas nombradas.
- Las pruebas de integración simulan de forma efectiva la interacción del usuario con el navegador de página a página.

5.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Como se sugiere en la Sección 5.2.2, realice los pasos para convertir manualmente el CSS del pie de página del Listado 5.13 a SCSS como se muestra en el Listado 5.15.
2. En la prueba de integración del Listado 5.25, agregue código para visitar la página de registro usando el método **get** y verifique que el título de la página resultante es correcto.

3. Es conveniente utilizar la función auxiliar `full_title` en las pruebas incluyendo la función auxiliar de la aplicación en la función auxiliar de prueba, como se muestra en el [Listado 5.36](#). Luego podemos probar que el título sea correcto usando código similar al del [Listado 5.37](#) (que extiende la solución del ejercicio anterior). Sin embargo, esto es frágil porque cualquier error al escribir el título base (tal como “Ruby on Rails Tutoial”) no será detectado por el conjunto de pruebas. Arregle este problema escribiendo una prueba directa de la función auxiliar `full_title`, lo que implica crear un archivo para probar la función auxiliar de la aplicación y luego llenar el código indicado con `FILL_IN` en el [Listado 5.38](#). (El [Listado 5.38](#) utiliza `assert_equal <esperado>, <real>`, que verifica que el resultado esperado coincide con el valor real cuando sea comparado mediante el operador `==`.)

Listado 5.36: Incluyendo la función auxiliar de la aplicación en las pruebas.

`test/test_helper.rb`

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

Listado 5.37: Usando la función auxiliar `full_title` en una prueba. VERDE

`test/integration/site_layout_test.rb`

```
require 'test_helper'  
  
class SiteLayoutTest < ActionDispatch::IntegrationTest  
  
  test "layout links" do  
    get root_path  
    assert_template 'static_pages/home'  
    assert_select "a[href=?]", root_path, count: 2
```

```
assert_select "a[href=?]", help_path
assert_select "a[href=?]", about_path
assert_select "a[href=?]", contact_path
get signup_path
assert_select "title", full_title("Sign up")
end
end
```

Listado 5.38: Una prueba directa de la función auxiliar `full_title`.

`test/helpers/application_helper_test.rb`

```
require 'test_helper'

class ApplicationHelperTest < ActionView::TestCase
  test "full title helper" do
    assert_equal full_title,           FILL_IN
    assert_equal full_title("Help"),   FILL_IN
  end
end
```

Capítulo 6

Modelando usuarios

En el [Capítulo 5](#), terminamos con una página simple para crear nuevos usuarios ([Sección 5.4](#)). En el transcurso de los siguientes cinco capítulos, cumpliremos la promesa implícita en esta página de registro. En este capítulo, daremos el primer paso crítico al crear un *modelo de datos* para los usuarios de nuestro sitio, junto con una forma de almacenar esos datos. En el [Capítulo 7](#), le daremos a los usuarios la capacidad de registrarse en nuestro sitio y crear una página de su perfil de usuario. Una vez que los usuarios puedan registrarse, les permitiremos iniciar y cerrar sesión ([Capítulo 8](#)), y en el [Capítulo 9](#) ([Sección 9.2.1](#)) aprenderemos a proteger las páginas de accesos indebidos. Finalmente, en el [Capítulo 10](#) agregaremos la activación de la cuenta (al confirmar que la dirección de correo electrónico es válida) y el reinicio de contraseña. Todo junto, el material del [Capítulo 6](#) al [Capítulo 10](#) desarrolla un sistema completo para inicio de sesión y autenticación en Rails. Como puede que usted ya sepa, existen varias soluciones pre-construídas de autenticación para Rails; el Recuadro 6.1 explica porqué, al menos la primera vez, es probablemente una mejor idea elaborar la suya.

Recuadro 6.1. Desarrollando su propio sistema de autenticación

Virtualmente todas las aplicaciones web requieren un sistema de autenticación e inicio de sesión de algún tipo. Como resultado, la mayoría de las bibliotecas

web tienen una pléthora de opciones para implementar tales sistemas, y Rails no es la excepción. Ejemplos de sistemas de autenticación y autorización incluyen [Clearance](#), [Authlogic](#), [Devise](#), y [CanCan](#) (así como soluciones específicas que no son Rails, construídas sobre [OpenID](#) o [OAuth](#)). Es sensato preguntar ¿porqué deberíamos de reinventar la rueda? ¿Porqué no sólo utilizar una solución ya lista para usarse en vez de crear la nuestra?

En primera, la experiencia en la práctica nos indica que la autenticación en la mayoría de los sitios requiere una personalización muy extensa, y que modificar un producto de terceros a menudo implica más trabajo que escribir el sistema desde cero. Además, los sistemas pre-construídos pueden ser “cajas negras”, con entrañas potencialmente misteriosas; cuando usted escribe su propio sistema, es más probable que lo entienda. Más aún, recientes adiciones a Rails ([Sección 6.3](#)) facilitan la escritura de un sistema de autenticación personalizado. Finalmente, si usted termina utilizando un sistema de terceros en lo sucesivo, estará mejor preparado para entenderlo y modificarlo si antes construyó uno por su cuenta.

6.1 Modelo usuario

Aunque la meta principal de los próximos tres capítulos es elaborar una página de registro para nuestro sitio (como esbozamos en la [Figura 6.1](#)), no tendría mayor beneficio aceptar en este momento información de los nuevos usuarios: no tenemos donde guardarla ahora. Entonces, el primer paso para registrar usuarios es crear una estructura de datos para capturar y almacenar su información.

En Rails, la estructura para un modelo de datos es llamada, como es natural, un *modelo* (la M en MVC de la [Sección 1.3.3](#)). La solución Rails por default al problema de la persistencia es utilizar una *base de datos* para almacenamiento de información a largo plazo, y la biblioteca por default para interactuar con la base de datos se llama *Active Record*.¹ *Active Record* viene con una serie de métodos para crear, guardar y buscar objetos de datos, todo sin tener que utilizar

¹El nombre viene del patrón “[registro activo](#)”, identificado y nombrado en el libro *Patterns of Enterprise Application Architecture* de Martin Fowler.



Figura 6.1: Un esbozo de la página de registro de usuarios.

el lenguaje estructurado de consulta (SQL, por sus siglas en inglés *Structured Query Language*) empleado en [bases de datos relacionales](#). Más aún, Rails tiene una funcionalidad llamada *migraciones* para permitir que las definiciones de datos sean escritas en Ruby puro, sin tener que aprender un lenguaje SQL para definición de datos (DDL, por sus siglas en inglés *Data Definition Language*). El efecto es que Rails lo aísla a usted por completo de los detalles del almacenamiento de datos. En este libro, usaremos SQLite para desarrollo y PostgreSQL (via Heroku) para producción ([Sección 1.5](#)), hemos desarrollado este tema aún más, al punto donde apenas tenemos que pensar acerca de cómo Rails almacena los datos, aún para aplicaciones en producción.

Como es usual, si usted está usando Git para el control de versiones, ahora es un buen momento para crear una rama para el modelado de usuarios:

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

6.1.1 Migraciones de base de datos

Recuerde de la [Sección 4.4.5](#) que ya habíamos trabajado, mediante una clase construída por nosotros, **User**, objetos de usuario con atributos **email** y **name**. Esa clase nos sirvió de ejemplo, pero carecía de la importante capacidad de *persistencia*: cuando creamos un objeto **User** en la consola Rails, desaparecía tan pronto terminábamos la sesión. Nuestro objetivo en esta sección es crear un modelo para los usuarios que no desaparezca tan fácilmente.

Igual que con la clase **User** de la [Sección 4.4.5](#), empezaremos modelando un usuario con dos atributos, **email** y **name**, el primero nos servirá como nombre de usuario único.² (Agregaremos un atributo para contraseñas en la [Sección 6.3](#).) En el [Listado 4.13](#), hicimos esto con el método **attr_accessor** de Ruby:

²Al utilizar la dirección electrónica como nombre de usuario, abrimos la posibilidad de comunicarnos con nuestros usuarios en un futuro ([Capítulo 10](#)).

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

Figura 6.2: Un diagrama de datos de ejemplo en la tabla **users**.

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

En contraste, cuando usamos Rails para modelar usuarios no necesitamos identificar los atributos explícitamente. Como mencionamos brevemente antes, para almacenar datos, Rails utiliza una base de datos relacional por default, que consiste de *tablas* compuestas de *registros* de datos, donde cada registro tiene *columnas* de atributos de datos. Por ejemplo, para guardar usuarios con nombres y direcciones de correo, crearemos una tabla **users** con columnas **email** y **name** (con un registro por cada usuario). Un ejemplo de la tabla aparece en la Figura 6.2, correspondiente al modelo de datos mostrado en la Figura 6.3. (La Figura 6.3 es sólo un bosquejo; el modelo de datos completo aparece en la Figura 6.4.) Al nombrar las columnas **email** y **name**, permitimos que *Active Record* descubra los atributos del objeto **User** por nosotros.

Recuerde del Listado 5.28 que creamos un controlador **Users** (junto con una acción **new**) mediante el comando

users	
id	integer
name	string
email	string

Figura 6.3: Un bosquejo del modelo de datos de **User**.

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

El comando análogo para crear un modelo es **generate model**, que podemos emplear para generar un modelo **User** con atributos **email** y **name**, como se muestra en el Listado 6.1.

Listado 6.1: Generando un modelo **User**.

```
$ rails generate model User name:string email:string
  invoke  active_record
  create    db/migrate/20140724010738_create_users.rb
  create    app/models/user.rb
  invoke  test_unit
  create    test/models/user_test.rb
  create    test/fixtures/users.yml
```

(Observe que, en contraste con la convención del plural para nombres de controladores, los nombres de modelo están en singular: un controlador *Users*, pero un modelo *User*.) Al pasar los parámetros opcionales **email:string** y **name:string**, le informamos a Rails acerca de los atributos que queremos, junto con el tipo de datos que deseamos para ellos (en este caso, **string**). Compare esto con los nombres de acción incluidos en los Listados 3.4 y 5.28.

Uno de los resultados del comando **generate** del Listado 6.1 es un archivo nuevo llamado *migración*. Las migraciones proporcionan una forma de alterar la estructura de la base de datos de forma incremental, de modo que nuestro

modelo de datos puede adaptarse a requerimientos cambiantes. En el caso del modelo **User**, la migración es creada de forma automática por el script de generación de modelos; esto crea una tabla **users** con dos columnas, **email** y **name**, como se muestra en el Listado 6.2. (Aprenderemos a partir de la Sección 6.2.5 cómo hacer una migración desde cero.)

Listado 6.2: Migración para el modelo **User** (para crear la tabla **users**).

db/migrate/[timestamp]_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps null: false
    end
  end
end
```

Observe que el nombre del archivo de la migración tiene como prefijo un *sello de tiempo* basado en cuándo se generó la migración. En los primeros días de las migraciones, los nombres de archivo tenían como prefijo un número entero incremental, pero causaba conflictos cuando al trabajar en equipo varios programadores tenían migraciones con el mismo número. A menos que las migraciones sean generadas en el mismo segundo, cosa que es poco probable que suceda, el uso de sellos de tiempo evita de forma conveniente tales colisiones.

La migración misma consiste de un método **change** que determina el cambio que será efectuado a la base de datos. En el caso del Listado 6.2, **change** utiliza un método Rails llamado **create_table** para crear una tabla para almacenar usuarios en la base de datos. El método **create_table** acepta un bloque (Sección 4.3.2) con una variable de bloque, en este caso llamada **t** (de “tabla”). Dentro del bloque, el método **create_table** utiliza el objeto **t** para crear las columnas **email** y **name** en la base de datos, ambas de tipo **string**.³ Aquí el nombre de la tabla es (**users**) aún cuando el nombre del modelo es

³No se preocupe acerca de cómo el objeto **t** se las arregla para hacer esto; la belleza de las *capas de abstracción* es que no necesitamos saber. Sólo confiemos en que el objeto **t** haga su trabajo.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime

Figura 6.4: El modelo de datos **User** producido por el [Listado 6.2](#).

singular (**User**), lo cual refleja una convención lingüística seguida por Rails: un modelo representa a un solo usuario, mientras que una tabla de la base de datos consiste de muchos usuarios. La línea final del bloque, **t.timestamps null: false**, es un comando especial que crea dos *columnas mágicas* llamadas **created_at** y **updated_at**, que son sellos de tiempo que automáticamente registran cuándo un usuario es creado y actualizado. (Veremos ejemplos concretos de las columnas mágicas empezando la [Sección 6.1.3](#).) El modelo de datos completo representado por la migración del [Listado 6.2](#) se muestra en la [Figura 6.4](#). (Observe la añadidura de las columnas mágicas, que no estaban presentes en el bosquejo de la [Figura 6.3](#).)

Podemos ejecutar la migración, mediante el comando **rake** (Recuadro 2.1) como sigue:

```
$ bundle exec rake db:migrate
```

(Puede recordar que ejecutamos este comando en un contexto similar en la [Sección 2.2](#).) La primera vez que **db:migrate** es ejecutado, crea un archivo llamado **db/development.sqlite3**, que es una base de datos **SQLite**. Podemos ver la estructura de la base abriendo **development.sqlite3** con el [Navegador para SQLite](#). (Si usted está utilizando el IDE en la nube, primero descargue el archivo de la base de datos a su disco local, como se muestra en la

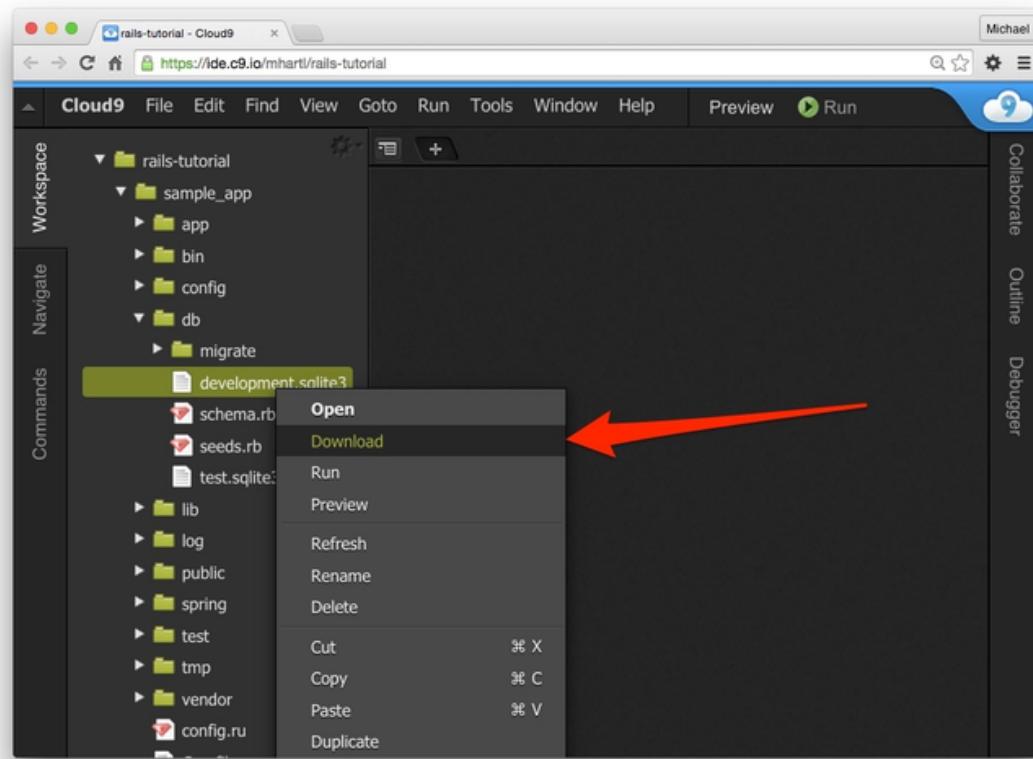


Figura 6.5: Descargando un archivo del IDE en la nube.

Figura 6.5.) El resultado aparece en la Figura 6.6; compárelo con el diagrama de la Figura 6.4. Puede observar que hay una columna en la Figura 6.6 que no ha sido considerada en la migración: la columna **id**. Como observamos brevemente en la Sección 2.2, esta columna es creada de forma automática, y es utilizada por Rails para identificar cada registro de forma única.

La mayoría de las migraciones (incluyendo todas las de este tutorial) son *reversibles*, lo que significa que podemos “migrar hacia atrás” y deshacerlas con una simple tarea Rake, llamada **db:rollback**:

```
$ bundle exec rake db:rollback
```

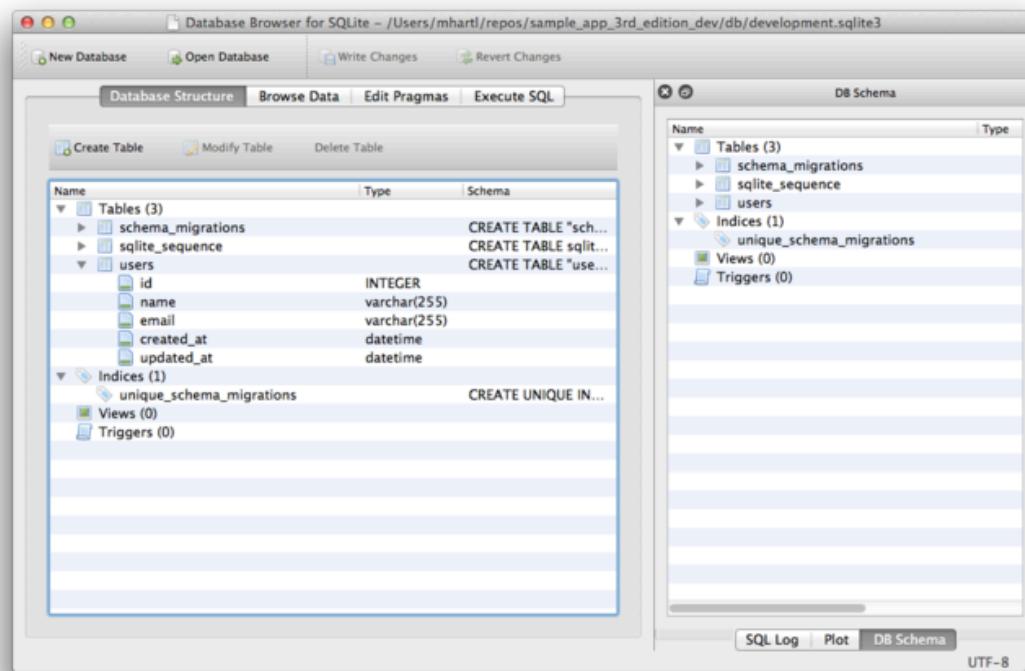


Figura 6.6: El Navegador para SQLite con nuestra nueva tabla **users**.

(Observe el Recuadro 3.1 para otra técnica útil para revertir migraciones.) Internamente, esta instrucción ejecuta el comando `drop_table` para eliminar la tabla `users` de la base de datos. La razón por la que esto funciona es que el método `change` sabe que `drop_table` es el inverso de `create_table`, lo que implica que la migración para revertir puede inferirse fácilmente. En el caso de una migración irreversible, tal como una para eliminar una columna de una tabla en base de datos, es necesario definir los métodos `up` y `down` en vez de un solo método `change`. Puede consultar [las guías de Rails](#) para mayor información acerca de migraciones.

Si usted ha regresado un cambio en la base de datos, lo puede volver a aplicar antes de continuar:

```
$ bundle exec rake db:migrate
```

6.1.2 El archivo modelo

Hemos visto cómo al generar el modelo `User` del Listado 6.1 se creó un archivo de migración (Listado 6.2), y vimos en la Figura 6.6 los resultados de ejecutar esta migración: se actualizó un archivo llamado `development.sqlite3` creando una tabla `users` con columnas `id`, `name`, `email`, `created_at`, y `updated_at`. El Listado 6.1 también creó el modelo mismo. El resto de la sección está dedicado a entender esto.

Empezamos echando un vistazo al código del modelo `User`, que vive en el archivo `user.rb` dentro del directorio `app/models/`. El archivo es, por decirlo así, muy compacto (Listado 6.3).

Listado 6.3: El flamante modelo `User`.

```
app/models/user.rb
```

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

Recuerde de la Sección 4.4.2 que la sintaxis `class User < ActiveRecord::Base` significa que la clase `User` hereda de `ActiveRecord::Base`,

de modo que el modelo `User` automáticamente obtiene toda la funcionalidad de la clase `ActiveRecord::Base`. Por supuesto, este conocimiento no nos sirve de nada a menos que sepamos qué contiene `ActiveRecord::Base`; por lo que empezaremos con unos ejemplos concretos.

6.1.3 Creando objetos usuario

De igual forma que en el [Capítulo 4](#), nuestra herramienta predilecta para explorar los modelos de datos es la consola Rails. Puesto que no queremos (aún) realizar cambios a nuestra base de datos, iniciaremos la consola en un *ambiente aislado*:

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

Como se indica en el mensaje de advertencia que se muestra al iniciar la consola en este modo, cualquier modificación que usted haga será deshecho al salir, es decir, que cualquier cambio que se haga en la base de datos durante esta sesión, es temporal.

En la sesión de la consola de la [Sección 4.4.5](#), creamos un nuevo objeto usuario con `User.new`, al cual tuvimos acceso luego de requerir el archivo de ejemplo de usuario del [Listado 4.13](#). Con los modelos, la situación es diferente; como puede recordar de la [Sección 4.4.4](#), la consola Rails automáticamente carga el ambiente Rails, el cual incluye los modelos. Esto implica que podemos crear un objeto usuario sin ningún esfuerzo extra:

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

Vemos aquí la representación por default que la consola muestra de un objeto usuario.

Cuando invocamos `User.new` sin argumentos, nos devuelve un objeto con todos los atributos en `nil`. En la [Sección 4.4.5](#), diseñamos el ejemplo de la

clase **User** para que aceptara un *arreglo hash de inicialización* para asignar los atributos del objeto; ese diseño fue motivado por *Active Record*, que nos permite inicializar objetos de la misma manera:

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

Aquí vemos que los atributos nombre y dirección electrónica han sido asociados como se espera.

La noción de *validez* es importante para entender los objetos de modelos de *Active Record*. Revisaremos este tema con más detalle en la [Sección 6.2](#), pero por ahora vale la pena mencionar que nuestro objeto inicial **user** es válido, lo cual podemos verificar invocando al método booleano **valid?** sobre él:

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

Hasta aquí, no hemos tocado la base de datos: **User.new** únicamente crea un objeto *en memoria*, mientras que **user.valid?** únicamente verifica que el objeto es válido. Con la finalidad de guardar el objeto **User** en la base de datos, necesitamos invocar el método **save** de la variable **user**:

```
>> user.save
(0.2ms)  begin transaction
User Exists (0.2ms)  SELECT 1 AS one FROM "users"  WHERE LOWER("users".
"email") = LOWER('mhartl@example.com') LIMIT 1
SQL (0.5ms)  INSERT INTO "users" ("created_at", "email", "name", "updated_at")
  VALUES (?, ?, ?, ?)  [[["created_at", "2014-09-11 14:32:14.199519"],
  ["email", "mhartl@example.com"], ["name", "Michael Hartl"], ["updated_at",
  "2014-09-11 14:32:14.199519"]]
(0.9ms)  commit transaction
=> true
```

El método **save** regresa **true** si todo sale bien; de otra forma regresa **false**. (Actualmente, todas las operaciones de guardado son exitosas porque aún no

tenemos validaciones; veremos algunos casos en la Sección 6.2 cuando algunas fallen.) Como referencia, la consola Rails también muestra el comando SQL correspondiente a `user.save` (digamos, `INSERT INTO "users" ...`). Difícilmente necesitaremos SQL puro en este libro,⁴ y en lo sucesivo omitiré la discusión sobre los comandos SQL, pero puede aprender mucho leyendo el SQL correspondiente a cada comando de *Active Record*.

Puede haber observado que el nuevo objeto `user` tenía valores `nil` para el `id` y para los atributos `created_at` y `updated_at`. Veamos si nuestro `save` cambió algo:

```
>> user
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
  created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">
```

Vemos que al `id` le ha sido asignado un valor de `1`, mientras que a las columnas mágicas se les ha asignado la fecha y hora actuales.⁵ En este momento, los sellos de tiempo para la hora de creación y modificación son idénticos; los veremos diferir en la Sección 6.1.5.

De igual forma que con la clase `User` de la Sección 4.4.5, las instancias del modelo `User` nos permiten acceder a sus atributos usando la notación punto:

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "mhartl@example.com"
>> user.updated_at
=> Thu, 24 Jul 2014 00:57:46 UTC +00:00
```

⁴La única excepción se encuentra en la Sección 12.3.3.

⁵En caso de que tenga curiosidad acerca de `"2014-07-24 00:57:46"`, no estoy escribiendo esto pasada la medianoche; los sellos de tiempo están guardados en [Tiempo Universal Coordinado](#) (UTC, por sus siglas en inglés *Coordinated Universal Time*), que para efectos prácticos es lo mismo que [El tiempo del meridiano de Greenwich](#). De la [Sección de Preguntas Frecuentes sobre de la hora y frecuencia del NIST](#): Q: Porqué UTC es utilizado como acrónimo “Coordinated Universal Time” en vez de CUT? A: En 1970 el sistema de tiempo coordinado universal fue inventado por un grupo asesor internacional de técnicos expertos dentro de la Unión de Telecomunicaciones Internacional (ITU, por sus siglas en inglés *International Telecommunication Union*). El ITU sintió que era mejor designar una sola abreviatura para que fuera utilizada en todos los lenguajes, con la finalidad de minimizar la confusión. No fue posible llegar a un acuerdo unánime, puesto que algunos proponían el orden en inglés, CUT, otros el orden en francés, TUC, y finalmente eligieron el acrónimo UTC como un compromiso.

Como veremos en el Capítulo 7, a menudo es conveniente crear y guardar un modelo en dos pasos como lo hicimos anteriormente, pero *Active Record* también le permite combinarlos en un solo paso con `User.create`:

```
>> User.create(name: "A Nother", email: "another@example.org")
=> #<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2014-07-24 01:05:24", updated_at: "2014-07-24 01:05:24">
>> foo = User.create(name: "Foo", email: "foo@bar.com")
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2014-07-24
01:05:42", updated_at: "2014-07-24 01:05:42">
```

Observe que `User.create`, en vez de regresar `verdadero` o `falso`, regresa el objeto `User`, que podemos asignar a una variable si así lo deseamos (tal como `foo` en el segundo comando del código anterior).

El inverso de `create` es `destroy`:

```
>> foo.destroy
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2014-07-24
01:05:42", updated_at: "2014-07-24 01:05:42">
```

Como `create`, `destroy` regresa el objeto en cuestión, aunque no puedo recordar que alguna vez haya utilizado el valor de retorno de `destroy`. Adicionalmente, el objeto destruido aún existe en memoria:

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2014-07-24
01:05:42", updated_at: "2014-07-24 01:05:42">
```

Entonces, ¿cómo sabemos si realmente destruimos un objeto? Y para objetos guardados y no destruidos, ¿cómo podemos recuperar usuarios de la base de datos? Para contestar estas preguntas, necesitamos aprender a utilizar *Active Record* para buscar objetos usuario.

6.1.4 Buscando objetos usuario

Active Record proporciona varias opciones para buscar objetos. Utilicémoslas para encontrar el primer usuario que creamos, mientras verificamos que el tercer

usuario (**foo**) ha sido destruido. Empezaremos con el usuario existente:

```
>> User.find(1)
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
  created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">
```

Aquí hemos pasado el **id** del usuario a **User.find**; *Active Record* regresa el usuario con ese **id**.

Veamos si el usuario con **id** igual a **3** aún existe en la base de datos:

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

Puesto que destruimos nuestro tercer usuario en la Sección 6.1.3, *Active Record* no puede encontrarlo en la base de datos. En su lugar, **find** arroja una *excepción*, que es una forma de indicar un evento excepcional en la ejecución de un programa—en este caso, un id que no existe, lo que ocasiona que **find** arroje la excepción **ActiveRecord::RecordNotFound**.⁶

Además del **find** genérico, *Active Record* también nos permite buscar usuarios por atributos específicos:

```
>> User.find_by(email: "mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
  created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">
```

Puesto que estaremos utilizando direcciones de correo electrónicas como nombres de usuarios, esta clase de **find** será útil cuando aprendamos cómo permitir que los usuarios inicien sesión en nuestro sitio (Capítulo 7). Si le preocupa que **find_by** sea ineficiente cuando hay un gran número de usuarios, está siendo visionario; nos ocuparemos de este tema y su solución mediante índices en la base de datos, en la Sección 6.2.5.

Terminaremos con otro par de formas generales de buscar usuarios. Para empezar, hay un **first**:

⁶Las excepciones y el manejo de excepciones son temas de Ruby algo avanzados, y no los necesitamos mucho en este libro. Son importantes, por lo que le sugiero que aprenda más de ellos utilizando uno de los libros de Ruby recomendados en la Sección 12.4.1.

```
>> User.first  
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",  
created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">
```

Naturalmente, **first** regresa el primer usuario en la base de datos. También hay un **all**:

```
>> User.all  
=> #<ActiveRecord::Relation [#<User id: 1, name: "Michael Hartl",  
email: "mhartl@example.com", created_at: "2014-07-24 00:57:46",  
updated_at: "2014-07-24 00:57:46">, #<User id: 2, name: "A Nother",  
email: "another@example.org", created_at: "2014-07-24 01:05:24",  
updated_at: "2014-07-24 01:05:24">]>
```

Como puede observar en la salida de la consola, **User.all** regresa a todos los usuarios de la base de datos, como un objeto de la clase **ActiveRecord::Relation**, que es propiamente un arreglo (Sección 4.3.1).

6.1.5 Actualizando objetos usuario

Una vez que hemos creado objetos, a menudo queremos actualizarlos. Hay dos formas básicas de hacer esto. Primero, podemos asignar atributos individualmente, como lo hicimos en la Sección 4.4.5:

```
>> user          # Just a reminder about our user's attributes  
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",  
created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">  
>> user.email = "mhartl@example.net"  
=> "mhartl@example.net"  
>> user.save  
=> true
```

Observe que el paso final es necesario para escribir los cambios en la base de datos. Podemos ver qué sucede sin guardar nuestros cambios, utilizando **reload**, que recarga el objeto con la información almacenada en la base de datos:

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

Ahora que hemos actualizado el usuario al ejecutar `user.save`, las columnas mágicas difieren, como lo prometimos en la Sección 6.1.3:

```
>> user.created_at
=> "2014-07-24 00:57:46"
>> user.updated_at
=> "2014-07-24 01:37:32"
```

La segunda forma principal de actualizar múltiples atributos es mediante `update_attributes`:⁷

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

El método `update_attributes` acepta un arreglo hash de atributos, y en el caso exitoso, actualiza y guarda en un solo paso (regresando `verdadero` para indicar que la operación terminó sin contratiempos). Observe que si alguna de las validaciones falla, como cuando una contraseña es requerida para guardar un registro (como implementaremos en la Sección 6.3), la llamada a `update_attributes` fallará. Si necesitamos actualizar únicamente un atributo, usando la versión en singular `update_attribute` evita esta restricción:

⁷El método `update_attributes` es un alias del método `update`, pero prefiero la versión larga por su similitud con la versión en singular del método, `update_attribute`.

```
>> user.update_attribute(:name, "The Dude")
=> true
>> user.name
=> "The Dude"
```

6.2 Validaciones de usuario

El modelo `User` que creamos en la Sección 6.1 ahora tiene atributos `name` y `email` funcionando, pero son completamente genéricos: cualquier cadena de caracteres (incluyendo una vacía) actualmente es válida en ambos casos. Además, los nombres y direcciones electrónicas son más específicos que esto. Por ejemplo, `name` no debería ser blanco, y `email` debería ajustarse al formato característico de las direcciones electrónicas. Más aún, puesto que estaremos usando las direcciones electrónicas como nombres de usuario únicos para entrar al sistema, no debemos permitir direcciones electrónicas duplicadas en la base de datos.

En resumen, no debemos permitir que `email` y `name` sean sólo cadenas de caracteres; debemos aplicar ciertas restricciones a sus valores. *Active Record* nos permite imponer estas restricciones usando *validaciones* (las revisamos brevemente en la Sección 2.3.2). En esta sección, cubriremos varios de los casos más comunes, validando *presencia*, *longitud*, *formato* y *unicidad*. En la Sección 6.3.2 agregaremos una validación común final: *confirmación*. Y veremos en la Sección 7.3 cómo las validaciones nos proporcionan mensajes de error útiles cuando los usuarios envían datos que las violan.

6.2.1 Una prueba de validez

Como observamos en el Recuadro 3.3, el desarrollo orientado a pruebas no siempre es la herramienta correcta para el trabajo, pero las validaciones del modelo son justo el tipo de características para las cuales TDD es perfecto. Es difícil tener confianza en que una validación dada está haciendo exactamente lo que esperamos si no escribimos una prueba de fallo y hacemos que lo pase.

Nuestra estrategia será empezar con un objeto del modelo *válido*, establecer alguno de sus atributos a algo que queremos que no sea válido, y luego probar que de hecho lo es. Como malla de seguridad, primero escribiremos la prueba para asegurarnos de que el objeto del modelo inicialmente es válido. De esta forma, cuando la prueba de validación falle, sabremos que es por la razón correcta (y no porque el objeto inicial era inválido desde el principio).

Para que podamos empezar, el comando del [Listado 6.1](#) generó una prueba inicial para usuarios, aunque en este caso está prácticamente vacía ([Listado 6.4](#)).

Listado 6.4: La prueba de usuario por default; prácticamente vacía.

test/models/user_test.rb

```
require 'test_helper'

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

Para escribir una prueba para un objeto válido, crearemos un objeto del modelo **User** inicialmente válido `@user` usando el método especial `setup` (discutido brevemente en los ejercicios del [Capítulo 3](#)), que automáticamente es ejecutado antes de cada prueba. Como `@user` es una variable de instancia, se encuentra automáticamente disponible en todas las pruebas, y podemos comprobar su validez usando el método `valid?` ([Sección 6.1.3](#)). El resultado aparece en el [Listado 6.5](#).

Listado 6.5: Una prueba para un usuario inicialmente válido. VERDE

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

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

El [Listado 6.5](#) utiliza el método plano `assert`, que en este caso termina exitosamente si `@user.valid?` regresa `verdadero` y falla si regresa `falso`.

Como nuestro modelo usuario aún no tiene ninguna validación, la prueba inicial debe pasar:

Listado 6.6: VERDE

```
$ bundle exec rake test:models
```

Aquí hemos usado `rake test:models` para ejecutar sólo las pruebas del modelo (compare con `rake test:integration` de la [Sección 5.3.4](#)).

6.2.2 Validación de presencia

Quizá la validación más elemental sea la de *presencia*, que simplemente verifica que un atributo dado se encuentra presente. Por ejemplo, en esta sección nos aseguraremos que los campos del nombre y de la dirección electrónica estén presentes antes de que un usuario sea almacenado en la base de datos. En la [Sección 7.3.3](#), veremos cómo propagar este requerimiento a la forma de registro para crear nuevos usuarios.

Empezaremos con una prueba para la presencia del atributo `name` tomando como base la prueba del [Listado 6.5](#). Como vimos en el [Listado 6.7](#), todo lo que necesitamos hacer es asignar al atributo `name` de la variable `@user` una cadena en blanco (en este caso, una cadena de espacios en blanco) y luego verificar (usando el método `assert_not`) que el objeto usuario resultante no es válido.

Listado 6.7: Una prueba para validar el atributo `name`. ROJO

```
test/models/user_test.rb
```

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

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

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

En este punto, las pruebas del modelo deberían estar en **ROJO**:

Listado 6.8: ROJO

```
$ bundle exec rake test:models
```

Como vimos brevemente en los ejercicios del Capítulo 2, la forma de validar la presencia del atributo `name` es mediante el método `validates` con argumento `presence: true`, como se muestra en el Listado 6.9. El argumento `presence: true` es un *arreglo hash de opciones* de un elemento; recuerde de la Sección 4.3.4 que las llaves son opcionales cuando pasamos arreglos hash como el último argumento de un método. (Como observamos en la Sección 5.1.1, el uso de los arreglos hash de opciones es un tema recurrente en Rails.)

Listado 6.9: Validando la presencia del atributo `name`. VERDE*app/models/user.rb*

```
class User < ActiveRecord::Base
  validates :name, presence: true
end
```

El [Listado 6.9](#) puede parecer mágico, pero `validates` es sólo un método. Una forma equivalente del [Listado 6.9](#) usando paréntesis se muestra a continuación:

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

Desde la consola de Rails veamos los efectos de agregar una validación a nuestro modelo usuario:⁸

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

Aquí verificamos la validez de la variable `user` mediante el método `valid?` que regresa `falso` cuando el objeto no pasa una o más validaciones, y `verdadero` cuando todas las validaciones pasan. En este caso, sólo tenemos una validación, por lo que sabremos cuál falló, pero aún puede ser útil verificar usando el objeto `errors` generado durante el fallo:

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

(El mensaje de error es una pista de que Rails validó la presencia de un atributo mediante el método `blank?`, que vimos al final de la [Sección 4.4.3](#).)

⁸Omitiré la salida de los comandos de la consola cuando no son particularmente ilustrativos—por ejemplo, la salida de `User.new`.

Como el usuario no es válido, el intento por guardar el usuario en la base de datos, automáticamente falla:

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

Como resultado, la prueba del [Listado 6.7](#) ahora debe estar en **VERDE**:

Listado 6.10: **VERDE**

```
$ bundle exec rake test:models
```

Siguiendo el modelo del [Listado 6.7](#), escribir una prueba para la presencia del atributo **email** es fácil ([Listado 6.11](#)), igual que el código de la aplicación para hacerla que pase ([Listado 6.12](#)).

Listado 6.11: Una prueba para validar el atributo **email**. **ROJO**

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

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

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

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

Listado 6.12: Validando la presencia del atributo `email`. VERDE*app/models/user.rb*

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

En este momento, las validaciones de presencia están completas, y el conjunto de pruebas debería estar en **VERDE**:

Listado 6.13: **VERDE**

```
$ bundle exec rake test
```

6.2.3 Validación de longitud

Hemos restringido nuestro modelo de usuario a que requiera un nombre para cada usuario, pero debemos ir más allá: los nombres de usuario serán mostrados en la aplicación de ejemplo, por lo que debemos imponer algún límite a su longitud. Con todo el trabajo que hicimos en la [Sección 6.2.2](#), este paso es sencillo.

No hay ninguna ciencia en escoger una longitud máxima; sólo la estableceremos en **50** como una cota superior razonable, lo que significa que al verificar nombres que tengan **51** caracteres, los consideraremos demasiado largos. Adicionalmente, aunque es muy poco probable que alguna vez se presente el problema, existe una posibilidad de que una dirección electrónica de un usuario rebase la longitud máxima de cadenas, que en muchas bases de datos es de 255. Debido a la validación de formato de la [Sección 6.2.4](#) no implementaremos esta restricción, sólo agregaremos una en esta sección por completez. El [Listado 6.14](#) muestra las pruebas resultantes.

Listado 6.14: Una prueba para la validación de longitud de `name`. ROJO*test/models/user_test.rb*

```

require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

  #
  #

  test "name should not be too long" do
    @user.name = "a" * 51
    assert_not @user.valid?
  end

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

```

Por conveniencia, hemos usado una “multiplicación de cadena” en el Listado 6.14 para crear una cadena de 51 caracteres de longitud. Podemos ver cómo funciona esto usando la consola:

```

>> "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> ("a" * 51).length
=> 51

```

La validación de la longitud de la dirección electrónica se encarga de construir una dirección electrónica válida que sobrepasa la longitud máxima por un carácter:

```

>> "a" * 244 + "@example.com"
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaa@example.com"
>> ("a" * 244 + "@example.com").length
=> 256

```

En este momento, las pruebas del Listado 6.14 deberían estar en **ROJO**:

Listado 6.15: ROJO

```
$ bundle exec rake test
```

Para hacerlas que pasen, necesitamos utilizar el argumento de validación para restringir la longitud, que es sólo **length**, junto con el parámetro **maximum** para imponer la cota máxima (Listado 6.16).

Listado 6.16: Agregando la validación de longitud para el atributo **name**.

VERDE

```
app/models/user.rb
```

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

Ahora las pruebas deberían estar en VERDE:

Listado 6.17: VERDE

```
$ bundle exec rake test
```

Con todas nuestras pruebas pasando de nuevo, podemos implementar una validación más retadora: el formato de dirección electrónica.

6.2.4 Validación de formato

Nuestras validaciones para el atributo **name** sólo imponen restricciones mínimas—cualquier cadena que no consista de espacios en blanco y que contenga menos de 51 caracteres pasará—pero por supuesto el atributo **email** debe satisfacer un requerimiento más riguroso para ser una dirección de correo electrónico válida. Hasta ahora sólo hemos rechazado direcciones electrónicas en blanco; en esta sección, requeriremos que las direcciones electrónicas se apeguen al patrón familiar **user@example.com**.

Ninguna de las pruebas o de las validaciones será exhaustiva, sólo lo suficientemente buenas para aceptar la mayoría de las direcciones electrónicas válidas y rechazar la mayoría de las que no son válidas. Empezaremos con un par de pruebas que involucran colecciones de direcciones válidas e inválidas. Para crear estas colecciones, es importante conocer la técnica `%w[]` para crear arreglos de cadenas, como se muestra en esta sesión de consola:

```
>> %w[foo bar baz]
=> ["foo", "bar", "baz"]
>> addresses = %w[USER@foo.COM THE_US-ER@foo.bar.org first.last@foo.jp]
=> ["USER@foo.COM", "THE_US-ER@foo.bar.org", "first.last@foo.jp"]
>> addresses.each do |address|
?>   puts address
>> end
USER@foo.COM
THE_US-ER@foo.bar.org
first.last@foo.jp
```

Aquí hemos iterado sobre los elementos del arreglo `addresses` usando el método `each` (Sección 4.3.2). Con esta técnica a la mano, estamos listos para escribir algunas pruebas básicas de validación para direcciones electrónicas.

Como la validación del formato de dirección electrónica es intrincado y tiende a errores, empezaremos con algunas pruebas exitosas para direcciones electrónicas válidas para atrapar cualquier error en la validación. En otras palabras, queremos asegurarnos no sólo de que direcciones electrónicas inválidas como `user@example.com` sean rechazadas, sino que también direcciones como `user@example.com` son aceptadas, aún después de imponer la restricción de validación. (En este momento, por supuesto, serán aceptadas porque todas las direcciones que no están en blanco son válidas.) El resultado de una muestra representativa de direcciones electrónicas válidas, aparece en el Listado 6.18.

Listado 6.18: Pruebas para el formato de direcciones electrónicas válidas.

VERDE

```
test/models/user_test.rb

require 'test_helper'
```

```

class UserTest < ActiveSupport::TestCase

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

.

.

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

```

Observe que hemos incluído un segundo argumento opcional a la aserción con un mensaje de error personalizado, que en este caso identifica las direcciones provocando que la prueba falle:

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

(Esto utiliza el método interpolado `inspect` que mencionamos en la Sección 4.3.3.) Incluir en el mensaje la dirección específica que causa el error, es especialmente útil en una prueba con un ciclo `each` como la del Listado 6.18, pues de otra forma, cualquier fallo identificaría únicamente el número de línea, que es el mismo para todas las direcciones de correo, y no sería suficiente para identificar el origen del problema.

A continuación, agregaremos pruebas de *validez* para una variedad de direcciones electrónicas inválidas, tales como `user@example,com` (con una coma en vez de un punto) y `user_at_foo.org` (donde falta el signo '@'). Como en los Listados 6.18 y 6.19, se incluye un mensaje de error personalizado para identificar la dirección exacta que causa el error.

Listado 6.19: Pruebas para validar el formato de la dirección electrónica. ROJO
test/models/user_test.rb

```

require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

  ...

  test "email validation should reject invalid addresses" do
    invalid_addresses = %w[user@example.com user_at_.org user.name@example.
                           foo@bar_baz.com foo@bar+baz.com]
    invalid_addresses.each do |invalid_address|
      @user.email = invalid_address
      assert_not @user.valid?, "#{invalid_address.inspect} should be invalid"
    end
  end
end

```

En este momento, las pruebas deben estar en ROJO:

Listado 6.20: ROJO

```
$ bundle exec rake test
```

El código de la aplicación que valida el formato de la dirección electrónica, utiliza el método **format**, que funciona como sigue:

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

Esto valida el atributo con la *expresión regular* dada (o *regex*), que es un lenguaje poderoso (y a menudo críptico) para detectar patrones en cadenas. Esto significa que necesitamos construir una expresión regular para detectar direcciones de correo válidas y al mismo tiempo detectar direcciones *no* válidas.

Actualmente existe una expresión regular completa para detectar direcciones de correo de acuerdo al estándar oficial de direcciones electrónicas, pero es enorme, obscura, y muy posiblemente contraproducente.⁹ En este tutorial, adoptare-

⁹Por ejemplo, ¿sabía usted que "**Michael Hartl**"@example.com, con comillas y espacio en medio, es una dirección electrónica válida de acuerdo a este estándar? Increíblemente, lo es—aunque es absurdo.

Expresión	Significado
/\A[\w+\-\.]+@[a-z\d\-\.\.]+\.[a-z]+\z/i	expresión regular completa
/	inicio de la expresión regular
\A	coincide con el inicio de la cadena
[\w+\-\.]+	al menos una palabra de letras, signo de suma, guion medio o punto
@	“signo de arroba”
[a-z\d\-\.\.]+	al menos una letra, dígito, guion medio o punto
\.	punto
[a-z]+	al menos una letra
\z	coincide con el fin de la cadena
/	fin de la expresión regular
i	ignora mayúsculas/minúsculas

Tabla 6.1: Desglosando la expresión regular para direcciones de correo válidas.

mos una expresión regular más práctica que ha probado ser robusta en la práctica. Así es como se ve:

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

Para ayudarle a entender cómo funciona esto, la Tabla 6.1 la desmenuza en pedacitos.¹⁰

Aunque puede aprender mucho estudiando la Tabla 6.1, para entender realmente las expresiones regulares, considero que usar un verificador de expresiones regulares interactivo como [Rubular](#) es esencial (Figura 6.7).¹¹ El sitio web de Rubular tiene una agradable interfaz interactiva para elaborar expresiones regulares, junto con una práctica referencia rápida sobre expresiones regulares. Lo animo a que estudie la Tabla 6.1 con una ventana de navegador abierta en Rubular—ninguna cantidad de lectura sobre expresiones regulares puede reemplazar el hecho de jugar con ellas interactivamente. (*Nota:* Si usted utiliza las expresiones regulares de la Tabla 6.1 en Rubular, le sugiero dejar fuera los caracteres \A y \z de forma que pueda detectar más de una dirección electrónica por vez en la cadena de prueba dada.)

¹⁰Observe que, en la Tabla 6.1, “letra” en realidad significa “letra minúscula”, pero la **i** al final de la expresión regular obliga a una detección que ignora mayúsculas de minúsculas.

¹¹Si usted lo encuentra tan útil como yo, lo invito a [donar a Rubular](#) para recompensar al desarrollador Michael Lovitt por su maravilloso trabajo.

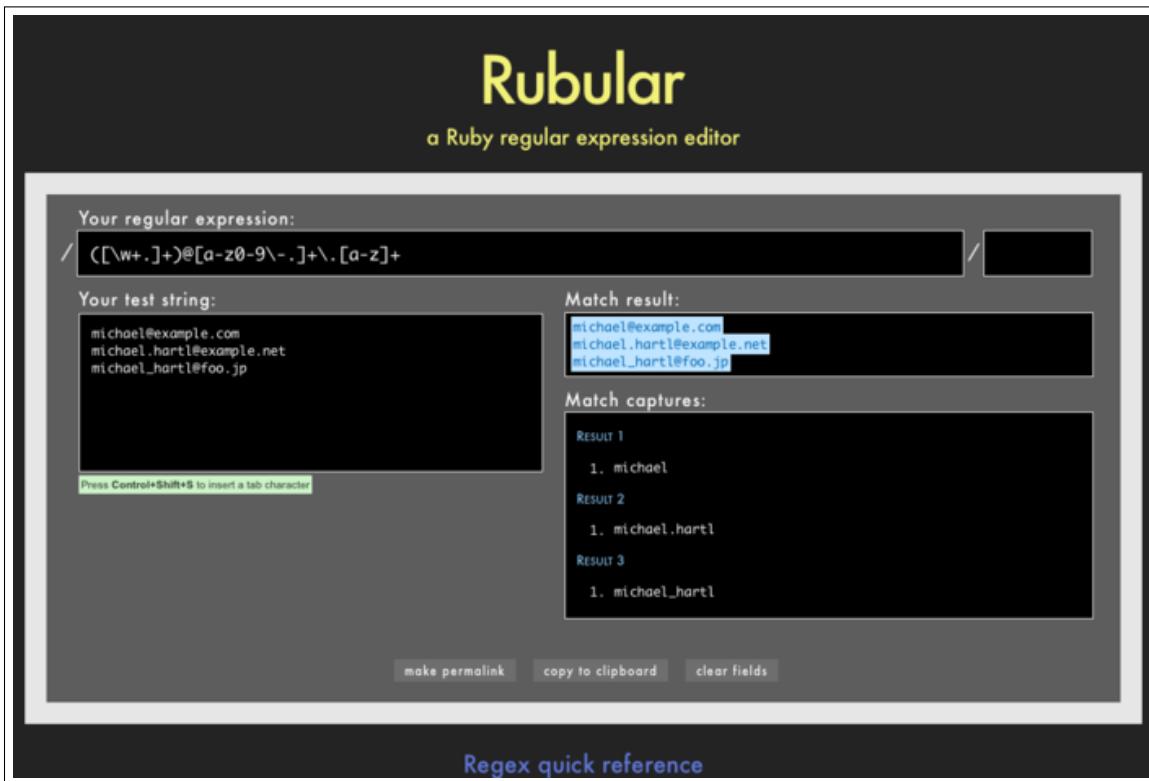


Figura 6.7: El impresionante editor de expresiones regulares de [Rubular](#).

Aplicando la expresión regular de la Tabla 6.1 a la validación de formato del **email** resulta en el código del Listado 6.21.

Listado 6.21: Validando el formato de la dirección electrónica con una expresión regular. **VERDE**

app/models/user.rb

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

Aquí la expresión regular **VALID_EMAIL_REGEX** es una *constante*, indicada así en Ruby porque su nombre empieza con una letra mayúscula. El código

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

asegura que sólo direcciones de correo que se apegan al patrón, serán consideradas válidas. (La expresión anterior tiene una debilidad notable: permite que direcciones no válidas que contienen puntos consecutivos, tales como **foo@bar..com** se consideren válidas. Arreglar este defecto requiere una expresión regular significativamente más complicada y se deja como ejercicio (Sección 6.5).)

En este momento, las pruebas deberían estar en **VERDE**:

Listado 6.22: **VERDE**

```
$ bundle exec rake test:models
```

Esto significa que sólo queda una restricción pendiente: obligar la unicidad de la dirección de correo.

6.2.5 Validación de unicidad

Para asegurar la unicidad de las direcciones electrónicas (de forma que podemos utilizarlas como nombres de usuario), estaremos usando la opción `:unique` del método `validates`. Pero les advierto: no lea superficialmente esta sección—léala cuidadosamente.

Empezaremos con algunas pruebas cortas. En nuestras pruebas previas a los modelos, principalmente utilizamos `User.new`, que sólo crea un objeto Ruby en memoria, pero para las pruebas de unicidad necesitamos guardar un registro en la base de datos.¹² La prueba inicial para detectar direcciones de correo duplicadas, aparece en el [Listado 6.23](#).

Listado 6.23: Una prueba para rechazar direcciones electrónicas duplicadas.

ROJO

```
test/models/user_test.rb

require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
  end
end
```

El algoritmo aquí es crear un usuario con la misma dirección de correo que `@user` usando `@user.dup`, lo cual crea un usuario duplicado con los mismos atributos. Al guardarlos, el usuario duplicado tiene una dirección electrónica que ya existe en la base de datos, y por tanto, no debe ser válido.

¹²Como se hizo notar brevemente en la introducción de esta sección, hay una base de datos dedicada a pruebas, `db/test.sqlite3`, para este propósito.

Podemos hacer que la prueba del Listado 6.23 pase agregando **uniqueness : true** a la validación del **email**, como se muestra en el Listado 6.24.

Listado 6.24: Validando la unicidad de la dirección electrónica. VERDE
app/models/user.rb

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

Sin embargo, no hemos terminado. Las direcciones electrónicas típicamente son procesadas como si no fueran sensibles a mayúsculas o minúsculas—es decir, **foo@bar.com** se considera igual que **FOO@BAR.COM** o **FoO@BAr.COM**—por lo que nuestra validación debería considerar esto también.¹³ Entonces, es importante probar que no hay sensibilidad a mayúsculas o minúsculas, lo que podemos lograr con el código del Listado 6.25.

Listado 6.25: Probando la unicidad de la dirección electrónica ignorando mayúsculas/minúsculas. ROJO

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

¹³Técnicamente, sólo la parte del dominio en la dirección electrónica no es sensible a mayúsculas o minúsculas: *foo@bar.com* en realidad es diferente de *Foo@bar.com*. En la práctica, es mala idea confiar en este hecho; como se hace notar en [about.com](#), “Puesto que la sensibilidad a mayúsculas o minúsculas en las direcciones electrónicas pueden crear gran confusión, problemas de interoperabilidad y dolores de cabeza, sería tonto solicitar que las direcciones de correo sean escritas con las mayúsculas y minúsculas correctas. Difícilmente cualquier servicio de correo electrónico o ISP obliga a utilizar direcciones sensibles a mayúsculas o minúsculas, devolviendo mensajes cuya dirección electrónica del destinatario no fue escrita correctamente (todo en mayúsculas, por ejemplo).” Agradezco al lector Riley Moses por hacer notar esto.

```

.
.
.

test "email addresses should be unique" do
  duplicate_user = @user.dup
  duplicate_user.email = @user.email.upcase
  @user.save
  assert_not duplicate_user.valid?
end
end

```

Aquí estamos utilizando el método `upcase` sobre cadenas (lo revisamos brevemente en la [Sección 4.3.2](#)). Esta prueba hace lo mismo que la prueba de dirección electrónica duplicada, pero con una dirección en mayúsculas. Si la prueba parece un poco abstracta, adelante, vaya a la consola:

```

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

```

Por supuesto, `duplicate_user.valid?` es en este momento **verdadero** porque la validación de unicidad es sensible a mayúsculas/minúsculas, pero queremos que sea **falso**. Por suerte, `:uniqueness` acepta una opción, `:case_sensitive` para lograr este propósito ([Listado 6.26](#)).

Listado 6.26: Validando la unicidad de las direcciones electrónicas, ignorando mayúsculas/minúsculas. **VERDE**

app/models/user.rb

```

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

```

Observe que simplemente hemos reemplazado `true` en el Listado 6.24 con `case_sensitive: false` en el Listado 6.26. (Rails deduce que `uniqueness` debería ser `true` también.)

En este momento, nuestra aplicación obliga la unicidad de la dirección electrónica—con una advertencia importante—, y nuestro conjunto de pruebas debería pasar:

Listado 6.27: VERDE

```
$ bundle exec rake test
```

Sólo hay un pequeño problema, y es que *la validación de unicidad de Active Record no garantiza unicidad a nivel de la base de datos*. Aquí hay un escenario que replica el problema:

1. Alicia se registra en la aplicación de ejemplo, con la dirección electrónica `alice@wonderland.com`.
2. Alicia accidentalmente presiona el botón “Submit” *dos veces*, enviando dos peticiones en una sucesión rápida.
3. La siguiente secuencia ocurre: la petición 1 crea un usuario en memoria que pasa la validación, la petición 2 hace lo mismo, la petición 1 es guardada en la base de datos, la petición 2 también.
4. Resultado: dos registros de usuario con la misma dirección electrónica, a pesar de la validación de unicidad

Si la secuencia anterior parece increíble, créame, no lo es: puede suceder en cualquier sitio web Rails con tráfico importante (lo cual alguna vez aprendí de la forma más dura). Afortunadamente, la solución es sencilla de implementar: sólo necesitamos obligar la unicidad a nivel base de datos así como a nivel modelo. Nuestro plan es crear un *índice* en base de datos para la columna `email` (Recuadro 6.2), y luego requerir que el índice sea único.

Recuadro 6.2. Índices de base de datos

Cuando creamos una columna en una tabla de la base de datos, es importante considerar si necesitaremos *buscar* registros mediante esa columna. Considere, por ejemplo, el atributo `email` creado por la migración del [Listado 6.2](#). Cuando permitamos que los usuarios inicien una sesión en la aplicación de ejemplo a partir del [Capítulo 7](#), necesitaremos encontrar el registro del usuario correspondiente a la dirección electrónica proporcionada. Desafortunadamente, basados en el ingenuo modelo de datos, la única forma de encontrar a un usuario dada su dirección de correo es buscar en *cada* registro de usuario en la base de datos y comparar su atributo `email` con el correo dado—lo que significa que tenemos que revisar *todos* los registros (puesto que el usuario puede ser el último en la base de datos). Esto se conoce en el negocio de las bases de datos como un *escaneo completo a la tabla*, y para un sitio real, con miles de usuarios, es una [algo malo](#).

Poner un índice a la columna `email` arregla el problema. Para entender qué es un índice de base de datos, es útil considerar la analogía con el índice de un libro. En un libro, para encontrar todas las ocurrencias de una cadena dada, digamos “foobar”, tendría que revisar cada página en busca de “foobar”—la versión en papel del escaneo completo a la tabla. Con un índice en el libro, por otra parte, puede buscar “foobar” en el índice para ver qué páginas contienen “foobar”. Un índice de base de datos funciona esencialmente de la misma forma.

El índice de la columna `email` representa una actualización a nuestros requerimientos del modelo de datos, el cual (como se revisó en la [Sección 6.1.1](#)) es manejado por Rails mediante migraciones. Revisamos en la [Sección 6.1.1](#) que cuando generamos el modelo `User`, automáticamente creaba una nueva migración ([Listado 6.2](#)); en este caso, estamos agregando estructura al modelo existente, por lo que necesitamos crear una migración directamente usando el generador de `migraciones`:

```
$ rails generate migration add_index_to_users_email
```

A diferencia de la migración para usuarios, la migración para la unicidad de la dirección electrónica no está predefinida, por lo que necesitamos agregarle el contenido del [Listado 6.28](#).¹⁴

Listado 6.28: La migración para aplicar la unicidad de la dirección electrónica.

```
db/migrate/[timestamp]_add_index_to_users_email.rb
```

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

Esto utiliza un método Rails llamado `add_index` para agregar un índice a la columna `email` de la tabla `users`. El índice por sí mismo no obliga la unicidad, pero con la opción `unique: true` sí lo hace.

El paso final es migrar la base de datos:

```
$ bundle exec rake db:migrate
```

(Si esto falla, trate de cerrar todas las sesiones de consola abiertas que estén ejecutándose en un ambiente aislado, lo cual puede bloquear la base de datos y evitar que se realicen migraciones.)

En este momento, el conjunto de pruebas debería estar en **ROJO** debido a una violación de la restricción de unicidad en los archivos de datos denominados *fixtures*, que contienen datos de ejemplo para las pruebas de la base de datos. Los datos de ejemplo de usuario fueron generados automáticamente en el [Listado 6.1](#), y como se observa en el [Listado 6.29](#) las direcciones electrónicas no

¹⁴Por supuesto, podemos únicamente editar el archivo de migración para la tabla `users` del [Listado 6.2](#), pero eso implicaría realizar una operación que deshaga esa migración y luego la vuelva a aplicar. El modo Rails es utilizar migraciones cada vez que descubrimos que nuestro modelo de datos necesita cambiar.

son únicas. (Tampoco son *válidas*, pero los datos de ejemplo no son procesados por las validaciones.)

Listado 6.29: Los datos de ejemplo de usuario. ROJO

test/fixtures/users.yml

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

one:
  name: MyString
  email: MyString

two:
  name: MyString
  email: MyString
```

Como no necesitaremos datos de ejemplo sino hasta el Capítulo 8, por ahora sólo los removeremos, dejando el archivo de datos de ejemplo vacío (Listado 6.30).

Listado 6.30: Un archivo de datos de ejemplo vacío. VERDE

test/fixtures/users.yml

```
# empty
```

Después de haber abordado el asunto de la unicidad, hay un cambio más que necesitamos hacer para asegurar la unicidad de la dirección electrónica. Algunos adaptadores de base de datos utilizan índices sensibles a mayúsculas/minúsculas, que consideran las cadenas “Foo@ExAMPLE.CoM” y “foo@example.com” como distintas, pero nuestra aplicación trata estas direcciones como iguales. Para evitar esta incompatibilidad, estandarizaremos todas nuestras direcciones de correo para que sean minúsculas, convirtiendo “Foo@ExAMPLE.CoM” a “foo@example.com” antes de guardarla en la base de datos. La forma de hacer esto es con una *llamada de regreso*, que es un método que es invocado en un punto particular del ciclo de vida de un objeto *Active Record*. En este caso, el punto es antes de que el objeto sea guardado, por lo que utilizaremos una llamada de regreso **before_save** para que convierta a minúsculas el atributo

`email` antes de almacenar el registro.¹⁵ El resultado aparece en el Listado 6.31. (Esto es sólo una primera implementación; discutiremos este tema nuevamente en la Sección 10.1.1, donde utilizaremos la convención preferida *método de referencia* para definir llamadas de regreso.)

Listado 6.31: Asegurando la unicidad de la dirección electrónica convirtiendo a minúsculas el atributo email. VERDE

`app/models/user.rb`

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

El código del Listado 6.31 pasa un bloque a la llamada de regreso `before_save` y asigna la dirección de correo del usuario con su valor actual convertido a minúsculas utilizando el método `downcase` para cadenas de caracteres. (Escribir una prueba para la conversión a minúsculas de la dirección electrónica se deja como ejercicio (Sección 6.5).)

En el Listado 6.31, pudimos haber escrito la asignación como

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

(donde `self` se refiere al usuario actual), pero dentro del modelo de usuario, la palabra reservada `self` es opcional en el lado derecho:

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

Nos encontramos con este concepto brevemente en el contexto de `reverse` en el método `palindrome` (Sección 4.4.2), en el que también observamos que `self` no es opcional en una asignación, por lo que

¹⁵Revise la [página del API de Rails sobre llamadas de regreso](#) para mayor información acerca de cuáles soporta Rails.

```
email = email.downcase
```

no funcionaría. (Discutiremos este tema con mayor detalle en la [Sección 8.4](#).)

En este momento, el escenario de Alicia funcionaría correctamente: la base de datos guardará un registro usuario basado en la primera petición, y rechazará el segundo por violar la restricción de unicidad. (Un error aparecerá en la bitácora de Rails, pero eso no causa ningún daño.) Más aún, agregar este índice en el atributo `email` logra un segundo objetivo, mencionado brevemente en la [Sección 6.1.4](#): como se observa en el Recuadro 6.2, el índice sobre el atributo `email` arregla un potencial problema de eficiencia al prevenir un escaneo completo a la tabla cuando buscamos usuarios a partir de su dirección electrónica.

6.3 Agregando una contraseña segura

Ahora que hemos definido validaciones para los campos `name` y `email`, estamos listos para agregar el último de los atributos de usuario básicos: una contraseña segura. El algoritmo es requerir que cada usuario tenga una contraseña (con su respectiva confirmación), y luego almacenar una versión *hash* de la contraseña en la base de datos. (Existe una potencial confusión aquí. En el contexto actual, un *hash* no se refiere a la estructura de datos Ruby que vimos en la [Sección 4.3.3](#) sino al resultado de aplicar una función irreversible llamada [función hash](#) a los datos de entrada.) También agregaremos una forma de *autenticar* al usuario basado en una contraseña dada, un método que utilizaremos en el [Capítulo 8](#) para permitir que los usuarios inicien sesión en el sitio.

La forma de autenticar usuarios será tomar la contraseña enviada, pasarla por la función `hash`, y comparar el resultado con el valor que tenemos almacenado en la base de datos. Si coinciden, entonces la contraseña proporcionada es correcta y el usuario es autenticado. Al comparar los valores de la función `hash` en vez de comparar las contraseñas directamente, seremos capaces de autenticar usuarios sin almacenar las contraseñas mismas. Esto significa que, aún si la base de datos se ve comprometida, las contraseñas de nuestros usuarios están seguras.

6.3.1 Una contraseña procesada con la función hash

La mayor parte de la maquinaria para la contraseña segura será implementada usando un sólo método de Rails llamado `has_secure_password`, que incluiremos en el modelo de usuario como sigue:

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

Cuando se incluye en un modelo como el anterior, este único método agrega la siguiente funcionalidad:

- La habilidad de guardar un atributo `password_digest` en la base de datos, que corresponde al hash de la contraseña
- Una pareja de atributos virtuales¹⁶ (`password` y `password_confirmation`), incluyendo las validaciones de presencia durante la creación del objeto y una validación que requiera que ambos atributos coincidan
- Un método `authenticate` que regresa al objeto usuario si la contraseña es correcta (o `falso` en caso contrario)

El único requerimiento para que `has_secure_password` haga su magia es que el modelo correspondiente tenga un atributo llamado `password_digest`. (El nombre *digest* viene de la terminología de [funciones hash criptográficas](#). En este contexto, *el hash de la contraseña y la digestión de la contraseña* significan lo mismo.)¹⁷ En el caso del modelo de usuario, esto nos lleva al modelo de datos que se muestra en la Figura 6.8.

¹⁶En este contexto, *virtual* significa que los atributos existen en el objeto del modelo, pero no corresponden a columnas en la base de datos.

¹⁷Los hash de las contraseñas a menudo son erróneamente conocidos como *contraseñas encriptadas*. Por ejemplo, el [código fuente de has_secure_password](#) comete este error, de igual forma que las dos primeras ediciones de este tutorial. Esta terminología está equivocada porque por diseño, la encripción es *reversible*—la habilidad de encriptar implica la habilidad de *desencriptar*. Por el contrario, la intención de calcular un hash sobre la contraseña es que sea *irreversible*, de forma que, computacionalmente no es posible deducir la contraseña original a partir del hash de la misma. (Agradezco al lector Andy Philips por resaltar este punto e invitarme a corregir la terminología.)

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

Figura 6.8: El modelo de datos de usuario con el atributo **password_digest** añadido.

Para implementar el modelo de datos de la Figura 6.8, primero generamos una migración apropiada para la columna **password_digest**. Podemos elegir el nombre que queramos para la migración, pero es conveniente agregarle el sufijo **to_users**, puesto que de esta forma Rails automáticamente construye una migración para agregar columnas a la tabla **users**. El resultado, con el nombre de migración **add_password_digest_to_users**, es el siguiente:

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

Aquí hemos proporcionado el argumento **password_digest:string** con el nombre y tipo de atributo que deseamos crear. (Compare esto con la generación original de la tabla **users** del Listado 6.1, que incluía los argumentos **name:string** y **email:string**.) Al incluir **password_digest:string**, le hemos dado suficiente información a Rails para construir la migración entera por nosotros, como se muestra en el Listado 6.32.

Listado 6.32: La migración para agregar una columna **password_digest** a la tabla **users**.

```
db/migrate/[timestamp]_add_password_digest_to_users.rb
```

```
class AddPasswordDigestToUsers < ActiveRecord::Migration
  def change
    users :password_digest :string
  end
end
```

El [Listado 6.32](#) utiliza el método `add_column` para agregar una columna `password_digest` a la tabla `users`. Para aplicarla, sólo migre la base de datos:

```
$ bundle exec rake db:migrate
```

Para obtener el hash de la contraseña, `has_secure_password` utiliza una moderna función hash llamada `bcrypt`. Al obtener el hash de la contraseña con bcrypt, nos aseguramos de que un atacante no será capaz de ingresar al sitio aún si se las arregla para obtener una copia de la base de datos. Para utilizar bcrypt en la aplicación de ejemplo, necesitamos agregar la gema `bcrypt` a nuestro `Gemfile` ([Listado 6.33](#)).

Listado 6.33: Agregando `bcrypt` al `Gemfile`.

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

gem 'rails',                  '4.2.2'
gem 'bcrypt',                  '3.1.7'
.
```

Luego ejecute `bundle install` como es usual:

```
$ bundle install
```

6.3.2 El usuario tiene una contraseña segura

Ahora que hemos agregado al modelo usuario el atributo `password_digest` requerido e instalado bcrypt, estamos listos para usar `has_secure_password`, como se muestra en el [Listado 6.34](#).

Listado 6.34: Agregando `has_secure_password` al modelo usuario. ROJO
`app/models/user.rb`

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

Como nos indica el ROJO del Listado 6.34, las pruebas están fallando en este momento, como puede confirmar en la línea de comandos:

Listado 6.35: ROJO

```
$ bundle exec rake test
```

La razón es que, como hicimos notar en la Sección 6.3.1, `has_secure_password` obliga las validaciones sobre los atributos virtuales `password` y `password_confirmation`, pero las pruebas del Listado 6.25 crean una variable `@user` sin estos atributos:

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

Por lo que, para hacer que las pruebas pasen, necesitamos agregar una contraseña y su confirmación, como se muestra en el Listado 6.36.

Listado 6.36: Agregando una contraseña y su confirmación. VERDE

`test/models/user_test.rb`

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
```

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

Ahora las pruebas deberían estar en **VERDE**:

Listado 6.37: **VERDE**

```
$ bundle exec rake test
```

En un momento más veremos los beneficios de agregar **has_secure_password** al modelo usuario ([Sección 6.3.4](#)), pero primero agregaremos un requerimiento mínimo a la seguridad de la contraseña.

6.3.3 Estándares mínimos de contraseña

En general, es una buena práctica obligar a que se cumplan los estándares mínimos sobre las contraseñas para hacer que sean más difíciles de adivinar. Hay muchas opciones para [determinar la fortaleza de la contraseña en Rails](#), pero por simplicidad sólo obligaremos a que la contraseña tenga una longitud mínima y que no esté en blanco. Escoger una longitud de 6 es un mínimo razonable que nos lleva a la prueba de validación que se muestra en el [Listado 6.38](#).

Listado 6.38: Probando una longitud de contraseña mínima. **ROJO**

```
test/models/user_test.rb
```

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

```

end
.
.
.

test "password should be present (nonblank)" do
  @user.password = @user.password_confirmation = " " * 6
  assert_not @user.valid?
end

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

```

Observe el uso de la asignación múltiple compacta

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

del [Listado 6.38](#). Esto se encarga de asignar un valor particular a la contraseña y su confirmación al mismo tiempo (en este caso, una cadena de longitud 5, construida usando la multiplicación de cadenas como se hizo en el [Listado 6.14](#)).

Puede que usted adivine el código para forzar la restricción de una longitud **mínima** si recordamos que aplicamos una restricción similar al nombre del usuario, donde restringíamos la longitud **máxima** de este atributo ([Listado 6.16](#)):

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

Combinando esto con la validación de presencia ([Sección 6.2.2](#)) para evitar contraseñas en blanco, llegamos al modelo de usuario que se muestra en el [Listado 6.39](#). (Sucede que el método **has_secure_password** incluye una validación de presencia, pero sólo aplica para los registros nuevos, lo que causa problemas cuando actualizamos usuarios.)

Listado 6.39: La implementación completa para contraseñas seguras. VERDE
app/models/user.rb

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

En este momento, las pruebas deberían estar en VERDE:

Listado 6.40: VERDE

```
$ bundle exec rake test:models
```

6.3.4 Creando y autenticando un usuario

Ahora que el modelo básico de usuario está completo, crearemos un usuario en la base de datos como preparación para crear una página que muestre la información del usuario en la Sección 7.1. También echaremos un vistazo más concreto a los efectos de agregar `has_secure_password` al modelo `User`, incluyendo una revisión al importante método `authenticate`.

Puesto que los usuarios aún no pueden registrarse en la aplicación de ejemplo a través de la web—ése es el objetivo del Capítulo 7—utilizaremos la consola Rails para crear un nuevo usuario manualmente. Por conveniencia, usaremos el método `create` que revisamos en la Sección 6.1.3, pero en el presente caso, nos preocuparemos de *no* iniciar en un ambiente aislado, de forma que el usuario resultante sea registrado en la base de datos. Esto significa que iniciaremos una sesión `rails console` ordinaria y luego crearemos el usuario con un nombre y dirección electrónica válidos junto con una contraseña y su confirmación también válidas:

```
$ rails console
>> User.create(name: "Michael Hartl", email: "mhartl@example.com",
?>                 password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2014-09-11 14:26:42", updated_at: "2014-09-11 14:26:42",
password_digest: "$2a$10$sLcMI2f8VglgirzjSJ0ln.Fv9NdLbqmR4rdTWIXY1G...">
```

Para verificar que esto funcionó, echemos un vistazo a la tabla `users` resultante, en la base de datos de desarrollo, usando el Navegador para SQLite, como se muestra en la [Figura 6.9](#).¹⁸ (Si usted está utilizando el IDE en la nube, debería descargar el archivo de la base de datos como se muestra en la [Figura 6.5](#).) Observe las columnas correspondientes a los atributos del modelo de datos definidos en la [Figura 6.8](#).

Regresando a la consola, podemos ver el efecto de `has_secure_password` del [Listado 6.39](#) observando el atributo `password_digest`:

```
>> user = User.find_by(email: "mhartl@example.com")
>> user.password_digest
=> "$2a$10$YmQTuuDNOszvu5yi7auOC.F4G//FGhyQSWCpghqRWQWITUY1G3XVy"
```

Esta es la versión hash de la contraseña ("`foobar`") utilizada para inicializar el objeto usuario. Como fue construída usando bcrypt, es computacionalmente imposible utilizar la digestión para descubrir la contraseña original.¹⁹

Como observamos en la [Sección 6.3.1](#), `has_secure_password` automáticamente agrega un método `authenticate` a los correspondientes objetos del modelo. Este método determina si una contraseña dada es válida para un usuario

¹⁸Si por alguna razón no funciona, puede reiniciar la base de datos como sigue:

1. Cierre la consola.
2. Ejecute `$ rm -f development.sqlite3` en la línea de comandos para eliminar la base de datos. (Aprenderemos un método más elegante para hacer esto en el [Capítulo 7](#).)
3. Vuelva a ejecutar las migraciones usando `$ bundle exec rake db:migrate`.
4. Reinicie la consola.

¹⁹Por diseño, el algoritmo bcrypt produce un [salted hash](#), que protege contra dos clases importantes de ataques ([los ataques de diccionario](#) y [los ataques de la tabla arcoiris](#)).

The screenshot shows the Database Browser for SQLite interface. The title bar reads "Database Browser for SQLite - /Users/mhartl/repos/sample_app_3rd_edition_dev/db/development.sqlite3". The main window displays the "users" table with the following data:

	id	name	email	created_at	updated_at	password_digest
1	1	Michael Hartl	mhartl@example.com	2014-09-11 14:26:42.28...	2014-09-11 14:26:42.28...	\$2a\$10\$slcMI2f8VgIgrzjSjOl...

At the bottom of the table view, there is a footer with navigation buttons: '< 1 - 1 of 1 >', 'Go to:', and a text input field containing '1'. The status bar at the bottom right shows 'UTF-8'.

Figura 6.9: Un registro de usuario en la base de datos SQLite `db/development.sqlite3`.

en particular, calculando su digestión y comparando el resultado con el `password_digest` de la base de datos. En el caso del usuario que recién creamos, podemos intentar con un par de contraseñas inválidas como sigue:

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

Aquí `user.authenticate` regresa `falso` para una contraseña inválida. Si por el contrario, nos autenticamos con la contraseña correcta, `authenticate` regresa el objeto usuario mismo:

```
>> user.authenticate("foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2014-07-25 02:58:28", updated_at: "2014-07-25 02:58:28",
password_digest: "$2a$10$YmQTuuDNOszvu5yi7auOC.F4G//FGhyQSWCpghqRWQW...">>
```

En el Capítulo 8, utilizaremos el método `authenticate` para iniciar la sesión de los usuarios registrados en nuestro sitio. De hecho, no es tan importante que el método `authenticate` nos regrese al usuario; lo único que nos interesa es que regrese un valor `verdadero` en un contexto booleano. Puesto que el objeto usuario no es `níl` ni `falso`, esto funciona bien:²⁰

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

6.4 Conclusión

Empezando desde cero, en este capítulo creamos un modelo `User` que funciona con atributos nombre, dirección electrónica y contraseña, junto con validaciones que aplican importantes restricciones en sus valores. Adicionalmente, tenemos la habilidad de autenticar de forma segura a los usuarios, mediante una

²⁰Recuerde de la Sección 4.2.3 que `!!` convierte un objeto en su correspondiente valor booleano.

contraseña. Esto representa una extraordinaria cantidad de funcionalidad para sólo doce líneas de código.

En el siguiente [Capítulo 7](#), elaboraremos una forma de registro que funcione para crear nuevos usuarios, junto con una página que muestre la información de cada usuario. En el [Capítulo 8](#), usaremos la maquinaria de autenticación de la [Sección 6.3](#) para permitir que los usuarios inicien sesión en el sitio.

Si usted está usando Git, es ahora buen momento para notificar sus cambios si es que hace rato que no lo hace:

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

Luego intégrelos en la rama principal y súbalos al repositorio remoto:

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

Para que el modelo **User** funcione en producción, necesitamos ejecutar las migraciones en Heroku, lo cual podemos hacer mediante el comando **heroku run**:

```
$ bundle exec rake test  
$ git push heroku  
$ heroku run rake db:migrate
```

Podemos verificar que esto funciona desde una consola en producción:

```
$ heroku run console --sandbox  
>> User.create(name: "Michael Hartl", email: "michael@example.com",  
?>           password: "foobar", password_confirmation: "foobar")  
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",  
created_at: "2014-08-29 03:27:50", updated_at: "2014-08-29 03:27:50",  
password_digest: "$2a$10$IViF0Q5j3hsEVgHgrrKH3uDou86Ka21EPz8zkwQopwj...">>
```

6.4.1 Qué aprendimos en este capítulo

- Las migraciones nos permiten modificar nuestro modelo de datos de la aplicación.
- *Active Record* viene con un gran número de métodos para crear y manipular los modelos de datos.
- Las validaciones de *Active Record* nos permiten imponer restricciones a los datos de nuestros modelos.
- Las validaciones comunes incluyen presencia, longitud y formato.
- Las expresiones regulares son críticas pero poderosas.
- Definir un índice en la base de datos mejora la eficiencia de búsqueda al mismo tiempo que nos permite imponer la unicidad a nivel base de datos.
- Podemos agregar una contraseña segura al modelo usando el método pre-construído `has_secure_password`.

6.5 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Agregue una prueba para la conversión a minúsculas de la dirección electrónica del Listado 6.31, como se muestra en el Listado 6.41. Esta prueba utiliza el método `reload` para volver a cargar un valor de la base de datos y el método `assert_equal` para comprobar que son iguales. Para verificar que el Listado 6.41 prueba lo correcto, comente la línea de código

before_save para hacer que esté en ROJO, luego remueva el comentario para hacer que pase a VERDE.

2. Al ejecutar el conjunto de pruebas, verique que la llamada de regreso **before_save** puede ser escrita usando el método **email.downcase!** para modificar el atributo **email** directamente, como se muestra en el Listado 6.42.
3. Como observamos en la Sección 6.2.4, la expresión regular para la dirección electrónica del Listado 6.21 nos permite direcciones inválidas con puntos consecutivos en el nombre del dominio, por ejemplo, direcciones con la forma *foo@bar.com*. Agregue esta dirección a la lista de direcciones inválidas del Listado 6.19 para hacer que la prueba falle, y luego utilice la expresión regular más compleja que mostramos en el Listado 6.43 para hacer que pasen las pruebas.

Listado 6.41: Una prueba para la conversión a minúsculas del correo electrónico como se indica en el Listado 6.31.

```
test/models/user_test.rb

require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

  test "email addresses should be unique" do
    duplicate_user = @user.dup
    duplicate_user.email = @user.email.upcase
    @user.save
    assert_not duplicate_user.valid?
  end

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

```

    assert_equal mixed_case_email.downcase, @user.reload.email
end

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

```

Listado 6.42: Una implementación alternativa de la función `before_save`.

VERDE

app/models/user.rb

```

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

```

Listado 6.43: Deshabilitando puntos dobles en los nombres de dominio de la dirección electrónica. VERDE*app/models/user.rb*

```

class User < ActiveRecord::Base
  before_save { email.downcase! }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\.-]+@[a-z\d\.-]+(\.[a-z\d\.-]+)*\.[a-z]+\z/i
  validates :email, presence: true,
    format: { with: VALID_EMAIL_REGEX },
    uniqueness: { case_sensitive: false }
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }
end

```

Capítulo 7

Registro

Ahora que tenemos un modelo de usuario funcionando, es momento de agregar una funcionalidad de la que pocos sitios pueden carecer: permitir a los usuarios registrarse. Utilizaremos un *formulario* HTML para enviar la información de registro del usuario a nuestra aplicación ([Sección 7.2](#)), la cual utilizaremos luego para crear un nuevo usuario y guardar sus atributos en la base de datos ([Sección 7.4](#)). Al final del proceso de registro, es importante mostrar una página del perfil con la información del usuario recién creado, por lo que empezaremos creando una página para *mostrar* usuarios, que nos servirá como primer paso hacia la implementación de la arquitectura REST para usuarios ([Sección 2.2.2](#)). En el camino, elaboraremos más sobre nuestro trabajo de la [Sección 5.3.4](#) para escribir pruebas de integración sencillas y expresivas.

En este capítulo, dependeremos de las validaciones al modelo de usuario del [Capítulo 6](#) para incrementar las probabilidades de que los nuevos usuarios proporcionen una dirección electrónica válida. En el [Capítulo 10](#), nos *aseguraremos* de la validez de la dirección de correo al agregar un paso adicional al registro de usuario, para la *activación de la cuenta*.

7.1 Mostrando usuarios

En esta sección, daremos nuestros primeros pasos hacia la versión final del perfil, elaborando una pagina que muestre el nombre del usuario y una foto, como

se indica en el bosquejo de la Figura 7.1.¹ Nuestro objetivo final para las páginas de perfil de usuarios es mostrar la imagen del perfil del usuario, datos de usuario básicos y una lista de microposts, como se bosquejó en la Figura 7.2.² (La Figura 7.2 contiene nuestro primer ejemplo de texto *lorem ipsum*, el cual tiene una fascinante historia que le sugiero lea en algún momento.) Terminaremos esta tarea, y con ella, la aplicación de ejemplo, en el Capítulo 12.

Si usted está utilizando Git para el control de versiones, cree una nueva rama como es usual:

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

7.1.1 Depuración y ambientes Rails

Los perfiles en esta sección conformarán las primeras páginas verdaderamente dinámicas de nuestra aplicación. Aunque la vista existirá como una sola página de código, cada perfil será personalizado utilizando la información recuperada de la base de datos de la aplicación. Como preparación para agregar páginas dinámicas a nuestra aplicación de ejemplo, ahora es un buen momento para agregar alguna información de depuración a nuestra estructura de diseño del sitio (Listado 7.1). Esto despliega información útil acerca de cada página que utiliza el método incorporado **debug** y la variable **params** (de los que aprenderemos más en la Sección 7.1.2).

Listado 7.1: Agregando información de depuración a la estructura de diseño del sitio.

```
app/views/layouts/application.html.erb
```

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

¹Mockingbird no soporta imágenes personalizadas como la foto de perfil de la Figura 7.1; Yo puse ésa manualmente utilizando GIMP.

²El hipopótamo aquí proviene de <http://www.flickr.com/photos/43803060@N00/24308857/>.

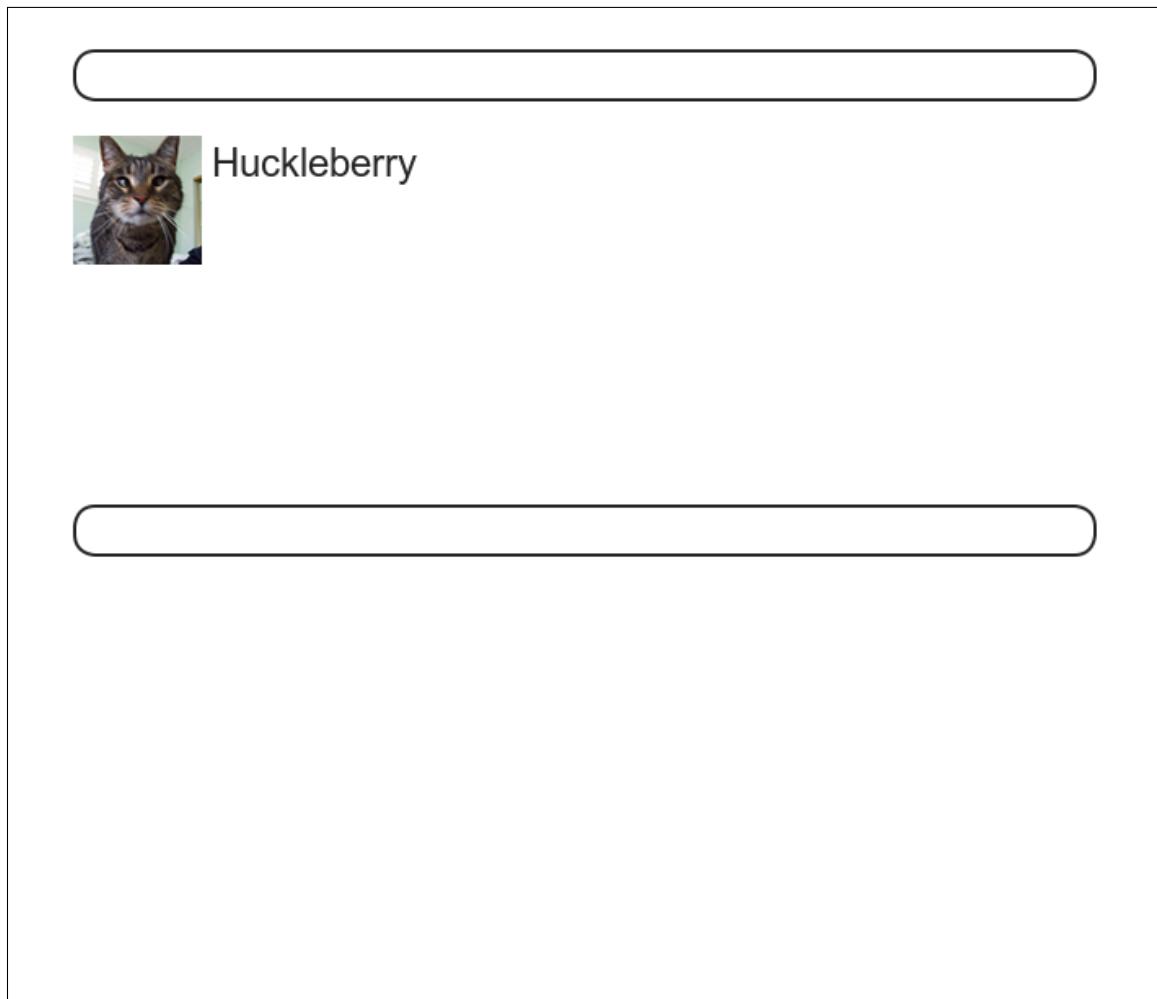


Figura 7.1: Un bosquejo del perfil de usuario elaborado en esta sección.

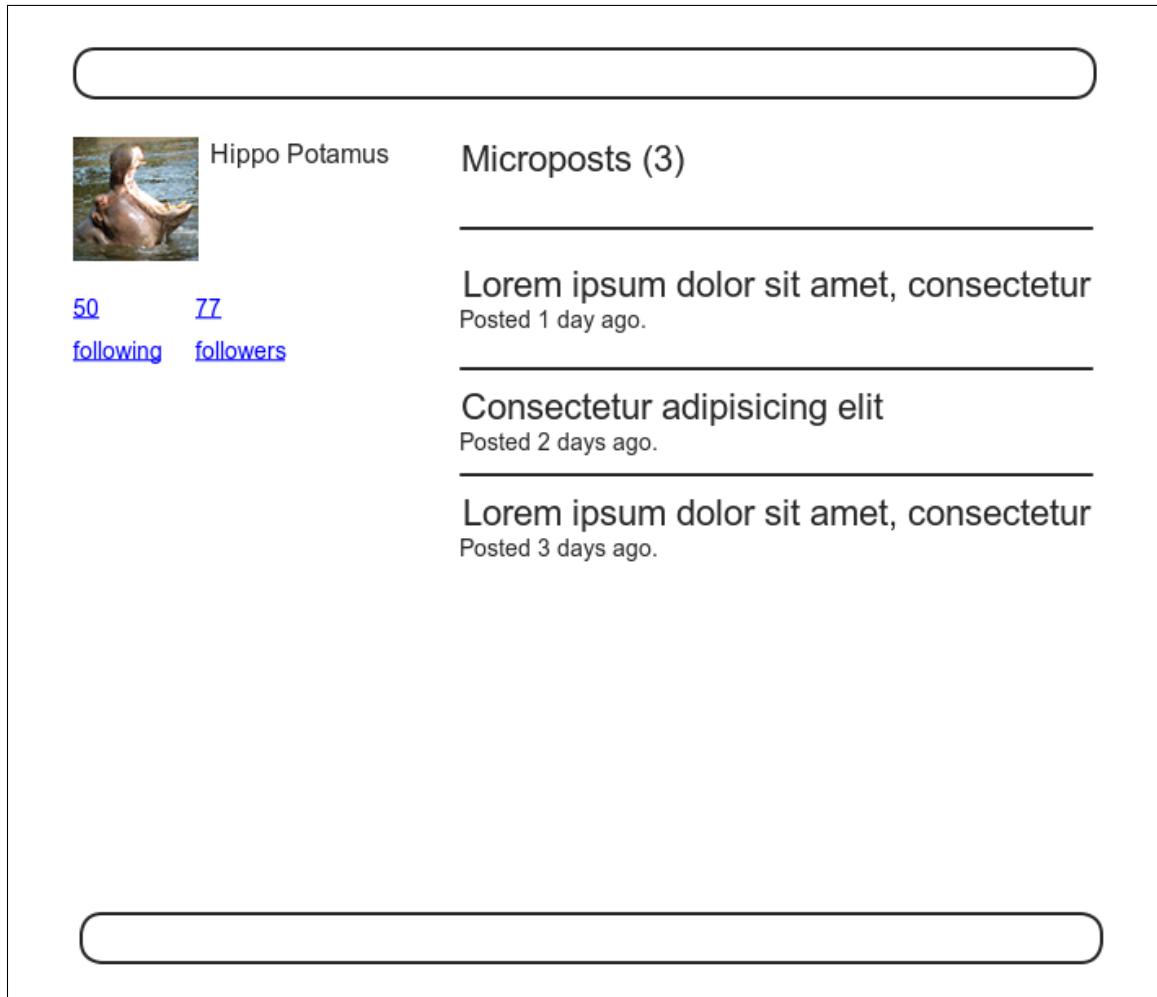


Figura 7.2: Un bosquejo de nuestra mejor hipótesis de la página del perfil final.

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

Puesto que no queremos desplegar esta información de depuración a los usuarios de la aplicación en producción, el [Listado 7.1](#) utiliza

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

para restringir la información de depuración al *ambiente de desarrollo*, que es uno de los tres ambientes definidos por default en Rails ([Recuadro 7.1](#)).³ En particular, `Rails.env.development?` es **verdadero** únicamente en un ambiente de desarrollo, por lo que el código Ruby embebido

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

no será insertado en aplicaciones de producción o pruebas. (Insertar la información de depuración en las pruebas probablemente no ocasiona ningún daño, pero tampoco aporta ningún beneficio, por lo que lo mejor es restringir el despliegue de la depuración al ambiente de desarrollo exclusivamente.)

Recuadro 7.1. Ambientes Rails

Rails viene equipado con tres ambientes: `pruebas`, `desarrollo`, y `producción`. El ambiente de la consola Rails por default es `desarrollo`:

³Usted puede definir sus propios ambientes personalizados también; revise el [RailsCast](#) acerca de cómo agregar un ambiente para mayor información.

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

Como puede observar, Rails proporciona un objeto `Rails` con un atributo `env` y métodos booleanos asociados al ambiente, de forma que, por ejemplo, `Rails.env.test?` regresa `verdadero` en un ambiente de pruebas y `falso` en cualquier otro.

Si en algún momento necesita ejecutar una consola en un ambiente diferente (para depurar una prueba, por ejemplo), puede pasar el ambiente como parámetro al script `console`:

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

De igual forma que con la consola, `desarrollo` es el ambiente por default para el servidor Rails, pero también puede ejecutarlo en un ambiente distinto:

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

Si usted visualiza su aplicación ejecutándose en producción, ésta no funcionará sin una base de datos de producción, la cual puede crear ejecutando `rake db:migrate` en producción:

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

(Me parece confuso que la consola, el servidor y los comandos de migración especifiquen ambientes que no son los de default en tres formas mutuamente incompatibles, lo cual es la razón por la que me tomé la molestia de mostrar los tres.)

Por cierto, si usted ha desplegado su aplicación de ejemplo en Heroku, puede revisar su ambiente mediante `heroku run console`:

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

Naturalmente, puesto que Heroku es una plataforma para sitios en producción, ejecuta cada aplicación en un ambiente de producción.

Para hacer que la salida de la depuración se vea bien, agregaremos algunas reglas a la hoja de estilo personalizada que creamos en el [Capítulo 5](#), como se muestra en el [Listado 7.2](#).

Listado 7.2: Agregando código para una caja de depuración atractiva, incluyendo un *mixin* de Sass.

app/assets/stylesheets/custom.css.scss

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

/* mixins, variables, etc. */

$gray-medium-light: #eaeaea;

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

```
/*
 * miscellaneous */

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

Esto nos presenta la funcionalidad Sass *mixin*, en este caso llamada **box-sizing**. Un *mixin* permite que un grupo de reglas CSS sean empacadas y utilizadas en múltiples elementos, convirtiendo

```
.debug_dump {
  .
  .
  .
  @include box-sizing;
}
```

en

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

Usaremos este *mixin* nuevamente en la Sección 7.2.1. El resultado de estilizar un recuadro de depuración se muestra en la Figura 7.3.

La salida de la depuración de la Figura 7.3 proporciona información potencialmente útil acerca de la página que está siendo desplegada:

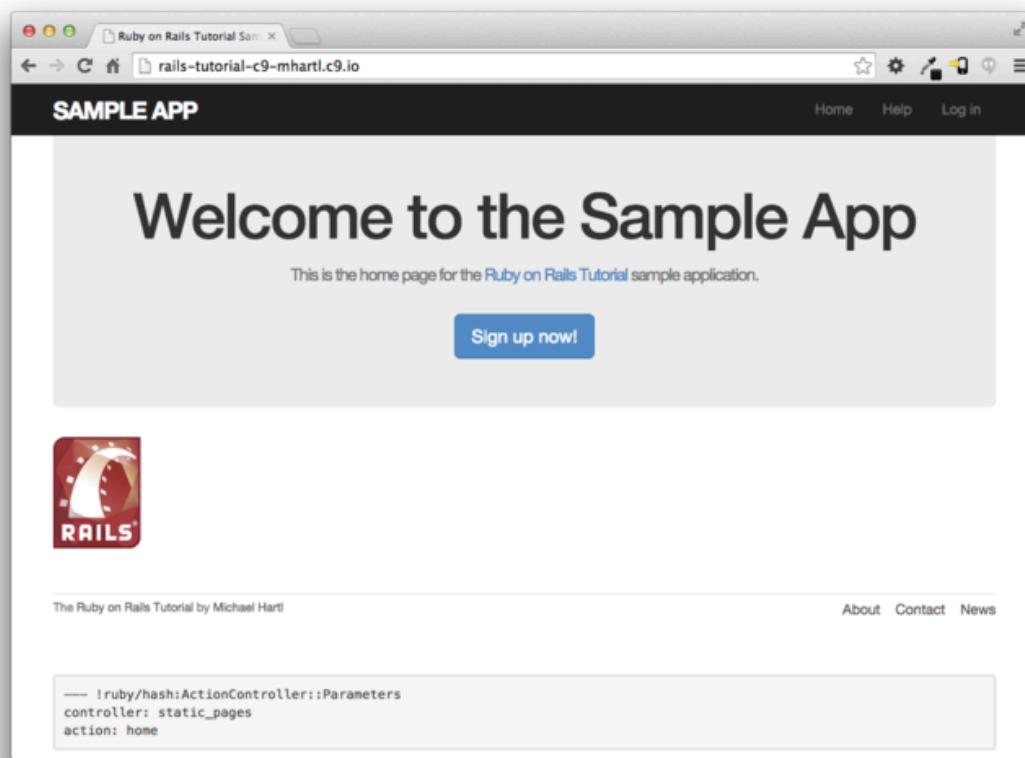


Figura 7.3: La página Home de la aplicación de ejemplo con información de depuración.

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

Esto es una representación de **params** en YAML⁴, que es básicamente un arreglo hash, y en este caso identifica el controlador y la acción de la página. Revisaremos otro ejemplo en la [Sección 7.1.2](#).

7.1.2 Un recurso usuarios

Con la finalidad de elaborar una página del perfil de usuario, necesitamos tener un registro de usuario en la base de datos, lo que nos conduce al problema del huevo y la gallina: ¿cómo puede el sitio tener un usuario antes de que exista una página de registro funcional? Felizmente, este problema ya ha sido resuelto: en la [Sección 6.3.4](#), creamos un registro de usuario manualmente utilizando la consola Rails, por lo que debería haber un usuario en la base de datos:

```
$ rails console
>> User.count
=> 1
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2014-08-29 02:58:28", updated_at: "2014-08-29 02:58:28",
password_digest: "$2a$10$YmQTuuDNOszvu5yi7auOC.F4G//FGhyQSWCpghqRWQW...">>
```

(Si usted no tiene en este momento un usuario en su base de datos, debería visitar la [Sección 6.3.4](#) ahora y completarla antes de continuar.) Observamos en la salida de la consola anterior que el usuario tiene un id **1**, y nuestro objetivo ahora es elaborar una página para mostrar la información de este usuario. Nos apegaremos a las convenciones de la arquitectura REST favorecida por las aplicaciones Rails (Recuadro 2.2), lo que implica que los datos sean representados como *recursos* que pueden ser creados, mostrados, actualizados o destruidos—cuatro

⁴La información de **depuración** de Rails se muestra como **YAML** (un **acrónimo recursivo**, por sus siglas en inglés *YAML Ain't Markup Language*), que es un formato de datos amigable diseñado para ser legible tanto por máquinas como por humanos.

```
ActionController::RoutingError (No route matches [GET] "/users/1"):
web-console (2.0.0.beta3) lib/action_dispatch/debug_exceptions.rb:22:in `middleware_call'
web-console (2.0.0.beta3) lib/action_dispatch/debug_exceptions.rb:13:in `call'
actionpack (4.2.0.beta1) lib/action_dispatch/middleware/show_exceptions.rb:30:in `call'
railties (4.2.0.beta1) lib/rails/rack/logger.rb:38:in `call_app'
railties (4.2.0.beta1) lib/rails/rack/logger.rb:20:in `block in call'
activesupport (4.2.0.beta1) lib/active_support/tagged_logging.rb:68:in `block in tagged'
activesupport (4.2.0.beta1) lib/active_support/tagged_logging.rb:26:in `tagged'
activesupport (4.2.0.beta1) lib/active_support/tagged_logging.rb:68:in `tagged'
railties (4.2.0.beta1) lib/rails/rack/logger.rb:20:in `call'
actionpack (4.2.0.beta1) lib/action_dispatch/middleware/request_id.rb:21:in `call'
rack (1.6.0.beta) lib/rack/methodoverride.rb:22:in `call'
rack (1.6.0.beta) lib/rack/runtime.rb:17:in `call'
```

Figura 7.4: La bitácora de error del servidor para /users/1.

acciones que corresponden a las cuatro operaciones fundamentales POST, GET, PATCH y DELETE definidas por el [estándar HTTP](#) (Recuadro 3.2).

Cuando nos apegamos a los principios REST, los recursos típicamente son referenciados usando el nombre del recurso y un identificador único. Lo que esto significa en el contexto de usuarios—que en este momento estamos considerando a los usuarios como un *recurso*—es que deberíamos visualizar el usuario con id 1 si emitimos una petición GET a la URL /users/1. Aquí la acción **show** es *implícita* en el tipo de petición—cuando las características REST de Rails sean activadas, las peticiones GET serán automáticamente tratadas por la acción **show**.

Revisamos en la [Sección 2.2.1](#) que la página para un usuario con id 1 tiene URL /users/1. Desafortunadamente, visitar esa URL en este momento sólo nos arroja un error, como podemos ver en la bitácora del servidor (Figura 7.4).

Podemos hacer que el direccionamiento para /users/1 funcione agregando una sola línea a nuestro archivo de rutas (**config/routes.rb**):

```
resources :users
```

El resultado aparece en el [Listado 7.3](#).

Petición HTTP	URL	Acción	Ruta nombrada	Propósito
GET	/users	index	users_path	página para enlistar todos los usuarios
GET	/users/1	show	user_path(user)	página para mostrar el usuario con id 1
GET	/users/new	new	new_user_path	página para crear un nuevo usuario (registro)
POST	/users	create	users_path	crea un nuevo usuario
GET	/users/1/edit	edit	edit_user_path(user)	página para actualizar el usuario con id 1
PATCH	/users/1	update	user_path(user)	actualiza el usuario con id 1
DELETE	/users/1	destroy	user_path(user)	borra el usuario con id 1

Tabla 7.1: Rutas RESTful proporcionadas por el recurso **Users** del Listado 7.3.

Listado 7.3: Agregando un recurso **Users** al archivo de rutas.

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help'    => 'static_pages#help'
  get 'about'   => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup'  => 'users#new'
  resources :users
end
```

Aunque nuestra motivación inmediata es elaborar una página para mostrar usuarios, la simple línea `resources :users` no sólo agrega una URL funcional `/users/1`; dota a nuestra aplicación de ejemplo con *todas* las acciones necesarias para un recurso **Users** RESTful,⁵ junto con una gran cantidad de rutas nombradas (Sección 5.3.3) para generar URLs de usuario. La relación resultante entre URLs, acciones y rutas nombradas se muestra en la Tabla 7.1. (Compárela con la Tabla 2.2.) En el transcurso de los siguientes tres capítulos, revisaremos las otras entradas de la Tabla 7.1 conforme vamos llenando todas las acciones necesarias para hacer de **Users** un recurso completamente RESTful.

Con el código del Listado 7.3, el direccionamiento funciona, pero aún no hay una página ahí (Figura 7.5). Para corregir esto, empezaremos con una ver-

⁵Esto significa que el *direcciónamiento* funciona, pero las páginas correspondientes no necesariamente funcionan en este momento. Por ejemplo, `/users/1/edit` es direccionado correctamente a la acción `edit` del controlador de usuarios, pero como la acción `edit` no existe aún realmente, al visitar esa URL nos devolverá un error.

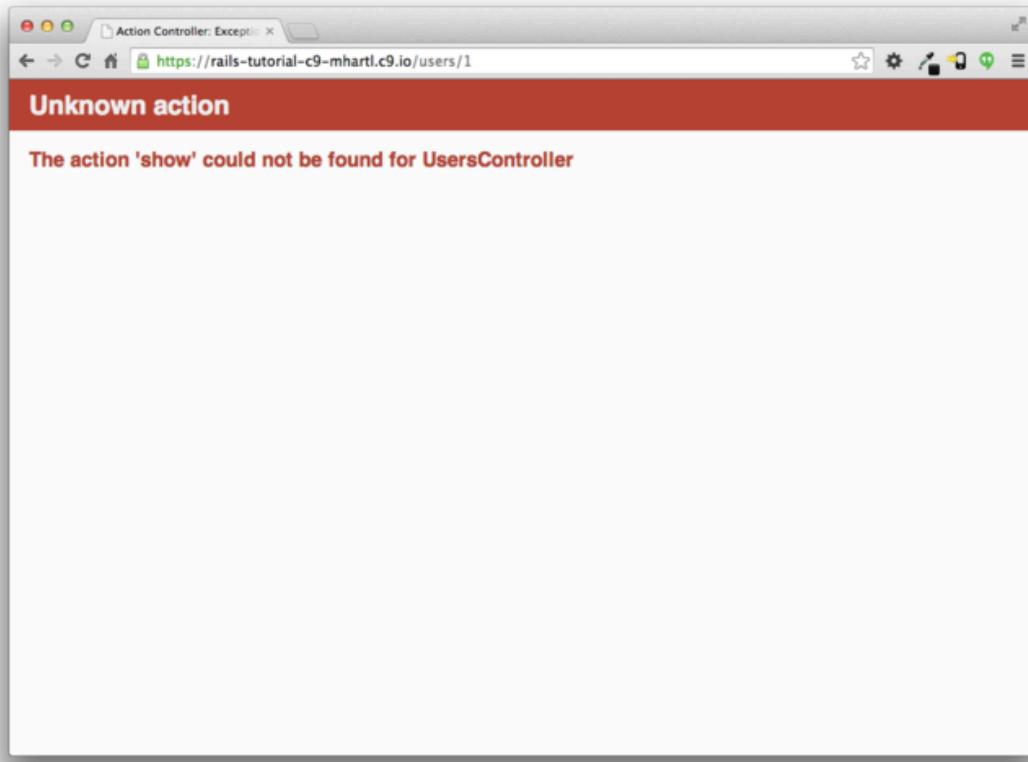


Figura 7.5: La URL /users/1 con direccionamiento pero sin página.

sión minimalista de la página del perfil, en la que profundizaremos en la Sección 7.1.4.

Emplearemos la ubicación estándar de Rails para mostrar un usuario, que es **app/views/users/show.html.erb**. A diferencia de la vista **new.html.-erb**, que creamos con el generador en el Listado 5.28, el archivo **show.html.-erb** aún no existe, por lo que deberá crearlo manualmente y luego asignarle el contenido que se muestra en el Listado 7.4.

Listado 7.4: Una vista mínima para mostrar información de usuario.

app/views/users/show.html.erb

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

Esta vista utiliza Ruby embebido para desplegar el nombre del usuario y la dirección electrónica, suponiendo la existencia de la variable de instancia llamada `@user`. Por supuesto, en algún momento la página real para mostrar usuarios lucirá muy distinta (y no mostrará la dirección de correo públicamente).

Con la finalidad de que la vista que muestra el usuario funcione, necesitamos definir una variable `@user` en la acción `show` correspondiente en el controlador de usuarios. Como puede usted esperar, utilizamos el método `find` del modelo `User` (Sección 6.1.4) para recuperar el usuario de la base de datos, como se muestra en el Listado 7.5.

Listado 7.5: El controlador `Users` con una acción `show`.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController

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

  def new
  end
end
```

Aquí hemos utilizado `params` para recuperar el `user id`. Cuando hacemos la petición apropiada al controlador de usuarios, `params[:id]` será el id del usuario 1, por lo que el efecto es el mismo que el del método `find: User.find(1)` que vimos en la Sección 6.1.4. (Técnicamente, `params[:id]` es la cadena "1", pero `find` es suficientemente inteligente para convertirla a entero.)

Con la vista de usuario definida, la URL `/users/1` funciona perfectamente, como vemos en la Figura 7.6. (Si usted no ha reiniciado el servidor Rails desde que agregó bcrypt, puede hacerlo en este momento.) Observe que la información de depuración de la Figura 7.6 confirma el valor de `params[:id]`:

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

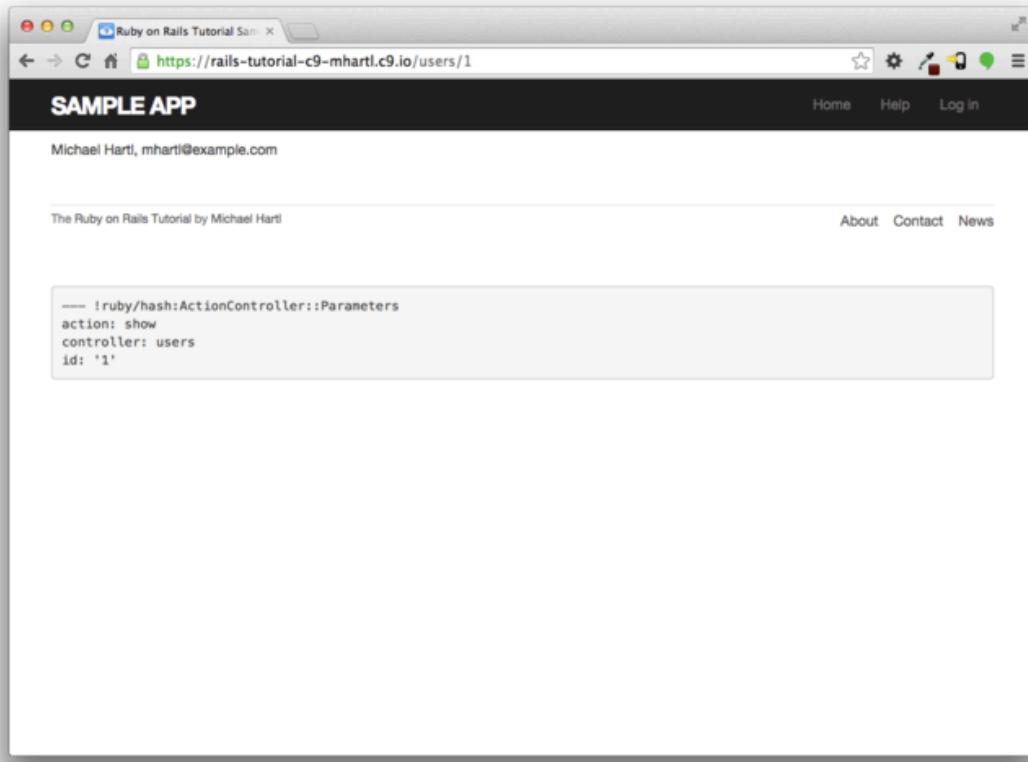


Figura 7.6: La página que muestra al usuario luego de agregar el recurso **Users**.

Es por esto que el código

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

del Listado 7.5 encuentra al usuario con id 1.

7.1.3 Depurador

Revisamos en la Sección 7.1.2 cómo la información de **depuración** puede ayudarnos a entender lo que está sucediendo en nuestra aplicación. En la versión 4.2 de Rails, existe una forma más directa de obtener información de depu-

ración usando la gema `byebug` (Listado 3.2). Para ver cómo funciona, sólo necesitamos agregar la línea `debugger` en nuestra aplicación, como se muestra en el Listado 7.6.

Listado 7.6: El controlador `Users` con un depurador.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController

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

  def new
  end
end
```

Ahora, cuando visitamos `/users/1`, el servidor Rails muestra un cursor de `byebug`:

```
(byebug)
```

Podemos tratarlo como si fuera una consola Rails, emitiendo comandos para entender el estado de la aplicación:

```
(byebug) @user.name
"Example User"
(byebug) @user.email
"example@railstutorial.org"
(byebug) params[:id]
"1"
```

Para liberar el cursor y continuar la ejecución de la aplicación, presione Ctrl-D, luego remueva la línea `debugger` de la acción `show` (Listado 7.7).

Listado 7.7: El controlador `Users` con la línea del depurador eliminada.

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController

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

  def new
  end
end
```

Cada vez que usted esté confundido acerca de algo en una aplicación Rails, es una buena práctica poner **debugger** cerca de la línea de código que usted sospecha que puede estar causando el problema. Inspeccionar el estado del sistema con ayuda de `byebug` es un método poderoso para rastrear los errores de la aplicación y depurar interactivamente su aplicación.

7.1.4 Una imagen gravatar y una barra lateral

Habiendo definido una página de usuario básica en la sección anterior, vamos a profundizar un poco en la imagen de perfil para cada usuario y a dar una primera aproximación de la barra lateral de la interfaz de usuario. Empezaremos agregando un “avatar reconocido globalmente”, o [Gravatar](#), al perfil de usuario.⁶ Gravatar es un servicio gratuito que permite a los usuarios subir imágenes y asociarlas con una dirección de correo que ellos controlan. Como resultado, los gravatares son una forma conveniente de incluir imágenes de perfil de usuario sin realizar el esfuerzo de administrar la carga de imágenes, su manipulación y almacenamiento; todo lo que necesitamos hacer es construir la URL de la imagen Gravatar correspondiente a la dirección de correo del usuario y la imagen automáticamente aparecerá. (Aprenderemos cómo manipular la carga de imágenes personalizadas en la [Sección 11.4](#).)

Nuestro plan es definir una función auxiliar `gravatar_for` que regrese una imagen Gravatar de un cierto usuario, como se muestra en el [Listado 7.8](#).

⁶En el hinduismo, un avatar es la manifestación de una deidad en forma humana o animal. Por consiguiente, el término *avatar* es comúnmente utilizado para expresar alguna forma de representación personal, especialmente en un ambiente virtual.

Listado 7.8: La vista que muestra el usuario con su nombre y su Gravatar.

app/views/users/show.html.erb

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

Por default, los métodos definidos en cualquier archivo auxiliar están automáticamente disponibles en cualquier vista, pero por conveniencia pondremos el método **gravatar_for** en el archivo para funciones auxiliares asociado con el controlador de usuarios. Como observamos en la [documentación de Gravatar](#), las URLs de Gravatar están basadas en el **hash MD5** de la dirección electrónica del usuario. En Ruby, el algoritmo de digestión MD5 es implementado mediante el método **hexdigest**, el cual forma parte de la biblioteca **Digest**:

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

Puesto que las direcciones de correo no son sensibles a mayúsculas/minúsculas ([Sección 6.2.4](#)) pero los hashes MD5 sí lo son, hemos utilizado el método **downcase** para asegurarnos de que el argumento de **hexdigest** está en minúsculas. (Debido a la llamada de regreso para pasar a minúsculas la dirección de correo electrónico del [Listado 6.31](#), esto nunca hará una diferencia en este tutorial, pero es recomendado en caso de que **gravatar_for** alguna vez utilice una dirección electrónica proveniente de otras fuentes.) El método auxiliar resultante **gravatar_for** aparece en el [Listado 7.9](#).

Listado 7.9: Definiendo un método auxiliar **gravatar_for**.

app/helpers/users_helper.rb

```
module UsersHelper

  # Returns the Gravatar for the given user.
  def gravatar_for(user)
```

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

El código del Listado 7.9 regresa una etiqueta HTML correspondiente a una imagen (de Gravatar) con una clase CSS `gravatar` y un atributo `alt` cuyo valor es el nombre de usuario (lo cual es especialmente conveniente para usuarios con discapacidad visual que utilizan un lector de pantalla).

La página del perfil que aparece como en la Figura 7.7, muestra la imagen Gravatar por default; esto sucede porque `user@example.com` no es una dirección electrónica real. (De hecho, como puede ver al visitar el sitio `example.com`, el dominio está reservado para ejemplos como éste.)

Para hacer que nuestra aplicación muestre un Gravatar personalizado, utilizaremos `update_attributes` (Sección 6.1.5) para cambiar la dirección de correo del usuario por algo que yo controlo:

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

Aquí hemos asignado al usuario la dirección electrónica `example@rails-tutorial.org`, a la que le he asociado el logo del Tutorial de Rails, como se muestra en la Figura 7.8.

El último elemento necesario para completar el bosquejo de la Figura 7.1 es la versión inicial de la barra lateral de la interfaz de usuario. La implementaremos usando la etiqueta `aside`, la cual es utilizada para contenidos (tales como barras laterales) que complementan el resto de la página pero que pueden ser autónomas. Incluimos las clases `row` y `col-md-4`, las cuales forman parte de Bootstrap. El código para la página que muestra el usuario modificada aparece en el Listado 7.10.

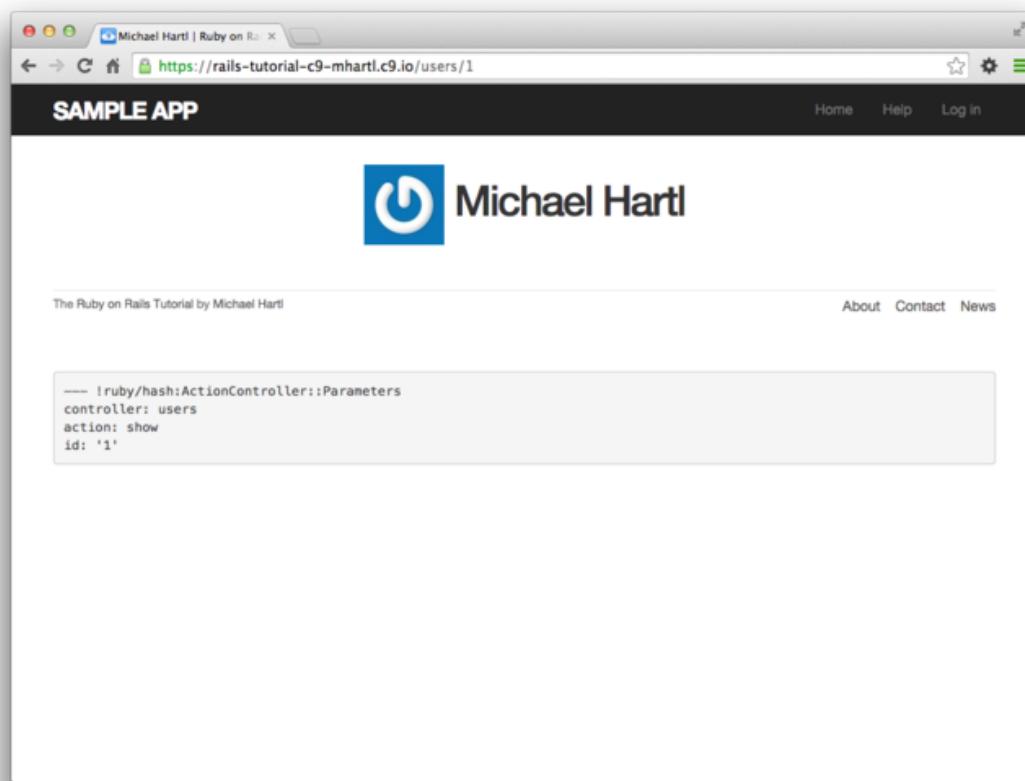


Figura 7.7: La página del perfil de usuario con el Gravatar por default.

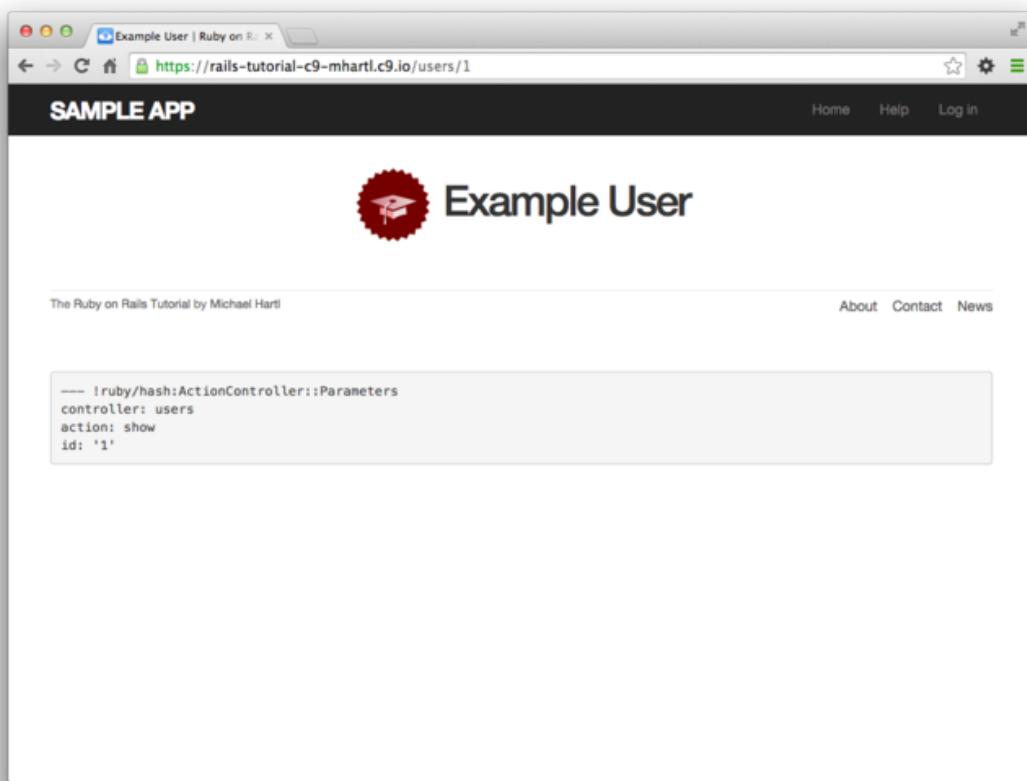


Figura 7.8: La página que muestra al usuario con un Gravatar personalizado.

Listado 7.10: Agregando una barra lateral a la vista de usuario `show`.

`app/views/users/show.html.erb`

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


<aside class="col-md-4">
    <section class="user_info">
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
  </aside>
</div>


```

Con los elementos HTML y las clases CSS en su lugar, podemos estilizar la página del perfil (incluyendo la barra lateral y el Gravatar) con el SCSS que se muestra en el [Listado 7.11](#).⁷ (Observe el anidado de las reglas CSS para la tabla, que funciona únicamente porque el motor de Sass es considerado dentro de la cadena de procesos conectados.) La página resultante se muestra en la Figura 7.9.

Listado 7.11: SCSS para estilizar la página que muestra el usuario, incluyendo la barra lateral.

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

```
.
.
.
/*
 * sidebar */
aside {
  section.user_info {
    margin-top: 20px;
  }
  section {
    padding: 10px 0;
    margin-top: 20px;
    &:first-child {
      border: 0;
      padding-top: 0;
```

⁷El [Listado 7.11](#) incluye la clase `.gravatar_edit`, la cual utilizaremos en el Capítulo 9.

```
        }
      span {
        display: block;
        margin-bottom: 3px;
        line-height: 1;
      }
    h1 {
      font-size: 1.4em;
      text-align: left;
      letter-spacing: -1px;
      margin-bottom: 3px;
      margin-top: 0px;
    }
  }
}

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

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

7.2 El formulario de registro

Ahora que tenemos una página del perfil de usuario funcionando (aunque aún no esté completa), estamos listos para crear un formulario de registro para nuestro sitio. Vimos en la [Figura 5.9](#) (y mostramos nuevamente en la [Figura 7.10](#)) que la página de registro está en este momento en blanco: inútil para registrar usuarios nuevos. El objetivo de esta sección es empezar a cambiar este triste estado creando el formulario de registro bosquejado en la [Figura 7.11](#).

Puesto que estamos a punto de agregar la funcionalidad para crear usuarios a través de la web, eliminemos al usuario creado desde la consola en la [Sección 6.3.4](#). La forma más limpia de hacer esto es reiniciando la base de datos con la tarea Rake `db:migrate:reset`:

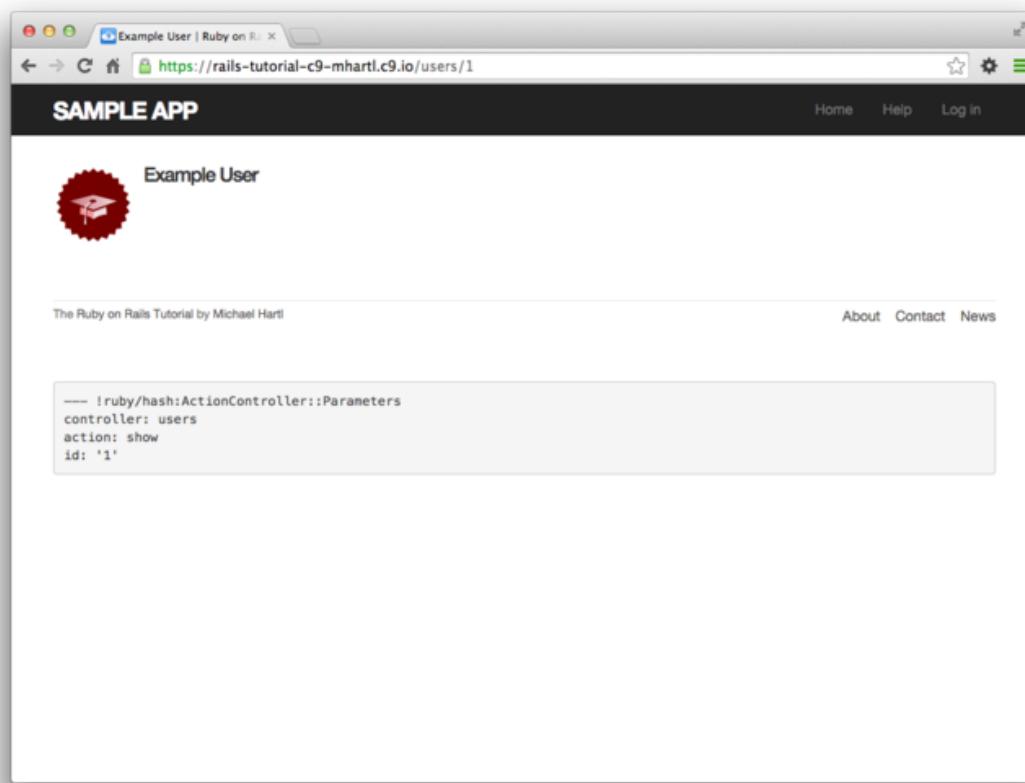


Figura 7.9: La página que muestra el usuario con una barra lateral y CSS.

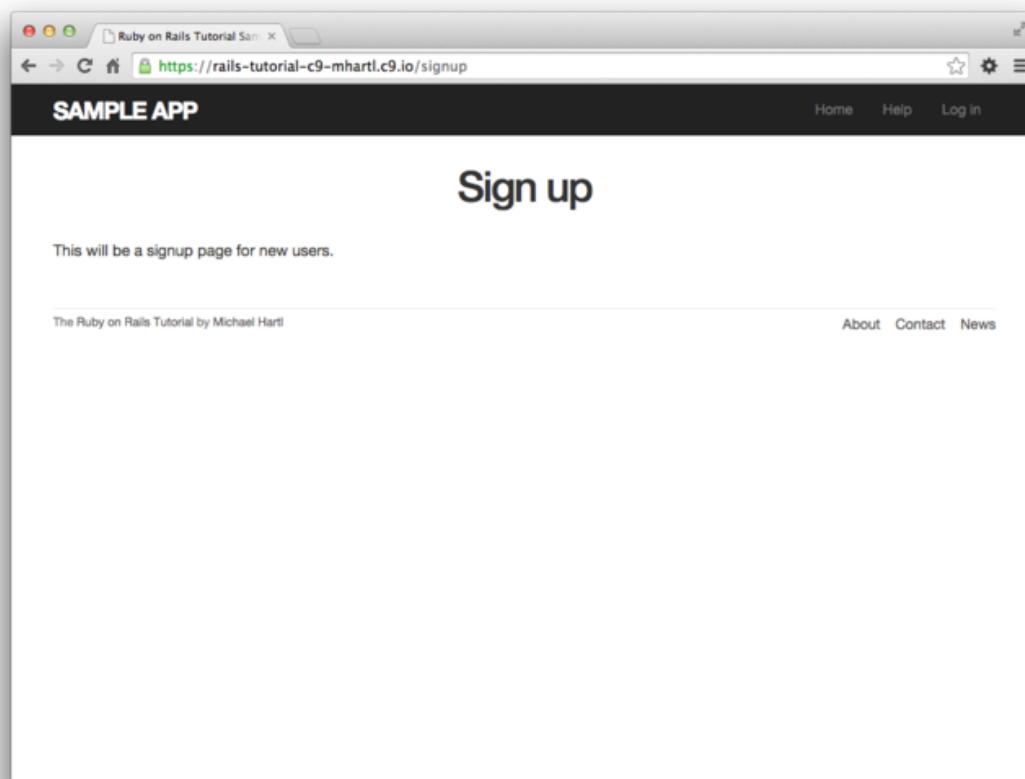


Figura 7.10: El estado actual de la página de registro [/signup](#).



Figura 7.11: Un bosquejo de la página de registro de usuario.

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

En algunos sistemas es probable que sea necesario reiniciar el servidor web (usando Ctrl-C) para que los cambios surtan efecto.

7.2.1 Usando `form_for`

El corazón de la página de registro es un *formulario* para proporcionar la información de registro relevante (nombre, dirección de correo electrónico, contraseña y confirmación). Podemos lograr esto en Rails con el método auxiliar `form_for`, el cual toma un objeto *Active Record* y construye un formulario utilizando los atributos del objeto.

Recordemos que la página de registro /signup es dirigida a la acción `new` del controlador de usuarios ([Listado 5.33](#)), nuestro primer paso es crear el objeto `User` requerido como argumento para `form_for`. La definición de la variable resultante `@user` aparece en el [Listado 7.12](#).

Listado 7.12: Agregando una variable `@user` a la acción `new`.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

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

El formulario mismo aparece en el [Listado 7.13](#). Discutiremos esto con mayor detalle en la [Sección 7.2.2](#), pero primero agreguemos un poco de estilo con el SCSS del [Listado 7.14](#). (Observe el reuso del *mixin* `box_sizing` del [Listado 7.2](#).) Una vez que estas reglas CSS han sido aplicadas, la página de registro luce como en la [Figura 7.12](#).

Listado 7.13: Un formulario para registrar nuevos usuarios.

app/views/users/new.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= f.label :name %>
      <%= f.text_field :name %>

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

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

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

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

Listado 7.14: CSS para el formulario de registro.

app/assets/stylesheets/custom.css.scss

```
.
.
.

/* forms */

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

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

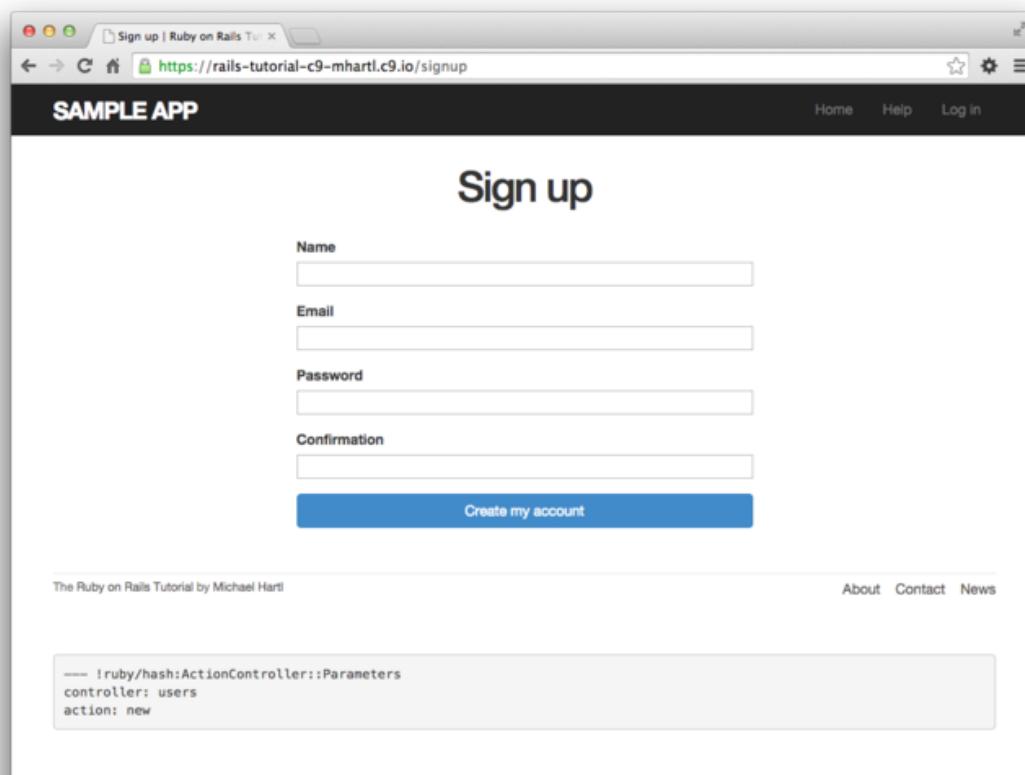


Figura 7.12: El formulario de registro de usuario.

7.2.2 El formulario HTML de registro

Para comprender el formulario definido en el [Listado 7.13](#), es útil dividirlo en partes. Primero echaremos un vistazo a la estructura externa, que consiste de Ruby embebido, que comienza con un llamado a `form_for` y termina con `end`:

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

La presencia de la palabra reservada `do` indica que `form_for` toma un bloque con una variable, la cual hemos llamado `f` (de “formulario”).

Como es usual con los métodos auxiliares Rails, no necesitamos conocer ningún detalle de la implementación, pero lo que *sí* necesitamos saber es lo que el objeto `f` hace: cuando lo invocamos junto con un método correspondiente a un [elemento del formulario HTML](#)—tal como un campo de texto, un grupo de opciones o un campo de contraseña—`f` regresa el código HTML de ese elemento específicamente diseñado para asociar un atributo del objeto `@user`. En otras palabras,

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

crea el HTML necesario para crear un campo de texto etiquetado apropiadamente para establecer el valor del atributo `nombre` a un modelo usuario.

Si usted echa un vistazo al HTML del formulario generado, presionando Ctrl-click y seleccionando la función “inspect element” de su navegador, el código fuente de la página debería verse como el del [Listado 7.15](#). Dediquemos un momento para discutir su estructura.

Listado 7.15: El HTML del formulario que aparece en la [Figura 7.12](#).

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

```

<input name="utf8" type="hidden" value="" />
<input name="authenticity_token" type="hidden"
       value="NNb6+J/j46LcrgYUC60wQ2titMuJQ51LqyAbnbAUkdo=" />
<label for="user_name">Name</label>
<input id="user_name" name="user[name]" type="text" />

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

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

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

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

```

Empezaremos con la estructura interna del documento. Comparando el [Listado 7.13](#) con el [Listado 7.15](#), vemos que el Ruby embebido

```

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

```

produce el HTML

```

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

```

mientras que

```

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

```

produce el HTML

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

y

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

produce el HTML

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

Como se muestra en la Figura 7.13, los campos de texto y correo electrónico (`type="text"` y `type="email"`) simplemente muestran sus contenidos, mientras que los campos de contraseña (`type="password"`) ocultan la entrada por seguridad, como se muestra en la Figura 7.13. (El beneficio de utilizar un campo de correo electrónico es que algunos sistemas le dan un trato especial y diferente de un campo de texto; por ejemplo, el código `type="email"` causará que algunos dispositivos móviles muestren un teclado especialmente optimizado para ingresar direcciones de correo electrónicas.)

Como veremos en la Sección 7.4, la clave para crear un usuario es el atributo especial `name` en cada etiqueta `input`:

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

Estos valores `name` le permiten a Rails construir un arreglo hash de inicialización (mediante la variable `params`) para crear usuarios utilizando los valores proporcionados por el usuario, como veremos en la Sección 7.3.

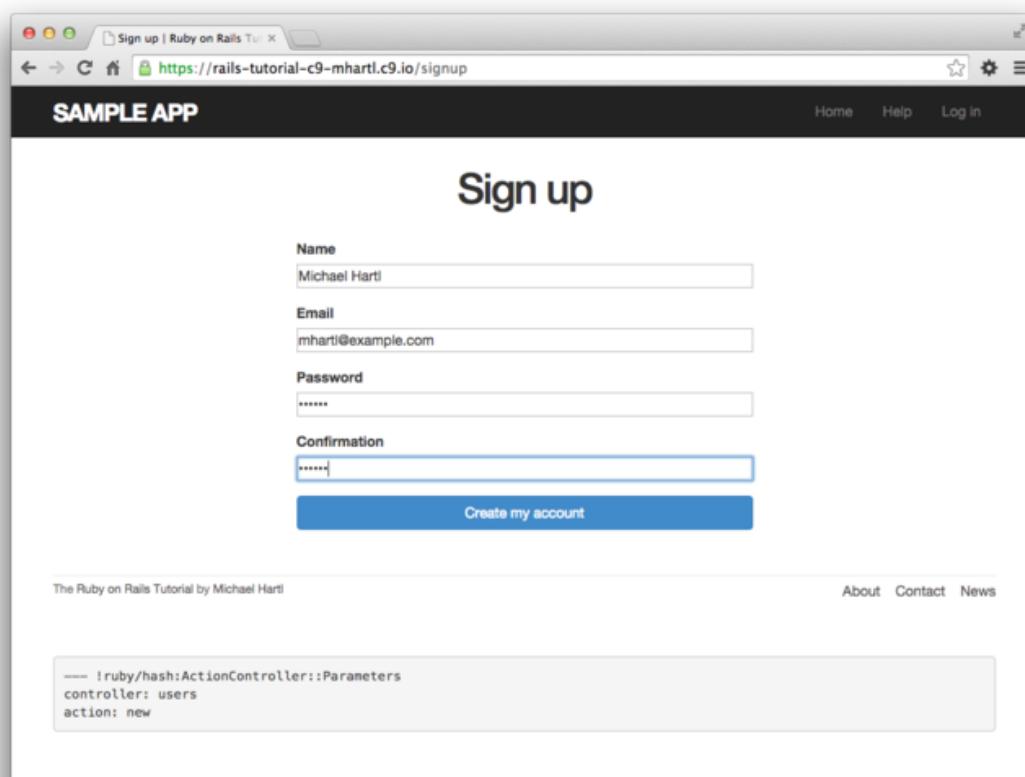


Figura 7.13: Un formulario con datos en los campos de tipo **text** y **password**.

El segundo elemento importante es la etiqueta `form`. Como todo objeto Ruby conoce su propia clase (Sección 4.4.1), Rails descubre que `@user` es de la clase `User`; y dado que `@user` es un *nuevo* usuario, Rails sabe cómo construir un formulario con el método `post`, que es el verbo apropiado para crear un nuevo objeto (Recuadro 3.2):

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

Aquí los atributos `class` y `id` son sumamente irrelevantes; lo que es importante aquí es `action= "/users"` y `method= "post"`. Juntos, constituyen las instrucciones para emitir una petición POST HTTP a la URL /users. Veremos en las próximas dos secciones el efecto que esto tiene.

(Puede que usted también haya observado el código que aparece justo dentro de la etiqueta `form`:

```
<div style="display:none">
  <input name="utf8" type="hidden" value="" />
  <input name="authenticity_token" type="hidden"
    value="NNb6+J/j46LcrgYUC60wQ2titMuJQ51LqyAbnbAUkdo=" />
</div>
```

Este código, que no es desplegado en el navegador, es utilizado internamente por Rails, por lo que no es importante para nosotros entender lo que hace. Brevemente, comentaremos que utiliza el carácter unicode `` (una marca de verificación ?) que obliga a los navegadores a enviar los datos utilizando la codificación de caracteres correcta, y luego la incluye en un *token de autenticidad*, que Rails utiliza para evitar un ataque denominado *falsificación de peticiones cruzadas* (CSRF, por sus siglas en inglés *Cross-Site Request Forgery*).⁸

⁸Vea la entrada en Stack Overflow acerca del token de autenticidad de Rails si es que usted está interesado en los detalles de cómo funciona esto.

7.3 Registros fallidos

Aunque hemos examinado brevemente el HTML del formulario que aparece en la [Figura 7.12](#) (que se muestra en el [Listado 7.15](#)), aún no hemos cubierto ningún detalle, y el formulario es mejor comprendido en el contexto de un *registro fallido*. En esta sección, crearemos un formulario de registro que acepte el envío de los datos y vuelva a desplegar la misma página con una lista de errores, como se esboza en la [Figura 7.14](#).

7.3.1 Un formulario funcional

Recuerde de la [Sección 7.1.2](#) que al agregar `resources :users` al archivo `routes.rb` ([Listado 7.3](#)) automáticamente asegura que nuestra aplicación Rails responde a las URLs RESTful de la [Tabla 7.1](#). En particular, asegura que una petición POST a `/users` es manejada por la acción `create`. Nuestra estrategia para la acción `create` es utilizar el envío del formulario para crear un nuevo objeto usuario mediante `User.new`, intente (y falle) el guardado del usuario, y luego despliegue la página de registro para permitir un re-envío de los datos. Empecemos revisando el código del formulario de registro:

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

Como observamos en la [Sección 7.2.2](#), este HTML emite una petición POST a la URL `/users`.

Nuestro primer paso hacia un formulario de registro funcional es agregar el código del [Listado 7.16](#). Este listado incluye un segundo uso del método `render`, que vimos por primera vez en el contexto de los parciales ([Sección 5.1.3](#)); como puede observar, `render` funciona en acciones del controlador también. Observe que hemos aprovechado esta oportunidad para presentar una estructura `if-else`, la cual nos permite manejar los casos de error y éxito de forma separada, con base en el valor de `@user.save`, que (como vimos en la [Sección 6.1.3](#)) es `verdadero` o `falso` dependiendo de si el guardado de los datos es exitoso o no.

The sketch depicts a user interface for a sign-up form. At the top center is the title "Sign up". Below it is a bulleted list of validation errors: "Name can't be blank", "Email is invalid", and "Password is too short". The form fields are labeled "Name", "Email", "Password", and "Confirmation", each followed by a horizontal input box. At the bottom is a button labeled "Create my account". The entire form is enclosed in a rectangular frame with rounded corners.

Sign up

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

Name

Email

Password

Confirmation

Create my account

Figura 7.14: Un esbozo de la página de registro fallido.

Listado 7.16: Una acción `create` que puede manejar los fallos en el registro.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

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

Observe el comentario: esta no es la implementación final. Pero es suficiente para que empecemos, y terminaremos la implementación en la Sección 7.3.2.

La mejor forma para comprender cómo funciona el código del Listado 7.16 es *enviar* el formulario con algunos datos de registro inválidos. El resultado aparece en la Figura 7.15, y la información de depuración completa (con un tamaño de letra mayor) aparece en la Figura 7.16. (La Figura 7.15 también muestra la *consola web*, la cual abre una consola Rails en el navegador para ayudar a la depuración. Es útil para examinar, por ejemplo, el modelo usuario, pero en este caso necesitamos inspeccionar `params`, la cual, hasta donde sé, no está disponible en la consola web.)

Para tener un mejor panorama de cómo maneja Rails el envío de datos, echemos un vistazo más de cerca al `user` que viene con los parámetros hash provenientes de la información de depuración (Figura 7.16):

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

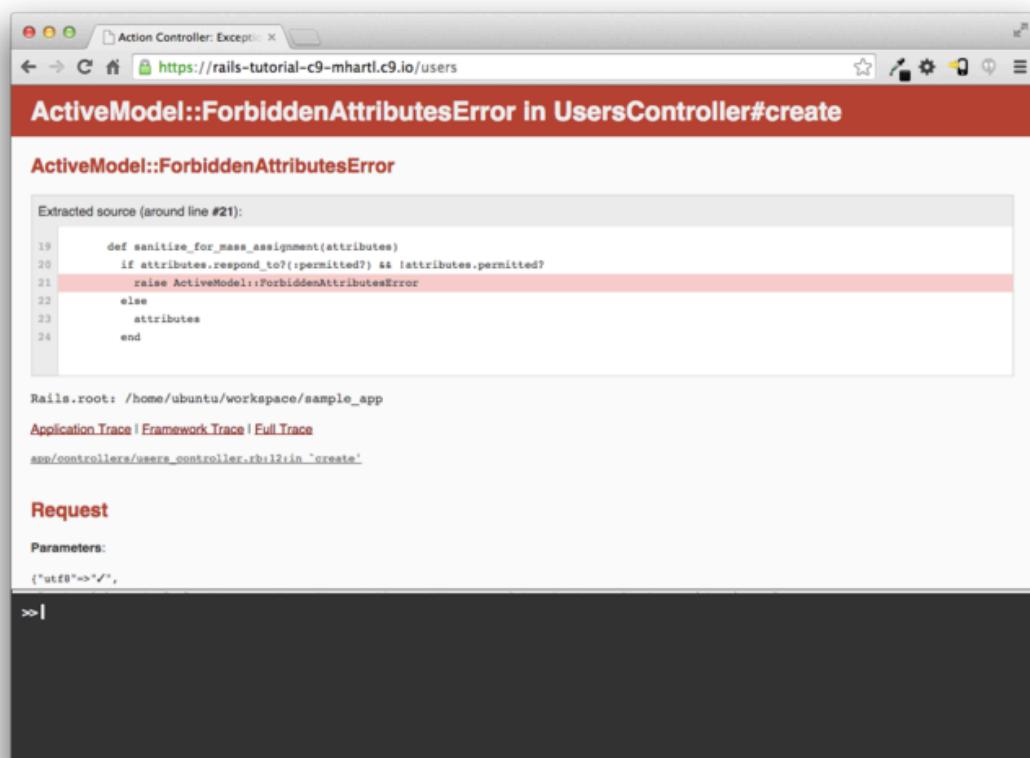


Figura 7.15: Registro fallido.

Request

Parameters:

```
{"utf8"=>"✓",
 "authenticity_token"=>"kicb677m0mxXCH5404ZKddGiYTTAWZnLhzAOP0ysmTM=",
 "user"=>{"name"=>"Foo Bar",
 "email"=>"foo@invalid",
 "password"=>"[FILTERED]",
 "password_confirmation"=>"[FILTERED]"},
 "commit"=>"Create my account"}
```

[Toggle session dump](#)

[Toggle env dump](#)

Figura 7.16: Registro fallido con información de depuración.

```
    "password_confirmation" => "[FILTERED]"
}
```

Esta digestión es pasada al controlador de usuarios como parte de `params`, y como vimos empezando la Sección 7.1.2, el arreglo hash `params` contiene información acerca de cada petición. En el caso de una URL como /users/1, el valor de `params[:id]` es el `id` del usuario correspondiente (1 en este ejemplo). En el caso de enviar datos al formulario de registro, `params` contiene en vez de un arreglo hash, un arreglo hash de hashes, una construcción que vimos por primera vez en la Sección 4.3.3, en la que presentamos la variable estratégicamente llamada `params` en una sesión de consola. La información de depuración anterior muestra que al enviar los datos del formulario obtenemos un arreglo hash `user` cuyos atributos corresponden a los valores enviados, donde las llaves están dadas por los atributos `name` de las etiquetas `input` que vimos en el Listado 7.13; por ejemplo, el valor de

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

con nombre "`user[email]`" es precisamente el atributo `email` del hash `user`.

Aunque las llaves del hash aparecen como cadenas de caracteres en la salida de la depuración, podemos tener acceso a ellas en el controlador de usuarios como símbolos, de modo que `params[:user]` es el hash de los atributos del usuario—de hecho, son los atributos estrictamente necesarios como argumentos para `User.new`, como vimos por primera vez en la Sección 4.4.5 y que aparece en el Listado 7.16. Esto significa que la línea

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

es prácticamente equivalente a

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

En versiones previas de Rails, emplear

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

realmente funcionaba, pero era inseguro por default y requería un procedimiento cuidadoso y propenso a errores para prevenir que usuarios potencialmente maliciosos modificaran la base de datos de la aplicación. En versiones de Rails posteriores a la 4.0, este código arroja un error (como vimos en las Figura 7.15 y 7.16 anteriores), lo que significa que es seguro por default.

7.3.2 Parámetros fuertes

Mencionamos brevemente en la Sección 4.4.5 la idea de *asignación masiva*, lo cual implica inicializar una variable Ruby usando un arreglo hash de valores, como en

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

El comentario incluído en el Listado 7.16 y reproducido anteriormente indica que esta no es la implementación final. La razón es que inicializar el arreglo hash **params** completo es *extremadamente* peligroso—se encarga de pasar a **User.new** *todos* los datos enviados por un usuario. En particular, suponga que, adicionalmente a los atributos actuales, el modelo de usuario incluye un atributo **admin** para identificar a los usuarios administrativos del sitio. (Implementaremos tal atributo en la Sección 9.4.1.) La manera de establecer un atributo de este tipo como **verdadero** es pasar el valor **admin='1'** como parte de **params[:user]**, una tarea que es fácil de realizar utilizando un cliente HTTP de línea de comandos tal como **curl**. El resultado sería que, al pasar el arreglo hash **params** completo a **User.new**, permitiríamos a cualquier usuario del sitio obtener privilegios de administrador tan sólo por incluir **admin='1'** en la petición web.

En versiones previas de Rails usamos un método llamado **attr_accessible** en la capa del *modelo* para resolver este problema, y aún puede encontrar ese

método en aplicaciones Rails legadas, pero a partir de Rails 4.0 la técnica preferida es utilizar los llamados *parámetros fuertes* en la capa del controlador. Esto nos permite especificar qué parámetros son *requeridos* y cuáles son *permitidos*. Adicionalmente, pasar un arreglo hash **params** como se mostró anteriormente arrojará un error, por lo que las aplicaciones Rails ahora son inmunes a las vulnerabilidades de la asignación masiva por default.

En la instancia actual, queremos que el arreglo hash **params** sea requerido, para tener un atributo **:user**, y queremos permitir los atributos nombre, correo electrónico, contraseña y confirmación de la contraseña (pero no otros). Podemos lograr esto como sigue:

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

Este código regresa una versión del arreglo hash **params** con los atributos permitidos únicamente (y arroja un error si el atributo **:user** no está presente).

Para facilitar el uso de esos parámetros, se acostumbra utilizar un método auxiliar llamado **user_params** (el cual regresa un arreglo hash de inicialización apropiado) y lo empleamos en vez de **params[:user]**:

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

Puesto que **user_params** sólo será utilizado de forma interna por el controlador de usuarios y no es necesario exponerlo a usuarios externos a través de la web, lo haremos *privado* utilizando la palabra reservada de Ruby **private**, como se muestra en el [Listado 7.17](#). (Discutiremos **private** con mayor detalle en la [Sección 8.4](#).)

Listado 7.17: Usando parámetros fueretes en la acción **create**.

```
app/controllers/users_controller.rb
```

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

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

private

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

Por cierto, el nivel extra de indentación en el método `user_params` es diseñado para hacer visualmente aparente qué métodos han sido definidos luego de `private`. (La experiencia muestra que esto es una práctica inteligente; en clases con una gran cantidad de métodos, es fácil definir un método privado accidentalmente, lo que provoca gran confusión cuando no está disponible para utilizarlo en el objeto correspondiente.)

En este momento, el formulario de registro está funcionando, al menos en el sentido de que no produce error al enviar los datos a guardar. Por otra parte, como vemos en la Figura 7.17, no despliega ninguna retroalimentación en caso de enviar datos inválidos (aparte del área de depuración exclusiva de desarrollo), lo cual es potencialmente confuso. De hecho, tampoco crea un nuevo usuario. Arreglaremos el primer problema en la Sección 7.3.3 y el segundo en la Sección 7.4.

7.3.3 Mensajes de error de registro

Como paso final en el manejo de errores al crear usuarios, agregaremos mensajes de error que nos ayuden a identificar los problemas que evitaron el registro exitoso. Convenientemente, Rails proporciona automáticamente tales mensajes basados en las validaciones del modelo `User`. Por ejemplo, intente guardar un usuario con una dirección electrónica inválida y con una contraseña demasiado corta:

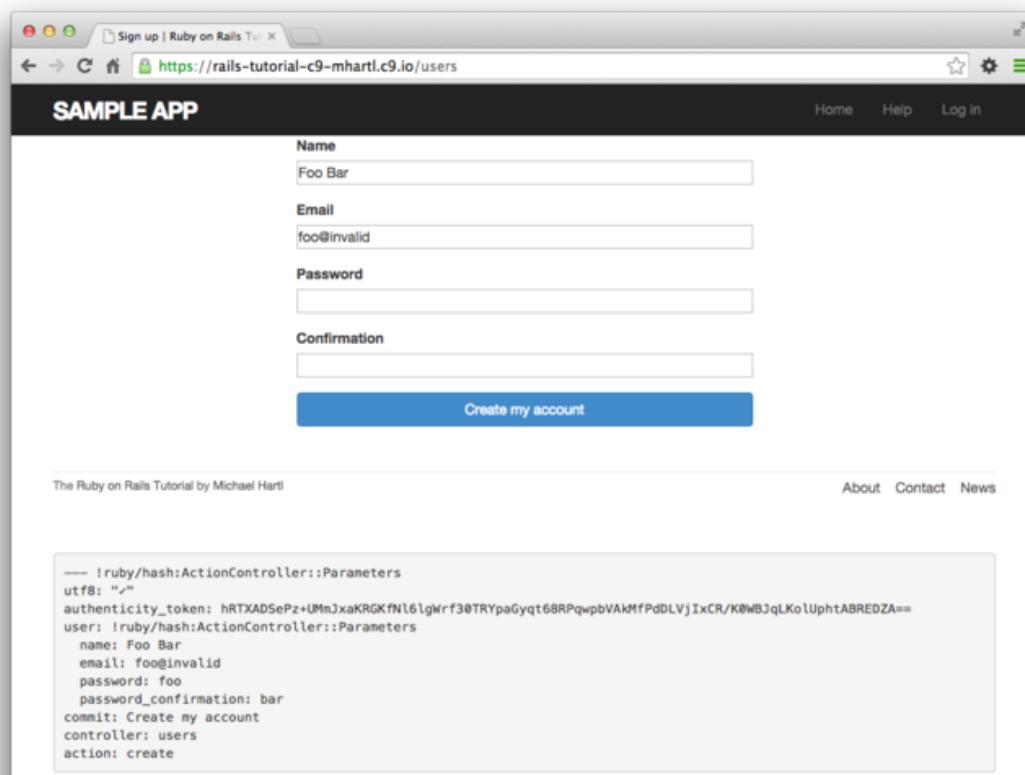


Figura 7.17: El formulario de registro con datos inválidos enviados a guardar.

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

Aquí el objeto `errors.full_messages` (el cual revisamos brevemente en la Sección 6.2.2) contiene un arreglo con mensajes de error.

Como en la sesión de consola anterior, el guardado fallido del Listado 7.16 genera una lista de mensajes de error asociados con el objeto `@user`. Para desplegar los mensajes en el navegador, mostraremos un parcial con los mensajes de error en la página de usuario `new`, mientras que asociamos las clases CSS `form-control` (las cuales tienen significado especial para Bootstrap) a cada campo de entrada, como se muestra en el Listado 7.18. Vale la pena observar que este parcial con mensajes de error es sólo una primera versión; la versión final aparece en la Sección 11.3.2.

Listado 7.18: Código para desplegar mensajes de error en el formulario de registro.

`app/views/users/new.html.erb`

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

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

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

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

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

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

Observe que aquí **desplegamos** un parcial llamado '**shared/error_messages**'; esto refleja la convención común a Rails de utilizar el directorio **shared/**, dedicado a los parciales que serán utilizados en vistas compartidas entre múltiples controles. (Veremos esta expectativa cumplida en la Sección 9.1.1.) Esto significa que tenemos que crear un nuevo directorio **app/views/shared** usando el comando **mkdir** (Tabla 1.1):

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

Luego necesitamos crear el archivo **_error_messages.html.erb** para el parcial usando nuestro editor de texto como es usual. El contenido del parcial aparece en el Listado 7.19.

Listado 7.19: Un parcial para desplegar los mensajes de error del formulario enviado.

app/views/shared/_error_messages.html.erb

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

Este parcial presenta varias construcciones nuevas de Rails y de Ruby, incluyendo dos métodos para los objetos de error de Rails. El primer método es **count**, que simplemente regresa el número de errores:

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

El otro método nuevo es `any?`, el cual (junto con `empty?`) forma una pareja de métodos complementarios:

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

Aquí vemos que el método `empty?` que vimos por primera vez en la Sección 4.2.3 en el contexto de cadenas de caracteres, también funciona con objetos de error de Rails, regresando `verdadero` para un objeto vacío y `falso` en caso contrario. El método `any?` es justo el opuesto de `empty?`, regresando `verdadero` si hay algún elemento presente y `falso` en caso contrario. (Por cierto, todos estos métodos—`count`, `empty?` y `any?`—funcionan sobre arreglos Ruby también. Aprovecharemos esto a partir de la Sección 11.2.)

La otra idea nueva es el método auxiliar `pluralize` para textos. Éste no está disponible en la consola de forma automática por default, pero podemos incluirlo explícitamente a través del módulo `ActionView::Helpers::TextHelper`:⁹

```
>> include ActionView::Helpers::TextHelper  
>> pluralize(1, "error")  
=> "1 error"  
>> pluralize(5, "error")  
=> "5 errors"
```

Aquí observamos que `pluralize` toma un argumento entero y luego regresa el número con una versión pluralizada adecuadamente de su segundo argumento. Debajo de este método existe un *inflector* poderoso que sabe cómo pluralizar una gran cantidad de palabras, incluyendo muchas con plurales irregulares:

⁹Me dí cuenta de esto buscando `pluralize` en el [API de Rails](#).

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

Como resultado de utilizar **pluralize**, el código

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

regresa "**0 errors**", "**1 error**", "**2 errors**", y así sucesivamente, dependiendo de cuántos errores haya, de este modo evitando frases gramaticalmente incorrectas como "**1 errors**" (un error penosamente común en las aplicaciones y en la Web).

Observe que el [Listado 7.19](#) incluye el id **error_explanation** que será utilizado en el CSS para asociar el estilo de los mensajes de error. (Recuerde de la [Sección 5.1.2](#) que CSS utiliza el signo de número `#` para seleccionar **ids**.) Adicionalmente, luego de un envío inválido de datos, Rails automáticamente envuelve los campos con error dentro de elementos **div** que contienen la clase CSS **field_with_errors**. Estas etiquetas nos permiten agregar estilo a los mensajes de error con el SCSS mostrado en el [Listado 7.20](#), el cual hace uso de la función de Sass **@extend** para incluir la funcionalidad de la clase Bootstrap **has-error**.

Listado 7.20: CSS para estilizar los mensajes de error.

app/assets/stylesheets/custom.css.scss

```
.
.
.

/* forms */

.
.
.

#error_explanation {
  color: red;
  ul {
    color: red;
    margin: 0 0 30px 0;
```

```
}

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

Con el código de los Listados 7.18 y 7.19 y el SCSS del Listado 7.20, aparecen mensajes de error útiles cuando se envían datos de registro inválidos al servidor, como se muestra en la Figura 7.18. Como los mensajes son generados por las validaciones del modelo, serán modificados automáticamente si en algún momento cambia de opinión al respecto, digamos, en el formato de direcciones de correo electrónicas, o la longitud mínima de las contraseñas.

7.3.4 Una prueba para envío de datos inválidos

Antes de que existieran las bibliotecas web poderosas con capacidades de prueba completas, los desarrolladores tenían que probar los formularios manualmente. Por ejemplo, para probar una página de registro, tendríamos que visitar la página en un navegador y luego enviar a guardar tanto datos válidos como inválidos, verificando en cada caso que el comportamiento de la aplicación era correcto. Es más, tendríamos que repetir el proceso cada vez que la aplicación cambiara. Este proceso era doloroso y tenía a errores.

Afortunadamente, con Rails podemos escribir pruebas para automatizar las pruebas de los formularios. En esta sección, escribiremos una de esas pruebas para verificar el correcto comportamiento en caso de enviar datos inválidos; en la Sección 7.4.4, escribiremos la prueba respectiva para datos válidos.

Para empezar, primero generaremos un archivo de prueba de integración para el registro de usuarios, que llamaremos `users_signup` (adoptando la convención del controlador de utilizar el nombre en plural del recurso):

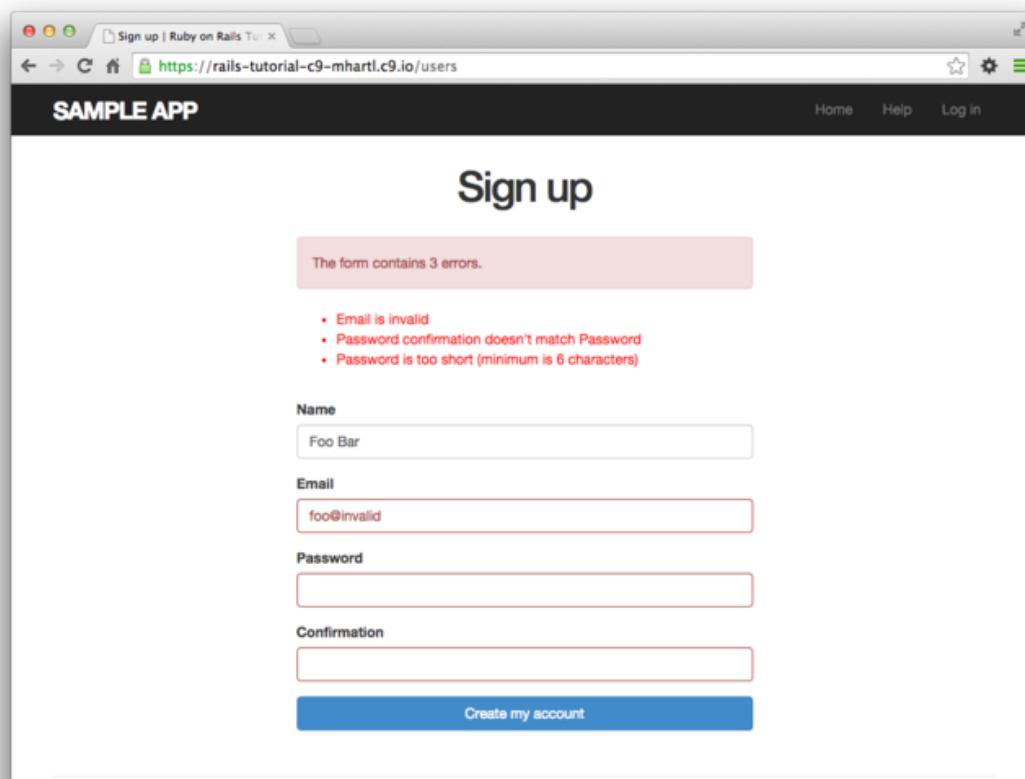


Figura 7.18: Registro fallido con mensajes de error.

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

(Utilizaremos este mismo archivo en la [Sección 7.4.4](#) para probar el caso del registro válido.)

El principal propósito de nuestra prueba es verificar que al dar click en el botón de registro obtendremos como resultado que *no* es creado un nuevo usuario si la información enviada es inválida. (Escribir una prueba para verificar los mensajes de error, se deja como ejercicio ([Sección 7.7](#)).) La forma de hacer esto es verificando la *cantidad* de usuarios, y debajo de nuestras pruebas utilizaremos el método **count** disponible en toda clase *Active Record*, incluyendo **User**:

```
$ rails console
>> User.count
=> 0
```

Aquí **User.count** es **0** porque reiniciamos la base de datos al inicio de la [Sección 7.2](#). De igual forma que en la [Sección 5.3.4](#), utilizaremos **assert_select** para probar los elementos HTML de las páginas relevantes, teniendo cuidado de verificar únicamente elementos que son poco probables que cambien en el futuro.

Empezaremos por visitar la ruta de registro mediante **get**:

```
get signup_path
```

Con la finalidad de probar el envío del formulario, necesitamos emitir una petición POST a la ruta **users_path** ([Tabla 7.1](#)), lo cual podemos hacer mediante la función **post**:

```
assert_no_difference 'User.count' do
  post users_path, user: { name: "",  
                        email: "user@invalid",
```

```

    password:          "foo",
password_confirmation: "bar" }

end

```

Aquí hemos incluído el arreglo hash `params[:user]` esperado por `User.new` en la acción `create` (Listado 7.24). Al invocar el `post` dentro del método `assert_no_difference` con el argumento de tipo cadena '`User.count`', nos preparamos para una comparación entre `User.count` antes y después del contenido del bloque `assert_no_difference`. Esto es equivalente a contar el número de usuarios, enviar los datos a guardar y luego verificar que el conteo sigue igual:

```

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

```

Aunque los dos son equivalentes, utilizar `assert_no_difference` es más limpio y más correcto para Ruby, idiomáticamente hablando.

Vale la pena observar que los pasos `get` y `post` anteriores, técnicamente no están relacionados, y de hecho no es necesario obtener la ruta de registro antes de enviar los datos a guardar. Aún así, yo prefiero incluir ambos pasos, tanto por claridad de conceptos como para realizar una doble verificación de que el formulario de registro se está mostrando sin error.

Juntando las ideas anteriores llegamos a la prueba del Listado 7.21. También hemos incluído una llamada a `assert_template` para verificar que un envío de datos fallido vuelve a desplegar la acción `new`. Agregar líneas para verificar la apariencia de los mensajes de error se deja como ejercicio (Sección 7.7).

Listado 7.21: Una prueba para un registro inválido. VERDE

`test/integration/users_signup_test.rb`

```

require 'test_helper'

```

```
class UsersController < ActionDispatch::IntegrationTest

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

Como escribimos el código de la aplicación antes que la prueba de integración, el conjunto de pruebas debería estar ahora en **VERDE**:

Listado 7.22: **VERDE**

```
$ bundle exec rake test
```

7.4 Registros exitosos

Habiendo tratado el envío de información inválida, es hora de completar el formulario de registro guardando un nuevo usuario (si es válido) en la base de datos. Primero, intentamos guardar al usuario; si esta operación es exitosa, la información del usuario es escrita en la base de datos automáticamente, y luego *redireccionamos* el navegador para que muestre el perfil del usuario (junto con una bienvenida amistosa), como se bosquejó en la Figura 7.19. Si falla, simplemente caemos en el comportamiento desarrollado en la Sección 7.3.

7.4.1 El formulario de registro terminado

Para terminar la funcionalidad de la forma de registro, necesitamos llenar la sección comentada del Listado 7.17 con el comportamiento adecuado. En este momento, la forma falla al enviar datos válidos. Como se muestra en la Figura 7.20, esto es porque el comportamiento por default de una acción Rails

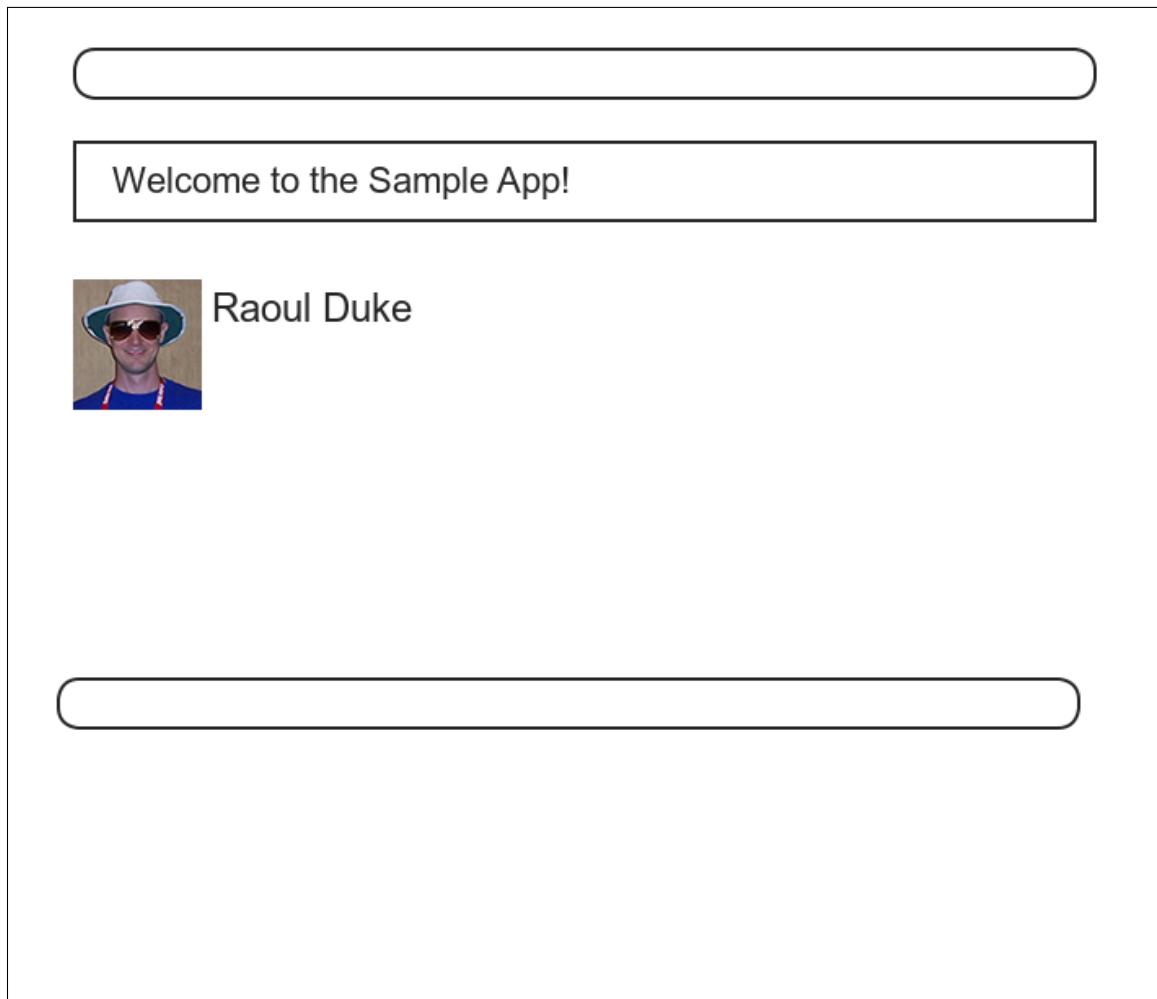


Figura 7.19: Un bosquejo de un registro exitoso.

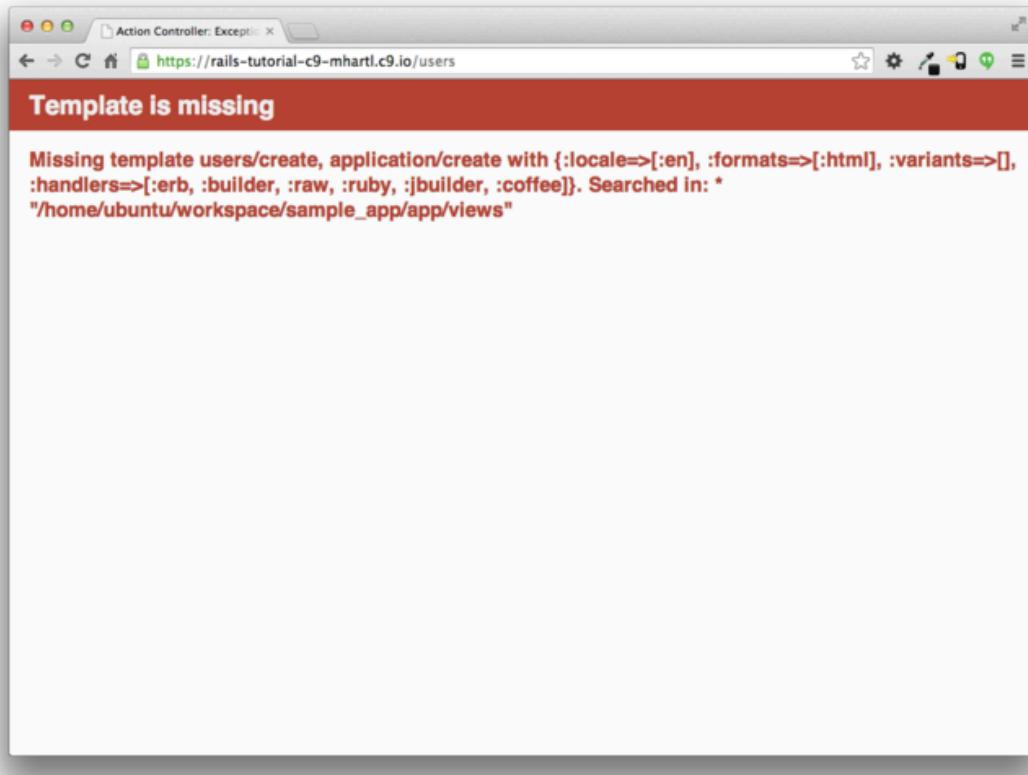


Figura 7.20: La página de error para un envío de registro válido.

en desplegar la vista correspondiente, y no existe una plantilla para la vista correspondiente a la acción `create`.

Al crear exitosamente un usuario, lo *redireccionaremos* a una página distinta. Seguiremos la convención común de redireccionarlo a su página del perfil, aunque la ruta raíz también funcionaría. El código de la aplicación, que introduce el método `redirect_to`, aparece en el [Listado 7.23](#).

Listado 7.23: La acción `create` de usuario con un guardado y una redirección.

```
app/controllers/users_controller.rb  
class UsersController < ApplicationController
```

```
•  
•  
•  
def create  
  @user = User.new(user_params)  
  if @user.save  
    redirect_to @user  
  else  
    render 'new'  
  end  
end  
  
private  
  
def user_params  
  params.require(:user).permit(:name, :email, :password,  
                                :password_confirmation)  
end  
end
```

Observe que hemos escrito

```
redirect_to @user
```

donde pudimos haber usado el equivalente

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

Esto es porque Rails automáticamente infiere de **redirect_to @user** que queremos redireccionarnos a **user_url(@user)**.

7.4.2 El flash

Con el código del [Listado 7.23](#), nuestro formulario de registro está funcionando, pero antes de enviar datos para un registro válido en el navegador, vamos a pulir un poco las partes comunes de la aplicación web: el mensaje que aparece en la página siguiente (en este caso, dando la bienvenida a nuestro nuevo usuario a la aplicación) y luego haciendo que desaparezca a partir de que visite la segunda página o que recargue la página.

El modo Rails para desplegar un mensaje temporal es utilizar un método especial llamado el **flash**, el cual podemos tratar como un arreglo hash. Rails adopta la convención de utilizar una llave **:success** para un mensaje que indica un resultado exitoso ([Listado 7.24](#)).

Listado 7.24: Agregando un mensaje **flash** al registro de usuario.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .

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

  private

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

Al asignar un mensaje a **flash**, estamos en posición de desplegar el mensaje en la primera página luego de la redirección. Nuestro método es iterar sobre el **flash** e insertar todos los mensajes relevantes en la estructura de diseño del sitio. Puede recordar el ejemplo de consola de la [Sección 4.3.3](#), donde vimos cómo iterar a través de un arreglo hash usando la variable estratégicamente llamada **flash**:

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

```
>> end
success
It worked!
danger
It failed.
```

Al seguir este patrón, podemos disponer que se despliegue el contenido del **flash** en todo el sitio utilizando código como éste:

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

(Este código es una combinación particularmente fea de HTML y ERb; hacerlo más elegante se deja como ejercicio (Sección 7.7).) Aquí el Ruby embebido

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

crea una clase CSS correspondiente al tipo de mensaje, de forma que para un mensaje :**success** la clase es

```
alert-success
```

(La llave :**success** es un símbolo, pero Ruby embebido, automáticamente lo convierte en una cadena de caracteres "**success**" antes de insertarlo en la plantilla.) Usar una clase diferente para cada llave nos permite aplicar diferentes estilos a diferentes tipos de mensajes. Por ejemplo, en la Sección 8.1.4 usaremos **flash[:danger]** para indicar un intento fallido por ingresar al sitio.¹⁰ (De hecho, hemos utilizado **alert-danger** en alguna ocasión, para estilizar **div** del mensaje de error del Listado 7.19.) El CSS de Bootstrap provee estilos para cuatro clases **flash** (**success**, **info**, **warning**, y **danger**), y encontraremos

¹⁰En realidad, utilizaremos el muy relacionado **flash.now**, pero aplazaremos el uso de esta sutileza hasta que la necesitemos.

ocasión de usarlas todas en el transcurso del desarrollo de la aplicación de ejemplo.

Como el mensaje también es insertado en la plantilla, el HTML producido por

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

se muestra como sigue:

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

Al colocar el Ruby embebido revisado anteriormente dentro de la estructura de diseño del sitio obtendremos el código del [Listado 7.25](#).

Listado 7.25: Agregando los contenidos de la variable `flash` a la estructura de diseño del sitio.

app/views/layouts/application.html.erb

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

7.4.3 El primer registro

Podemos ver el resultado de todo este trabajo registrando a nuestro primer usuario bajo el nombre de “Rails Tutorial” y con la dirección electrónica “example@railstutorial.org” (Figura 7.21). La página resultante (Figura 7.22) muestra un mensaje amigable en caso de un registro exitoso, incluyendo un agradable estilo verde para la clase **success**, que viene incluido con la biblioteca CSS de Bootstrap CSS de la Sección 5.1.2. (Si usted obtiene un mensaje de error indicando que la dirección electrónica ya ha sido registrada, asegúrese de ejecutar la tarea Rake **db:migrate:reset** como se indica en la Sección 7.2 y reinicie el servidor web de desarrollo.) Luego, al recargar la página que muestra al usuario, el mensaje **flash** desaparece como prometimos (Figura 7.23).

Ahora podemos revisar nuestra base de datos sólo para verificar que el nuevo usuario fue realmente creado:

```
$ rails console
>> User.find_by(email: "example@railstutorial.org")
=> #<User id: 1, name: "Rails Tutorial", email: "example@railstutorial.org",
  created_at: "2014-08-29 19:53:17", updated_at: "2014-08-29 19:53:17",
  password_digest: "$2a$10$zthScEx9x6EkuLa4NolGye600Zgrkp1B6LQ12pTHlNB...">
```

7.4.4 Una prueba para el envío de datos válidos

Antes de continuar, escribiremos una prueba para el envío de datos válidos de modo que verifiquemos el comportamiento de nuestra aplicación y atrapemos regresiones. De igual forma que con la prueba para envío de datos inválidos en la Sección 7.3.4, nuestro principal propósito es verificar el contenido de la base de datos. En este caso, queremos enviar información válida para su registro y luego confirmar que un usuario *ha sido* creado. Análogamente al Listado 7.21, que utilizó

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

aquí utilizaremos el método correspondiente **assert_difference**:

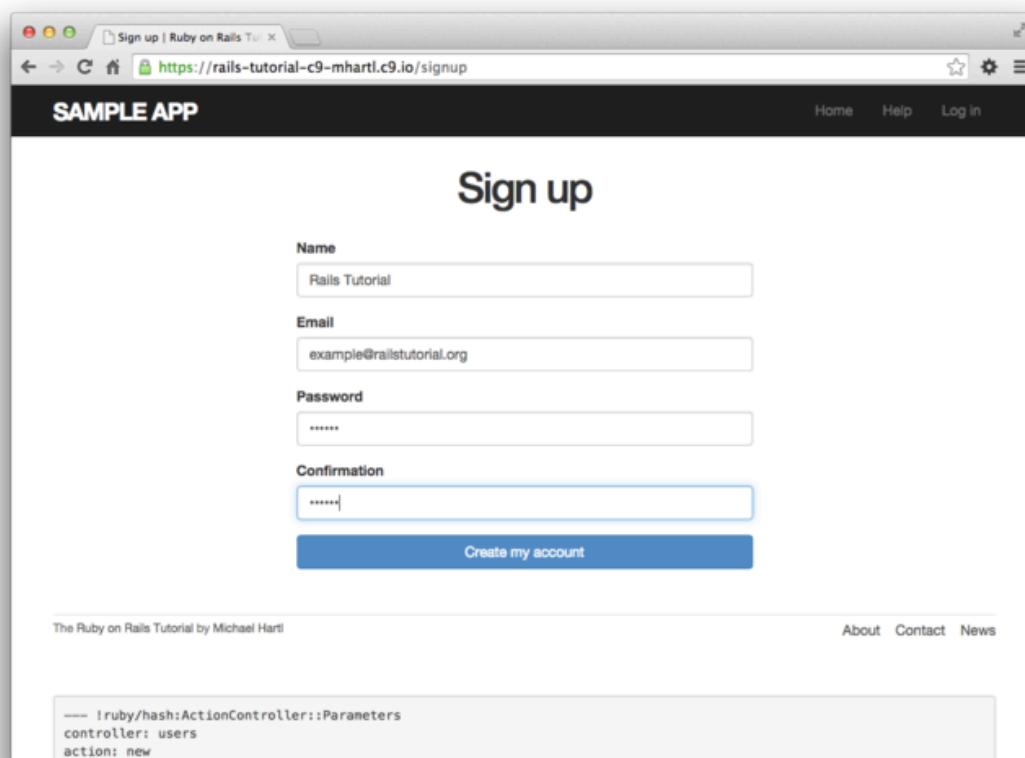


Figura 7.21: Llenando el formulario para el primer registro.

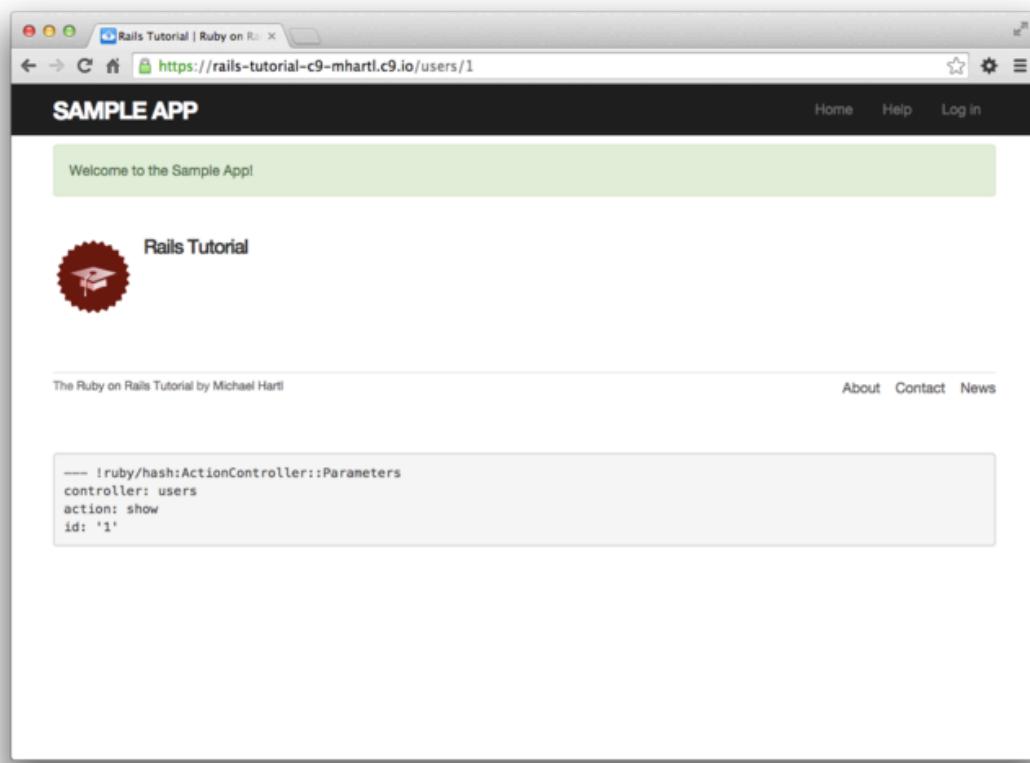


Figura 7.22: Los resultados de un registro de usuario exitoso, con un mensaje **flash**.

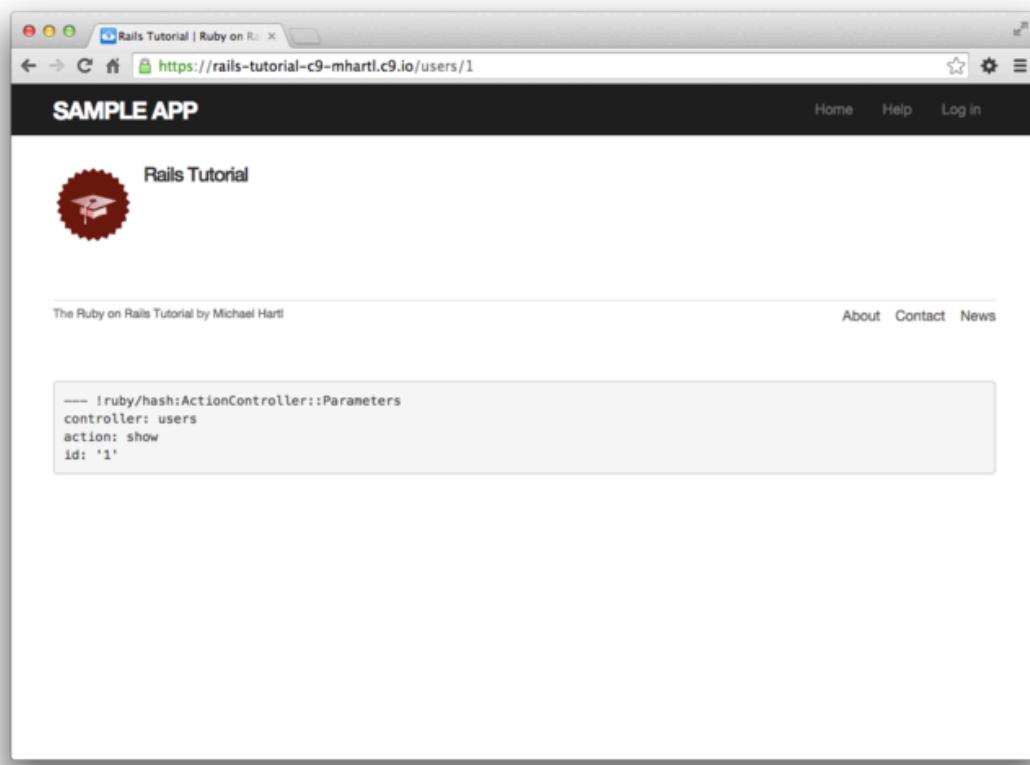


Figura 7.23: La página del perfil sin **flash** luego de que el navegador ha re-cargado.

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

De igual forma que con `assert_no_difference`, el primer argumento es la cadena '`User.count`', que se encarga de comparar entre `User.count` antes y después del contenido del bloque `assert_difference`. El segundo argumento (opcional) especifica el tamaño de la diferencia (en este caso, 1).

Incorporar `assert_difference` en el archivo del [Listado 7.21](#) nos conduce a la prueba que se muestra en el [Listado 7.26](#). Observe que hemos usado la variante `post_via_redirect` para enviar los datos a procesar a la ruta de usuario. Esto se encarga simplemente de seguir la redirección luego de enviar los datos, resultando en el despliegue de la plantilla '`users/show`'. (Probablemente sea buena idea escribir una prueba para el `flash` también, lo cual se deja como ejercicio ([Sección 7.7](#)).)

Listado 7.26: Una prueba para un registro válido. [VERDE](#)

`test/integration/users_signup_test.rb`

```
require 'test_helper'

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

Observe que el [Listado 7.26](#) también verifica que la página que muestra al usuario se despliega a continuación de un registro exitoso. Para que esta prueba funcione, es necesario que las rutas de `Users` ([Listado 7.3](#)), la acción `show`

de **Users** (Listado 7.5) y la vista **show.html.erb** (Listado 7.8) funcionen correctamente. Como resultado, la única línea

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

es una prueba suficiente para casi todo lo relacionado con la página del perfil del usuario. Esta clase de cobertura de principio a fin de características importantes de la aplicación es una de las razones por las cuales las pruebas de integración son tan útiles.

7.5 Despliegue de grado profesional

Ahora que tenemos una página de registro funcionando, es hora de desplegar nuestra aplicación y hacerla funcionar en producción. Aunque empezamos desplegando nuestra aplicación en el Capítulo 3, esta es la primera vez que realmente *hará* algo, por lo que aprovecharemos esta oportunidad para hacer un despliegue de grado profesional. En particular, agregaremos una característica importante a la aplicación de producción para hacer que el registro sea seguro, y reemplazaremos el servidor web por default con uno más apropiado para ser utilizado en el mundo real.

Como preparación para el despliegue, en este momento debería integrar sus cambios con la rama **master**:

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

7.5.1 SSL en producción

Cuando enviamos a registrar el formulario desarrollado en este capítulo, el nombre, la dirección electrónica y la contraseña son enviados por la red, y por tanto

son vulnerables de ser interceptados. Esto es una falla de seguridad potencialmente seria en nuestra aplicación, y la forma de arreglarla es utilizando una [Capa de Conexión Segura \(SSL, por sus siglas en inglés *Secure Sockets Layer*\)¹¹](#) para encriptar toda la información relevante antes de que salga del navegador local. Aunque podemos utilizar SSL sólo en la página de registro, realmente es más fácil implementarlo en todo el sitio, lo cual tiene beneficios adicionales al asegurar el inicio de sesión del usuario ([Capítulo 8](#)) y hacer nuestra aplicación inmune a la vulnerabilidad crítica de *secuestro de sesión* que discutimos en la [Sección 8.4](#).

Habilitar SSL es tan sencillo como descomentar una línea de código en el archivo **`production.rb`**, que contiene la configuración de producción para aplicaciones. Como se muestra en el [Listado 7.27](#), todo lo que necesitamos hacer es definir la variable **`config`** para forzar el uso de SSL en producción.

Listado 7.27: Configurando la aplicación para que utilice SSL en producción.
`config/environments/production.rb`

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

En este punto, necesitamos configurar el SSL en el servidor remoto. Habilitar un servidor en producción para que utilice SSL implica la compra y configuración de un *certificado SSL* para su dominio. Eso implica mucho trabajo, sin embargo y por suerte no necesitamos hacerlo aquí: para una aplicación ejecutándose en un dominio Heroku (tal como la aplicación de ejemplo), podemos aprovechar el certificado SSL de Heroku. Como resultado, cuando despleguemos la aplicación en la [Sección 7.5.2](#), SSL automáticamente estará ha-

¹¹Técnicamente hablando, SSL es ahora TLS, (por sus siglas en inglés *Transport Layer Security*, pero todas las personas que conozco aún la conocen como “SSL”).

bilitado. (Si usted quiere ejecutar SSL en un dominio personalizado, como www.example.com, revise la [página de Heroku acerca de SSL](#).)

7.5.2 Servidor web de Producción

Habiendo habilitado SSL, ahora necesitamos configurar nuestra aplicación para que utilice un servidor web apropiado para aplicaciones en producción. Por default, Heroku utiliza un servidor web escrito en Ruby puro, llamado WEBrick, que es fácil de configurar y ejecutar pero no es muy bueno cuando se trata de manejar tráfico significativo. Como resultado, WEBrick [no es apropiado para ser utilizado en producción](#), por lo que [reemplazaremos WEBrick con Puma](#), un servidor HTTP que es capaz de manejar una gran cantidad de peticiones.

Para agregar el nuevo servidor web, simplemente siga la [documentación de Puma en Heroku](#). El primer paso es incluir la gema `puma` en nuestro `Gemfile`, como se muestra en el Listado 7.28. Como no necesitamos la gema `Puma` localmente, el Listado 7.28 la considera en el grupo `:production`.

Listado 7.28: Agregando Puma al `Gemfile`.

```
source 'https://rubygems.org'  
.  
.  
.  
group :production do  
  gem 'pg',          '0.17.1'  
  gem 'rails_12factor', '0.0.2'  
  gem 'puma',         '2.11.1'  
end
```

Como configuramos Bundler para que no instale gemas de producción ([Sección 3.1](#)), el Listado 7.28 no agregará ninguna gema al ambiente de desarrollo, pero aún así necesitamos ejecutar Bundler para actualizar `Gemfile.lock`:

```
$ bundle install
```

El siguiente paso es crear un archivo llamado **config/puma.rb** y agregarle el contenido del [Listado 7.29](#). Este código viene directo de la [documentación de Heroku](#),¹² y no es necesario entenderlo.

Listado 7.29: El archivo de configuración para el servidor web de producción.

config/puma.rb

```
workers Integer(ENV['WEB_CONCURRENCY'] || 2)
threads_count = Integer(ENV['MAX_THREADS'] || 5)
threads threads_count, threads_count

preload_app!

rackup      DefaultRackup
port        ENV['PORT']     || 3000
environment ENV['RACK_ENV'] || 'development'

on_worker_boot do
  # Worker specific setup for Rails 4.1+
  # See: https://devcenter.heroku.com/articles/
  # deploying-rails-applications-with-the-puma-web-server#on-worker-boot
  ActiveRecord::Base.establish_connection
end
```

Finalmente, necesitamos crear el conocido **Procfile** para indicarle a Heroku que ejecute un proceso Puma en producción, como se muestra en el [Listado 7.30](#). El **Procfile** debería ser creado en el directorio raíz de su aplicación (es decir, en la misma ubicación que el archivo **Gemfile**).

Listado 7.30: Definiendo un **Procfile** para Puma.

./Procfile

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

Con la configuración del servidor web de producción terminada, estamos listos para subir nuestros cambios y desplegar:¹³

¹²El [Listado 7.29](#) cambia el formato ligeramente de forma que se ajuste al estándar de las 80 columnas.

¹³No hemos cambiado el modelo de datos en este capítulo, por lo que correr la migración en Heroku no debería ser necesario, pero sólo lo haremos si usted sigue los pasos de la [Sección 6.4](#). Como varios lectores han reportado tener problemas, he agregado **heroku run rake db:migrate** como paso final sólo para estar seguros.

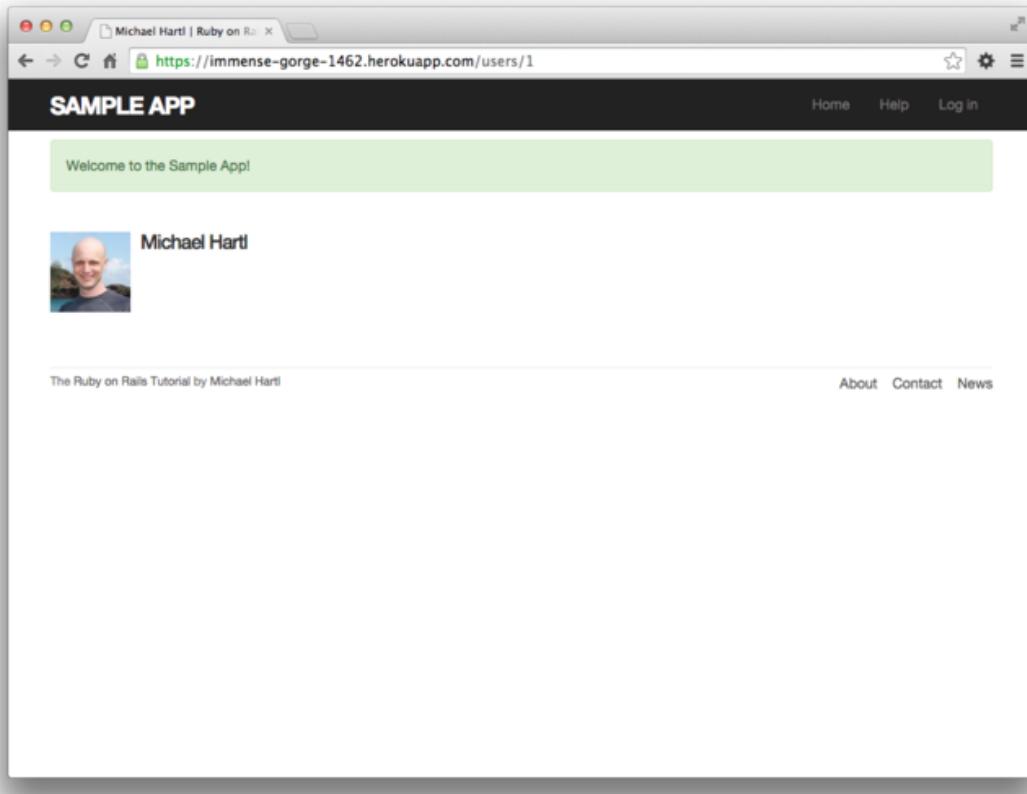


Figura 7.24: Registrándose en vivo en la Web.

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Use SSL and the Puma webserver in production"
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

El formulario de registro ahora está trabajando correctamente, y el resultado de un registro exitoso se muestra en la Figura 7.24. Observe la presencia de `https://` y el ícono de candado en la barra de direcciones de la Figura 7.24, lo cual indica que el SSL está funcionando.

7.5.3 Número de versión de Ruby

Cuando desplegamos en Heroku, puede ser que vea un mensaje de advertencia como éste:

```
##### WARNING:  
You have not declared a Ruby version in your Gemfile.  
To set your Ruby version add this line to your Gemfile:  
ruby '2.1.5'
```

La experiencia muestra que, al nivel de este tutorial, el precio de incluir un número de versión explícito de Ruby es más alto que los (escasos) beneficios, por lo que debería ignorar esta advertencia por ahora. El principal problema es que mantener su aplicación de ejemplo y el sistema en sincronía con la última versión de Ruby puede ser un gran inconveniente,¹⁴ y aún así, casi nunca hace una diferencia el número de versión exacta de Ruby que usted utilice. A pesar de eso, usted debería tener en cuenta que, si alguna vez termina ejecutando una aplicación de misión crítica en Heroku, especificar una versión exacta de Ruby en el **Gemfile** es recomendado para asegurar la máxima compatibilidad entre los ambientes de desarrollo y producción.

7.6 Conclusión

Ser capaces de registrar usuarios es un logro importante para nuestra aplicación. Aunque la aplicación de ejemplo aún tiene que hacer algo útil, hemos sentado las bases esenciales para el resto del desarrollo. En el Capítulo 8, completaremos nuestra maquinaria de autenticación al permitir que los usuarios inicien y cierren sesión en la aplicación. En el Capítulo 9, le permitiremos a todos los usuarios actualizar la información de su cuenta, y a los administradores del sitio, borrar usuarios; completando de esa forma el conjunto completo de acciones REST del recurso **Users** de la Tabla 7.1.

¹⁴Por ejemplo, luego de desperdiciar varias horas tratando infructuosamente de instalar Ruby 2.1.4 en mi máquina local, descubrí que Ruby 2.1.5 había sido liberado el día anterior. Los intentos para instalar la versión 2.1.5 también fallaron.

7.6.1 Qué aprendimos en este capítulo

- Rails despliega información de depuración útil mediante el método **debug**.
- Los *mixins* de Sass nos permiten agrupar reglas CSS para ser empaquetadas y reutilizadas en múltiples lugares.
- Rails viene con tres ambientes estándar: **desarrollo, pruebas y producción**.
- Podemos interactuar con los usuarios como un *recurso* a través de un conjunto estándar de URLs REST.
- Los Gravatares proporcionan una forma conveniente de desplegar imágenes para representar usuarios.
- El método auxiliar **form_for** es utilizado para crear formularios que interactúan con objetos *Active Record*.
- Los registros fallidos despliegan la página de creación de usuarios y despliegan mensajes de error automáticamente determinados por *Active Record*.
- Rails proporciona el **flash** como una forma estándar de desplegar mensajes temporales.
- Los registros exitosos crean un usuario en la base de datos y redireccionan a la página que muestra el usuario y despliegan un mensaje de bienvenida.
- Podemos usar las pruebas de integración para verificar el comportamiento del envío de datos y atrapar regresiones.
- Podemos configurar nuestra aplicación en producción para que utilice SSL con la finalidad de que las comunicaciones seguras y **Puma** para asegurar un alto desempeño.

7.7 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Verifique que el código del Listado 7.31 permite al método auxiliar `gravatar_for` definido en la Sección 7.1.4 tomar un parámetro opcional `size`, permitiendo código como `gravatar_for user, size: 50` en la vista. (Utilizaremos esta función auxiliar mejorada en la Sección 9.3.1.)
2. Escriba una prueba para los mensajes de error implementados en el Listado 7.18. Qué tan detalladas desea hacer sus pruebas, es su decisión; una plantilla sugerida aparece en el Listado 7.32.
3. Escriba una prueba para el `flash` implementado en la Sección 7.4.2. Qué tan detalladas desea hacer sus pruebas, es su decisión; una plantilla sugerida ultra-minimalista aparece en el Listado 7.33, donde debe reemplazar `FILL_IN` con el código apropiado. (Probar que la llave es correcta, ó el texto, es una prueba frágil, por lo que prefiero probar únicamente que el `flash` no está vacío.)
4. Como observó en la Sección 7.4.2, el `flash` HTML del Listado 7.25 es feo. Verifique ejecutando el conjunto de pruebas, que el código más limpio del Listado 7.34, que utiliza la función auxiliar de Rails `content_tag`, también funciona.

Listado 7.31: Agregando un arreglo hash de opciones a la función auxiliar `gravatar_for`.

`app/helpers/users_helper.rb`

```
module UsersHelper

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

Listado 7.32: Una plantilla para las pruebas de los mensajes de error.

`test/integration/users_signup_test.rb`

```
require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

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

Listado 7.33: Una plantilla para una prueba del `flash`.

`test/integration/users_signup_test.rb`

```
require 'test_helper'
.
```

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

Listado 7.34: El `flash` dentro de la estructura de diseño del sitio usando `content_tag`.

`app/views/layouts/application.html.erb`

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

Capítulo 8

Inicio y Cierre de Sesión

Ahora que nuevos usuarios pueden registrarse en nuestro sitio ([Capítulo 7](#)), es tiempo de permitirles que puedan iniciar y cerrar sesión. Implementaremos los tres modelos más comunes para la funcionalidad de inicio / cierre de sesión en la web: “olvidar” a los usuarios cuando el navegador se cierra ([Secciones 8.1](#) y [8.2](#)), recordar usuarios *automáticamente* ([Sección 8.4](#)) y *opcionalmente* recordar usuarios dependiendo de si eligen la opción “recuérdame” ([Sección 8.4.5](#)).¹

El sistema de autenticación que desarrollaremos en este capítulo nos permitirá personalizar el sitio e implementar un modelo de autorización basado en el status del login y en la identidad del usuario actual. Por ejemplo, en este capítulo actualizaremos el encabezado del sitio con enlaces para iniciar / cerrar sesión y para el perfil. En el [Capítulo 9](#), implementaremos un modelo de seguridad en el que únicamente los usuarios que hayan iniciado sesión pueden visitar la página principal del usuario, y en el que únicamente el usuario correcto puede tener acceso a la página para editar su información y además de restringir a que únicamente los usuarios administrativos puedan eliminar a otros usuarios de la base de datos. Finalmente, en el [Capítulo 11](#), utilizaremos la identidad de un usuario que ha iniciado sesión para crear microposts asociados con él mismo, y en el [Capítulo 12](#) le permitiremos al usuario actual, seguir a otros usuarios de la aplicación (recibiendo de esa forma retroalimentación de los microposts de esos usuarios).

¹Otro modelo común es hacer que la sesión expire después de una cierta cantidad de tiempo. Esto es especialmente apropiado para sitios que contienen información confidencial, tales como bancos e instituciones financieras.

Este es un capítulo largo y desafiante en el que cubriremos muchos aspectos detallados del inicio de sesión de sistemas comunes, por lo que le recomiendo que se enfoque en completarlo sección por sección. Adicionalmente, muchos usuarios han recomendado darle un segundo repaso para mayor beneficio.

8.1 Sesiones

HTTP es un *protocolo sin estado*, que trata cada petición como una transacción independiente y que es incapaz de utilizar información de peticiones previas. Esto significa que no hay forma *dentro del protocolo de transferencia de hipertexto* de recordar la identidad de un usuario de página a página; en vez de esto, las aplicaciones web que requieren que el usuario pueda iniciar sesión, deben utilizar una *sesión*, que es una conexión semi-permanente entre dos computadoras (tales como una computadora cliente ejecutando un navegador web y un servidor ejecutando Rails).

Las técnicas más comunes para implementar sesiones en Rails involucran el uso de *galletas*, que son pequeños pedazos de texto ubicados en el navegador del usuario. Como las galletas persisten de una página a otra, pueden almacenar información (tal como un id de usuario) que puede ser utilizada por la aplicación para recuperar el usuario que está en sesión de la base de datos. En esta sección y en la Sección 8.2, utilizaremos el método de Rails llamado **session** para crear sesiones temporales que expiran automáticamente cuando el navegador se cierra,² y luego en la Sección 8.4 agregaremos sesiones que duren por más tiempo utilizando otro método Rails llamado **cookies**.

Es conveniente modelar las sesiones como un recurso RESTful: visitar la página para iniciar sesión desplegará un formulario para sesiones *nuevas*, iniciar la sesión *creará* una sesión, y cerrar la sesión la *destruirá*. A diferencia del recurso de usuarios, que utiliza una base de datos en el servidor (mediante el modelo de usuario) para persistir datos, el recurso de sesiones utilizará galletas, y mucho del trabajo que implica el inicio de sesión proviene de construir esta maquinaria de autenticación basada en galletas. En esta sección y

²Algunos navegadores ofrecen la opción de restaurar tales sesiones mediante la opción “continúe donde se quedó”, pero por supuesto que Rails no tiene control sobre este comportamiento.

la próxima, nos prepararemos para este trabajo construyendo un controlador de sesiones, un formulario de inicio de sesión y las acciones del controlador relevantes. Luego terminaremos el inicio de sesión de usuario en la [Sección 8.2](#) agregando el código necesario para la manipulación de la sesión.

Como en capítulos anteriores, realizaremos nuestro trabajo sobre una rama nueva e integraremos nuestros cambios al final:

```
$ git checkout master
$ git checkout -b log-in-log-out
```

8.1.1 Controlador de Sesiones

El inicio y cierre de sesión corresponden a acciones REST particulares del controlador de sesiones: el formulario para el inicio de sesión es manejado por la acción **new** (que cubriremos en esta sección), de hecho el inicio de sesión es manejado a través de enviar una petición POST a la acción **create** ([Sección 8.2](#)), y el cierre de sesión es manejado al enviar una petición DELETE a la acción **destroy** ([Sección 8.3](#)). (Recuerde la asociación de los verbos HTTP con las acciones REST de la [Tabla 7.1](#).)

Para empezar, generaremos el controlador de sesiones con una acción **new**:

```
$ rails generate controller Sessions new
```

(Al incluir **new** también generamos las *vistas*, razón por la cual no incluimos las acciones como **create** y **destroy** que no requieren vistas.) Siguiendo el modelo de la [Sección 7.2](#) para la página de registro, nuestro plan es crear un formulario para iniciar sesión que permita crear sesiones nuevas, como se bosqueja en la [Figura 8.1](#).

A diferencia del recurso de usuarios, en el que utilizamos el método especial **resources** para obtener un conjunto completo de rutas RESTful de manera automática ([Listado 7.3](#)), el recurso **Sessions** sólo utilizará rutas nombradas, manejando peticiones GET y POST con la ruta **login** y peticiones DELETE con

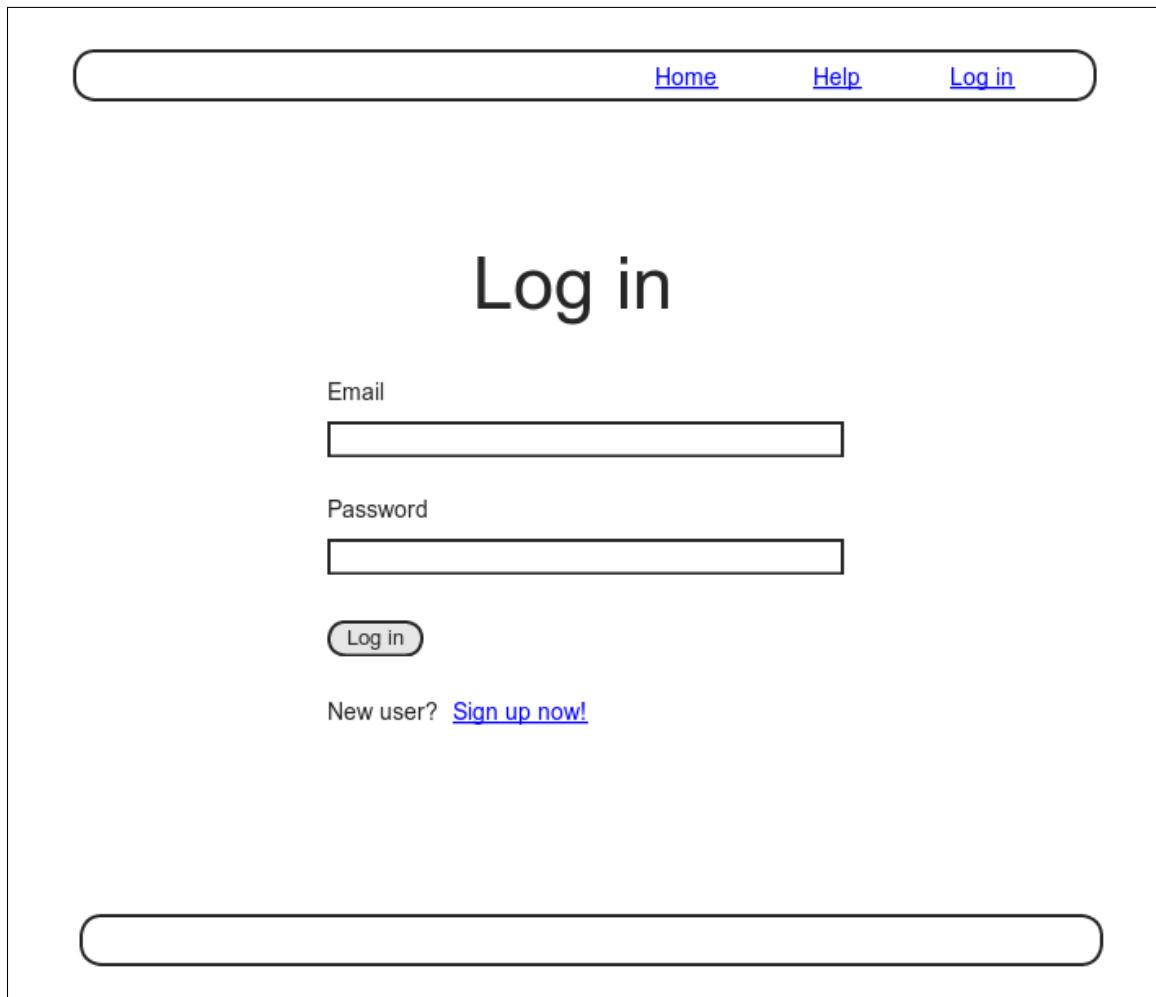


Figura 8.1: Un bosquejo del formulario de inicio de sesión.

Petición HTTP	URL	Ruta nombrada	Acción	Propósito
GET	/login	<code>login_path</code>	<code>new</code>	página para crear una nueva sesión (login)
POST	/login	<code>login_path</code>	<code>create</code>	crea una nueva sesión (login)
DELETE	/logout	<code>logout_path</code>	<code>destroy</code>	borra una sesión (log out)

Tabla 8.1: Rutas proporcionadas por las reglas de sesiones del Listado 8.1.

la ruta **logout**. El resultado aparece en el Listado 8.1 (el cual también elimina las rutas innecesarias generadas por **rails generate controller**).

Listado 8.1: Agregando un recurso para obtener las acciones RESTful estándar para las sesiones.

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help' => 'static_pages#help'
  get 'about' => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup' => 'users#new'
  get 'login' => 'sessions#new'
  post 'login' => 'sessions#create'
  delete 'logout' => 'sessions#destroy'
  resources :users
end
```

Las rutas definidas en el Listado 8.1 corresponden a URLs y acciones, similares a las de los usuarios (Tabla 7.1), como se muestra en la Tabla 8.1.

Puesto que ahora hemos agregado varias rutas nombradas personalizadas, es útil echar un vistazo a la lista completa de rutas de nuestra aplicación, la cual podemos generar mediante **rake routes**:

\$ bundle exec rake routes	
Prefix Verb URI Pattern	Controller#Action
root GET /	static_pages#home
help GET /help(.:format)	static_pages#help
about GET /about(.:format)	static_pages#about
contact GET /contact(.:format)	static_pages#contact
signup GET /signup(.:format)	users#new
login GET /login(.:format)	sessions#new

```

    POST /login(.:format)           sessions#create
  logout DELETE /logout(.:format) sessions#destroy
   users GET  /users(.:format)      users#index
          POST   /users(.:format)      users#create
 new_user GET   /users/new(.:format) users#new
edit_user GET   /users/:id/edit(.:format) users#edit
   user GET   /users/:id(.:format)   users#show
        PATCH  /users/:id(.:format)   users#update
        PUT    /users/:id(.:format)   users#update
     DELETE /users/:id(.:format)   users#destroy

```

No es necesario entender los resultados en detalle, pero visualizar las rutas de esta manera nos proporciona una panorama general de las acciones soportadas por nuestra aplicación.

8.1.2 Formulario para inicio de sesión

Habiendo definido el controlador relevante y la ruta, ahora rellenaremos la vista para las sesiones nuevas, es decir, el formulario para inicio de sesión. Comparando las Figuras 8.1 con la 7.11, vemos que el formulario para inicio de sesión es similar en apariencia al formulario de registro, excepto que tiene dos campos (dirección electrónica y contraseña) en vez de cuatro.

Como podemos observar en la Figura 8.2, cuando la información para iniciar sesión es inválida, queremos volver a mostrar la página de inicio de sesión y mostrar un mensaje de error. En la Sección 7.3.3, utilizamos un parcial para mensajes de error, pero vimos en esa sección que esos mensajes eran proporcionados de forma automática por *Active Record*. Esto no funcionaría para los errores en la creación de sesiones porque la sesión no será un objeto *Active Record*, por lo que mostraremos el error como un mensaje flash.

Recuerde del Listado 7.13 que el formulario de registro utiliza la función auxiliar **form_for**, tomando como argumento la variable de instancia del usuario **@user**:

```

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

```

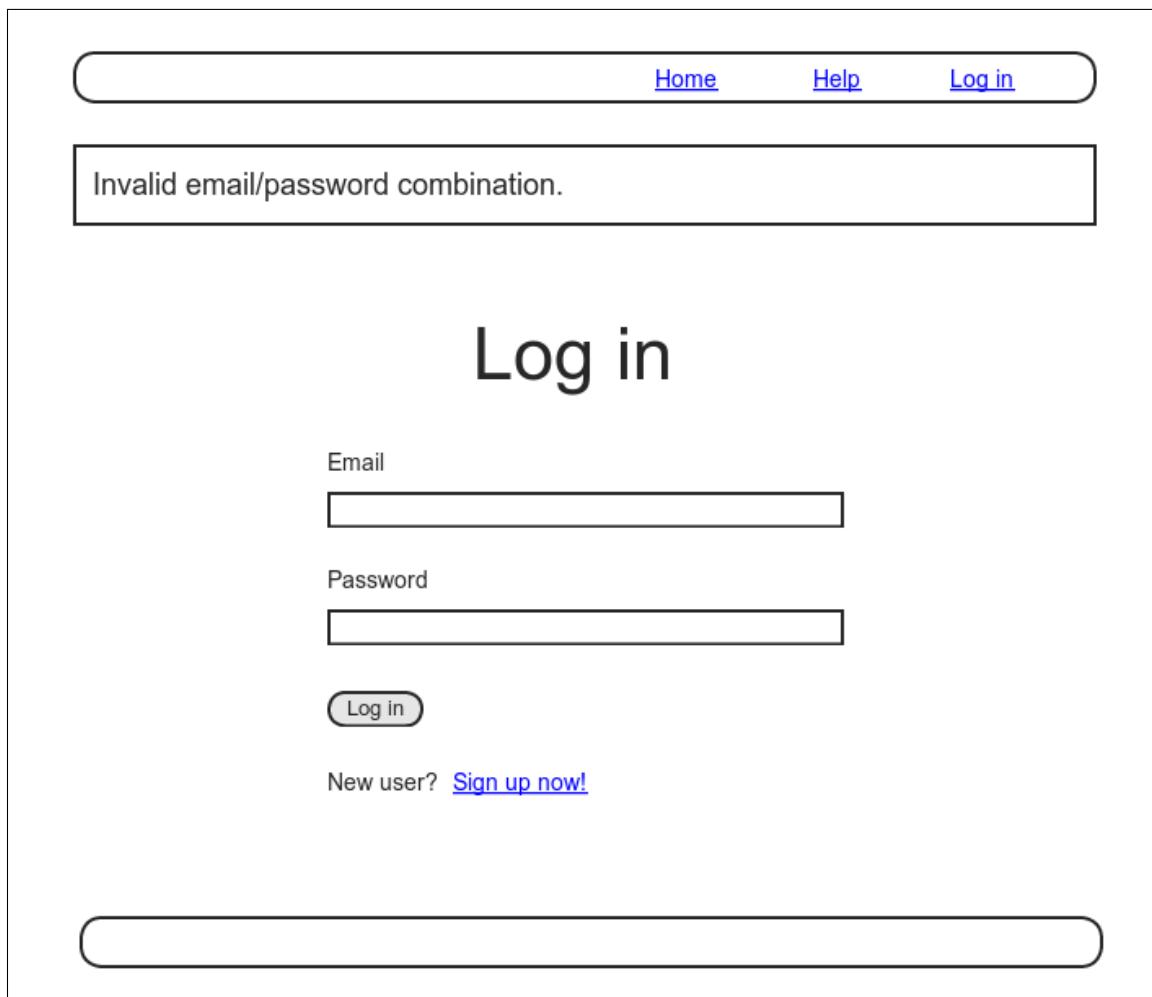


Figura 8.2: Un bosquejo de un inicio de sesión fallido.

La principal diferencia entre el formulario de sesión y el de registro es que no tenemos un modelo de sesión, y por tanto, tampoco un análogo de la variable `@user`. Esto significa que, al construir el formulario para la nueva sesión, necesitamos proporcionarle a `form_for` un poco más de información: en particular,

```
form_for(@user)
```

le permitía a Rails deducir que la **acción** del formulario debía ser un POST a la URL /users, en el caso de sesiones necesitamos indicar el *nombre* del recurso y la correspondiente URL:³

```
form_for(:session, url: login_path)
```

Con el apropiado `form_for` a mano, es fácil crear un formulario para inicio de sesión que corresponda con el bosquejo de la Figura 8.1 usando el formulario de registro como modelo (Listado 7.13), como se muestra en el Listado 8.2.

Listado 8.2: Código para el formulario de inicio de sesión.

app/views/sessions/new.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session, url: login_path) do |f| %>

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

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

      <%= f.submit "Log in", class: "btn btn-primary" %>
    <% end %>
```

³Una segunda opción es utilizar `form_tag` en vez de `form_for`, lo cual puede ser más correcto para Rails en términos idiomáticos, pero tiene menos en común con el formulario de registro, y en este momento lo que quiero es enfatizar el paralelismo de las estructuras.

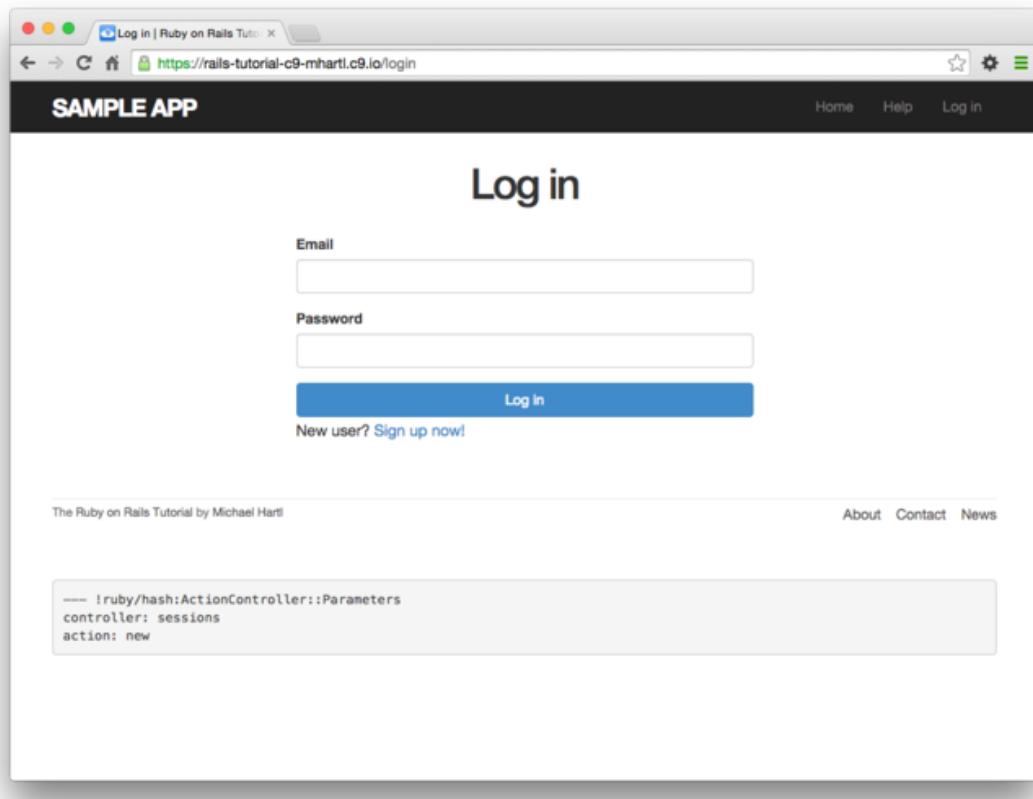


Figura 8.3: El formulario para inicio de sesión.

```
<p>New user? <%= link_to "Sign up now!", signup_path %></p>
</div>
</div>
```

Observe que hemos agregado un enlace a la página de registro por conveniencia. Con el código del [Listado 8.2](#), el formulario para iniciar sesión aparece como en la [Figura 8.3](#). (Como el enlace de navegación “Log in” no ha sido rellenado aún, usted tendrá que escribir la URL /login directamente en la barra de direcciones de su navegador. Arreglaremos este defecto en la [Sección 8.2.3](#).)

El formulario HTML generado aparece en el [Listado 8.3](#).

Listado 8.3: HTML para el formulario de inicio de sesión producido por el Listado 8.2.

```
<form accept-charset="UTF-8" action="/login" method="post">
  <input name="utf8" type="hidden" value="" />
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ51LqyAbnbAUkdo=" />
  <label for="session_email">Email</label>
  <input class="form-control" id="session_email"
         name="session[email]" type="text" />
  <label for="session_password">Password</label>
  <input id="session_password" name="session[password]"
         type="password" />
  <input class="btn btn-primary" name="commit" type="submit"
         value="Log in" />
</form>
```

Comparando el Listado 8.3 con el Listado 7.15, usted puede ser capaz de adivinar que al enviar este formulario al servidor, se creará un arreglo hash `params` donde `params[:session][:email]` y `params[:session][:password]` corresponden a los campos de dirección electrónica y contraseña respectivamente.

8.1.3 Encontrando y autenticando al usuario

Como en el caso de crear usuarios (registro), el primer paso para crear sesiones (inicio de sesión) es manejar los datos de entrada *inválidos*. Empezaremos recordando qué sucede cuando un formulario es enviado al servidor, y luego nos encargaremos de los mensajes de error que aparecerán en caso de que el inicio de sesión falle (como se bosquejó en la Figura 8.2.) Luego estableceremos las bases para un inicio de sesión exitoso (Sección 8.2) al evaluar cada petición de inicio de sesión basados en la validez de la pareja dirección de correo / contraseña.

Empecemos por definir una acción `create` minimalista para el controlador de sesiones, junto con acciones `new` y `destroy` vacías (Listado 8.4). La acción `create` del Listado 8.4 no hace nada excepto mostrar la vista `new`, pero es suficiente para que empecemos. Enviar el formulario `/sessions/new` nos lleva al resultado que se muestra en la Figura 8.4.

Listado 8.4: Una versión preliminar de la acción `create` del controlador de sesiones.

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end

  def create
    render 'new'
  end

  def destroy
  end
end
```

Revisando cuidadosamente la información de depuración de la Figura 8.4 nos muestra que, como vislumbramos al final de la Sección 8.1.2, el envío de los datos al servidor resulta en la creación de un arreglo hash `params` que contiene la dirección electrónica y la contraseña bajo la llave `session`, el cual (omitiendo algunos detalles irrelevantes que son usados internamente por Rails) aparece como sigue:

```
---
session:
  email: 'user@example.com'
  password: 'foobar'
commit: Log in
action: create
controller: sessions
```

Como con el caso del registro de usuario (Figura 7.15), estos parámetros forman un arreglo hash *anidado* como el que vimos en el Listado 4.10. En particular, `params` contiene un arreglo hash anidado de la forma

```
{ session: { password: "foobar", email: "user@example.com" } }
```

Esto significa que

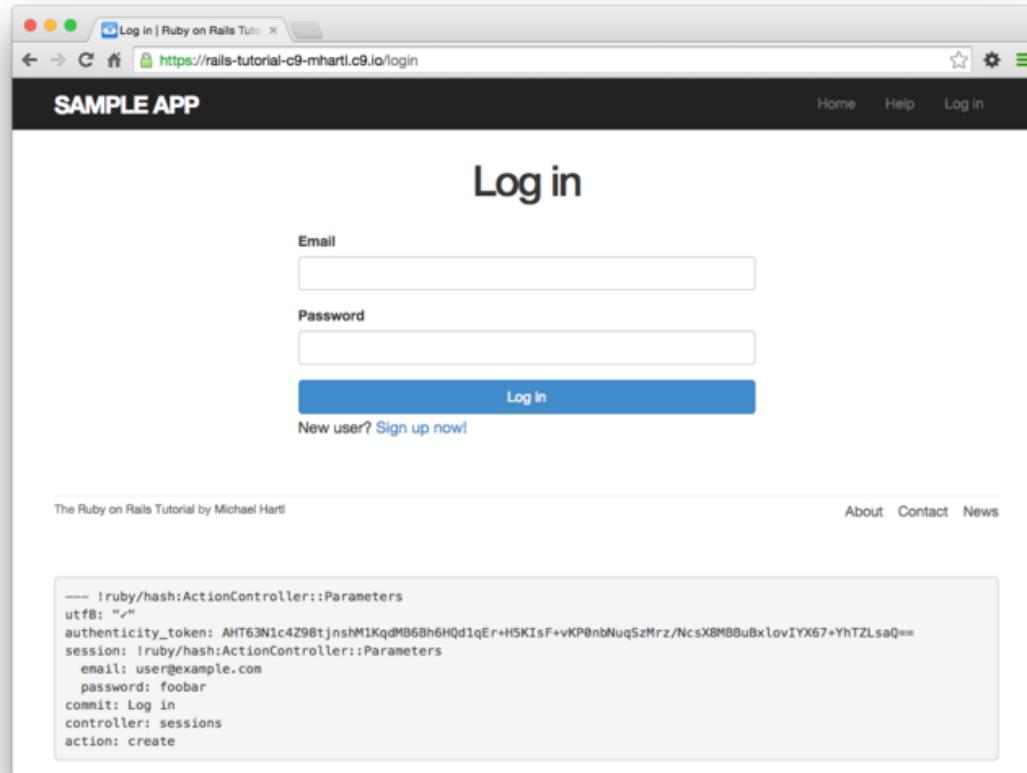


Figura 8.4: El primer inicio de sesión fallido, con **create** como en el [Listado 8.4](#).

```
params[:session]
```

es en sí mismo un arreglo hash:

```
{ password: "foobar", email: "user@example.com" }
```

Como resultado,

```
params[:session][:email]
```

es la dirección electrónica enviada y

```
params[:session][:password]
```

es la contraseña enviada.

En otras palabras, dentro de la acción `create` el arreglo hash `params` contiene toda la información necesaria para autenticar usuarios mediante su dirección de correo y contraseña. No es coincidencia que ya tenemos exactamente los métodos que necesitamos: el método `User.find_by` proporcionado por *Active Record* (Sección 6.1.4) y el método `authenticate` method proporcionado por `has_secure_password` (Sección 6.3.4). Recuerde que `authenticate` regresa `falso` para una autenticación fallida (Sección 6.3.4), nuestra estrategia para que el usuario inicie sesión puede resumirse como se muestra en el Listado 8.5.

Listado 8.5: Encontrando y autenticando al usuario.

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  def new
  end
```

Usuario	Contraseña	a && b
no existente	cualquier cosa	(nil && [cualquier cosa]) == falso
usuario válido	contraseña errónea	(verdadero && falso) == falso
usuario válido	contraseña correcta	(verdadero && verdadero) == verdadero

Tabla 8.2: Posibles resultados de `user && user.authenticate(...)`.

```

def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    # Log the user in and redirect to the user's show page.
  else
    # Create an error message.
    render 'new'
  end
end

def destroy
end
end

```

La primera línea resaltada del Listado 8.5 extrae el usuario de la base de datos usando la dirección electrónica que se ha enviado. (Recuerde que en la Sección 6.2.5 vimos que las direcciones de correo electrónico son guardadas en minúsculas, por lo que debemos utilizar aquí el método `downcase` para asegurarnos de que coincidan cuando la dirección que envía el usuario es válida.) La siguiente línea puede ser un poco confusa pero es bastante común en la programación en Rails:

```
user && user.authenticate(params[:session][:password])
```

Esto utiliza `&&` (*and* lógico) para determinar si el usuario resultante es o no válido. Tomando en cuenta que cualquier otro objeto que no sea `nil` o `falso` es `verdadero` en un contexto booleano (Sección 4.2.3), las posibilidades aparecen en la Tabla 8.2. Podemos observar en ella que la sentencia `if` es `verdadera` sólo si un usuario con la dirección electrónica dada existe en la base de datos y tiene asociada la contraseña proporcionada, exactamente como es requerido.

8.1.4 Mostrando un mensaje flash

Recuerde de la Sección 7.3.3 que podemos notificar errores al registrarse usando los mensajes de error del modelo **User**. Estos errores están asociados con un objeto *Active Record* particular, pero esta estrategia no va a funcionar aquí porque la sesión no corresponde a un modelo de *Active Record*. En vez de esto, colocaremos un mensaje en el flash para ser desplegado en caso de un inicio de sesión fallido. El primer intento, ligeramente incorrecto, aparece en el Listado 8.6.

Listado 8.6: Un intento (ineficaz) de manejar el inicio de sesión fallido.

```
app/controllers/sessions_controller.rb

class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Log the user in and redirect to the user's show page.
    else
      flash[:danger] = 'Invalid email/password combination' # Not quite right!
      render 'new'
    end
  end

  def destroy
  end
end
```

Como el mensaje flash forma parte de la estructura de diseño del sitio (Listado 7.25), el mensaje **flash[:danger]** automáticamente se muestra; y debido al CSS de Bootstrap, automáticamente adquiere estilo (Figura 8.5).

Desafortunadamente, como se hace notar en el texto y en el comentario del Listado 8.6, este código no es del todo correcto. Sin embargo, la página se ve bien; entonces, ¿cuál es el problema? El asunto es que el contenido del flash persiste durante una *peticIÓN*, pero—a diferencia de un redireccionamiento, el cual utilizamos en el Listado 7.24—volver a mostrar una página con **render** no cuenta como una petición. El resultado es que el mensaje flash persiste una

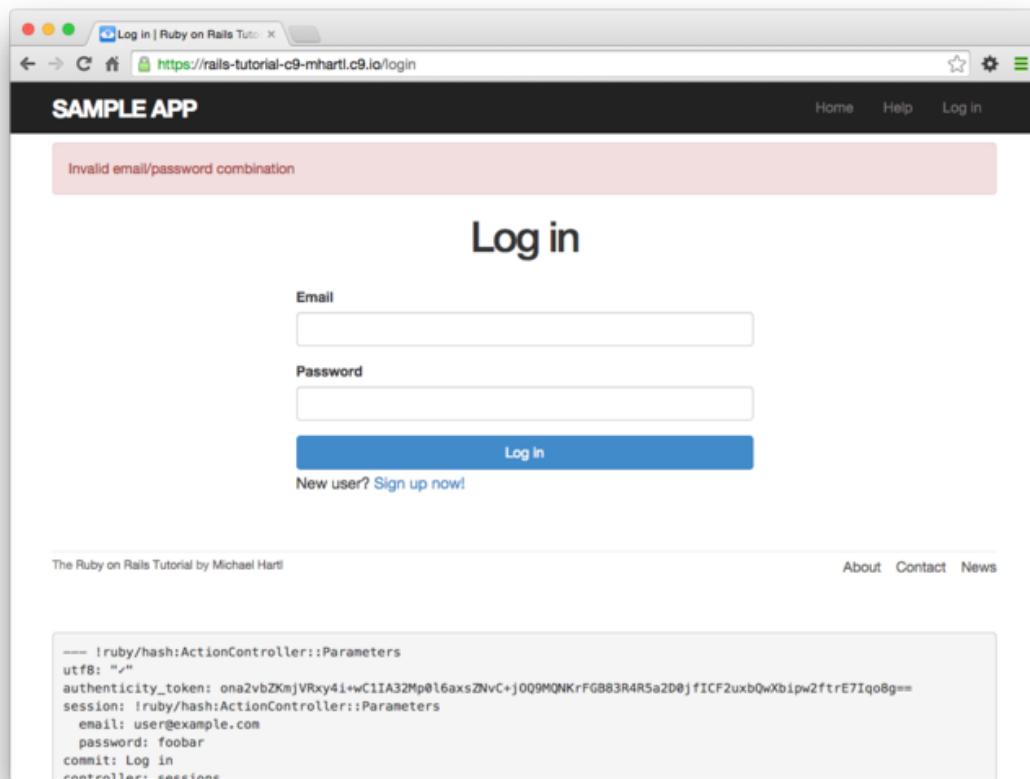


Figura 8.5: El mensaje flash para un inicio de sesión fallido.

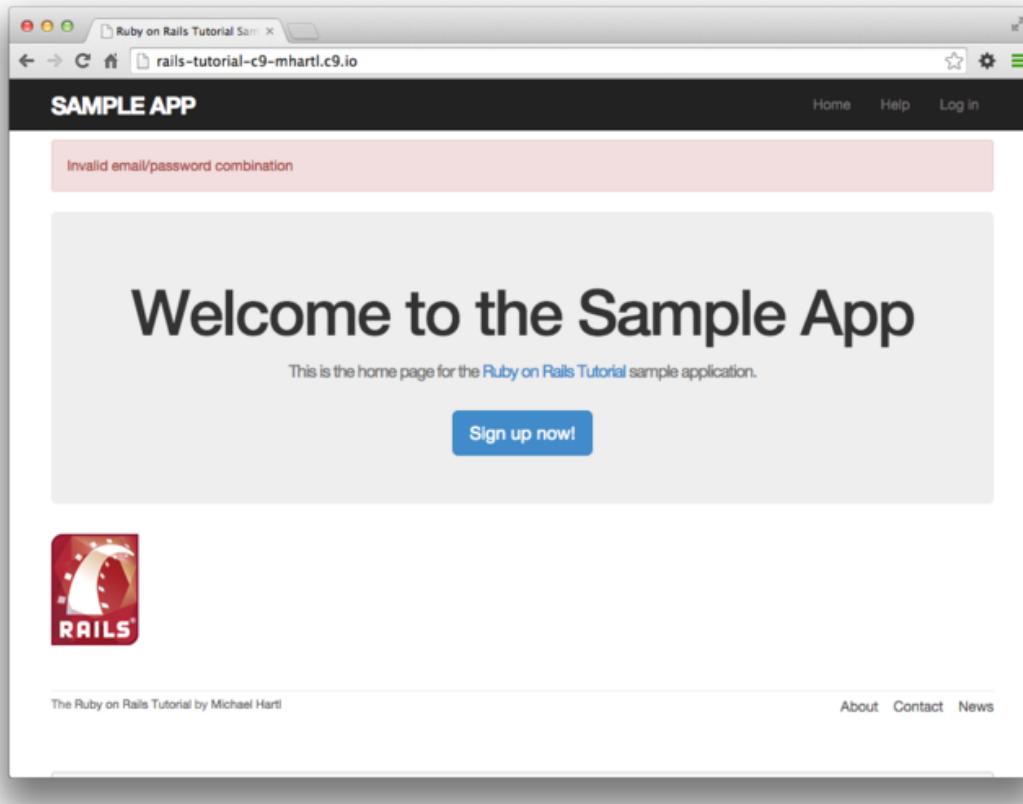


Figura 8.6: Un ejemplo de la perseverancia de flash.

petición más de lo que queremos. Por ejemplo, si enviamos información para iniciar sesión con datos erróneos y luego damos click en la página principal, el mensaje flash será mostrado por segunda vez ([Figura 8.6](#)). Arreglar este defecto es la tarea de la [Sección 8.1.5](#).

8.1.5 Una prueba para flash

El comportamiento incorrecto del flash es un defecto menor en nuestra aplicación. De acuerdo con las directrices de pruebas del Recuadro 3.3, esta es exactamente la clase de situación en la que debemos escribir una prueba para

atrapar un error de forma que no vuelva a producirse. Entonces escribiremos una pequeña prueba de integración para el envío de datos del formulario de inicio de sesión antes de continuar. Aparte de documentar el defecto y prevenir una regresión, esto también nos proporciona una buena base para futuras pruebas de integración de inicio y cierre de sesión.

Empezamos por generar una prueba de integración para el comportamiento de inicio de sesión de nuestra aplicación:

```
$ rails generate integration_test users_login
  invoke  test_unit
  create    test/integration/users_login_test.rb
```

A continuación, necesitamos una prueba para capturar la secuencia mostrada en las Figuras 8.5 y 8.6. Los pasos básicos son los siguientes:

1. Visitar la ruta de inicio de sesión.
2. Verificar que el formulario se muestra apropiadamente.
3. Enviar a la ruta de sesiones un arreglo hash **params** con datos erróneos.
4. Verificar que el formulario vuelve a mostrarse y que el mensaje flash aparece.
5. Visitar otra página (por ejemplo, la página Home).
6. Verificar que el mensaje flash *no* aparece en la nueva página.

Una prueba que implementa los pasos anteriores se muestra en el Listado 8.7.

Listado 8.7: Una prueba para detectar la perseverancia no deseada de flash.

ROJO

```
test/integration/users_login_test.rb

require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  test "login with invalid information" do
    get login_path
    assert_template 'sessions/new'
    post login_path, session: { email: "", password: "" }
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
    assert flash.empty?
  end
end
```

Luego de agregar la prueba del [Listado 8.7](#), el resultado de la misma debe estar en ROJO:

Listado 8.8: ROJO

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb
```

Esto muestra cómo ejecutar un (y sólo un) archivo de prueba, usando el argumento **TEST** y la ruta completa del archivo.

La forma de hacer que la prueba fallida del [Listado 8.7](#) pase, es reemplazando **flash** con la variante especial **flash.now**, que fue diseñada específicamente para mostrar mensajes flash en páginas desplegadas. A diferencia del contenido de **flash**, el contenido de **flash.now** desaparece tan pronto como existe una nueva petición, que es exactamente el comportamiento que queremos probar en el [Listado 8.7](#). Con esta substitución, el código de la aplicación corregida aparece en el [Listado 8.9](#).

Listado 8.9: Código correcto para un inicio de sesión fallido. VERDE
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      # Log the user in and redirect to the user's show page.
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Luego podemos verificar que tanto la prueba de integración del inicio de sesión, como el conjunto completo de pruebas están en VERDE:

Listado 8.10: VERDE

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb
$ bundle exec rake test
```

8.2 Entrar al sistema

Ahora que nuestro formulario de inicio de sesión puede manejar datos inválidos, el siguiente paso es manejar correctamente datos válidos permitiendo que el usuario inicie su sesión. En esta sección, iniciaremos la sesión del usuario junto con una galleta de sesión temporal que expira automáticamente cuando el navegador se cierra. En la [Sección 8.4](#), agregaremos sesiones que persisten aún después de haber cerrado el navegador.

Implementar sesiones involucrará definir un gran número de funciones relacionadas entre sí que serán utilizadas en múltiples controladores y vistas. Puede

recordar de la Sección 4.2.5 que Ruby provee de *módulos*, un medio para empaquetar tales funciones en un solo lugar. Convenientemente, un módulo auxiliar de Sesiones fue generado automáticamente cuando generamos el controlador de Sesiones (Sección 8.1.1). Más aún, tales auxiliares estn automáticamente incluídos en las vistas Rails; al incluir el módulo en la clase base de todos los controladores (el controlador de la Aplicación), logramos hacerlos disponibles en nuestros controladores también (Listado 8.11).

Listado 8.11: Incluyendo el módulo auxiliar de Sesiones en el controlador de la aplicación.

```
app/controllers/application_controller.rb
```

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper
end
```

Una vez terminada esta configuración, estamos listos para escribir el código para que los usuarios inicien sesión.

8.2.1 El método `log_in`

Iniciar la sesión del usuario es simple con la ayuda del método `session` definido por Rails. (Este método está separado y es distinto del que generó el controlador de Sesiones en la Sección 8.1.1.) Podemos tratar la `sesión` como si fuera un arreglo hash, y asignarle valores como sigue:

```
session[:user_id] = user.id
```

Esto coloca una galleta temporal en el navegador del usuario, que contiene una versión encriptada del id del usuario, lo que nos permite recuperar el id en páginas siguientes mediante `session[:user_id]`. En contraste con la galleta persistente creada por el método `cookies` (Sección 8.4), la galleta temporal creada por el método `session` expira al momento en que el navegador es cerrado.

Como queremos usar la misma técnica de inicio de sesión en un par de lugares más, definiremos un método llamado `log_in` en el auxiliar de Sesiones, como se muestra en el [Listado 8.12](#).

Listado 8.12: La función `log_in`.

```
app/helpers/sessions_helper.rb

module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
end
```

Como las galletas temporales que son creadas con el método `session` están automáticamente encriptadas, el código del [Listado 8.12](#) es seguro, y no hay forma de que un atacante use la información de la sesión para iniciar una sesión como si fuera el usuario. Esto aplica sólo para sesiones temporales iniciadas con el método `session`, sin embargo, *no* es el caso de sesiones persistentes creadas con el método `cookies`. Las galletas permanentes son vulnerables a los ataques de *secuestro de sesión*, por lo que en la [Sección 8.4](#) tendremos mucho más cuidado acerca de la información que ponemos en el navegador del usuario.

Con el método `log_in` definido en el [Listado 8.12](#), estamos listos para completar la acción `create` del controlador de Sesión, al iniciar la sesión del usuario y redireccionándolo a su página del perfil. El resultado aparece en el [Listado 8.13](#).⁴

Listado 8.13: Iniciando la sesión del usuario.

```
app/controllers/sessions_controller.rb

class SessionsController < ApplicationController

  def new
```

⁴El método `log_in` está disponible en el controlador de Sesiones debido a la inclusión del módulo del [Listado 8.11](#).

```
end

def create
  user = User.find_by(email: params[:session][:email].downcase)
  if user && user.authenticate(params[:session][:password])
    log_in user
    redirect_to user
  else
    flash.now[:danger] = 'Invalid email/password combination'
    render 'new'
  end
end

def destroy
end
end
```

Observe la redirección compacta

```
redirect_to user
```

que vimos anteriormente en la [Sección 7.4.1](#). Rails automáticamente convierte esto a la ruta de la página del perfil del usuario:

```
user_url(user)
```

Con la acción `create` definida en el [Listado 8.13](#), el formulario de inicio de sesión del [Listado 8.2](#) ahora debe estar funcionando. No tiene ningún efecto en la forma en que se muestra la aplicación, por lo que, a menos que se inspeccione la información de la sesión directamente en el navegador, no hay forma de decir si el usuario ha iniciado o no su sesión. Como primer paso para hacer cambios más visibles, en la [Sección 8.2.2](#) recuperaremos al usuario actual de la base de datos utilizando el id de la sesión. En la [Sección 8.2.3](#), cambiaremos los enlaces en la estructura de diseño de la aplicación, incluyendo la URL del perfil del usuario actual.

8.2.2 Usuario actual

Habiendo colocado el id del usuario de forma segura en una sesión temporal, ahora podemos recuperarlo de ahí en páginas subsecuentes, lo cual haremos definiendo un método `current_user` para buscar al usuario en la base de datos que corresponde al id de la sesión. El propósito de `current_user` es permitir construcciones tales como

```
<%= current_user.name %>
```

y

```
redirect_to current_user
```

Para buscar al usuario actual, una posibilidad es utilizar el método `find`, como hicimos en la página del perfil del usuario ([Listado 7.5](#)):

```
User.find(session[:user_id])
```

Pero recuerde de la [Sección 6.1.4](#) que `find` lanza una excepción si el id del usuario no existe. Este comportamiento es apropiado en una página de perfil porque sólo sucederá si el id es inválido, pero en este caso `session[:user_id]` a menudo será `nil` (por ejemplo, para usuarios que no han iniciado sesión). Para manejar estos casos, utilizaremos el mismo método `find_by` que utilizamos para buscar direcciones electrónicas en el método `create`, con un `id` en vez de `email`:

```
User.find_by(id: session[:user_id])
```

En vez de arrojar una excepción, este método regresa `nil` (indicando que no existe tal usuario) si el id no es válido.

Ahora podemos definir el método `current_user` como sigue:

```
def current_user
  User.find_by(id: session[:user_id])
end
```

Esto funcionaría bien, pero haría demasiadas consultas a la base de datos si, por ejemplo, `current_user` apareciera varias veces en una misma página. En vez de esto, seguiremos una convención común en Ruby guardando el resultado de `User.find_by` en una variable de instancia, que consulta la base de datos la primera vez, pero regresa el valor de la variable en las invocaciones siguientes:⁵

```
if @current_user.nil?
  @current_user = User.find_by(id: session[:user_id])
else
  @current_user
end
```

Recordando el operador *or* `||` que vimos en la Sección 4.2.3, podemos reescribir esto como sigue:

```
@current_user = @current_user || User.find_by(id: session[:user_id])
```

Como un objeto `User` es verdadero en un contexto booleano, la llamada a `find_by` sólo se ejecuta si la variable `@current_user` no ha sido asignada.

Aunque el código anterior podría funcionar, no es idiomáticamente correcto en Ruby; en su lugar, la forma apropiada de escribir la asignación a la variable `@current_user` es:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Esto utiliza el operador potencialmente confuso pero frecuentemente usado `||=` (“o igual”) (Recuadro 8.1).

⁵ Esta práctica de recordar asignaciones de variables de la invocación de un método a otra invocación del mismo método, se conoce como *memoization*. (Observe que éste es un término técnico; en particular, no es un error de ortografía inglesa para la palabra “memorization”.)

Recuadro 8.1. ¿Qué *\$@! es | |= ?

El operador de asignación `|=` (“o igual”) es una expresión común en Ruby y de ahí que es importante para los aspirantes a desarrollador en Rails reconocerlo. Aunque al principio puede parecer misterioso, *o igual* es fácil de entender con una analogía.

Empezaremos observando el patrón común de incrementar una variable:

```
x = x + 1
```

Muchos lenguajes proporcionan una expresión sintácticamente más corta para esta operación; en Ruby (y en C, C++, Perl, Python, Java, etc.), puede escribirse como sigue:

```
x += 1
```

También existen constucciones análogas para otros operadores:

```
$ rails console
>> x = 1
=> 1
>> x += 1
=> 2
>> x *= 3
=> 6
>> x -= 8
=> -2
>> x /= 2
=> -1
```

En cada caso, el patrón es que `x = x O` y `x O= y` son equivalentes para cualquier operador O.

Otro patrón común en Ruby es la asignación de una variable si ésta es `nil` pero dejarla sin cambios en otro caso. Recordando el operador *or* `||` que vimos en la [Sección 4.2.3](#), podemos escribir esto como sigue:

```
>> @foo
=> nil
>> @foo = @foo || "bar"
=> "bar"
>> @foo = @foo || "baz"
=> "bar"
```

Puesto que `nil` es falso en un contexto booleano, la primera asignación de `@foo` es la evaluación de `nil || "bar"`, que es `"bar"`. Análogamente, la segunda asignación es `@foo || "baz"`, es decir, `"bar" || "baz"`, que al evaluarla resulta en `"bar"`. Esto es porque otra cosa diferente de `nil` o `falso` es `verdadero` en un contexto booleano, y la serie de expresiones `||` termina luego de que la primera expresión verdadera es evaluada. (Esta práctica de evaluar expresiones `||` de izquierda a derecha y detenerse en el primer valor verdadero se conoce como *evaluación de corto-circuito*. El mismo principio se aplica a las expresiones `&&`, excepto que en este caso la evaluación se detiene en el primer valor `falso`.)

Comparando las sesiones de consola con los diferentes operadores, vemos que `@foo = @foo || "bar"` sigue el patrón `x = x O y` con `||` en vez de `O`:

<code>x = x + 1</code>	\rightarrow	<code>x += 1</code>
<code>x = x * 3</code>	\rightarrow	<code>x *= 3</code>
<code>x = x - 8</code>	\rightarrow	<code>x -= 8</code>
<code>x = x / 2</code>	\rightarrow	<code>x /= 2</code>
<code>@foo = @foo "bar"</code>	\rightarrow	<code>@foo = "bar"</code>

Entonces vemos que `@foo = @foo || "bar"` y `@foo ||= "bar"` son equivalentes. En el contexto del usuario actual, esto sugiere la siguiente construcción:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Voilà !

(Por cierto, internamente Ruby evalúa realmente la expresión `@foo || @foo = "bar"`, la cual evita una asignación innecesaria cuando `@foo` es `nil` o `falso`. Pero esta expresión no explica la notación `| |=`, por lo que el comentario anterior utiliza el casi equívocamente `@foo = @foo || "bar"`.)

Aplicando los resultados de la discusión anterior llegamos al sucinto método `current_user` que se muestra en el [Listado 8.14](#).

Listado 8.14: Buscando al usuario actual en la sesión.

app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
    @current_user ||= User.find_by(id: session[:user_id])
  end
end
```

Con el método `current_user` del Listado 8.14 funcionando, estamos ahora en posición de hacer cambios a nuestra aplicación en base al status de la sesión del usuario.

8.2.3 Actualizando los enlaces de la estructura de diseño

La primera aplicación práctica de iniciar sesión involucra el cambio de los enlaces en la estructura de diseño, basados en el status de la sesión. En particular, como se muestra en el bosquejo de la [Figura 8.7](#),⁶ agregaremos enlaces para las páginas del cierre de la sesión, para la configuración del usuario, para enlistar a

⁶Imagen de <http://www.flickr.com/photos/hermanusbackpackers/3343254977/>.

todos los usuarios, y para el perfil del usuario actual. Observe en la Figura 8.7 que los enlaces del perfil y del cierre de sesión aparecen en un menú colapsable “Account”; que en el Listado 8.16 veremos cómo crearlo con Bootstrap.

En este momento, en la vida real yo consideraría escribir una prueba de integración para capturar el comportamiento recién descrito. Como observamos en el Recuadro 3.3, conforme usted se vuelve más familiar con las herramientas de prueba de Rails, se sentirá más inclinado a escribir las pruebas primero. Aunque en este caso, tal prueba involucra varias ideas nuevas, por lo que por ahora es mejor postergarlo (Sección 8.2.4).

La forma de cambiar los enlaces en la estructura de diseño involucra el uso de sentencias if-else dentro de Ruby embebido para mostrar un conjunto de enlaces si el usuario está en sesión y otro conjunto en cualquier otro caso:

```
<% if logged_in? %>
  # Links for logged-in users
<% else %>
  # Links for non-logged-in-users
<% end %>
```

Esta clase de código requiere de la existencia del método booleano `logged_in?`, el cual definiremos ahora.

Un usuario está en sesión si `current_user` no es `nil`. Verificar esto requiere el uso del operador “not” (Sección 4.2.3), el cual se escribe utilizando un signo de exclamación `!`. El método resultante `logged_in?` aparece en el Listado 8.15.

Listado 8.15: El método auxiliar `logged_in?`.

```
app/helpers/sessions_helper.rb

module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Returns the current logged-in user (if any).
  def current_user
```

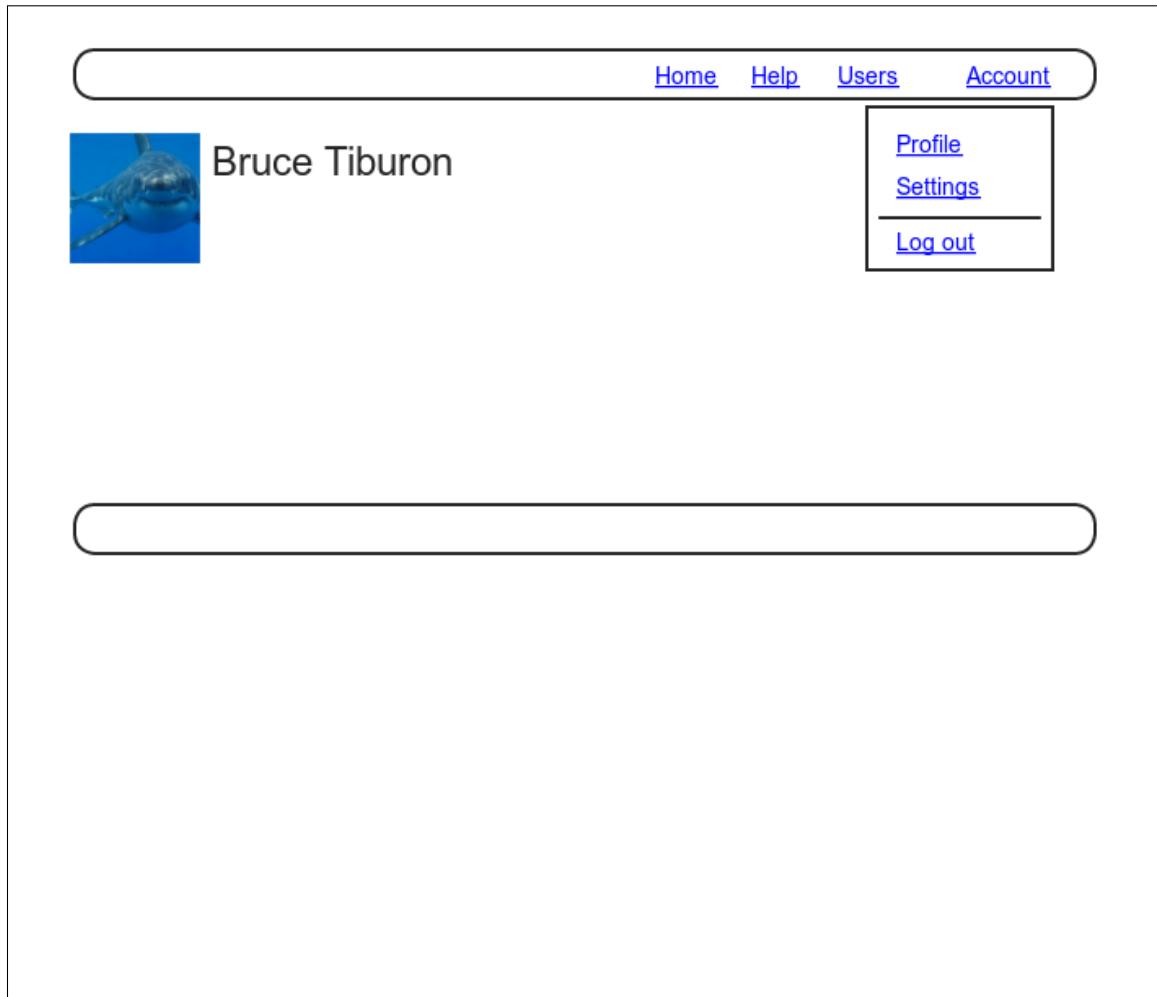


Figura 8.7: Un bosquejo del perfil del usuario luego de un inicio de sesión exitoso.

```


    @current_user ||= User.find_by(id: session[:user_id])
end

# Returns true if the user is logged in, false otherwise.
def logged_in?
  !current_user.nil?
end
end


```

Con la adición del Listado 8.15, ahora estamos listos para cambiar los enlaces de la estructura de diseño si un usuario está en sesión. Hay cuatro nuevos enlaces, dos de los cuales están pendientes (los completaremos en el Capítulo 9):

```


<%= link_to "Users", '#' %>
<%= link_to "Settings", '#' %>


```

Mientras tanto, el enlace para cerrar sesión, utiliza la ruta definida en el Listado 8.1:

```


<%= link_to "Log out", logout_path, method: :delete %>


```

Observe que el enlace para cierre de sesión pasa como argumento un arreglo hash indicando que debería enviarse con una petición HTTP cuyo método sea DELETE.⁷ También agregaremos un enlace al perfil como sigue:

```


<%= link_to "Profile", current_user %>


```

Aquí podemos escribir

```


<%= link_to "Profile", user_path(current_user) %>


```

⁷Los navegadores realmente no son capaces de emitir peticiones DELETE; Rails las simula con JavaScript.

pero como siempre, Rails nos permite relacionar directamente al usuario al convertir automáticamente `current_user` en `user_path(current_user)` en este contexto. Finalmente, cuando los usuarios *no están* en sesión, utilizaremos la ruta de inicio de sesión definida en el [Listado 8.1](#) para crear un enlace al formulario de inicio de sesión:

```
<%= link_to "Log in", login_path %>
```

Poniendo todo junto obtenemos la versión actualizada del parcial del encabezado, la cual se muestra en el [Listado 8.16](#).

Listado 8.16: Actualizando los enlaces en la estructura de diseño para usuarios que han iniciado sesión.

app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Account <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li><%= link_to "Profile", current_user %></li>
              <li><%= link_to "Settings", '#' %></li>
              <li class="divider"></li>
              <li>
                <%= link_to "Log out", logout_path, method: "delete" %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

Como parte de incluir los nuevos enlaces en la estructura de diseño, el [Listado 8.16](#) aprovecha la funcionalidad de Bootstrap para crear menús desplegables.⁸ Observe en particular, la inclusión de las clases CSS especiales de Bootstrap CSS, tales como `dropdown`, `dropdown-menu`, etc. Para activar el menú desplegable, necesitamos incluir la biblioteca de JavaScript de Bootstrap dentro del archivo de la cadena de procesos conectados de Rails `application.js`, como se muestra en el [Listado 8.17](#).

Listado 8.17: Agregando la biblioteca de JavaScript de Bootstrap a `application.js`.

`app/assets/javascripts/application.js`

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require turbolinks
//= require_tree .
```

En este momento, debería visitar la página de inicio de sesión e iniciar una sesión con un usuario válido, con lo cual prueba de forma efectiva el código de las tres secciones anteriores.⁹ Con el código de los Listados 8.16 y 8.17, debería ver el menú desplegable y los enlaces para los usuarios que están en sesión, como se muestra en la [Figura 8.8](#). Si usted cierra su navegador completamente, también debería poder verificar que la aplicación olvida su status de inicio de sesión, pidiéndole que inicie sesión nuevamente para ver los cambios descritos anteriormente.

8.2.4 Probando los cambios a la estructura de diseño

Habiendo verificado manualmente que la aplicación se está comportando adecuadamente al iniciar exitosamente una sesión, y antes de continuar, escribiremos una prueba de integración para capturar este comportamiento y atrapar

⁸Consulte la [página de componentes de Bootstrap](#) para mayor información.

⁹Si usted está utilizando el IDE en la nube, le recomiendo utilizar un navegador diferente para probar el comportamiento de inicio de sesión, de forma que no tenga que cerrar el navegador donde está corriendo el IDE.

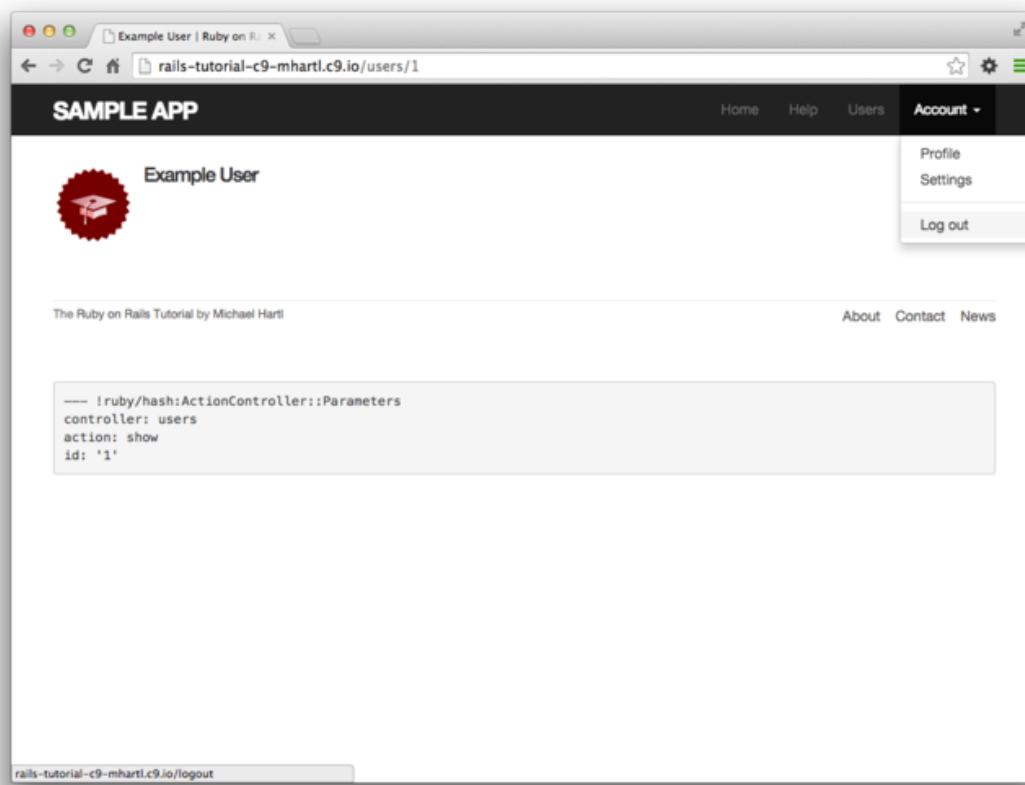


Figura 8.8: Un usuario en sesión con los nuevos enlaces y un menú desplegable.

regresiones. Lo haremos sobre la prueba del [Listado 8.7](#) y escribiremos una serie de pasos para verificar la siguiente secuencia de acciones:

1. Visitar la página de inicio de sesión.
2. Enviar información válida a la ruta de sesiones.
3. Verificar que el enlace a la página de inicio de sesión desaparece.
4. Verificar que aparece un enlace para cerrar sesión.
5. Verificar que aparece un enlace a la página del perfil.

Con la finalidad de visualizar estos cambios, nuestra prueba necesita iniciar sesión con un usuario previamente registrado, lo que significa que tal usuario debe existir previamente en la base de datos. El modo Rails por default para hacer esto es utilizar *fixtures*, que es un mecanismo para organizar datos que deben ser cargados en la base de datos de pruebas. Como descubrimos en la [Sección 6.2.5](#), necesitamos borrar el contenido del archivo *fixtures*, de forma que nuestras pruebas de unicidad de la dirección electrónica pasaran ([Listado 6.30](#)). Ahora estamos listos para empezar a llenar este archivo vacío con nuestros propios datos.

En este caso, sólo necesitamos un usuario, cuya información consista de un nombre y dirección electrónica válidos. Como necesitaremos iniciar sesión con este usuario, también necesitamos incluir una contraseña válida para compararla con la contraseña enviada a la acción **create** del controlador de Sesiones. Siguiendo el modelo de datos de la [Figura 6.8](#), vemos que esto implica crear un atributo **password_digest** para el usuario del *fixture*, lo cual lograremos definiendo un método **digest** por nuestra cuenta.

Como vimos en la [Sección 6.3.1](#), la digestión de la contraseña es creada utilizando bcrypt (mediante **has_secure_password**), por lo que necesitamos crear la contraseña del *fixture* con el mismo método. Al revisar el código fuente de **secure password**, encontramos que este método es

```
BCrypt::Password.create(string, cost: cost)
```

donde **string** es la cadena que será digerida y **cost** es el *parámetro de costo* que determina el costo computacional de calcular la digestión. Utilizar un costo alto genera una digestión a partir de la cual es computacionalmente imposible deducir la contraseña original, lo cual es una medida de seguridad importante en un ambiente de producción, pero en pruebas queremos que el método **digest** sea tan rápido como sea posible. El código fuente para la contraseña segura contiene una línea para este caso también:

```
cost = ActiveRecord::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :  
      BCrypt::Engine.cost
```

Este código más bien obscuro, que usted no entiende en detalle, se encarga del comportamiento recién descrito: utiliza el parámetro del costo mínimo en pruebas, y un costo normal (alto) en producción. (Aprenderemos más acerca de la extraña notación `?-:` en la [Sección 8.4.5](#).)

Hay varios lugares donde podemos guardar el resultado del método **digest**, pero tendremos oportunidad en la [Sección 8.4.1](#) de reutilizar **digest** en el modelo **User**. Esto sugiere que el lugar adecuado para el método está en el archivo **user.rb**. Como no necesariamente tendremos acceso a un objeto usuario cuando calculemos la digestión (como es el caso del archivo *fixtures*), crearemos el método **digest** a nivel de la clase **User**, lo cual (como vimos brevemente en la [Sección 4.4.1](#)) lo convierte en un *método de clase*. El resultado aparece en el [Listado 8.18](#).

Listado 8.18: Agregando un método de digestión para ser utilizado en los *fixtures*.

app/models/user.rb

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

```

validates :email, presence: true, length: { maximum: 255 },
          format: { with: VALID_EMAIL_REGEX },
          uniqueness: { case_sensitive: false }
has_secure_password
validates :password, presence: true, length: { minimum: 6 }

# Returns the hash digest of the given string.
def User.digest(string)
  cost = ActiveRecord::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                BCrypt::Engine.cost
  BCrypt::Password.create(string, cost: cost)
end
end

```

Con el método **digest** del Listado 8.18, estamos listos para crear los datos de usuario para un usuario válido, como se muestra en el Listado 8.19.

Listado 8.19: Datos para probar el inicio de sesión del usuario.

test/fixtures/users.yml

```

michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>

```

Observe en particular, que el archivo *fixtures* puede incluir Ruby embebido, lo cual nos permite utilizar

```
<%= User.digest('password') %>
```

para crear la digestión de una contraseña válida para el usuario de pruebas.

Aunque hemos definido el atributo **password_digest** requerido por **has_secure_password**, algunas veces es conveniente hacer referencia a la contraseña plana (virtual) también. Desafortunadamente, esto es imposible de arreglar con *fixtures*, y agregar un atributo **password** al Listado 8.19 provoca que Rails se queje de que no existe tal columna en la base de datos (lo cual es cierto). Lo haremos adoptando la convención de que todos los usuarios en el *fixture* tienen la misma contraseña ('**password**').

Habiendo creado un *fixture* con un usuario válido, podemos utilizarlo dentro de una prueba como sigue:

```
user = users(:michael)
```

Aquí **users** corresponde al nombre del archivo *fixture* **users.yml**, mientras que el símbolo **:michael** hace referencia al usuario con la llave que se muestra en el [Listado 8.19](#).

Con el usuario del *fixture* anterior, ahora podemos escribir una prueba para los enlaces de la estructura de diseño convirtiendo la lista al inicio de esta sección en código, como se observa en el [Listado 8.20](#).

Listado 8.20: Una prueba para iniciar sesión de usuario con información válida. [VERDE](#)

```
test/integration/users_login_test.rb

require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "login with valid information" do
    get login_path
    post login_path, session: { email: @user.email, password: 'password' }
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
  end
end
```

Aquí hemos usado

```
assert_redirected_to @user
```

para verificar el correcto redireccionamiento y

```
follow_redirect!
```

para visitar realmente la página objetivo. El [Listado 8.20](#) también verifica que el enlace a la página de inicio de sesión desaparece, verificando que existen *cero* enlaces a esta ruta dentro de la página:

```
assert_select "a[href=?]", login_path, count: 0
```

Al incluir la opción extra **count: 0**, le indicamos a **assert_select** que esperamos que haya cero enlaces que coincidan con el patrón dado. (Compárelo con **count: 2** del [Listado 5.25](#), el cual verifica lo mismo pero para exactamente 2 enlaces coincidentes.)

Como el código de la aplicación ya estaba funcionando, esta prueba debería estar en **VERDE**:

Listado 8.21: VERDE

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb \
> TESTOPTS="--name test_login_with_valid_information"
```

Esto nos muestra cómo ejecutar una prueba específica dentro de un archivo de pruebas, al pasar la opción

```
TESTOPTS="--name test_login_with_valid_information"
```

que contiene el nombre de la prueba. (El nombre de la prueba es sólo la palabra “test” y las palabras en la descripción de la prueba se unen con guiones bajos.) Observe que el **>** en la segunda línea es un carácter “de continuación de línea” insertado automáticamente por la consola, y no debe ser escrito literalmente.

8.2.5 Inicio de sesión al registrarse

Aunque nuestro sistema de autenticación está funcionando, los usuarios recién registrados pueden estar confundidos, puesto que no han iniciado sesión por default. Como sería extraño forzar a los usuarios a iniciar sesión justo después de haberse registrado, iniciaremos su sesión automáticamente como parte del proceso de registro. Para implementar este comportamiento, todo lo que necesitamos hacer es agregar una llamada a `log_in` en la acción `create` del controlador de usuarios, como se muestra en el Listado 8.22.¹⁰

Listado 8.22: Iniciando sesión al registrar un nuevo usuario.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

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

  private

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

¹⁰De igual forma que con el controlador de sesiones, el método `log_in` está disponible en el controlador de usuarios debido a la inclusión del módulo del Listado 8.11.

Para probar el comportamiento del Listado 8.22, podemos agregar una línea a la prueba del Listado 7.26 para verificar que el usuario tiene una sesión. Es útil en este contexto definir un método auxiliar `is_logged_in?` similar al auxiliar `logged_in?` que definimos en el Listado 8.15, el cual regresa `verdadero` si existe un id de usuario en la sesión (de prueba) y falso en cualquier otro caso (Listado 8.23). (Como los métodos auxiliares no están disponibles en las pruebas, no podemos utilizar el `current_user` como en el Listado 8.15, pero el método `session` está disponible, por lo que utilizaremos éste en su lugar.) Aquí utilizamos `is_logged_in?` en vez de `logged_in?` de forma que los métodos auxiliares de pruebas tienen nombres diferentes de los métodos auxiliares del controlador de sesiones, lo cual evita que sean confundidos unos con otros.¹¹

Listado 8.23: Un método booleano para el status de inicio de sesión dentro de las pruebas.

```
test/test_helper.rb

ENV['RAILS_ENV'] ||= 'test'
.
.
.

class ActiveSupport::TestCase
  fixtures :all

  # Returns true if a test user is logged in.
  def is_logged_in?
    !session[:user_id].nil?
  end
end
```

Con el código del Listado 8.23, podemos afirmar que el usuario ha iniciado sesión luego del registro usando la línea mostrada en el Listado 8.24.

¹¹Por ejemplo, una vez tuve un conjunto de pruebas que estaba en VERDE aún después de haber borrado accidentalmente el método `log_in` principal en el auxiliar de `Sessions`. El motivo fue que las pruebas estaban utilizando un método auxiliar con el mismo nombre, haciendo que pasaran aún cuando la aplicación estaba totalmente sin funcionar. Como con `is_logged_in?`, evitaremos este caso definiendo el auxiliar de la prueba `log_in_as` como en el Listado 8.50.

Listado 8.24: Una prueba de sesión luego del registro. VERDE

```
test/integration/users_signup_test.rb
```

```
require 'test_helper'

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

En este momento, el conjunto de pruebas debería seguir en VERDE:

Listado 8.25: VERDE

```
$ bundle exec rake test
```

8.3 Cerrando sesión

Como lo discutimos en la [Sección 8.1](#), nuestro modelo de autenticación consiste en mantener a los usuarios en sesión hasta que ellos la cierren explícitamente. En esta sección, agregaremos esta funcionalidad necesaria. Como el enlace “Cerrar Sesión” ya había sido definido ([Listado 8.16](#)), todo lo que necesitamos escribir es una acción válida en el controlador para destruir sesiones de usuario.

Hasta ahora, las acciones del controlador de sesiones han seguido la convención RESTful de utilizar **new** para una página de inicio de sesión y **create** para completarlo. Continuaremos en esta dirección utilizando una acción **destroy**

para eliminar sesiones, es decir, para cerrar sesión. A diferencia de la funcionalidad de inicio de sesión, que usamos tanto en el [Listado 8.13](#) como en el [Listado 8.22](#), sólo estaremos cerrando sesión en un solo lugar, por lo que pondremos el código relevante directamente en la acción **destroy**. Como veremos en la [Sección 8.4.6](#), este diseño (con un poco de refactorización) hará que la maquinaria de autenticación sea más fácil de probar.

Cerrar la sesión se encarga de deshacer los efectos del método **log_in** del [Listado 8.12](#), lo cual implica borrar el id del usuario de la sesión.¹² Para hacer esto, usamos el método **delete** como sigue:

```
session.delete(:user_id)
```

También haremos que el usuario actual sea **nil**, aunque en este caso no importa, porque se realiza un redireccionamiento inmediato a la URL raíz.¹³ Como con **log_in** y sus métodos asociados, pondremos el método resultante **log_out** en el módulo auxiliar de sesiones, como se muestra en el [Listado 8.26](#).

Listado 8.26: El método **log_out**.

```
app/helpers/sessions_helper.rb

module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  .

  .

  .

  # Logs out the current user.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

¹² Algunos navegadores ofrecen la funcionalidad “recordar dónde me quedé”, que restaura la sesión automáticamente, por lo que asegúrese de deshabilitar estas características antes de cerrar sesión.

¹³ Hacer que **@current_user** sea **nil** sólo importa si **@current_user** fue creado antes de invocar la acción **destroy** (lo que no sucede) y si no emitíramos un redireccionamiento inmediato (lo que sí sucede). Esta es una combinación poco probable de eventos, y con la aplicación como está construida en este momento, es innecesario, pero como es un tema relacionado con la seguridad, lo incluiré aquí por completez.

Podemos poner el método **log_out** para ser utilizado en la acción del controlador de sesiones **destroy**, como se muestra en el [Listado 8.27](#).

Listado 8.27: Destruyendo una sesión (cierre de sesión de usuario).

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out
    redirect_to root_url
  end
end
```

Para probar la maquinaria de cierre de sesión, podemos agregar algunos pasos a las pruebas de inicio de sesión del usuario del [Listado 8.20](#). Luego de iniciar sesión, utilizamos **delete** para emitir una petición DELETE a la ruta de cierre de sesión ([Tabla 8.1](#)) y verificamos que el usuario ha cerrado su sesión y ha sido redirigido a la URL raíz. También verificamos que el enlace para iniciar sesión reaparece, y que los enlaces al perfil y al cierre de sesión desaparecen. Los nuevos pasos aparecen en el [Listado 8.28](#).

Listado 8.28: Una prueba para cierre de sesión de usuario. **VERDE**

test/integration/users_login_test.rb

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest
```

```
•  
•  
•  
test "login with valid information followed by logout" do  
  get login_path  
  post login_path, session: { email: @user.email, password: 'password' }  
  assert is_logged_in?  
  assert_redirected_to @user  
  follow_redirect!  
  assert_template 'users/show'  
  assert_select "a[href=?]", login_path, count: 0  
  assert_select "a[href=?]", logout_path  
  assert_select "a[href=?]", user_path(@user)  
  delete logout_path  
  assert_not is_logged_in?  
  assert_redirected_to root_url  
  follow_redirect!  
  assert_select "a[href=?]", login_path  
  assert_select "a[href=?]", logout_path, count: 0  
  assert_select "a[href=?]", user_path(@user), count: 0  
end  
end
```

(Ahora que tenemos `is_logged_in?` disponible en las pruebas, hemos ganado una bonificación `assert is_logged_in?` inmediatamente luego de enviar datos válidos a la ruta de sesiones.)

Con la acción `destroy` de la sesión, definida y probada, el flujo de registro-inicio de sesión/cierre de sesión está completo, y el conjunto de pruebas debería estar en **VERDE**:

Listado 8.29: VERDE

```
$ bundle exec rake test
```

8.4 Recuérdame

El sistema de inicio de sesión que terminamos en la Sección 8.2 es auto-contenido y completamente funcional, pero la mayoría de los sitios web tienen la funcionalidad adicional de recordar sesiones de usuario aún después de que los usuarios han cerrado sus navegadores. En esta sección, empezaremos por recordar sesiones de usuario por default, expirándolas sólo cuando explícitamente

cierren sesión. En la Sección 8.4.5, habilitaremos un modelo alternativo común, una casilla “recuérdame” que proporciona a los usuarios la opción de ser recordados. Ambos modelos tienen grado profesional, el primero es usado en sitios como GitHub y Bitbucket, y el segundo, en sitios como Facebook y Twitter.

8.4.1 Recordando el token y la digestión

En la Sección 8.2, usamos el método Rails **session** para almacenar el id del usuario, pero esta información desaparece cuando el usuario cierra su navegador. En esta sección, daremos un paso adelante hacia las sesiones persistentes al generar un *token de recordatorio* adecuado para crear galletas permanentes usando el método **cookies**, junto con una *digestión del recordatorio* segura para autenticar esos tokens.

Como observamos en la Sección 8.2.1, la información almacenada usando **session** está segura automáticamente, pero no es el caso de la información almacenada utilizando **cookies**. En particular, las galletas persistentes son vulnerables al [secuestro de sesión](#), en el que un atacante utiliza un token recordado que roba para iniciar sesión como un usuario en particular. Hay cuatro formas principales de robar galletas: (1) usando un [detector de paquetes](#) para detectar galletas que viajan sobre redes inseguras,¹⁴ (2) comprometiendo una base de datos que contiene los tokens recordados, (3) usando [Scripts entre sitios \(XSS, por sus siglas en inglés Cross-Site Scripting\)](#), y (4) obteniendo acceso físico al equipo donde el usuario está en sesión. Evitamos el primer problema en la Sección 7.5 al utilizar la [Capa de Conexión Segura \(SSL, por sus siglas en inglés Secure Sockets Layer\)](#) en todo el sitio, lo que protege nuestros datos que viajan por la red de los detectores de paquetes. Evitaremos el segundo problema almacenando una digestión del token recordado en vez del token mismo, de una forma muy similar a la que almacenamos las digestiones de las contraseñas en vez de las contraseñas planas en la Sección 6.3. Rails automáticamente evita el tercer problema evitando que cualquier contenido sea inyectado en las vistas. Finalmente, aunque no hay un camino férreo para detener a los atacantes

¹⁴El secuestro de sesión ha sido ampliamente difundido por la aplicación Firesheep, que mostró que recordar tokens en muchos sitios de alto perfil eran visibles cuando se conectaban a redes Wi-Fi públicas.

que pueden tener acceso físico a una computadora con una sesión abierta, minimizaremos el cuarto problema cambiando los tokens cada vez que el usuario cierra una sesión y teniendo cuidado de *firmar criptográficamente* cualquier información potencialmente sensible que guardemos en el navegador.

Con estas consideraciones de diseño y seguridad en mente, nuestro plan para crear sesiones persistentes aparece como sigue:

1. Crear una cadena de dígitos aleatoria para ser utilizada como un token recordado.
2. Guardar el token en las galletas del navegador con una fecha de expiración lejana en el futuro.
3. Guardar la digestión del token en la base de datos.
4. Guardar una versión encriptada del id del usuario en las galletas del navegador.
5. Cuando una galleta contenga un id de usuario, buscar el usuario en la base de datos usando el id dado, y verificar que el token recordado en la galleta coincide con la digestión almacenada en la base de datos.

Observe las similitudes del último paso con el inicio de sesión de usuario, cuando recuperamos el usuario dada su dirección electrónica y luego verificando (mediante el método **authenticate**) que la contraseña enviada coincide con la digestión de la contraseña ([Listado 8.5](#)). Como resultado, nuestra implementación tiene aspectos similares a los de **has_secure_password**.

Empezaremos agregando el atributo requerido **remember_digest** al modelo **User**, como se muestra en la [Figura 8.9](#).

Para agregar el modelo de datos de la [Figura 8.9](#) a nuestra aplicación, generaremos una migración:

```
$ rails generate migration add_remember_digest_to_users remember_digest:string
```

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

Figura 8.9: El modelo **User** con el nuevo atributo **remember_digest**.

(Compare con la migración de la digestión de la contraseña en la [Sección 6.3.1.](#)) Como en migraciones previas, hemos utilizado un nombre de migración que termine con **_to_users** para indicarle a Rails que la migración está diseñada para modificar la tabla **users** en la base de datos. Como también incluimos el atributo (**remember_digest**) y el tipo (**string**), Rails genera una migración por default para nosotros, como se muestra en el [Listado 8.30.](#)

Listado 8.30: La migración generada para la digestión recordada.

```
db/migrate/[timestamp]_add_remember_digest_to_users.rb
```

```
class AddRememberDigestToUsers < ActiveRecord::Migration
  def change
    add_column :users, :remember_digest, :string
  end
end
```

Como no esperamos recuperar usuarios por la digestión recordada, no hay necesidad de asociar un índice a la columna **remember_digest**, y podemos utilizar la migración por default como se generó anteriormente:

```
$ bundle exec rake db:migrate
```

Ahora tenemos que decidir qué utilizar como token recordado. Hay muchas posibilidades casi todas equivalentes—esencialmente, cualquier cadena aleatoria suficientemente larga funciona. El método `urlsafe_base64` del módulo `SecureRandom` en la biblioteca estándar de Ruby cubre nuestras necesidades.¹⁵ devuelve una cadena aleatoria de longitud 22 compuesta de los caracteres A–Z, a–z, 0–9, “-”, and “_” (para un total de 64 posibilidades, en consecuencia “base64”). Una cadena típica en base64 se muestra como sigue:

```
$ rails console
>> SecureRandom.urlsafe_base64
=> "q5lt38hQDc_959PVoo6b7A"
```

De igual forma que está bien si dos usuarios utilizan la misma contraseña,¹⁶ no hay necesidad de que los tokens recordados sean únicos, pero es más seguro si lo son.¹⁷ En el caso de la cadena en base64 anterior, cada uno de los 22 caracteres tiene 64 posibilidades, por lo que la probabilidad de que dos tokens recordados colisionen es sumamente pequeña $1/64^{22} = 2^{-132}$ aproximadamente 10^{-40} . Como beneficio adicional, al usar cadenas en base64 específicamente diseñadas para ser seguras en las URLs (como se indica en el nombre `urlsafe_base64`), podremos utilizar el mismo generador de tokens para los enlaces que permitirán activar la cuenta y reestablecer la contraseña en el [Capítulo 10](#).

Recordar usuarios involucra crear un token recordado y guardar la digestión del mismo en la base de datos. Ya hemos definido un método `digest` para ser utilizado en los *fixtures* de pruebas ([Listado 8.18](#)), y podemos utilizar el resultado de la discusión anterior para crear un método `new_token` que genere un token nuevo. Como con `digest`, el nuevo método no necesita un objeto

¹⁵ Esta elección está basada en el [RailsCast acerca de ‘recuérdame’](#).

¹⁶ En efecto, más vale que esté bien, porque con las `digestiones salpicadas` de bcrypt no hay forma de que sepamos si las contraseñas de dos usuarios coinciden.

¹⁷ Con tokens recordados únicos, un atacante siempre necesita tanto el id del usuario como la galleta con el token recordado para secuestrar la sesión.

usuario, por lo que lo crearemos como un método de clase.¹⁸ El resultado es el modelo **User** que se muestra en el Listado 8.31.

Listado 8.31: Agregando un método para generar tokens.

app/models/user.rb

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

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                 BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end
end
```

Nuestro plan para la implementación es crear un método **user.remember** que asocie un token recordado con el usuario y guarde la correspondiente digestión recordada en la base de datos. Debido a la migración del Listado 8.30, el modelo **User** ya cuenta con un atributo **remember_digest**, pero aún no tiene un atributo **remember_token**. Necesitamos una forma de hacer el token accesible vía **user.remember_token** (para guardarlo en las galletas) *sin* guardarla en la base de datos. Resolvimos un problema similar con las contraseñas seguras en la Sección 6.3, que hacía pareja con un atributo virtual **password** con un atributo seguro **password_digest** en la base de datos. En ese caso, el atributo virtual **password** fue creado automáticamente por

¹⁸Como regla general, si un método no necesita una instancia de un objeto, debería ser un método de clase. En efecto, esta decisión resultará importante en la Sección 10.1.2.

`has_secure_password`, pero tendremos que escribir el código para `remember_token` nosotros mismos. La forma de hacer esto es utilizando `attr_accessor` para crear un atributo accesible, que vimos anteriormente en la Sección 4.4.5:

```
class User < ActiveRecord::Base
  attr_accessor :remember_token
  .
  .
  .
  def remember
    self.remember_token = ...
    update_attribute(:remember_digest, ...)
  end
end
```

Observe la forma de la sentencia en la primera línea del método `remember`. Por la forma en que Ruby maneja las sentencias dentro de objetos, sin `self` la sentencia crearía una variable *local* llamada `remember_token`, que no es lo que queremos. Al utilizar `self` nos aseguramos que la sentencia asigna el atributo del usuario `remember_token`. (Ahora ya sabe porqué la invocación `before_save` del Listado 6.31 utiliza `self.email` en vez de sólo `email`.) Mientras tanto, la segunda línea de `remember` utiliza el método `update_attribute` para actualizar la digestión recordada. (Como observamos en la Sección 6.1.5, este método evita las validaciones, lo cual es necesario en este caso porque no tenemos acceso a la contraseña del usuario o a la confirmación.)

Con estas consideraciones en mente, podemos crear un token válido y una digestión asociada creando en primer lugar un nuevo token recordado utilizando `User.new_token`, y luego actualizando la digestión recordada con el resultado de aplicar `User.digest`. Este procedimiento nos lleva al método `remember` que se muestra en el Listado 8.32.

Listado 8.32: Agregando un método `remember` al modelo `User`. VERDE
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
```

```

validates :name, presence: true, length: { maximum: 50 }
VALID_EMAIL_REGEX = /\A[\w+\-\.]+@[a-z\d\-\.]+\.[a-z]+\z/i
validates :email, presence: true, length: { maximum: 255 },
  format: { with: VALID_EMAIL_REGEX },
  uniqueness: { case_sensitive: false }
has_secure_password
validates :password, presence: true, length: { minimum: 6 }

# Returns the hash digest of the given string.
def User.digest(string)
  cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                BCrypt::Engine.cost
  BCrypt::Password.create(string, cost: cost)
end

# Returns a random token.
def User.new_token
  SecureRandom.urlsafe_base64
end

# Remembers a user in the database for use in persistent sessions.
def remember
  self.remember_token = User.new_token
  update_attribute(:remember_digest, User.digest(remember_token))
end
end

```

8.4.2 Inicio de sesión con recuerdo

Habiendo creado un método funcional `user.remember`, ahora estamos en posición de crear una sesión persistente guardando el id del usuario (encriptado) y el token recordado como galletas permanentes en el navegador. La forma de hacer esto es con el método `cookies`, el cual (como con `session`) podemos tratar como una digestión. Una galleta consiste de dos piezas de información, un `valor` y una fecha de expiración opcional. Por ejemplo, podríamos crear una sesión persistente creando una galleta con un valor igual al token recordado que expira en unos 20 años a partir de ahora:

```

cookies[:remember_token] = { value:   remember_token,
                           expires: 20.years.from_now.utc }

```

(Esto utiliza uno de los auxiliares de tiempo de Rails, que vimos en el Re-

cuadro 8.2.) Este patrón de guardar una galleta que expira en 20 años es tan común que Rails tiene un método especial para implementarlo: **permanent**, de forma que podemos escribir simplemente

```
cookies.permanent[:remember_token] = remember_token
```

Esto hace que Rails establezca la fecha de expiración a **20.years.from_now** automáticamente.

Recuadro 8.2. Galletas que exiran en 20 años

Puede recordar de la [Sección 4.4.2](#) que Ruby permite agregar métodos a *cualquier* clase, aún a las preconstruídas. En esa sección, agregamos un método `palindrome?` a la clase `String` (y descubrimos como resultado que "reconocer" es un palíndromo), también vimos cómo Rails agrega un método `blank?` a la clase `Object` (de forma que `"".blank?`, `" ".blank?`, y `nil.blank?` son todos verdaderos). El método `cookies.permanent`, que crea galletas "permanentes" con una expiración a 20 años, nos proporciona otro ejemplo de esta práctica mediante uno de los *auxiliares de tiempo* de Rails, que son métodos agregados a `Fixnum` (la clase base para enteros):

```
$ rails console
>> 1.year.from_now
=> Sun, 09 Aug 2015 16:48:17 UTC +00:00
>> 10.weeks.ago
=> Sat, 31 May 2014 16:48:45 UTC +00:00
```

Rails agrega otros auxiliares también:

```
>> 1.kilobyte
=> 1024
>> 5.megabytes
=> 5242880
```

Éstos son útiles para validaciones cuando se suben archivos, haciendo fácil restringir, digamos, que las imágenes subidas no excedan de 5.megabytes.

Aunque debería utilizarse con precaución, la flexibilidad de agregar métodos a clases preconstruídas permite extender extraordinariamente Ruby de forma natural. De hecho, bastante de la elegancia de Rails deriva a últimas de la maleabilidad del lenguaje subyacente, Ruby.

Para almacenar el id del usuario en las galletas, podemos seguir el patrón utilizado con el método **session** (Listado 8.12) usando algo como

```
cookies[:user_id] = user.id
```

Como guarda el id como texto plano, este método expone la forma de las galletas de la aplicación y hace más fácil para un atacante comprometer las cuentas de usuario. Para evitar este problema, utilizaremos una galleta *firmada*, que encripta de forma segura la galleta antes de guardarla en el navegador:

```
cookies.signed[:user_id] = user.id
```

Como queremos que el id del usuario haga pareja con el token recordado permanente, deberíamos hacerlo permanente también, lo cual podemos hacer encadenando los métodos **signed** y **permanent**:

```
cookies.permanent.signed[:user_id] = user.id
```

Luego de que las galletas han sido guardadas, en páginas subsecuentes podemos recuperar el usuario con este código

```
User.find_by(id: cookies.signed[:user_id])
```

donde `cookies.signed[:user_id]` automáticamente descripta de la galleta, el id del usuario. Luego podemos utilizar bcrypt para verificar que `cookies[:remember_token]` coincide con el `remember_digest` generado en el [Listado 8.32](#). (En caso de que usted se esté preguntando porqué no sólo utilizamos el id firmado del usuario, sin el token recordado, es porque permitiría a un atacante que poseyera el id encriptado, iniciar sesión como el usuario por toda la eternidad. Con el diseño actual, un atacante con ambas galletas puede iniciar sesión como el usuario sólo hasta que el usuario cierre su sesión.)

La pieza final del rompecabezas es verificar que un token dado coincide con la digestión recordada del usuario, y en este contexto hay un par de formas equivalentes de utilizar bcrypt para verificar que coinciden. Si echa un vistazo al [código fuente de la contraseña segura](#), encontrará una comparación como esta:¹⁹

```
BCrypt::Password.new(password_digest) == unencrypted_password
```

En nuestro caso, el código análogo se observaría como esto:

```
BCrypt::Password.new(remember_digest) == remember_token
```

Si piensa en esto, este código es realmente extraño: parece estar comparando una digestión de una contraseña de bcrypt con un token, lo cual implicaría *desencriptar* la digestión con el fin de compararla usando `==`. Pero el punto de utilizar bcrypt es para que la digestión sea irreversible, por lo que esto no está bien. De hecho, revisando el [código fuente de la gema bcrypt](#) verificamos que el operador de comparación `==` está siendo *redefinido*, y en el fondo, la comparación anterior es equivalente a:

¹⁹Como observamos en la [Sección 6.3.1](#), “contraseña sin encriptar” es un mal nombre, puesto que la contraseña segura es *digerida*, no encriptada.

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

En vez de `==`, utiliza el método booleano `is_password?` para realizar la comparación. Como su significado es un poco más claro, preferiremos la segunda expresión en el código de la aplicación.

La discusión anterior sugiere poner la comparación del token con la digestión en un método `authenticated?` en el modelo `User`, que juega un rol similar al método `authenticate` proporcionado por `has_secure_password` para autenticar a un usuario (Listado 8.13). La implementación aparece en el Listado 8.33. (Aunque el método `authenticated?` del Listado 8.33 está amarrado específicamente a la digestión recordada, sería útil en otros contextos también, por lo que lo generalizaremos en el Capítulo 10.)

Listado 8.33: Agregando un método `authenticated?` al modelo `User`.

`app/models/user.rb`

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

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
      BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
  def remember
    self.remember_token = User.new_token
```

```

    update_attribute(:remember_digest, User.digest(remember_token))
end

# Returns true if the given token matches the digest.
def authenticated?(remember_token)
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
end

```

Observe que el argumento `remember_token` del método `authenticated?` definido en el Listado 8.33 no es el mismo que el descriptor de acceso que se definió en el Listado 8.32 usando `attr_accessor :remember_token`; en vez de esto, es una variable local del método. (Como el argumento se refiere al token recordado, no es raro utilizar un argumento en el método con el mismo nombre.) Observe también el uso del atributo `remember_digest`, que es el mismo que `self.remember_digest` y, de igual forma que `name` y `email` en el Capítulo 6, es creado automáticamente por *Active Record* basándose en el nombre de la columna de la base de datos correspondiente (Listado 8.30).

Ahora estamos en posición de recordar a un usuario que ha iniciado sesión, lo cual haremos agregando una función auxiliar `remember` que vaya junto con `log_in`, como se muestra en el Listado 8.34.

Listado 8.34: Iniciando sesión y recordando a un usuario.

`app/controllers/sessions_controller.rb`

```

class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      remember user
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end
end

```

```
def destroy
  log_out
  redirect_to root_url
end
end
```

Como con `log_in`, el Listado 8.34 delega el trabajo real al auxiliar `Sessions`, donde definimos un método `remember` que invoca `user.remember`, de este modo generamos un token recordado y guardamos su digestión en la base de datos. Luego utiliza `cookies` para crear galletas permanentes para el id del usuario y el token recordado como se describió anteriormente. El resultado aparece en el Listado 8.35.

Listado 8.35: Recordando al usuario.

app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the current logged-in user (if any).
  def current_user
    @current_user ||= User.find_by(id: session[:user_id])
  end

  # Returns true if the user is logged in, false otherwise.
  def logged_in?
    !current_user.nil?
  end

  # Logs out the current user.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

Con el código del Listado 8.35, un usuario que ha iniciado sesión será recordado en el sentido de que su navegador recordará un token válido, pero aún no es útil para nosotros porque el método `current_user` que definimos en el Listado 8.14 sólo conoce de la sesión temporal:

```
@current_user ||= User.find_by(id: session[:user_id])
```

En el caso de sesiones persistentes, queremos recuperar el usuario de la sesión temporal si `session[:user_id]` existe, pero de otra forma, debemos buscar `cookies[:user_id]` para recuperar (e iniciar sesión) el usuario correspondiente a la sesión persistente. Podemos lograr esto como sigue:

```
if session[:user_id]
  @current_user ||= User.find_by(id: session[:user_id])
elsif cookies.signed[:user_id]
  user = User.find_by(id: cookies.signed[:user_id])
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

(Esto sigue el mismo patrón `user && user.authenticated?` que vimos en el Listado 8.5.) El código anterior funcionará, pero observe el uso repetido tanto de `session` como de `cookies`. Podemos eliminar esta duplicidad como sigue:

```
if (user_id = session[:user_id])
  @current_user ||= User.find_by(id: user_id)
elsif (user_id = cookies.signed[:user_id])
  user = User.find_by(id: user_id)
  if user && user.authenticated?(cookies[:remember_token])
    log_in user
    @current_user = user
  end
end
```

Esto utiliza la común pero potencialmente confusa construcción

```
if (user_id = session[:user_id])
```

A pesar de las apariencias, esto *no* es una comparación (la cual usaría doble signo de igual `==`), más bien es una *asignación*. Si fuera a leerlo con palabras, diríamos “Si el id del usuario es igual al id del usuario en la sesión...”, pero es más bien algo como “Si el id del usuario existe en la sesión (mientras que damos valor al id del user con el id del usuario en la sesión)...”.²⁰

Definiendo el auxiliar `current_user` como comentamos anteriormente nos lleva a la implementación mostrada en el [Listado 8.36](#).

Listado 8.36: Actualizando `current_user` para sesiones persistentes. **ROJO**
`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end

  # Returns true if the user is logged in, false otherwise.
```

²⁰Generalmente uso la convención de poner estas asignaciones entre paréntesis, lo cual es un recordatorio visual de que no es una comparación.

Name	remember_token
Value	vb4IQ7Oy3dCLv2R2TEdQ0g
Host	rails-tutorial-c9-mhartl.c9.io
Path	/
Expires	Sun, 30 Jul 2034 00:18:56 GMT
Secure	No
HttpOnly	No

Figura 8.10: La galleta del token recordado en el navegador local.

```

def logged_in?
  !current_user.nil?
end

# Logs out the current user.
def log_out
  session.delete(:user_id)
  @current_user = nil
end
end

```

Con el código del [Listado 8.36](#), usuarios que recién crearon su sesión son recordados correctamente, lo cual puede verificar iniciando una sesión, cerrando el navegador y verificando que aún está en sesión cuando reinicie la aplicación de ejemplo y vuelva a visitar la página de la aplicación. Si lo desea, puede inspeccionar las galletas del navegador para ver el resultado directamente ([Figura 8.10](#)).²¹

Sólo hay un problema con nuestra aplicación como está: tenemos que limpiar las galletas de nuestro navegador (o esperar 20 años), para que los usuarios cier-

²¹Busque en Google “<el nombre de su navegador> inspeccionar galletas” para saber más de cómo inspeccionar las galletas en su sistema.

ren sesión, pues no hay otra forma de hacerlo. Esta es exactamente la clase de cosas que nuestro conjunto de pruebas debería verificar, y de hecho lo hace puesto que actualmente está en ROJO:

Listado 8.37: ROJO

```
$ bundle exec rake test
```

8.4.3 Olvidando usuarios

Para permitir a los usuarios cerrar su sesión, definiremos métodos que olviden usuarios en analogía con los que los recuerdan. El método `user.forget` resultante sólo deshace `user.remember` al actualizar la digestión recordada con `nil`, como se muestra en el Listado 8.38.

Listado 8.38: Agregando un método `forget` al modelo `User`.

`app/models/user.rb`

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

  # Returns the hash digest of the given string.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
      BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Remembers a user in the database for use in persistent sessions.
```

```
def remember
  self.remember_token = User.new_token
  update_attribute(:remember_digest, User.digest(remember_token))
end

# Returns true if the given token matches the digest.
def authenticated?(remember_token)
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end

# Forgets a user.
def forget
  update_attribute(:remember_digest, nil)
end
end
```

Con el código del Listado 8.38, estamos listos para olvidar una sesión permanente al agregar un auxiliar **forget** e invocándolo desde el auxiliar **log_out** (Listado 8.39). Como vimos en el Listado 8.39, el auxiliar **forget** invoca **user.forget** y luego borra las galletas del **user_id** y **remember_token**.

Listado 8.39: Cerrando una sesión persistente.

app/helpers/sessions_helper.rb

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end
  .

  .

  # Forgets a persistent session.
  def forget(user)
    user.forget
    cookies.delete(:user_id)
    cookies.delete(:remember_token)
  end

  # Logs out the current user.
  def log_out
    forget(current_user)
    session.delete(:user_id)
    @current_user = nil
  end
end
```

8.4.4 Dos defectos sutiles

Hay dos sutilezas relacionadas entre sí que aún están pendientes. La primer sutileza es que, aún el enlace “Cerrar sesión” aparece únicamente cuando se ha iniciado sesión, un usuario podría potencialmente tener múltiples ventanas de navegador abiertas en el sitio. Si el usuario saliera de sesión en una de ellas, y por tanto estableciendo el `current_user` en `nil`, al dar click en el enlace “Cerrar sesión” en otra ventana, provocaría un error debido al `forget(current_user)` del método `log_out` (Listado 8.39).²² Podemos evitar esto al cerrar sesión sólo si el usuario ha iniciado una.

La segunda sutileza es que un usuario podría estar en sesión (recordada) en múltiples navegadores, tales como Chrome o Firefox, lo que provoca un problema si el usuario cierra su sesión en el primer navegador pero no en el segundo, y luego cierra y reabre la segunda.²³ Por ejemplo, suponga que el usuario cierra su sesión en Firefox, por lo tanto establece su digestión recordada a `nil` (via `user.forget` en el Listado 8.38). La aplicación seguirá funcionando en Firefox; porque el método `log_out` del Listado 8.39 borra el id del usuario, los dos condicionales resaltados son **falsos**:

```
# Returns the user corresponding to the remember token cookie.
def current_user
  if (user_id = session[:user_id])
    @current_user ||= User.find_by(id: user_id)
  elsif (user_id = cookies.signed[:user_id])
    user = User.find_by(id: user_id)
    if user && user.authenticated?(cookies[:remember_token])
      log_in user
      @current_user = user
    end
  end
end
```

Como resultado, la evaluación desciende al final del método `current_user`, por lo tanto regresa `nil` como es requerido.

Por el contrario, si cerramos Chrome, establecemos el `session[:user_id]` a `nil` (porque todas las variables `session` expiran automáticamente al cerrar

²²Agradezco al lector Paulo Célio Júnior por resaltar este punto.

²³Agradezco al lector Niels de Ron por señalar esto.

el navegador), pero la galleta `user_id` seguirá estando presente. Esto significa que el usuario correspondiente aún será extraído de la base de datos:

```
# Returns the user corresponding to the remember token cookie.
def current_user
  if (user_id = session[:user_id])
    @current_user ||= User.find_by(id: user_id)
  elsif (user_id = cookies.signed[:user_id])
    user = User.find_by(id: user_id)
    if user && user.authenticated?(cookies[:remember_token])
      log_in user
      @current_user = user
    end
  end
end
```

En consecuencia, la condición `if` interna será evaluada:

```
user && user.authenticated?(cookies[:remember_token])
```

En particular, porque `user` no es `nil`, la *segunda* expresión será evaluada, lo cual arroja un error. Esto es porque la digestión recordada del usuario fue borrada como parte del cierre de la sesión ([Listado 8.38](#)) en Firefox, por lo que cuando accedemos la aplicación en Chrome terminamos invocando

```
BCrypt::Password.new(remember_digest).is_password?(remember_token)
```

con una digestión recordada `nil`, lanzando de esta forma una excepción dentro de la biblioteca bcrypt. Para arreglar esto, queremos que el método `authenticated?` regrese `falso`.

Estas son exactamente la clase de sutilezas que se benefician del desarrollo orientado a pruebas, por lo que escribiremos pruebas para reproducir los dos errores antes de corregirlos. Primero tenemos la prueba de integración del [Listado 8.28](#) en ROJO, como se muestra en el [Listado 8.40](#).

Listado 8.40: Una prueba para el cierre de sesión del usuario. ROJO

```
test/integration/users_login_test.rb
```

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest
  .
  .
  .
  test "login with valid information followed by logout" do
    get login_path
    post login_path, session: { email: @user.email, password: 'password' }
    assert is_logged_in?
    assert_redirected_to @user
    follow_redirect!
    assert_template 'users/show'
    assert_select "a[href=?]", login_path, count: 0
    assert_select "a[href=?]", logout_path
    assert_select "a[href=?]", user_path(@user)
    delete logout_path
    assert_not is_logged_in?
    assert_redirected_to root_url
    # Simulate a user clicking logout in a second window.
    delete logout_path
    follow_redirect!
    assert_select "a[href=?]", login_path
    assert_select "a[href=?]", logout_path,      count: 0
    assert_select "a[href=?]", user_path(@user), count: 0
  end
end
```

La segunda llamada a `delete logout_path` en el Listado 8.40 debería arrojar un error debido a que falta `current_user`, lo que nos lleva al ROJO en el conjunto de pruebas:

Listado 8.41: ROJO

```
$ bundle exec rake test
```

El código de la aplicación simplemente involucra el llamado a `log_out` sólo si `logged_in?` es verdadero, como se muestra en el Listado 8.42.

Listado 8.42: Sólo cerramos sesión si estamos en una. VERDE

app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
  .
  .
  .
  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

El segundo caso, involucra un escenario con dos navegadores diferentes, lo cual es más complicado de simular con una prueba de integración, pero es fácil verificar las pruebas sobre el modelo **User** directamente. Todo lo que necesitamos es iniciar con un usuario que no tiene una digestión recordada (lo cual es cierto para la variable **@user** definida en el método **setup**) y luego invocar **authenticated?**, como se muestra en el Listado 8.43. (Observe que hemos dejado a un lado el token recordado; no importa qué valor tiene, porque el error ocurre antes de que éste sea utilizado.)

Listado 8.43: Una prueba de **authenticated?** con una digestión no existente. ROJO

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "authenticated? should return false for a user with nil digest" do
    assert_not @user.authenticated?(' ')
  end
end
```

Como `BCrypt::Password.new(nil)` arroja error, el conjunto de pruebas debería estar en ROJO:

Listado 8.44: ROJO

```
$ bundle exec rake test
```

Para arreglar el error y hacer que esté en VERDE, todo lo que necesitamos hacer es regresar `falso` si la digestión recordada es `nil`, como se muestra en el Listado 8.45.

Listado 8.45: Actualizando `authenticated?` para que maneje una digestión inexistente. VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns true if the given token matches the digest.
  def authenticated?(remember_token)
    return false if remember_digest.nil?
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end
end
```

Esto utiliza la palabra reservada `return` para terminar inmediatamente si la digestión recordada es `nil`, lo cual es una forma común de enfatizar que el resto del método es ignorado en ese caso. El código equivalente

```
if remember_digest.nil?
  false
else
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

también funcionaría correctamente, pero prefiero la claridad de la versión del Listado 8.45 (que además sucede que es ligeramente más corta).

Con el código del Listado 8.45, nuestro conjunto de pruebas debería estar en **VERDE**, y ambas sutilezas deberían ser reproducidas:

Listado 8.46: **VERDE**

```
$ bundle exec rake test
```

8.4.5 Casilla “Recuérdame”

Con el código de la Sección 8.4.3, nuestra aplicación tiene un sistema de autenticación completo, de grado profesional. Como último paso, veremos cómo hacer que el usuario pueda optar por no cerrar su sesión usando una casilla “recuérdame”. Un bosquejo del formulario de inicio de sesión con tal casilla se muestra en la Figura 8.11.

Para escribir la implementación, empezaremos por agregar la casilla al formulario de inicio de sesión del Listado 8.2. Como con las etiquetas, campos de texto, de contraseña y botones de envío, las casillas pueden ser creadas con un método auxiliar de Rails. Aunque, con la finalidad de que sean estilizadas correctamente, debemos de *anidar* la casilla dentro de la etiqueta, como sigue:

```
<%= f.label :remember_me, class: "checkbox inline" do %>
<%= f.check_box :remember_me %>
<span>Remember me on this computer</span>
<% end %>
```

Colocando esto en el formulario de inicio de sesión, obtenemos el código que se muestra en el Listado 8.47.

Listado 8.47: Agregando una casilla “recuérdame” al formulario de inicio de sesión.

app/views/sessions/new.html.erb

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



Figura 8.11: Un bosquejo de una casilla “recuérdame”.

```

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session, url: login_path) do |f| %>

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

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

      <%= f.label :remember_me, class: "checkbox inline" do %>
        <%= f.check_box :remember_me %>
        <span>Remember me on this computer</span>
      <% end %>

      <%= f.submit "Log in", class: "btn btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>

```

En el Listado 8.47, hemos incluído las clases CSS `checkbox` e `inline`, las cuales son utilizadas por Bootstrap para poner la casilla y el texto (“Remember me on this computer”) en la misma línea. Para completar el estilo, necesitamos unas cuantas reglas CSS más, como podemos ver en el Listado 8.48. El formulario de inicio de sesión resultante se muestra en la Figura 8.12.

Listado 8.48: El CSS para la casilla “recuérdame”.

app/assets/stylesheets/custom.css.scss

```

.
.
.
/*
 forms */
.

.

.checkbox {
  margin-top: -10px;
  margin-bottom: 10px;
  span {
    margin-left: 20px;
    font-weight: normal;
  }
}

```

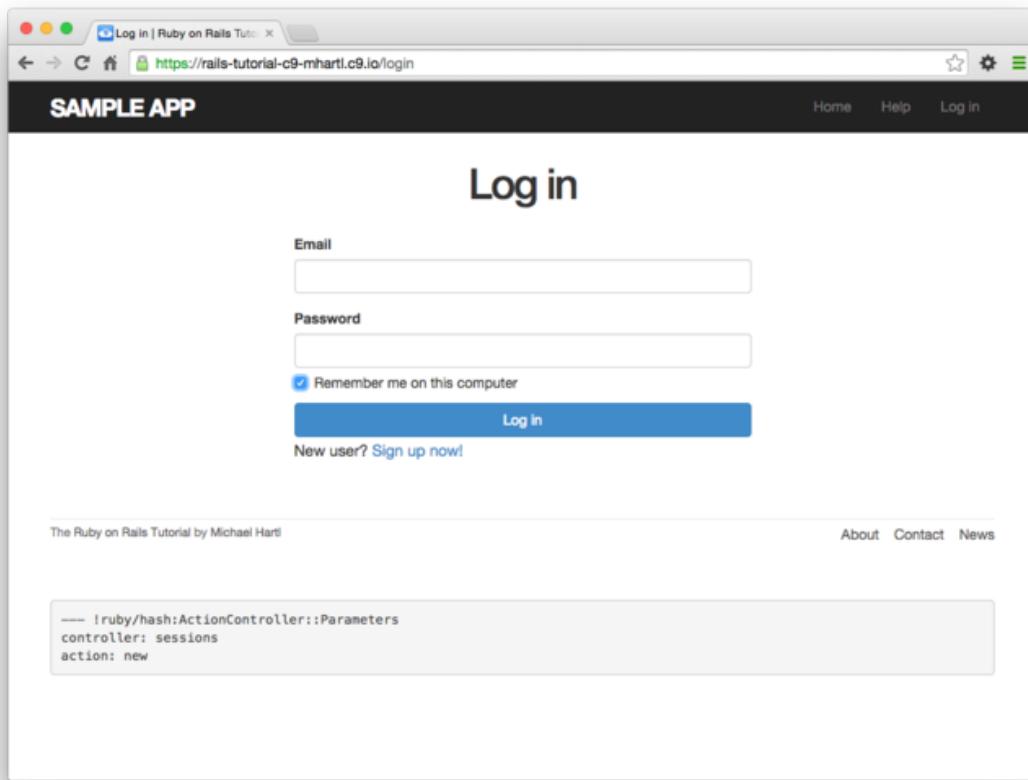


Figura 8.12: El formulario de inicio de sesión con una casilla “recuérdame” agregada.

```
#session_remember_me {
  width: auto;
  margin-left: 0;
}
```

Habiendo editado el formulario, estamos listos para recordar usuarios si marcan la casilla y para olvidarlos en caso contrario. Increíblemente, gracias a todo nuestro trabajo de las secciones previas, la implementación puede reducirse a una sola línea. Empecemos por observar que el arreglo hash **params** del formulario enviado incluye un valor basado en la casilla (como puede ve-

rificar al enviar datos desde el formulario del Listado 8.47 con información inválida e inspeccionando los valores en la sección de depuración de la página). En particular, el valor de

```
params[:session][:remember_me]
```

es '**1**' si la casilla está marcada y '**0**' si no lo está.

Al probar el valor relevante del arreglo hash **params**, podemos recordar u olvidar al usuario basados en el valor enviado:²⁴

```
if params[:session][:remember_me] == '1'
  remember(user)
else
  forget(user)
end
```

Como se explica en el Recuadro 8.3, eta clase de estructura **if-then** puede escribirse en una sola línea utilizando el *operador ternario* como sigue:²⁵

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

Agregar esto al método **create** del controlador de sesiones, produce un código sorprendentemente compacto que se muestra en el Listado 8.49. (Ahora está en posición de entender el código del Listado 8.18, whel cual utiliza el operador ternario para definir la variable bcrypt **cost**.)

Listado 8.49: Manejando el envío de la casilla “recuérdame”.

app/controllers/sessions_controller.rb

²⁴Observe que esto significa que desmarcar la casilla cerrará la sesión del usuario en todos los navegadores en todas las computadoras. El diseño alternativo de recordar sesiones de usuario en cada navegador de forma independiente es potencialmente más conveniente para los usuarios, pero es menos segura y también más complicada para implementar. Los lectores ambiciosos están invitados a intentar por su cuenta esta implementación.

²⁵Antes escribimos **remember user** sin paréntesis, pero cuando usamos el operador ternario, omitirlos produce un error de sintaxis.

```

class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      params[:session][:remember_me] == '1' ? remember(user) : forget(user)
      redirect_to user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end

```

Con la implementación del [Listado 8.49](#), nuestro sistema de manejo de sesión está completo, como puede verificar marcando y desmarcando la casilla en su navegador.

Recuadro 8.3. 10 tipos de personas

Hay una vieja broma que dice que existen 10 tipos de personas en el mundo: los que entienden binario y los que no (10, por supuesto, considerado 2 en binario). Siguendo la analogía, podemos decir que existen 10 tipos de personas en el mundo: los que les gusta el operador ternario, los que no, y los que no saben de su existencia todavía. (Si usted está en la tercera categoría, pronto dejara de estarlo.)

Cuando usted programa bastante, pronto aprende que uno de los flujos de control más comunes se parecen a éste:

```

if boolean?
  do_one_thing
else

```

```
do_something_else  
end
```

Ruby, de igual forma que muchos otros lenguajes (incluyendo C/C++, Perl, PHP, y Java), le permite reemplazar esto con una expresión mucho más compacta usando el *operador ternario* (se llama así porque consiste de tres partes):

```
boolean? ? do_one_thing : do_something_else
```

También puede utilizar el operador ternario para reemplazar una asignación como ésta

```
if boolean?  
    var = foo  
else  
    var = bar  
end
```

se convierte en

```
var = boolean? ? foo : bar
```

Finalmente, a menudo es conveniente utilizar el operador ternario al devolver el resultado de una función:

```
def foo  
    do_stuff  
    boolean? ? "bar" : "baz"  
end
```

Puesto que Ruby implícitamente regresa el valor de la última expresión de una función, el método `foo` regresa "bar" o "baz" dependiendo de si `boolean?` es verdadero o falso.

8.4.6 Pruebas para Recuérdame

Aunque nuestra funcionalidad “recuérdame” está funcionando, es importante escribir algunas pruebas para verificar su comportamiento. Una de los motivos es para atrapar errores de implementación, como comentamos en su momento. Aunque aún más importante, es que la parte principal del código de persistencia de usuario carece de toda prueba en este momento. Arreglar estos detalles requerirá algunos trucos, pero el resultado será un conjunto de pruebas mucho más poderoso.

Probando la casilla “recuérdame”

Cuando originalmente implementé el manejo de la casilla en el [Listado 8.49](#), en vez del correcto

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

realmente utilicé

```
params[:session][:remember_me] ? remember(user) : forget(user)
```

En este contexto, `params[:session][:remember_me]` es '`0`' o '`1`', cualquiera de los dos son **verdadero** en un contexto booleano, por lo que la expresión resultante es *siempre verdadera*, y la aplicación actúa como si la casilla estuviera siempre marcada. Esta es exactamente la clase de error que una prueba puede detectar.

Como recordar usuarios requiere que estén en sesión, nuestro primer paso es definir un auxiliar para iniciar sesión de usuario dentro de las pruebas. En el [Listado 8.20](#), iniciamos la sesión de un usuario utilizando el método **post** y un arreglo hash válido **session**, pero es poco práctico realizar esto cada vez. Para evitar repetición innecesaria, escribiremos un método auxiliar llamado **log_in_as** que inicie sesión por nosotros.

Nuestro método para iniciar la sesión de un usuario depende del tipo de prueba. Dentro de las pruebas de integración, podemos enviar datos a la ruta de

sesiones como en el Listado 8.20, pero en otras pruebas (tales como las pruebas del controlador y del modelo) esto no funcionaría, y necesitamos manipular el método `session` directamente. Como resultado, `log_in_as` debería detectar la clase de prueba que estamos utilizando y ajustarse en consecuencia. Podemos determinar la diferencia entre las pruebas de integración y otras clases de pruebas, usando el método `defined?` de Ruby, que regresa verdadero si su argumento está definido y falso en cualquier otro caso. En este caso, el método `post_via_redirect` (que vimos anteriormente en el Listado 7.26) está disponible sólo en las pruebas de integración, por lo que el código

```
defined?(post_via_redirect) ...
```

regresará `verdadero` dentro de una prueba de integración y falso en cualquier otro caso. Esto sugiere definir un método booleano `integration_test?` y escribir una sentencia if-then como sugiere este esquema:

```
if integration_test?
  # Log in by posting to the sessions path
else
  # Log in using the session
end
```

Rellenando los comentarios con código obtenemos el método auxiliar `log_in_as` que se muestra en el Listado 8.50. (Este es un método bastante avanzado, por lo que usted va por buen camino si puede comprenderlo por completo.)

Listado 8.50: Agregando un auxiliar `log_in_as`.

`test/test_helper.rb`

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

  # Returns true if a test user is logged in.
```

```

def is_logged_in?
  !session[:user_id].nil?
end

# Logs in a test user.
def log_in_as(user, options = {})
  password = options[:password] || 'password'
  remember_me = options[:remember_me] || '1'
  if integration_test?
    post login_path, session: { email: user.email,
                                 password: password,
                                 remember_me: remember_me }
  else
    session[:user_id] = user.id
  end
end

private

# Returns true inside an integration test.
def integration_test?
  defined?(post_via_redirect)
end

```

Observe que, para mayor flexibilidad, el método `log_in_as` del Listado 8.50 acepta un arreglo hash `options` (como en el Listado 7.31), con opciones por default para la contraseña y la casilla “recuérdame” con los valores `'password'` y `'1'`, respectivamente. En particular, como el hash regresa `nil` para llaves no-existentes, el código

```
remember_me = options[:remember_me] || '1'
```

evalúa la opción dada si está presente y en otro caso, regresa el valor por default (una aplicación de la evaluación del corto circuito descrito en el Recuadro 8.1).

Para verificar el comportamiento de la casilla “recuérdame”, escribiremos dos pruebas, una para enviar la casilla marcada y otra para enviar la casilla sin marcar. Esto es fácil utilizando el auxiliar de inicio de sesión definido en el Listado 8.50, con los dos casos como siguen

```
log_in_as(@user, remember_me: '1')
```

y

```
log_in_as(@user, remember_me: '0')
```

(Como '1' es el valor por default para `remember_me`, podemos omitir la opción correspondiente en el primer caso, pero debemos incluirlo para que su estructura sea claramente similar.)

Luego de haber iniciado sesión, podemos verificar si el usuario ha sido recordado revisando la llave `remember_token` en las galletas. Idealmente, verificaremos que el valor de la galleta es igual al token recordado del usuario, pero a como está diseñado en este momento, no hay forma de que la prueba tenga acceso a éste: la variable `user` en el controlador tiene un atributo de token recordado (porque `remember_token` es virtual), pero la variable `@user` en la prueba no lo tiene. Arreglar esta pequeño defecto se deja como ejercicio ([Sección 8.6](#)), pero por ahora sólo probaremos si la galleta relevante es `nil` o no.

Hay una sutileza adicional, que es que por alguna razón, dentro de las pruebas, el método `cookies` no funciona con símbolos como llaves, por lo que

```
cookies[:remember_token]
```

es siempre `nil`. Afortunadamente, `cookies` *funciona* con llaves que son cadenas, por lo que

```
cookies['remember_token']
```

tiene el valor que necesitamos. Las pruebas resultantes aparecen en el [Listado 8.51](#). (Recuerde del [Listado 8.20](#) que `users(:michael)` hace referencia al *fixture* del usuario del [Listado 8.19](#).)

Listado 8.51: Una prueba de la casilla “recuérdame”. VERDE

```
test/integration/users_login_test.rb
```

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  .
  .
  .

  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
    assert_not_nil cookies['remember_token']
  end

  test "login without remembering" do
    log_in_as(@user, remember_me: '0')
    assert_nil cookies['remember_token']
  end
end
```

Suponiendo que usted no cometió el mismo error en la implementación que yo, las pruebas deberían estar en VERDE:

Listado 8.52: VERDE

```
$ bundle exec rake test
```

Probando la rama recuérdame

En la Sección 8.4.2, verificamos manualmente que la sesión persistente implementada en secciones anteriores está funcionando, pero de hecho la rama relevante en el método `current_user` está en este momento sin probar. Mi forma favorita para manejar este tipo de situación es arrojar una excepción en el bloque de código que se sospecha no ha sido probado: si el código no está cubierto, las pruebas pasarán; si está cubierto, el error resultante identificará la prueba relevante. El resultado en este caso aparece en el Listado 8.53.

Listado 8.53: Lanzando una excepción en una rama no probada. VERDE
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      raise "The tests still pass, so this branch is currently untested." # The tests still pass, so this branch is currently untested.
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

En este momento, las pruebas están en VERDE:

Listado 8.54: VERDE

```
$ bundle exec rake test
```

Esto es un problema, por supuesto, porque el código del [Listado 8.53](#) no funciona. Más aún, las sesiones persistentes son incómodas de probar manualmente, por lo que si en algún momento deseamos refactorizar el método **current_user** (como haremos en el [Capítulo 10](#)) es importante probarlo.

Como el método auxiliar **log_in_as** definido en el [Listado 8.50](#) automáticamente establece **session[:user_id]**, probar la rama “recuérdame” del método **current_user** es difícil en una prueba de integración. Por suerte, podemos evitar esta restricción probando el método **current_user** directamente en una prueba del auxiliar de sesiones, cuyo archivo debemos crear:

```
$ touch test/helpers/sessions_helper_test.rb
```

La secuencia de la prueba es simple:

1. Definir una variable **user** usando los *fixtures*.
2. Invocar el método **remember** para recordar el usuario dado.
3. Verificar que **current_user** es igual al usuario dado.

Como el método **remember** no establece **session[:user_id]**, este procedimiento probará la rama “remember” deseada. El resultado aparece en el [Listado 8.55](#).

Listado 8.55: Una prueba para las sesiones persistentes.

```
test/helpers/sessions_helper_test.rb

require 'test_helper'

class SessionsHelperTest < ActionView::TestCase

  def setup
    @user = users(:michael)
    remember(@user)
  end

  test "current_user returns right user when session is nil" do
    assert_equal @user, current_user
    assert is_logged_in?
  end

  test "current_user returns nil when remember digest is wrong" do
    @user.update_attribute(:remember_digest, User.digest(User.new_token))
    assert_nil current_user
  end
end
```

Observe que hemos agregado una segunda prueba, que verifica que el usuario actual es **nil** si la digestión recordada del usuario no corresponde correctamente al token recordado, por lo tanto probando la expresión **authenticated?** en la sentencia **if** anidada:

```
if user && user.authenticated?(cookies[:remember_token])
```

De paso, en el [Listado 8.55](#) podemos escribir

```
assert_equal current_user, @user
```

y funcionaría igual, pero (como mencionamos brevemente en la [Sección 5.6](#)) el orden convencional para los argumentos de **assert_equal** es *expected, actual*:

```
assert_equal <expected>, <actual>
```

el cual en el caso del [Listado 8.55](#) nos da

```
assert_equal @user, current_user
```

Con el código del [Listado 8.55](#), la prueba está en **ROJO** como es requerido:

Listado 8.56: ROJO

```
$ bundle exec rake test TEST=test/helpers/sessions_helper_test.rb
```

Podemos hacer que las pruebas del [Listado 8.55](#) pasen removiendo el **raise** y restaurando el método original **current_user**, como se muestra en el [Listado 8.57](#). (También puede verificar removiendo la expresión **authenticated?** en el [Listado 8.57](#) que la segunda prueba del [Listado 8.55](#) falla, lo que confirma que prueba lo correcto.)

Listado 8.57: Removiendo la excepción arrojada. VERDE app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Returns the user corresponding to the remember token cookie.
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

En este momento, el conjunto de pruebas debería estar en **VERDE**:

Listado 8.58: VERDE

```
$ bundle exec rake test
```

Ahora que la rama “remember” de **current_user** es probada, podemos tener confianza de atrapar regresiones sin necesidad de verificar manualmente.

8.5 Conclusión

Hemos cubierto mucho material en los últimos dos capítulos, transformando nuestra promisoria aplicación que no tenía forma, en un sitio con todas las capacidades de registro e inicio de sesión. Todo lo que se necesita para completar la funcionalidad de autenticación es restringir el acceso a las páginas basados en el status de inicio de sesión y en la identidad del usuario. Realizaremos esta tarea cuando les demos a los usuarios la posibilidad de editar su información, lo cual es el objetivo principal del Capítulo 9.

Antes de continuar, mezcle sus cambios con la rama principal:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Finish log in/log out"
$ git checkout master
$ git merge log-in-log-out
```

Luego súbalos al repositorio remoto y al servidor de producción:

```
$ bundle exec rake test
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

Observe que la aplicación estará brevemente en un estado inválido luego de subir los cambios y hasta antes de que termine la migración. En un sitio de producción con tráfico significativo, es buena idea cambiar a *modo de mantenimiento* antes de realizar los cambios:

```
$ heroku maintenance:on
$ git push heroku
$ heroku run rake db:migrate
$ heroku maintenance:off
```

Esto se encarga de mostrar una página estándar de error durante el despliegue y la migración. (No nos ocuparemos de este paso nuevamente, pero es bueno verlo al menos una vez.) Para mayor información, vea la documentación de Heroku en las [páginas de error](#).

8.5.1 Qué aprendimos en este capítulo

- Rails puede mantener el estado de una página a la siguiente usando galletas tanto temporales como persistentes.
- El formulario de inicio de sesión está diseñado para permitir al usuario el ingreso al sitio.

- El método `flash.now` es utilizado por los mensajes flash en páginas desplegadas.
- El desarrollo orientado a pruebas es útil cuando depuramos reproduciendo el defecto en una prueba.
- Usando el método `session`, podemos guardar de forma segura el id del usuario en el navegador para crear una sesión temporal.
- Podemos cambiar características tales como enlaces en la estructura de diseño basados en el status de la sesión.
- Las pruebas de integración pueden verificar correctamente las rutas, actualizaciones a la base de datos y cambios propios de la estructura de diseño.
- Asociamos a cada usuario un token recordado su respectiva una digestión recordada para utilizarlos en las sesiones persistentes.
- Usando el método `cookies`, creamos una sesión persistente al guardar un token recordado en una galleta permanente en el navegador.
- El status de sesión está determinado por la presencia del usuario actual basados en el id del usuario de la sesión temporal o en el token recordado de la sesión permanente.
- La aplicación cierra la sesión de los usuarios borrando el id del usuario de la sesión y removiendo la galleta permanente del navegador.
- El operador ternario es una forma compacta de escribir sentencias if-then simples.

8.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. En el Listado 8.32, definimos los métodos de clase para el nuevo token y la digestión al ponerles en el nombre, el prefijo `User`. Esto funciona bien y como realmente son *invocados* usando `User.new_token` y `User.digest`, es probablemente la forma más explícita de definirlos. Pero hay un par de formas idiomáticamente más correctas para definir métodos de clase, una ligeramente confusa y la otra extremadamente confusa. Al ejecutar el conjunto de pruebas, verifique que las implementaciones del Listado 8.59 (ligeramente confuso) y del Listado 8.60 (extremadamente confuso) son correctas. (Observe que, en el contexto de los Listados 8.59 y 8.60, `self` es la clase `User`, mientras que los otros usos de `self` en el modelo `User` se refieren a una *instancia* de un objeto user. Esto es parte de lo que los hace confusos.)
2. Como se mencionó en la Sección 8.4.6, a como está diseñada la aplicación en este momento, no hay forma de tener acceso al atributo virtual `remember_token` en la prueba de integración del Listado 8.51. Aunque es posible, usar un método de prueba especial llamado `assigns`. Dentro de una prueba, usted puede accesar variables de *instancia* definidas en el controlador utilizando `assigns` con el símbolo correspondiente. Por ejemplo, si la acción `create` define una variable `@user`, podemos accesarla en una prueba utilizando `assigns(:user)`. En este momento, la acción `create` del controlador `Sessions` define una variable normal (no de instancia) llamada `user`, pero si la cambiamos a una variable de instancia podemos probar que las galletas contienen correctamente el token recordado del usuario. Rellenando los elementos faltantes de los Listados 8.61 y 8.62 (indicados con signos de interrogación `?` y `FILL_IN`), complete esta versión mejorada de la prueba de la casilla “recuérdame”.

Listado 8.59: Definiendo los métodos para el nuevo token y digestión utilizando **self**. VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns the hash digest of the given string.
  def self.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                  BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Returns a random token.
  def self.new_token
    SecureRandom.urlsafe_base64
  end
  .
  .
  .
end
```

Listado 8.60: Definiendo los métodos para el nuevo token y digestión utilizando **class << self**. VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  class << self
    # Returns the hash digest of the given string.
    def digest(string)
      cost = ActiveModel::SecurePassword.min_cost ? BCrypt::Engine::MIN_COST :
                                                    BCrypt::Engine.cost
      BCrypt::Password.create(string, cost: cost)
    end

    # Returns a random token.
    def new_token
      SecureRandom.urlsafe_base64
    end
  end
  .
end
```

```
•  
•
```

Listado 8.61: Una plantilla para utilizar una variable de instancia en la acción **create**.

```
app/controllers/sessions_controller.rb

class SessionsController < ApplicationController

  def new
  end

  def create
    ?user = User.find_by(email: params[:session][:email].downcase)
    if ?user && ?user.authenticate(params[:session][:password])
      log_in ?user
      params[:session][:remember_me] == '1' ? remember(?user) : forget(?user)
      redirect_to ?user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

Listado 8.62: Una plantilla para una prueba mejorada de “recuérdame”. **VERDE**

```
test/integration/users_login_test.rb

require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  .
  .
  .

  test "login with remembering" do
    log_in_as(@user, remember_me: '1')
```

```
assert_equal FILL_IN, assigns(:user).FILL_IN
end

test "login without remembering" do
  log_in_as(@user, remember_me: '0')
  assert_nil cookies['remember_token']
end
.
.
.
end
```

Capítulo 9

Actualizando, mostrando y borrando usuarios

En este capítulo, completaremos las acciones REST para el recurso Users (Tabla 7.1) agregando las acciones `edit`, `update`, `index`, y `destroy`. Empezaremos permitiendo a los usuarios que actualicen sus perfiles, lo cual nos proporcionará de forma natural, una oportunidad para aplicar un modelo de autorización (lo cual es posible gracias al código de autenticación del Capítulo 8). Luego elaboraremos un listado de todos los usuarios (requiriendo autenticación también), lo cual motivará la introducción de datos de muestra y paginación. Finalmente, añadiremos la funcionalidad para destruir usuarios, limpiándolos de la base de datos. Puesto que por seguridad no podemos permitir que cualquier usuario tenga tales poderes, tendremos cuidado de crear una clase privilegiada de usuarios administrativos autorizados para borrar a otros usuarios.

9.1 Actualizando usuarios

El patrón para editar la información de un usuario es muy similar al que utilizamos para crear nuevos usuarios (Capítulo 7). En vez de una acción `new` que muestra una vista para los nuevos usuarios, tenemos una acción `edit` que muestra una vista para editar usuarios; en vez de que `create` responda a una petición POST, tendremos una acción `update` que responde a una petición PATCH

(Recuadro 3.2). La mayor diferencia es que, mientras que cualquier persona puede registrarse, sólo el usuario actual puede actualizar su información. La maquinaria de autenticación del Capítulo 8 nos permitirá utilizar un *filtro previo* para asegurar que este es el caso.

Para empezar, trabajemos en una rama **updating-users**:

```
$ git checkout master
$ git checkout -b updating-users
```

9.1.1 Formulario de edición

Empezaremos con el formulario de edición, cuyo bosquejo aparece en la Figura 9.1.¹ Para convertir este bosquejo en una página funcional, necesitamos llenar tanto la acción **edit** en el controlador de usuarios, como la vista para editar al usuario. Empezaremos con la acción **edit**, lo cual requiere de extraer al usuario respetivo de la base de datos. Observe en la Tabla 7.1 que la URL apropiada para la página de edición del usuario es **/users/1/edit** (suponiendo que el id del usuario es 1). Recuerde que el id del usuario está disponible en la variable **params[:id]**, lo cual significa que podemos encontrar al usuario con el código del Listado 9.1.

Listado 9.1: La acción **edit** del usuario.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
  end

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

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      redirect_to @user
    else
      render :edit
    end
  end
end
```

¹Imagen de <http://www.flickr.com/photos/sashawolff/4598355045/>.

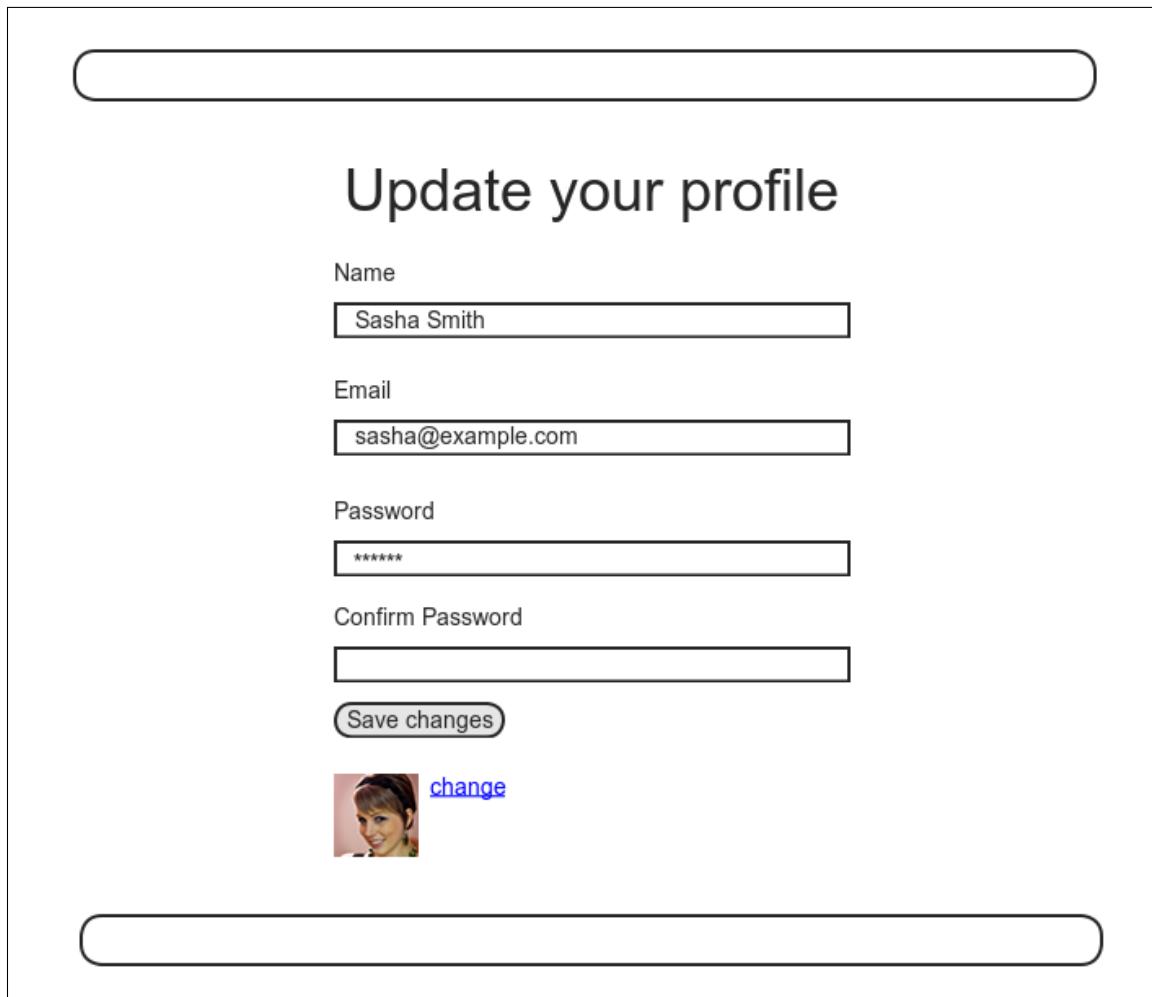


Figura 9.1: Un bosquejo de la página de edición del usuario.

```

if @user.save
  log_in @user
  flash[:success] = "Welcome to the Sample App!"
  redirect_to @user
else
  render 'new'
end
end

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

private

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

```

La vista respectiva para editar al usuario (que usted tendrá que crear manualmente) se muestra en el Listado 9.2. Observe qué tanto se asemeja a la vista de usuarios nuevos del Listado 7.13; la gran intersección sugiere que refactoricemos el código repetido en un parcial, lo cual se deja como ejercicio (Sección 9.6).

Listado 9.2: La vista para editar al usuario.

app/views/users/edit.html.erb

```

<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages' %>

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

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

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

```

```

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

<%= f.submit "Save changes", class: "btn btn-primary" %>
<% end %>

<div class="gravatar_edit">
  <%= gravatar_for @user %>
  <a href="http://gravatar.com/emails" target="_blank">change</a>
</div>
</div>
</div>

```

Aquí hemos reutilizado el parcial compartido `error_messages` que se mostró en la Sección 7.3.3. Por cierto, el uso de `target="_blank"` en el enlace Gravatar es un truco elegante para hacer que el navegador abra la página en una nueva ventana o pestaña, que es un comportamiento conveniente cuando enlazamos con un sitio de terceros.

Con la variable de instancia `@user` del Listado 9.1, la página de edición debería desplegarse adecuadamente, como se muestra en la Figura 9.2. Los campos “Name” y “Email” de la Figura 9.2 también muestran cómo Rails automáticamente pre-llena los campos mencionados utilizando los atributos de la variable `@user` existente.

Revisando el código fuente HTML de la Figura 9.2, vemos una etiqueta `form` como es esperado, como en el Listado 9.3 (puede diferir en algunos detalles).

Listado 9.3: HTML para el formulario de edición definido en el Listado 9.2 y que se muestra en la Figura 9.2.

```

<form accept-charset="UTF-8" action="/users/1" class="edit_user"
  id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="patch" />
  .
  .
  .
</form>

```

Observe aquí el campo de entrada oculto

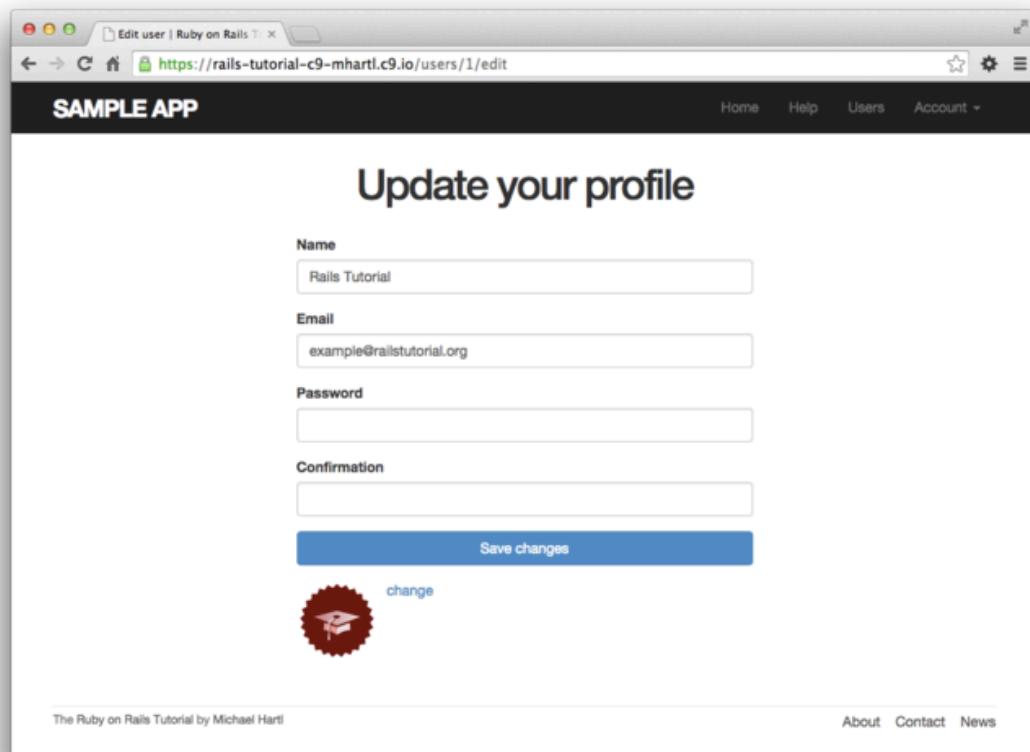


Figura 9.2: La página de edición inicial con el nombre y dirección electrónica pre-llenados.

```
<input name="_method" type="hidden" value="patch" />
```

Puesto que los navegadores web no pueden enviar peticiones PATCH de forma nativa (como es requerido por las convenciones REST de la Tabla 7.1), Rails la simula con una petición POST y un campo **input** oculto.²

Hay otra sutileza que debemos mencionar aquí: el código **form_for(@user)** del Listado 9.2 es *exactamente* el mismo que el del Listado 7.13—por lo que ¿cómo es que Rails determina si debe utilizar una petición POST para crear nuevos usuarios o utilizar una petición PATCH para editar usuarios? La respuesta es que es posible distinguir si un usuario es nuevo o ya existe en la base de datos mediante el método booleano **new_record?** de Active Record:

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

Cuando creamos un formulario utilizando **form_for(@user)**, Rails utiliza POST si **@user.new_record?** es **verdadero** y PATCH si es **falso**.

Como toque final, rellenaremos la URL del enlace “Settings” en la navegación del sitio. Esto es fácil usando la ruta nombrada **edit_user_path** de la Tabla 7.1, junto con el método auxiliar **current_user** definido en el Listado 8.36:

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

El código completo de la aplicación aparece en el Listado 9.4.

²No se preocupe acerca de cómo funciona esto; los detalles son de interés para los desarrolladores de Rails, y por diseño no son importantes para los desarrolladores de aplicaciones Rails.

Listado 9.4: Agregando una URL al enlace “Settings” de la estructura de diseño.

app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Account <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li><%= link_to "Profile", current_user %></li>
              <li><%= link_to "Settings", edit_user_path(current_user) %></li>
              <li class="divider"></li>
              <li>
                <%= link_to "Log out", logout_path, method: "delete" %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

9.1.2 Ediciones fallidas

En esta sección trataremos las ediciones fallidas, siguiendo la analogía de los registros de nuevos usuarios fallidos (Sección 7.3). Empezaremos creando una acción **update**, que utiliza **update_attributes** (Sección 6.1.5) para actualizar al usuario basado en el arreglo hash **params** enviado, como se muestra en el Listado 9.5. Con información no válida, el intento de actualización regresa **falso**, por lo que el bloque **else** regresa la página de edición. Hemos visto este patrón antes; la estructura es muy similar a la primera versión de la acción

create (Listado 7.16).

Listado 9.5: La acción inicial **update** de usuario.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

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

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

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      # Handle a successful update.
    else
      render 'edit'
    end
  end

  private

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

Observe el uso de **user_params** en la llamada a **update_attributes**, que

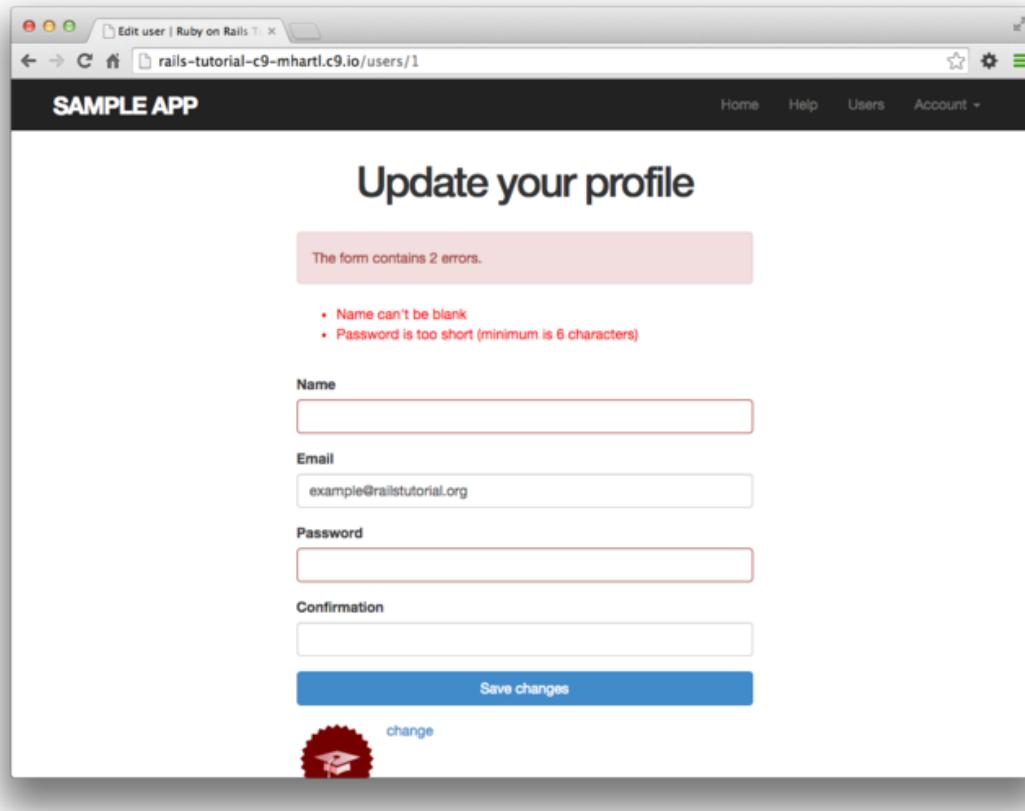


Figura 9.3: Mensaje de error al enviar el formulario de actualización.

utiliza parámetros fuertes para prevenir la vulnerabilidad de asignación masiva (que se describió en la [Sección 7.3.2](#)).

Debido a las validaciones existentes al modelo **User** y a los mensajes de error del parcial del [Listado 9.2](#), el envío de información inválida produce mensajes de error útiles (Figura 9.3).

9.1.3 Probando las ediciones fallidas

Dejamos la [Sección 9.1.2](#) con un formulario de edición funcional. Siguiendo las directrices de prueba del Recuadro 3.3, escribiremos ahora una prueba de

integración para atrapar regresiones. Nuestro primer paso es generar una prueba de integración como es usual:

```
$ rails generate integration_test users_edit
  invoke  test_unit
  create    test/integration/users_edit_test.rb
```

Después escribiremos una prueba simple de una edición fallida, como se muestra en el [Listado 9.6](#). La prueba del [Listado 9.6](#) revisa el correcto funcionamiento al verificar que la plantilla de edición se muestra luego de obtener la página de edición y se vuelve a mostrar a continuación de haber enviado información inválida. Observe el uso del método **patch** para emitir una petición PATCH, el cual sigue el mismo patrón que **get**, **post**, y **delete**.

Listado 9.6: Una prueba para una edición fallida. [VERDE](#)

test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    patch user_path(@user), user: { name: "",
                                    email: "foo@invalid",
                                    password:           "foo",
                                    password_confirmation: "bar" }
    assert_template 'users/edit'
  end
end
```

En este punto, la suite de pruebas debería estar aún en [VERDE](#):

Listado 9.7: VERDE

```
$ bundle exec rake test
```

9.1.4 Ediciones exitosas (con TDD)

Es hora de poner al formulario de edición a trabajar. Editar las imágenes del perfil ya es funcional puesto que externalizamos la carga de imágenes a través del sitio web de Gravatar; podemos editar Gravatares al dar click en el enlace “change” de la Figura 9.2, como se muestra en la Figura 9.4. Hagamos que el resto de la funcionalidad de la edición de usuarios funcione también.

Conforme se sienta más cómodo con las pruebas, puede encontrar útil escribir las pruebas de integración antes de escribir el código de la aplicación en vez de hacerlo después. En este contexto, tales pruebas son algunas veces conocidas como *pruebas de aceptación*, puesto que determinan cuándo una característica en particular puede ser aceptada como completa. Para ver cómo funciona esto, completaremos la edición del usuario utilizando el desarrollo orientado a pruebas.

Probaremos el correcto comportamiento de actualizar usuarios escribiendo una prueba similar a la mostrada en el Listado 9.6, sólo que esta vez le enviaremos información válida. Luego verificaremos que un mensaje flash no vacío aparece y que una redirige exitosamente a la página del perfil, mientras que también verificamos que la información del usuario ha sido correctamente modificada en la base de datos. El resultado aparece en el Listado 9.8. Observe que la contraseña y la confirmación en el Listado 9.8 aparecen en blanco, lo cual es conveniente para usuarios que no desean actualizar sus contraseñas cada vez que actualizan sus nombres o direcciones electrónicas. Observe también el uso de `@user.reload` (que vimos por primera vez en la Sección 6.1.5) para recargar los valores del usuario desde la base de datos y confirmar que fueron actualizados exitosamente. (Esta es la clase de detalle que usted puede olvidar fácilmente al inicio, y es por lo que las pruebas de aceptación –y TDD en general– requieren cierto nivel de experiencia para ser efectivas.)



Figura 9.4: La interfaz de recorte de imágenes de [Gravatar](#), con una foto de [alguien dudos](#)o.

Listado 9.8: Una prueba de una edición exitosa. ROJO

```
test/integration/users_edit_test.rb
```

```
require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  .
  .
  .

  test "successful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    name = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), user: { name: name,
                                    email: email,
                                    password: "",
                                    password_confirmation: "" }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name, @user.name
    assert_equal email, @user.email
  end
end
```

La acción **update** necesaria para hacer que las pruebas del Listado 9.8 pasen, es similar al formulario final de la acción **create** (Listado 8.22), como vemos en el Listado 9.9.

Listado 9.9: La acción **update** de usuario. ROJO

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  .
  .
  .

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    end
  end
end
```

```

else
  render 'edit'
end
end
.
.
.
end

```

Como se indica en el título del Listado 9.9, el conjunto de pruebas estará aún en **ROJO**, lo cual es el resultado de la validación de longitud de la contraseña (Listado 6.39) que falla debido a la contraseña vacía y su confirmación en el Listado 9.8. Para hacer que las pruebas estén en **VERDE**, necesitamos hacer una excepción a la validación de la contraseña si ésta es vacía. Podemos hacer esto pasando la opción **allow_nil: true** a **validates**, como vemos en el Listado 9.10.

Listado 9.10: Permitiendo contraseñas vacías en la actualización. **VERDE**
app/models/user.rb

```

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

```

En caso de que le preocupe que el Listado 9.10 pueda permitir a los nuevos usuarios registrarse con contraseñas vacías, recuerde que en la Sección 6.3.3 el método **has_secure_password** incluye una validación de presencia separada de la creación del objeto.

Con el código de esta sección, la página de edición del usuario debería estar funcionando (Figura 9.5), como puede volver a verificar ejecutando de nuevo

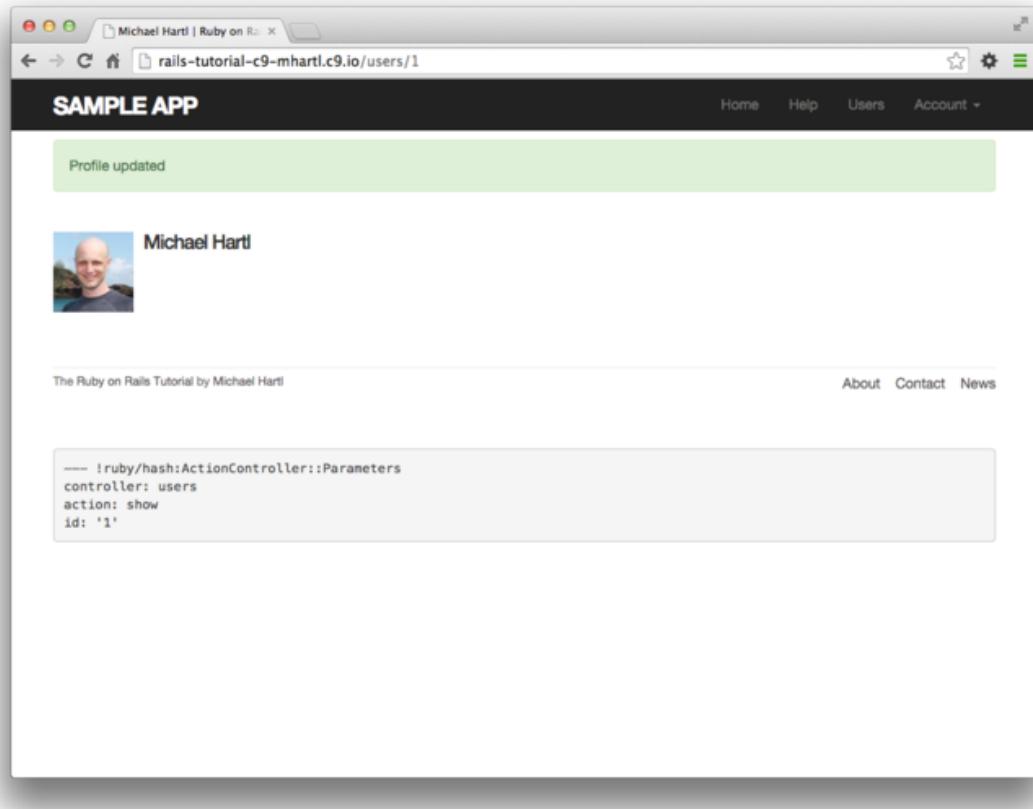


Figura 9.5: El resultado de una edición exitosa.

el conjunto de pruebas, que ahora debería estar en **VERDE**:

Listado 9.11: **VERDE**

```
$ bundle exec rake test
```

9.2 Autorización

En el contexto de aplicaciones web, la *autenticación* nos permite identificar a los usuarios de nuestro sitio, y la *autorización* nos permite controlar lo que

pueden hacer. Lo bueno de haber construído la maquinaria de autenticación en el Capítulo 8 es que ahora estamos en posición de implementar la autorización también.

Aunque las acciones de edición y actualización de la Sección 9.1 están completamente funcionales, tienen una falla de seguridad: permiten a todos (incluso a los usuarios que no han iniciado sesión) el acceso a cualquier acción, y a cualquier usuario que ha iniciado sesión, actualizar la información de cualquier otro usuario. En esta sección, implementaremos un modelo de seguridad que requiera a los usuarios iniciar sesión y que evite que actualicen cualquier información que no sea la suya.

En la Sección 9.2.1, manejaremos el caso de los usuarios que no han iniciado sesión que intenten acceder a una página protegida a la cual normalmente tendrían acceso. Como esto podría suceder fácilmente en el transcurso de utilizar la aplicación, tales usuarios serán redirigidos a la página de inicio de sesión con un mensaje de ayuda, como se bosquejó en la Figura 9.6. Por otra parte, los usuarios que intenten acceder a una página para la cual no han tenido nunca autorización (tal como es el caso de un usuario que está en sesión e intenta tener acceso a la página de edición de un usuario diferente) serán redirigidos a la URL raíz (Sección 9.2.2).

9.2.1 Requeriendo a los usuarios que inicien sesión

Para implementar el comportamiento mostrado en la Figura 9.6, utilizaremos un filtro *before* en el controlador `Users`. Los filtros de este tipo utilizan el comando `before_action` para que un método en particular sea invocado antes que las acciones dadas.³ Para solicitar a los usuarios que inicien sesión, definimos un método `logged_in_user` y lo invocamos utilizando `before_action :logged_in_user`, como se muestra en el Listado 9.12.

Listado 9.12: Agregando un filtro a `logged_in_user`. ROJO
`app/controllers/users_controller.rb`

³El comando para filtros *previos* se utilizaba para invocar `before_filter`, pero el equipo de Rails decidió renombrarlo para enfatizar que el filtro se invoca antes que las acciones particulares del controlador.

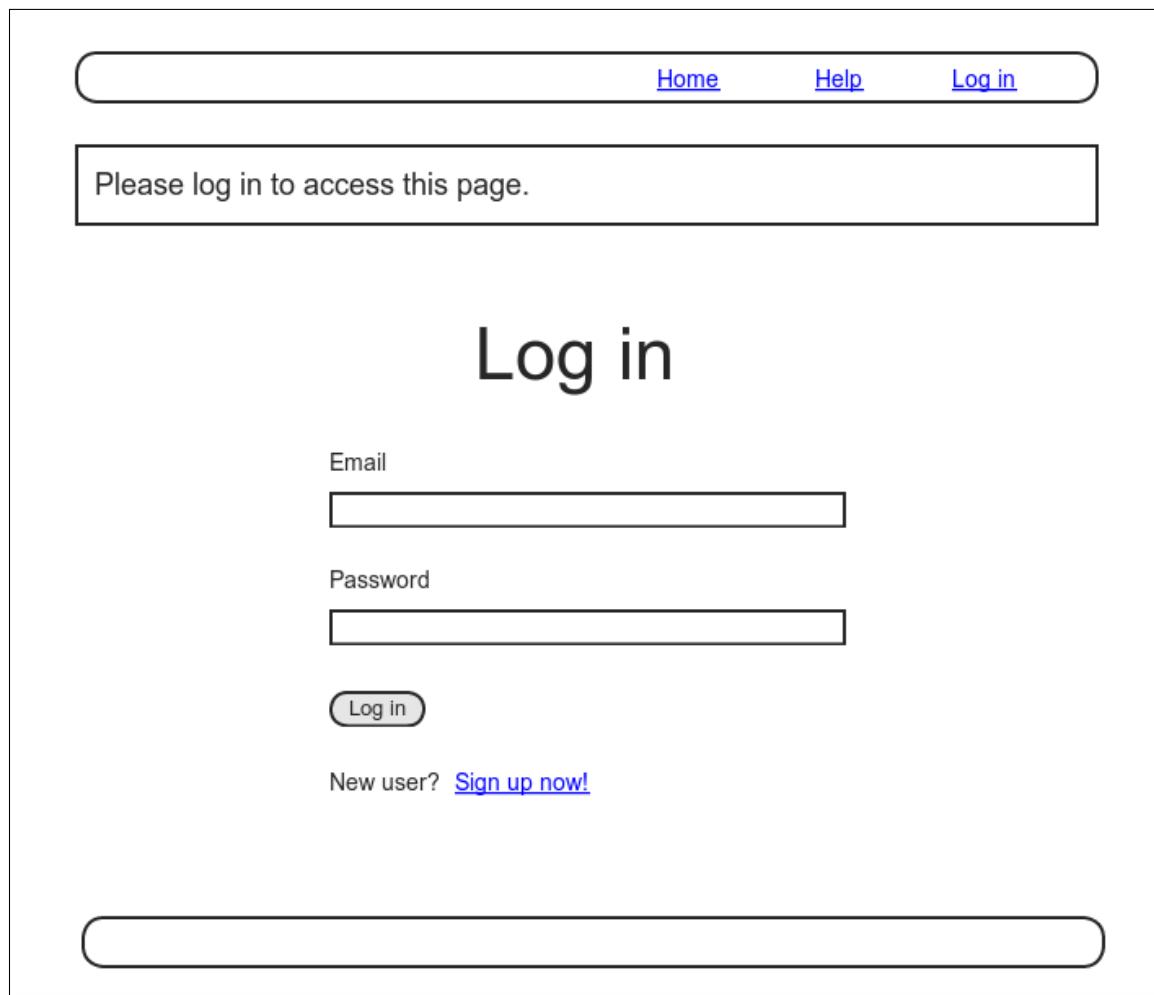


Figura 9.6: Un bosquejo del resultado de visitar una página protegida

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
  private

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

  # Before filters

  # Confirms a logged-in user.
  def logged_in_user
    unless logged_in?
      flash[:danger] = "Please log in."
      redirect_to login_url
    end
  end
end
```

Por default, los filtros *previos* aplican a *toda* acción en un controlador, por lo que restringiremos el filtro para que actúe sólo en las acciones **:edit** y **:update** pasando el arreglo hash de opciones apropiado **:only**.

Podemos ver el resultado del filtro *previo* en el Listado 9.12 al cerrar sesión e intentar acceder a la página de edición del usuario [/users/1/edit](#), como se muestra en la Figura 9.7.

Como se indica en el título del Listado 9.12, nuestro conjunto de pruebas está actualmente en **ROJO**:

Listado 9.13: ROJO

```
$ bundle exec rake test
```

La razón es que las acciones **edit** y **update** ahora requieren un usuario en sesión, pero ningún usuario está en sesión dentro de las pruebas correspondientes.

Corregiremos nuestro conjunto de pruebas iniciando la sesión del usuario antes de invocar las acciones de edición o actualización. Esto es fácil utilizando

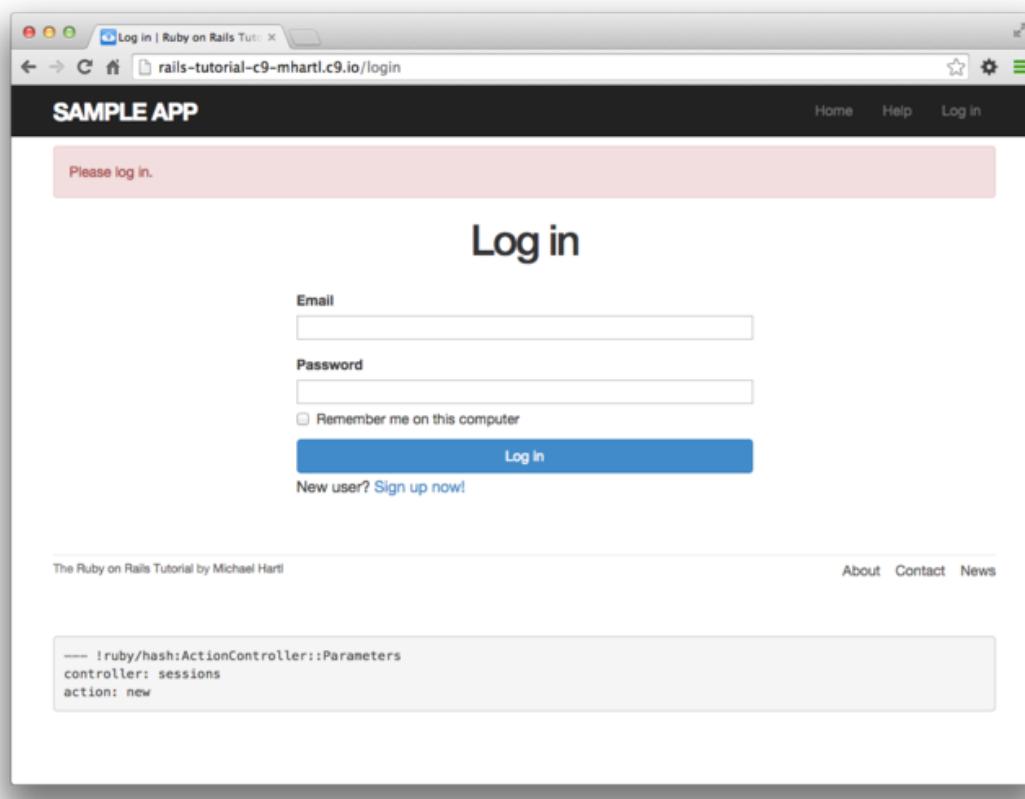


Figura 9.7: El formulario de inicio de sesión luego de intentar acceder una página protegida.

el auxiliar **log_in_as** desarrollado en la Sección 8.4.6 (Listado 8.50), como se muestra en el Listado 9.14.

Listado 9.14: Iniciando la sesión del usuario de pruebas. VERDE

```
test/integration/users_edit_test.rb

require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
    .
    .
  end

  test "successful edit" do
    log_in_as(@user)
    get edit_user_path(@user)
    .
    .
    .
  end
end
```

(Podemos eliminar código duplicado poniendo el inicio de sesión de pruebas en el método **setup** del Listado 9.14, pero en la Sección 9.2.3 cambiaremos una de las pruebas para visitar la página de edición *antes* de iniciar sesión, lo cual no es posible si el inicio de sesión sucede durante el inicio de las pruebas.)

En este momento, nuestro conjunto de pruebas debería estar en VERDE:

Listado 9.15: VERDE

```
$ bundle exec rake test
```

Aún cuando nuestro conjunto de pruebas esté pasando, no hemos terminado con el filtro *previo*, porque el conjunto de pruebas está en VERDE aún si elimi-

namos nuestro modelo de seguridad, como puede verificar comentándolo ([Listado 9.16](#)). Esto es [algo malo](#)—de todas las regresiones que nos gustaría que nuestro conjunto de pruebas atrapara, un agujero de seguridad masivo es probablemente el más importante, por lo que el código del [Listado 9.16](#) debería estar definitivamente en [ROJO](#). Escribamos pruebas que arreglen eso.

Listado 9.16: Comentando el filtro *previo* para probar nuestro modelo de seguridad. [VERDE](#)

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  # before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

Como el filtro *previo* opera en base a una acción, pondremos las pruebas correspondientes en el archivo de pruebas del controlador **Users**. El plan es invocar las acciones **edit** y **update** con el tipo de peticiones correctas y verificar que el flash es correcto y que el usuario es redirigido a la ruta de inicio de sesión. De la [Tabla 7.1](#), vemos que las peticiones apropiadas son GET y PATCH, respectivamente, lo que significa que debemos utilizar los métodos **get** y **patch** dentro de las pruebas. El resultado aparece en el [Listado 9.17](#).

Listado 9.17: Probando que **edit** y **update** son protegidas. [ROJO](#)

```
test/controllers/users_controller_test.rb
```

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user = users(:michael)
  end

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

```
test "should redirect edit when not logged in" do
  get :edit, id: @user
  assert_not flash.empty?
  assert_redirected_to login_url
end

test "should redirect update when not logged in" do
  patch :update, id: @user, user: { name: @user.name, email: @user.email }
  assert_not flash.empty?
  assert_redirected_to login_url
end
end
```

Observe que los argumentos de `get` y `patch` involucran código como

```
get :edit, id: @user
```

y

```
patch :update, id: @user, user: { name: @user.name, email: @user.email }
```

Esto utiliza la convención Rails de `id: @user`, la cual (como en las redirecciones del controlador) automáticamente utiliza `@user.id`. En el segundo caso, necesitamos proporcionar un arreglo hash `user` adicional con la finalidad de que las rutas funcionen adecuadamente. (Si usted observa las pruebas generadas para el controlador Users de la aplicación de juguete en el Capítulo 2, podrá observar el código anterior.)

El conjunto de pruebas debería estar ahora en ROJO, como es requerido. Para hacer que esté en VERDE, sólo descomente el filtro *previo* (Listado 9.18).

Listado 9.18: Descomentando el filtro *previo*. VERDE

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

Con esto, nuestro conjunto de pruebas debería estar en VERDE:

Listado 9.19: VERDE

```
$ bundle exec rake test
```

Cualquier exposición accidental de los métodos de edición a usuarios no autorizados será detectada inmediatamente por nuestro conjunto de pruebas.

9.2.2 Requeriendo el usuario correcto

Por supuesto, requerir que los usuarios inicien sesión no es suficiente; los usuarios deberían tener permiso de editar únicamente *su propia* información. Como vimos en la [Sección 9.2.1](#), es fácil tener un conjunto de pruebas que omita una agujero de seguridad esencial, por lo que procederemos usando el desarrollo orientado a pruebas para asegurarnos de que nuestro código implementa el modelo de seguridad correctamente. Para hacer esto, agregaremos pruebas al controlador **Users** que complementen las del [Listado 9.17](#).

Para asegurarnos de que los usuarios no pueden editar la información de otros usuarios, necesitamos ser capaces de iniciar sesión con un segundo usuario. Esto significa que debemos agregar un segundo usuario al archivo *fixture*, como se muestra en el [Listado 9.20](#).

Listado 9.20: Agregando un segundo usuario al archivo *fixture*.

```
test/fixtures/users.yml
```

```

michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>

```

Al utilizar el método `log_in_as` definido en el Listado 8.50, podemos probar las acciones `edit` y `update` como en el Listado 9.21. Observe que esperamos redireccionar a los usuarios a la ruta raíz en vez de a la ruta de inicio de sesión porque un usuario tratando de editar a un usuario diferente debería estar ya en sesión.

Listado 9.21: Pruebas para intentar editar como un usuario equivocado. ROJO
test/controllers/users_controller_test.rb

```

require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user      = users(:michael)
    @other_user = users(:archer)
  end

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

  test "should redirect edit when not logged in" do
    get :edit, id: @user
    assert_not flash.empty?
    assert_redirected_to login_url
  end

  test "should redirect update when not logged in" do
    patch :update, id: @user, user: { name: @user.name, email: @user.email }
    assert_not flash.empty?
    assert_redirected_to login_url
  end

  test "should redirect edit when logged in as wrong user" do

```

```

log_in_as(@other_user)
get :edit, id: @user
assert flash.empty?
assert_redirected_to root_url
end

test "should redirect update when logged in as wrong user" do
  log_in_as(@other_user)
  patch :update, id: @user, user: { name: @user.name, email: @user.email }
  assert flash.empty?
  assert_redirected_to root_url
end
end

```

Para redireccionar usuarios que intentan editar otro perfil de usuario, agregaremos un segundo método llamado **correct_user**, junto con el filtro *previo* para invocarlo (Listado 9.22). Observe que el **correct_user** del filtro define una variable **@user**, por lo que el Listado 9.22 también muestra que podemos eliminar las asignaciones a **@user** en las acciones **edit** y **update**.

Listado 9.22: **correct_user** como filtro *previo* para proteger las páginas edit / update. VERDE

app/controllers/users_controller.rb

```

class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user, only: [:edit, :update]

  def edit
  end

  def update
    if @user.update_attributes(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
end

private

```

```

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

# Before filters

# Confirms a logged-in user.
def logged_in_user
  unless logged_in?
    flash[:danger] = "Please log in."
    redirect_to login_url
  end
end

# Confirms the correct user.
def correct_user
  @user = User.find(params[:id])
  redirect_to(root_url) unless @user == current_user
end
end

```

En este punto, nuestro conjunto de pruebas debería estar en **VERDE**:

Listado 9.23: VERDE

```
$ bundle exec rake test
```

Como reestructuración final, adoptaremos una convención común y definiremos un método booleano **current_user?** para usar en el filtro *previo* **correct_user**, el cual definimos en el auxiliar **Sessions**(Listado 9.24). Utilizaremos este método para reemplazar código como el que sigue

```
unless @user == current_user
```

con el (ligeramente) más expresivo

```
unless current_user?(@user)
```

Listado 9.24: El método `current_user?`.

`app/helpers/sessions_helper.rb`

```
module SessionsHelper

  # Logs in the given user.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Remembers a user in a persistent session.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Returns true if the given user is the current user.
  def current_user?(user)
    user == current_user
  end

  .
  .
  .

end
```

Al reemplazar la comparación directa con el método booleano obtenemos el código que se muestra en el Listado 9.25.

Listado 9.25: La versión final del filtro *previo* `correct_user`. VERDE

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]

  .
  .
  .

  def edit
  end

  def update
    if @user.update_attributes(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
  end
```

```
    render 'edit'
  end
end
.
.
.
private

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

# Before filters

# Confirms a logged-in user.
def logged_in_user
  unless logged_in?
    flash[:danger] = "Please log in."
    redirect_to login_url
  end
end

# Confirms the correct user.
def correct_user
  @user = User.find(params[:id])
  redirect_to(root_url) unless current_user?(@user)
end
end
```

9.2.3 Redirección amigable

La autorización de nuestro sitio está completa tal como está escrita, pero tiene un pequeño defecto: cuando los usuarios intentan acceder a una página protegida, en este momento están siendo redirigidos a sus páginas de perfil sin importar a dónde intentaban ir. En otras palabras, si un usuario que no ha iniciado sesión intenta visitar la página de edición, luego de iniciar su sesión, será redirigido a `/users/1` en vez de `/users/1/edit`. En vez de esto, sería mucho más amigable redirigirlo al destino que deseaba visitar.

El código de la aplicación se volverá relativamente complicado, pero podemos escribir una prueba muy sencilla para la redirección amigable tan sólo si invertimos el orden del inicio de sesión y la visita a la página de edición del Listado 9.14. Como vimos en el Listado 9.26, las pruebas resultantes intentan

visitar la página de edición, luego inician sesión, y después verifican que el usuario sea redirigido a la página de edición en vez de a la página del perfil. (El Listado 9.26 también elimina la prueba para desplegar la plantilla de edición puesto que ya no es el comportamiento esperado.)

Listado 9.26: Una prueba para el reenvío amigable. ROJO

test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "successful edit with friendly forwarding" do
    get edit_user_path(@user)
    log_in_as(@user)
    assert_redirected_to edit_user_path(@user)
    name = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), user: { name: name,
                                    email: email,
                                    password: "",
                                    password_confirmation: "" }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name, @user.name
    assert_equal email, @user.email
  end
end
```

Ahora que tenemos una prueba fallando, estamos listos para implementar la redirección amigable.⁴ Con la finalidad de redirigir a los usuarios a su destino deseado, necesitamos almacenar la ubicación de la página solicitada en algún lugar, y luego redirigirlos hacia esa ubicación en vez de a la ubicación por default. Lograremos esto con un par de métodos, `store_location` y `redirect_back_or`, ambos definidos en el auxiliar `Sessions` (Listado 9.27).

⁴El código de esta sección es una adaptación de la gema `Clearance` de [thoughtbot](#).

Listado 9.27: Código para implementar la redirección amigable.

app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Redirects to stored location (or to the default).
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
  end

  # Stores the URL trying to be accessed.
  def store_location
    session[:forwarding_url] = request.url if request.get?
  end
end
```

Aquí el mecanismo de almacenamiento para redirigir la URL es la misma utilería de **session** que usamos en la Sección 8.2.1 para iniciar la sesión del usuario. El Listado 9.27 también utiliza el objeto **request** (via **request.url**) para obtener la URL de la página solicitada.

El método **store_location** del Listado 9.27 almacena la URL solicitada en la variable **session** bajo la llave **:forwarding_url**, pero sólo para una petición **GET**. Esto evita que se almacene la URL destino si el usuario, digamos, envía un formulario cuando no ha iniciado sesión (lo cual es un caso extremo, pero podría suceder si por ejemplo, un usuario borrara las galletas de sesión manualmente antes de enviar el formulario). En tal caso, la redirección resultante emitiría una petición **GET** a una URL que espera un **POST**, **PATCH**, o **DELETE**, causando de esta forma un error. Incluir **if request.get?** previene que esto suceda.⁵

Para utilizar el **store_location**, necesitamos agregarlo al filtro *previo* **logged_in_user**, como se muestra en el Listado 9.28.

⁵Agradezco al lector Yoel Adler por señalar este detalle y por descubrir la solución.

Listado 9.28: Agregando `store_location` al filtro *previo* del inicio de sesión.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  before_action :correct_user,   only: [:edit, :update]

  .

  .

  def edit
  end

  .

  .

  private

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

    # Before filters

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Confirms the correct user.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

Para implementar el reenvío como tal, utilizamos el método `redirect_back_or` para redireccionar a la URL solicitada si existe, o en caso contrario, a alguna otra URL por default, la cual agregamos a la acción `create` del controlador de sesiones para redirigir luego de haber iniciado sesión exitosamente ([Listado 9.29](#)). El método `redirect_back_or` utiliza el operador `o ||` como sigue:

```
session[:forwarding_url] || default
```

Esto evalúa como `session[:forwarding_url]` a menos que sea `nil`, en cuyo caso evalúa a la URL dada por default. Observe que el Listado 9.27 tiene el cuidado de eliminar la URL de reenvío (mediante `session.delete(:forwarding_url)`); de otra forma, los siguientes intentos de iniciar sesión redirigirían a la página protegida hasta que el usuario cerrara su navegador. (Probar esto se deja como ejercicio (Sección 9.6).) Observe también que la sesión se elimina aún cuando la línea con la redirección aparece primero; el redireccionamiento no ocurre hasta que una instrucción `return` o el final del método ocurre, por lo que cualquier código que siga después de la redirección es ejecutado.

Listado 9.29: La acción `create` del controlador de sesiones con un redireccionamiento amigable.

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      log_in user
      params[:session][:remember_me] == '1' ? remember(user) : forget(user)
      redirect_back_or user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end
  .
  .
  .
end
```

Con esto, la prueba de integración del redireccionamiento amigable del Listado 9.26 debería pasar, y la autenticación de usuario básica y la imple-

mentación de la protección de páginas están completas. Como es usual, es una buena idea verificar que el conjunto de pruebas está en **VERDE** antes de continuar:

Listado 9.30: **VERDE**

```
$ bundle exec rake test
```

9.3 Mostrando todos los usuarios

En esta sección, agregaremos la **penúltima** acción de usuario, la acción **index**, que está diseñada para mostrar *todos* los usuarios en vez de sólo uno. De paso, aprenderemos cómo alimentar la base de datos con usuarios ejemplo y cómo *paginar* la salida de usuario de forma que la página del listado de usuarios pueda potencialmente mostrar un gran número de resultados. Un bosquejo del resultado—usuarios, enlaces de paginación, y un enlace de navegación “Users”—aparece en la [Figura 9.8](#).⁶ En la [Sección 9.4](#), agregaremos una interfaz administrativa al índice de usuarios de forma que éstos también puedan ser destruidos.

9.3.1 Listado de Usuarios

Para empezar con el listado de usuarios, primero implementaremos un modelo de seguridad. Aunque mantendremos las páginas individuales **show** que muestran a un usuario visibles a todos los visitantes del sitio, el listado de usuarios estará restringido a usuarios en sesión de forma que existe un límite por default a cuánto pueden ver los usuarios que no están registrados.⁷

Para proteger la página **index** de acceso no autorizado, primero agregaremos una pequeña prueba para verificar que la acción **index** es redirigida apropiadamente ([Listado 9.31](#)).

⁶Foto de bebé de <http://www.flickr.com/photos/glasgows/338937124/>.

⁷Este es el mismo modelo de autorización utilizado por Twitter.

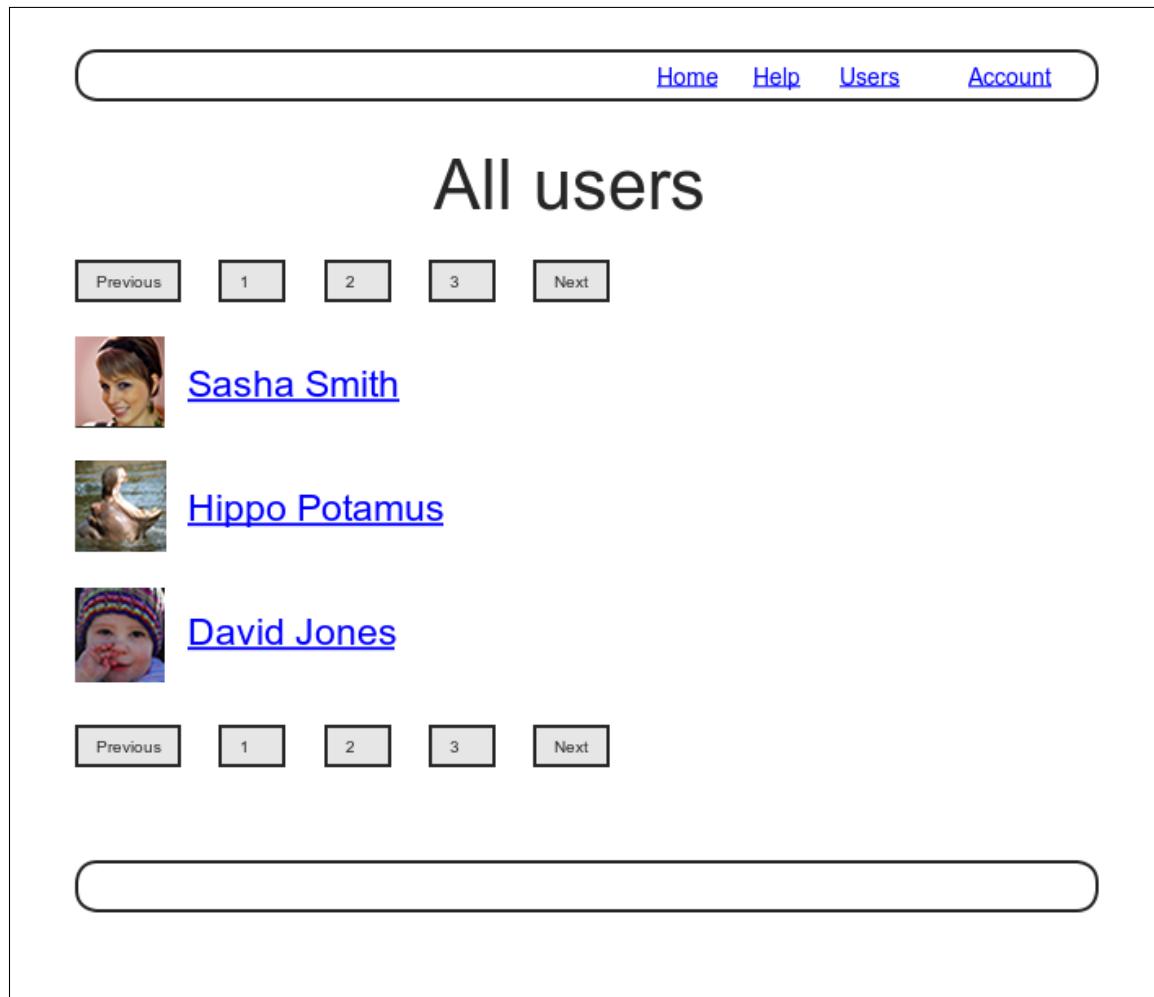


Figura 9.8: Un bosquejo de la página del listado de usuarios.

Listado 9.31: Probando el redireccionamiento de la acción `index`. ROJO

```
test/controllers/users_controller_test.rb
```

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user      = users(:michael)
    @other_user = users(:archer)
  end

  test "should redirect index when not logged in" do
    get :index
    assert_redirected_to login_url
  end
  .
  .
  .
end
```

Luego sólo necesitamos agregar una acción `index` e incluirla en la lista de acciones protegidas por el filtro `logged_in_user` (Listado 9.32).

Listado 9.32: Requeriendo un usuario en sesión para la acción `index`. VERDE

```
app/controllers/users_controller.rb
```

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update]
  before_action :correct_user,   only: [:edit, :update]

  def index
  end

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

Para mostrar a los usuarios, necesitamos crear una variable que contenga a todos los usuarios del sitio y luego mostrar cada uno iterando sobre la variable

en la vista del listado. Como puede recordar de la acción correspondiente en la aplicación de juguete (Listado 2.5), podemos utilizar `User.all` para traer a todos los usuarios de la base de datos, asignándolos a una variable de instancia `@users` que utilizamos en la vista, como se observa en el Listado 9.33. (Si desplegar todos los usuarios a la vez parece una mala idea, usted está en lo correcto, removeremos este defecto en la Sección 9.3.3.)

Listado 9.33: La acción `index` de usuario.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update]
  .
  .
  .

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

Para crear la página del listado, generaremos una vista (cuyo archivo tendremos que crear) que itere sobre los usuarios y los encapsule en una etiqueta `li`. Logramos esto con el método `each`, desplegando el nombre y Gravatar de cada usuario, al mismo tiempo que envolvemos todo en una etiqueta `ul` (Listado 9.34).

Listado 9.34: La vista del listado de usuarios.

`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 50 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>
```

530 CAPÍTULO 9. ACTUALIZANDO, MOSTRANDO Y BORRANDO USUARIOS

El código del Listado 9.34 utiliza el resultado del Listado 7.31 de la Sección 7.7, que nos permite pasar una opción al auxiliar Gravatar especificando un tamaño diferente del default. Si usted no hizo ese ejercicio, actualice su archivo auxiliar **Users** con el contenido del Listado 7.31 antes de continuar.

Agreguemos un poco de CSS (o más bien, SCSS) para estilizar (Listado 9.35).

Listado 9.35: CSS para el listado de usuarios.

app/assets/stylesheets/custom.css.scss

```
.  
./* Users index */  
  
.users {  
  list-style: none;  
  margin: 0;  
  li {  
    overflow: auto;  
    padding: 10px 0;  
    border-bottom: 1px solid $gray-lighter;  
  }  
}
```

Finalmente, agregaremos la URL al enlace de usuarios en el encabezado de la navegación del sitio mediante **users_path**, empleando de esta forma la última de las rutas nombradas de la Tabla 7.1 que faltaban de utilizar. El resultado aparece en el Listado 9.36.

Listado 9.36: Agregando la URL al enlace de usuarios.

app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">  
  <div class="container">  
    <%= link_to "sample app", root_path, id: "logo" %>  
    <nav>  
      <ul class="nav navbar-nav navbar-right">  
        <li><%= link_to "Home", root_path %></li>  
        <li><%= link_to "Help", help_path %></li>  
        <% if logged_in? %>  
          <li><%= link_to "Users", users_path %></li>  
          <li class="dropdown">
```

```

<a href="#" class="dropdown-toggle" data-toggle="dropdown">
  Account <b class="caret"></b>
</a>
<ul class="dropdown-menu">
  <li><%= link_to "Profile", current_user %></li>
  <li><%= link_to "Settings", edit_user_path(current_user) %></li>
  <li class="divider"></li>
  <li>
    <%= link_to "Log out", logout_path, method: "delete" %>
  </li>
</ul>
</li>
<% else %>
  <li><%= link_to "Log in", login_path %></li>
<% end %>
</ul>
</nav>
</div>
</header>
```

Con esto, el listado de usuarios está funcionando en su totalidad, con todas las pruebas en **VERDE**:

Listado 9.37: **VERDE**

```
$ bundle exec rake test
```

Por otra parte, como se observa en la [Figura 9.9](#), esto está un poco...solitario. Pongamos remedio a esta situación.

9.3.2 Usuarios de ejemplo

En esta sección, le daremos a nuestro usuario solitario un poco de compañía. Por supuesto, para crear suficientes usuarios de forma que la página de listado se viera decente, *podríamos* utilizar nuestro navegador para visitar la página de registro y crear nuevos usuarios, uno por uno, pero una solución mucho mejor es utilizar Ruby (y Rake) para crear estos usuarios.

Primero, agregaremos la gema *Faker* al archivo **Gemfile**, el cual nos permitirá crear usuarios de ejemplo con nombres y direcciones electrónicas semi-reales ([Listado 9.38](#)).

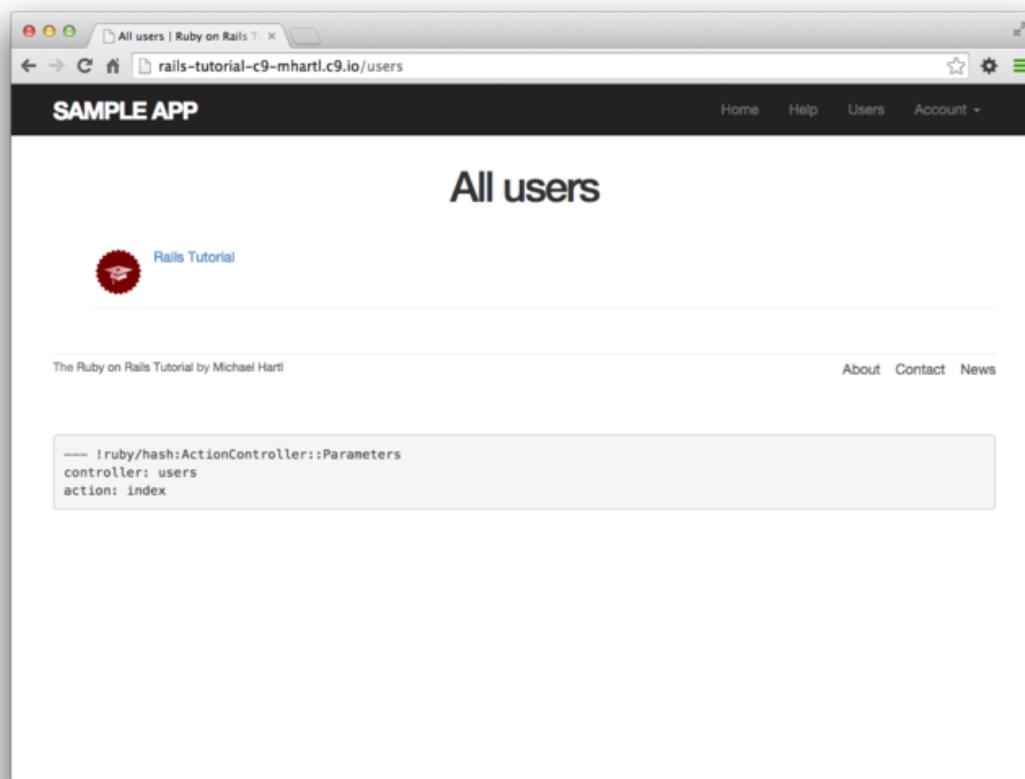


Figura 9.9: La página del listado de usuarios con sólo un usuario.

Listado 9.38: Agregando la gema Faker al archivo **Gemfile**.

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

gem 'rails',          '4.2.2'
gem 'bcrypt',          '3.1.7'
gem 'faker',           '1.4.2'
.
.
```

Luego instálela como es usual:

```
$ bundle install
```

A continuación, agregaremos una tarea Rake para alimentar la base de datos con los usuarios de ejemplo, para esto Rails utiliza el archivo estándar **db/seeds.rb**. El resultado aparece en el [Listado 9.39](#). (El código del [Listado 9.39](#) es un poco avanzado, por lo que no se preocupe demasiado sobre sus detalles.)

Listado 9.39: Una tarea Rake para alimentar la base de datos con usuarios de ejemplo.

db/seeds.rb

```
User.create!(name:  "Example User",
            email: "example@railstutorial.org",
            password:          "foobar",
            password_confirmation: "foobar")

99.times do |n|
  name = Faker::Name.name
  email = "example-#{n+1}@railstutorial.org"
  password = "password"
  User.create!(name:  name,
              email: email,
              password:          password,
              password_confirmation: password)
end
```

El código del [Listado 9.39](#) crea un usuario de ejemplo con nombre y dirección electrónica replicando el anterior y creando 99 más. El método **create!** es

similar al método `create`, excepto que arroja una excepción (Sección 6.1.4) para un usuario no válido en vez de regresar `falso`. Este comportamiento facilita la depuración al evitar errores silenciosos.

Con el código del Listado 9.39, podemos reiniciar la base de datos y luego invocar la tarea Rake usando `db:seed`:⁸

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

Alimentar la base de datos puede ser lento, y en algunos sistemas puede tomar algunos minutos.

Luego de ejecutar la tarea Rake `db:seed`, nuestra aplicación tiene 100 usuarios de ejemplo. Como se observa en la Figura 9.10, me he tomado la libertad de asociar las primeras direcciones electrónicas con Gravatares de forma que no todas ellas muestran la imagen Gravatar por default. (En este momento es probable que necesite reiniciar su servidor web.)

9.3.3 Paginación

Nuestro usuario original ya no padece de soledad, pero ahora tenemos el problema opuesto: nuestro usuario tiene *demasiada* compañía, y todos aparecen en la misma página. En este momento hay un ciento, que es un número razonablemente grande, pero en un sitio real podría alcanzar la cifra de miles. La solución es *paginar* los resultados, de forma que (por ejemplo) se muestren de 30 en 30.

Existen varios métodos de paginación en Rails; nosotros utilizaremos uno de los más sencillos y robustos, llamado `will_paginate`. Para usarlo, necesitamos incluir las gemas `will_paginate` y `bootstrap-will_paginate`, que configura `will_paginate` para ser utilizado con los estilos de paginación de Bootstrap. El archivo `Gemfile` actualizado aparece en el Listado 9.40.

⁸En principio, estas dos tareas pueden combinarse en `rake db:reset`, pero a la fecha en que se escribe esto, el comando no está funcionando en la última versión de Rails.

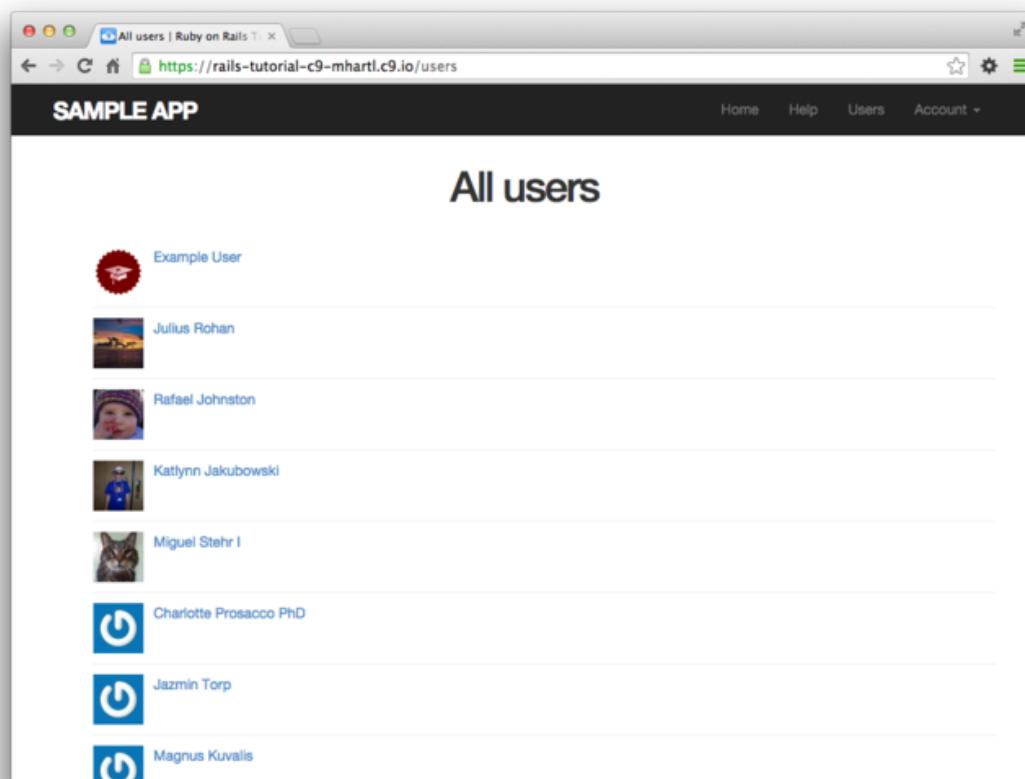


Figura 9.10: La página de listado de usuarios con 100 usuarios de ejemplo.

Listado 9.40: Incluyendo la gema `will_paginate` en el archivo `Gemfile`.

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

gem 'rails',                  '4.2.2'
gem 'bcrypt',                  '3.1.7'
gem 'faker',                   '1.4.2'
gem 'will_paginate',           '3.0.7'
gem 'bootstrap-will_paginate', '0.0.10'

.'
```

Luego ejecute `bundle install`:

```
$ bundle install
```

También se le sugiere reiniciar el servidor web para asegurarse de que las nuevas gemas sean cargadas apropiadamente.

Para hacer que la paginación funcione, necesitamos agregar un poco de código a la vista del listado indicándole a Rails que pague a los usuarios, y necesitamos reemplazar `User.all` en la acción `index` con un objeto que sepa de paginación. Empezaremos agregando el método especial `will_paginate` en la vista ([Listado 9.41](#)); en un momento veremos porqué el código aparece tanto antes como después de la lista de usuarios.

Listado 9.41: El listado de usuarios con paginación.

app/views/users/index.html.erb

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= gravatar_for user, size: 50 %>
      <%= link_to user.name, user %>
    </li>
  <% end %>
</ul>

<%= will_paginate %>
```

El método `will_paginate` es algo mágico; dentro de la vista `users`, automáticamente busca un objeto `@users`, y luego despliega los enlaces de paginación para accesar las otras páginas. La vista del Listado 9.41 no funciona aún, pero se debe a que en este momento `@users` contiene el resultado de `User.all` (Listado 9.33), mientras que `will_paginate` requiere que paginemos los resultados explícitamente utilizando el m??todo `paginate`:

```
$ rails console
>> User.paginate(page: 1)
User Load (1.5ms)  SELECT "users".* FROM "users" LIMIT 30 OFFSET 0
(1.7ms)  SELECT COUNT(*) FROM "users"
=> #<ActiveRecord::Relation:0x0000000000000000>
```

Observe que `paginate` toma como argumento un arreglo hash con una llave `:page` y un valor igual a la página requerida. `User.paginate` trae los usuarios de la base de datos parte por parte (30 por default), basado en el parámetro `:page`. De forma que, por ejemplo, la página 1 muestra a los usuarios 1–30, la página 2 a los usuarios 31–60, etc. Si `page` es `nil`, `paginate` entonces regresa a la primera página.

Usando el método `paginate`, podemos paginar a los usuarios de la aplicación de ejemplo utilizando `paginate` en vez de `all` en la acción `index`

(Listado 9.42). Aquí el parámetro `page` viene de `params[:page]`, el cual es generado automáticamente por `will_paginate`.

Listado 9.42: Paginando a los usuarios en la acción `index`.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update]
  .

  .

  .

  def index
    @users = User.paginate(page: params[:page])
  end
  .

  .

  .
end
```

La página del listado de usuarios debería estar funcionando ahora, como se muestra en la Figura 9.11. (En algunos sistemas, podría ser necesario reiniciar el servidor de Rails en este momento.) Como incluimos `will_paginate` tanto antes como después de la lista de usuarios, los enlaces de la paginación aparecen en ambos lugares.

Si usted da click ya sea en el enlace `2` o en `Next`, se mostrará la segunda página de resultados, como se observa en la Figura 9.12.

9.3.4 Prueba al listado de usuarios

Ahora que nuestra página del listado de usuarios está funcionando, escribiremos una prueba ligera para ella, incluyendo una prueba mínima para la paginación de la Sección 9.3.3. La idea es iniciar una sesión, visitar la ruta del listado, verificar que la primera página del listado está presente y luego confirmar que la paginación está presente en la página. Para que estos dos últimos pasos funcionen, necesitamos tener suficientes usuarios en la base de datos de pruebas para invocar la paginación, es decir, más de 30.

Creamos un segundo usuario en el archivo fixtures del Listado 9.20, pero 30 o más usuarios son muchos para crearlos manualmente. Afortunadamente,

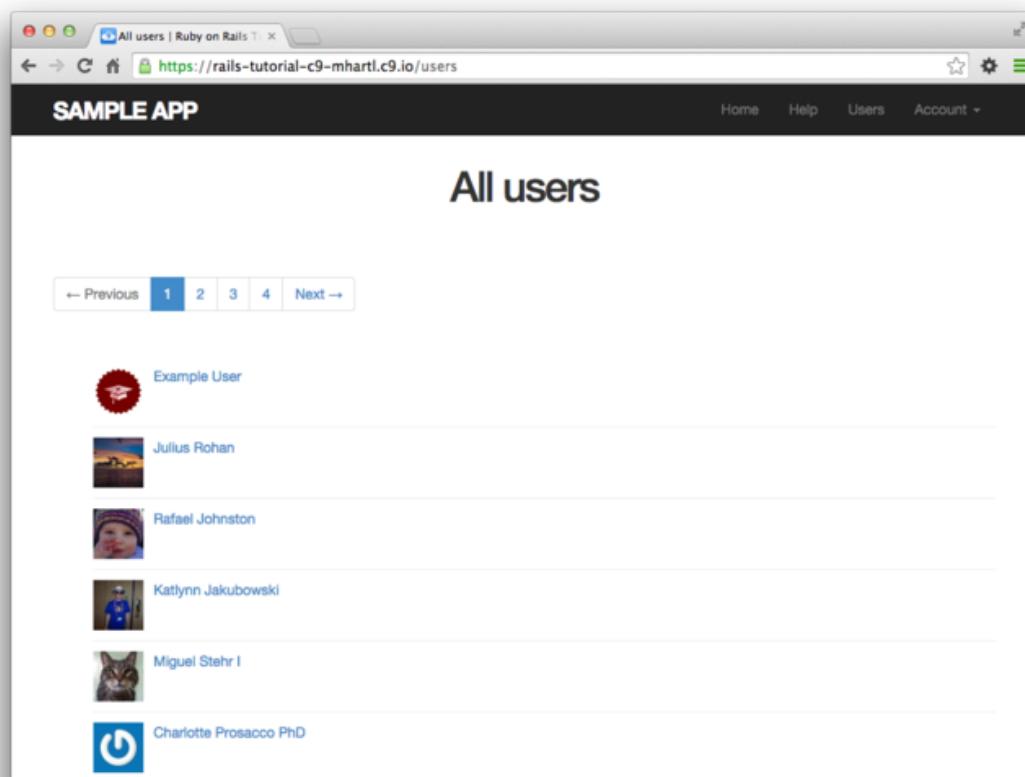


Figura 9.11: La página del listado de usuarios con paginación.

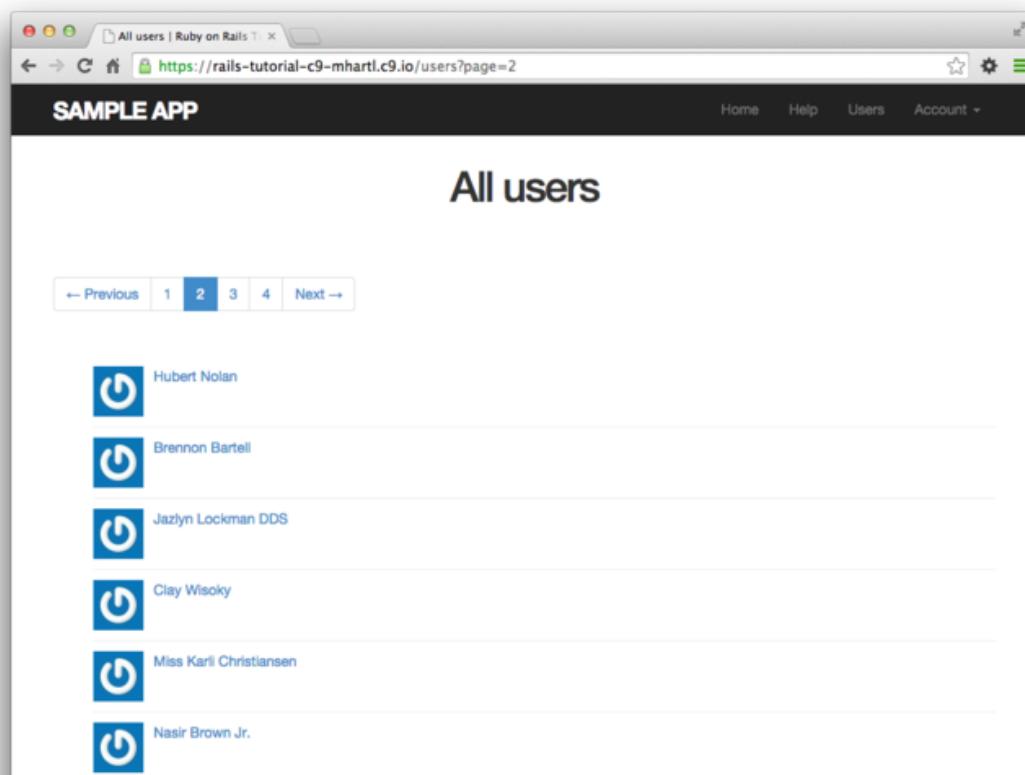


Figura 9.12: La página 2 del listado de usuarios.

como vimos con el atributo **password_digest** del fixture de usuario, este tipo de archivos permiten utilizar Ruby embebido, lo que significa que podemos crear los usuarios adicionales como se muestra en el [Listado 9.43](#). ([Listado 9.43](#) también crea un par de otros usuarios nombrados para futura referencia.)

Listado 9.43: Agregando 30 usuarios extra en el fixture.

test/fixtures/users.yml

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>

lana:
  name: Lana Kane
  email: hands@example.gov
  password_digest: <%= User.digest('password') %>

mallory:
  name: Mallory Archer
  email: boss@example.gov
  password_digest: <%= User.digest('password') %>

<% 30.times do |n| %>
user_<%= n %>:
  name: <%= "User #{n}" %>
  email: <%= "user-#{n}@example.com" %>
  password_digest: <%= User.digest('password') %>
<% end %>
```

Con el archivo fixtures definido como en el [Listado 9.43](#), estamos listos para escribir una prueba del listado de usuarios. Primero generamos la prueba relevante:

```
$ rails generate integration_test users_index
  invoke test_unit
    create  test/integration/users_index_test.rb
```

La prueba en sí misma involucra la verificación de que un **div** con la clase **pagination** requerida existe y verifica que la primera página del listado está presente. El resultado aparece en el Listado 9.44.

Listado 9.44: Una prueba para el listado de usuarios, incluyendo paginación.

VERDE

```
test/integration/users_index_test.rb

require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "index including pagination" do
    log_in_as(@user)
    get users_path
    assert_template 'users/index'
    assert_select 'div.pagination'
    User.paginate(page: 1).each do |user|
      assert_select 'a[href=?]', user_path(user), text: user.name
    end
  end
end
```

The result should be a VERDE test suite:

Listado 9.45: VERDE

```
$ bundle exec rake test
```

9.3.5 Refactorización parcial

El listado de usuarios paginados está copleto, pero hay una mejora que no puedo resistirme a incluir: Rails tiene algunas herramientas increíblemente rápidas para hacer vistas compactas, y en esta sección refactorizaremos la página del listado para utilizarlas. Como nuestro código está bien probado, podemos refactorizar con confianza, seguros de que es poco probable que descompongamos la funcionalidad del sitio.

El primer paso en nuestra refactorización es reemplazar el usuario `1i` del Listado 9.41 con una llamada a `render` (Listado 9.46).

Listado 9.46: El primer intento de refactorización en la página del listado.

`app/views/users/index.html.erb`

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <% @users.each do |user| %>
    <%= render user %>
  <% end %>
</ul>

<%= will_paginate %>
```

Aquí invocamos `render` no sobre una cadena con el nombre del parcial, sino con una variable `user` de la clase `User`.⁹ En este contexto, Rails automáticamente busca un parcial llamado `_user.html.erb`, el cual debemos crear (Listado 9.47).

Listado 9.47: Un parcial para mostrar a un solo usuario.

`app/views/users/_user.html.erb`

```
<li>
  <%= gravatar_for user, size: 50 %>
  <%= link_to user.name, user %>
</li>
```

Esto definitivamente es una mejora, pero podemos mejorar aún más: podemos invocar `render` directamente en la variable `@users` (Listado 9.48).

⁹El nombre `user` es irrelevante —pudimos haber escrito `@users.each do |foobar|` y luego utilizar `render foobar`. La clave es la *class* del objeto—en este caso, `User`.

Listado 9.48: El listado de usuarios completamente refactorizado. **VERDE**
app/views/users/index.html.erb

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <%= render @users %>
</ul>

<%= will_paginate %>
```

Aquí Rails deduce que **@users** es una lista de objetos de tipo **User**; y más aún, cuando invocamos una colección de usuarios, Rails automáticamente itera sobre ellos y despliega cada uno con el parcial **_user.html.erb**. El resultado es el código impresionantemente compacto del [Listado 9.48](#).

Como con cualquier refactorización, debería verificar que el conjunto de pruebas está aún en **VERDE** luego de cambiar el código de la aplicación:

Listado 9.49: **VERDE**

```
$ bundle exec rake test
```

9.4 Borrando usuarios

Ahora que la página del listado de usuarios está completa, hay sólo una acción REST canónica pendiente: **destroy**. En esta sección, agregaremos enlaces para eliminar usuarios, como se bosquejó en la [Figura 9.13](#), y definiremos la acción **destroy** necesaria para lograr el borrado. Pero primero, crearemos la clase de usuarios administrativos, o *admins*, autorizados para tal fin.

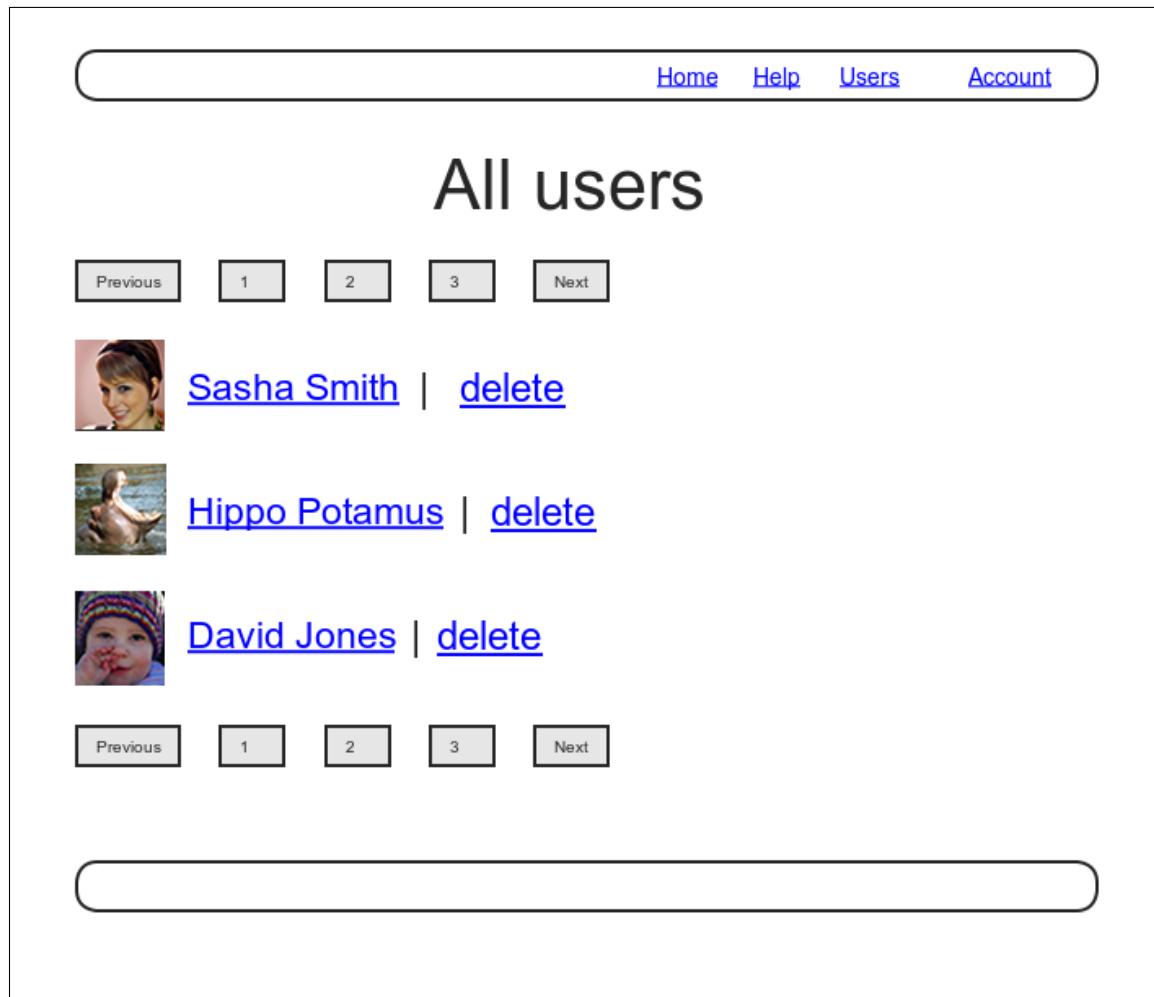


Figura 9.13: Un bosquejo del listado de usuarios con los enlaces para borrarlos.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string
admin	boolean

Figura 9.14: El modelo User con un atributo booleano **admin** agregado.

9.4.1 Usuarios administrativos

Identificaremos a los privilegiados usuarios administrativos con un atributo booleano **admin** en el modelo User, lo cual nos lleva automáticamente al método booleano **admin?** para probar el status de administrador. El modelo de datos resultante aparece en la Figura 9.14.

Como es usual, agregamos el atributo **admin** con una migración, indicando el tipo **booleano** en la línea de comandos:

```
$ rails generate migration add_admin_to_users admin:boolean
```

La migración agrega la columna **admin** a la tabla **users**, como se observa en el Listado 9.50. Observe que hemos agregado el argumento **default: false** a **add_column** en el Listado 9.50, lo que significa que los usuarios no serán administradores por default. (Sin el argumento **default: false**, **admin** sería **nil** por default, lo cual aún es **falso**, por lo que este paso no es estrictamente necesario. Aunque es más explícito, y comunica nuestra intención más claramente tanto a Rails como a los lectores de nuestro código.)

Listado 9.50: La migración para agregar un atributo booleano `admin` a los usuarios.

```
db/migrate/[timestamp]_add_admin_to_users.rb
```

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

A continuación, migramos como es usual:

```
$ bundle exec rake db:migrate
```

Como es esperado, Rails descubre la naturaleza booleana del atributo `admin` y automáticamente añade el método `admin?` con el signo de interrogación:

```
$ rails console --sandbox
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

Aquí hemos utilizado el método `toggle!` para cambiar el atributo `admin` de **falso a verdadero**.

Como paso final, actualicemos nuestros datos semilla para crear el primer usuario admin por default ([Listado 9.51](#)).

Listado 9.51: El código para los datos semilla con un usuario admin.

```
db/seeds.rb
```

```
User.create!(name: "Example User",
            email: "example@railstutorial.org",
            password: "foobar",
            password_confirmation: "foobar",
```

```

    admin: true)

99.times do |n|
  name = Faker::Name.name
  email = "example-#{n+1}@railstutorial.org"
  password = "password"
  User.create!(name: name,
              email: email,
              password: password,
              password_confirmation: password)
end

```

Luego reiniciamos nuestra base de datos:

```

$ bundle exec rake db:migrate:reset
$ bundle exec rake db:seed

```

Recordando los parámetros fuertes

Puede haber notado que el [Listado 9.51](#) crea un usuario como administrador al incluir `admin: true` en el arreglo hash de inicialización. Esto subraya el peligro de exponer nuestros objetos a la red salvaje: si sólo pasamos un arreglo de inicialización en una petición web arbitraria, un usuario malicioso podría enviar una petición PATCH como sigue:¹⁰

```
patch /users/17?admin=1
```

Esta petición haría que el usuario 17 se convirtiera en admin, lo cual podría ser hueco de seguridad potencialmente serio.

A causa de este peligro, es esencial que sólo actualicemos los atributos que son seguros de editar a través de la red. Como se observa en la [Sección 7.3.2](#), esto se logra utilizando *parámetros fuertes* al invocar `require` y `permit` en el arreglo hash `params`:

¹⁰Las herramientas de línea de comandos tales como `curl` pueden emitir peticiones PATCH de esta forma.

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

Observe en particular que **admin** no está en la lista de atributos permitidos. Esto es lo que evita que usuarios arbitrarios se otorguen acceso administrativo a nuestra aplicación. Debido a su importancia, es buena idea escribir una prueba para cualquier atributo que no sea editable, y la escritura de tal prueba para el atributo **admin** se deja como ejercicio (Sección 9.6).

9.4.2 La acción `destroy`

El paso final que necesitamos para completar el recurso Users es agregar los enlaces para borrar usuarios y una acción **destroy**. Empezaremos por agregar el enlace para borrar cada usuario en la página del listado de usuarios, restringiendo el acceso a los usuarios administrativos. Los enlaces resultantes "**delete**" serán mostrados únicamente si el usuario actual tiene privilegios administrativos (Listado 9.52).

Listado 9.52: Enlaces para borrar usuarios (visibles sólo para administrativos).

```
app/views/users/_user.html.erb
```

```
<li>
  <%= gravatar_for user, size: 50 %>
  <%= link_to user.name, user %>
  <% if current_user.admin? && !current_user?(user) %>
    | <%= link_to "delete", user, method: :delete,
      data: { confirm: "You sure?" } %>
  <% end %>
</li>
```

Observe el argumento **method: :delete**, que se encarga del enlace para emitir las peticiones DELETE necesarias. También hemos encapsulado cada enlace dentro de una sentencia **if** de forma que sólo los usuarios administrativos puedan verla. El resultado para nuestro administrador aparece en la Figura 9.15.

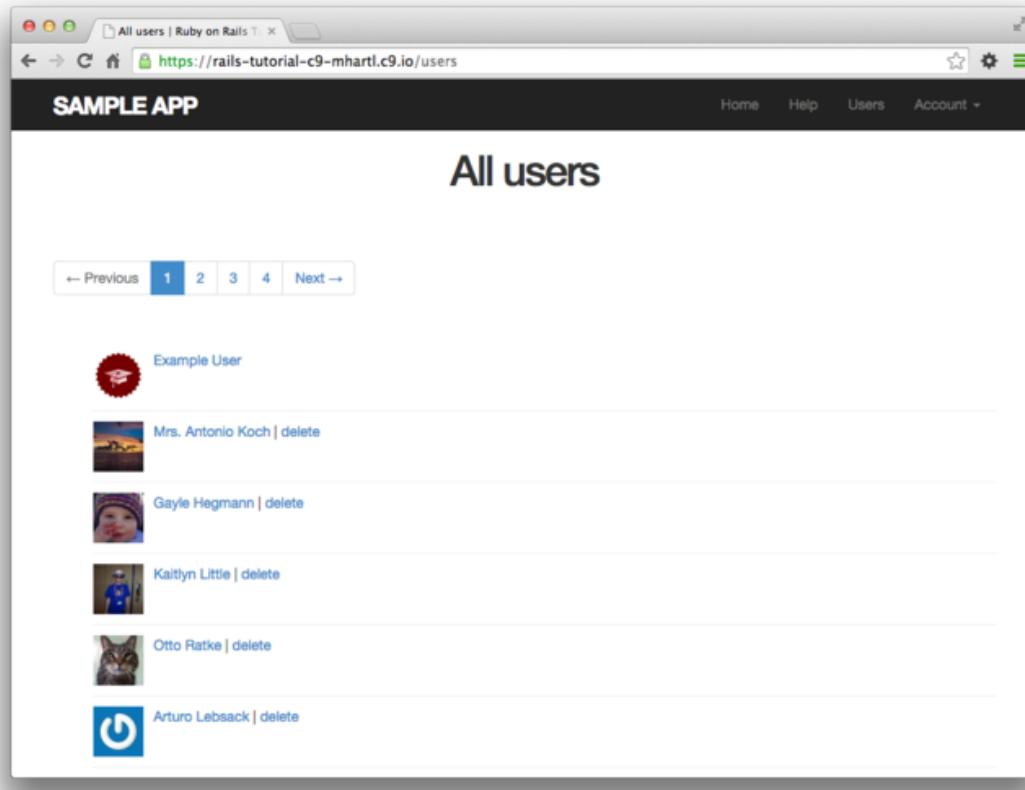


Figura 9.15: El listado de usuarios con enlaces para borrar.

Los navegadores Web no pueden emitir peticiones DELETE de forma nativa, por lo que Rails las simula con JavaScript. Esto significa que los enlaces para borrar no funcionarán si el usuario tiene deshabilitado en su navegador el uso de JavaScript. Si usted debe aceptar navegadores en estas condiciones, puede simular una petición DELETE usando un formulario y una petición POST, lo cual funciona aún sin JavaScript.¹¹

Para hacer que los enlaces funcionen, necesitamos agregar una acción **destroy** (Tabla 7.1), lo cual encuentra al usuario correspondiente y lo destruye con el método **destroy** de Active Record, finalmente redireccionamos a la página

¹¹Vea el RailsCast acerca de “Destruyendo sin JavaScript” para mayor detalle.

del listado, como se muestra en el Listado 9.53. Como los usuarios tienen que estar en sesión para borrar usuarios, Listado 9.53 también agrega `:destroy` al filtro `logged_in_user`.

Listado 9.53: Agregando una acción `destroy` que funciona.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update, :destroy]
  before_action :correct_user,   only: [:edit, :update]
  .

  def destroy
    User.find(params[:id]).destroy
    flash[:success] = "User deleted"
    redirect_to users_url
  end
  .
  .
  .
end
```

Observe que la acción `destroy` utiliza métodos encadenados para combinar `find` y `destroy` en una sola línea:

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

Como se construyó, sólo los usuarios con privilegios administrativos pueden destruir usuarios a través de la red puesto que sólo ellos pueden tener acceso a los enlaces de borrado, pero aún hay un terrible hueco de seguridad: un atacante suficientemente sofisticado podría simplemente emitir una petición DELETE directamente desde línea de comandos para eliminar a cualquier usuario de nuestro sitio. Para asegurar el sitio apropiadamente, también necesitamos controlar el acceso a la acción `destroy`, de forma que *sólo* administradores puedan borrar usuarios.

Como en las Secciones 9.2.1 y 9.2.2, pondremos en vigor el control de acceso usando un filtro *previo*, esta vez para restringir el acceso a la acción

destroy únicamente para los administrativos. El filtro resultante **admin_user** aparece en el Listado 9.54.

Listado 9.54: Un filtro para restringir la acción **destroy** a los administrativos.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update, :destroy]
  before_action :correct_user,   only: [:edit, :update]
  before_action :admin_user,    only: [:destroy]

  .
  .
  .

  private

  .
  .
  .

  # Confirms an admin user.
  def admin_user
    redirect_to(root_url) unless current_user.admin?
  end
end
```

9.4.3 Pruebas al borrado de usuario

Con algo tan peligroso como destruir usuarios, es importante tener buenas pruebas para el comportamiento esperado. Empezaremos haciendo que uno de nuestros usuarios del archivo fixture sea administrador, como se muestra en el Listado 9.55.

Listado 9.55: Haciendo que uno de los usuarios del archivo fixture sea administrador.

test/fixtures/users.yml

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>
  admin: true
```

```

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>

lana:
  name: Lana Kane
  email: hands@example.gov
  password_digest: <%= User.digest('password') %>

mallory:
  name: Mallory Archer
  email: boss@example.gov
  password_digest: <%= User.digest('password') %>

<% 30.times do |n| %>
user_<%= n %>:
  name: <%= "User #{n}" %>
  email: <%= "user-#{n}@example.com" %>
  password_digest: <%= User.digest('password') %>
<% end %>

```

Siguiendo la práctica de la Sección 9.2.1, pondremos pruebas de control de acceso a nivel de acción en el archivo de prueba del controlador Users. Como con la prueba de cierre de sesión del Listado 8.28, utilizaremos **delete** para emitir una petición DELETE directamente a la acción **destroy**. Necesitamos verificar dos casos: primero, que los usuarios que no están en sesión sean redirigidos a la página de inicio de sesión; segundo, que los usuarios que están en sesión pero no tienen privilegios administrativos sean redirigidos a la página principal. El resultado se muestra en el Listado 9.56.

Listado 9.56: Pruebas a nivel de acción para el control de acceso administrativo. **VERDE**

```

test/controllers/users_controller_test.rb

require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user      = users(:michael)
    @other_user = users(:archer)
  end

  .

```

```

.
.
test "should redirect destroy when not logged in" do
  assert_no_difference 'User.count' do
    delete :destroy, id: @user
  end
  assert_redirected_to login_url
end

test "should redirect destroy when logged in as a non-admin" do
  log_in_as(@other_user)
  assert_no_difference 'User.count' do
    delete :destroy, id: @user
  end
  assert_redirected_to root_url
end
end

```

Observe que el [Listado 9.56](#) también asegura que el conteo de usuarios no cambia, mediante el método **assert_no_difference** (que vimos anteriormente en el [Listado 7.21](#)).

Las pruebas del [Listado 9.56](#) verifican el comportamiento en el caso de un usuario no autorizado (no administrativo), pero también queremos verificar que un usuario administrativo puede utilizar un enlace de borrado para eliminar exitosamente a un usuario. Puesto que los enlaces de borrado aparecen en el listado de usuarios, agregaremos estas pruebas al archivo de pruebas del listado de usuarios como se observa en el [Listado 9.44](#). La única parte realmente truculenta es la verificación de que un usuario es borrado cuando un administrador da click en un enlace de borrado, lo cual lograremos como sigue:

```

assert_difference 'User.count', -1 do
  delete user_path(@other_user)
end

```

Esto utiliza el método **assert_difference** que vimos por primera vez en el [Listado 7.26](#) cuando creamos un usuario, esta vez verificando que un usuario es *destruido* al verificar que **User.count** cambia por -1 cuando emitimos una petición **delete** a la ruta respectiva del usuario.

Poniendo todo junto nos da las pruebas de paginación y borrado del [Listado 9.57](#), que incluyen pruebas tanto para usuarios administrativos como para los que no lo son.

Listado 9.57: Una prueba de integración para los enlaces de borrado y la destrucción de usuarios. **VERDE**

```
test/integration/users_index_test.rb

require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest

  def setup
    @admin      = users(:michael)
    @non_admin = users(:archer)
  end

  test "index as admin including pagination and delete links" do
    log_in_as(@admin)
    get users_path
    assert_template 'users/index'
    assert_select 'div.pagination'
    first_page_of_users = User.paginate(page: 1)
    first_page_of_users.each do |user|
      assert_select 'a[href=?]', user_path(user), text: user.name
      unless user == @admin
        assert_select 'a[href=?]', user_path(user), text: 'delete'
      end
    end
    assert_difference 'User.count', -1 do
      delete user_path(@non_admin)
    end
  end

  test "index as non-admin" do
    log_in_as(@non_admin)
    get users_path
    assert_select 'a', text: 'delete', count: 0
  end
end
```

Observe que el [Listado 9.57](#) verifica los enlaces de borrado correctos, incluyendo evitar la prueba si el usuario es no administrativo (el cual carece de un enlace de borrado debido al [Listado 9.52](#)).

En este punto, nuestro código de borrado está bien probado, y el conjunto de pruebas debería estar en **VERDE**:

Listado 9.58: VERDE

```
$ bundle exec rake test
```

9.5 Conclusión

Hemos recorrido un largo camino desde que empezamos el controlador de usuarios en la [Sección 5.4](#). Esos usuarios no podían siquiera registrarse en el sitio, iniciar sesión, cerrarla, visualizar sus perfiles, editar sus datos o ver un listado de todos los usuarios—y algunos podían incluso destruir a otros usuarios.

Como actualmente se encuentra, la aplicación de ejemplo conforma una base sólida para cualquier sitio web que requiera usuarios con autenticación y autorización. En el [Capítulo 10](#), agregaremos dos detalles adicionales: un enlace para activar la cuenta de los usuarios recién registrados (verificando en el proceso que la dirección de correo electrónico sea válida) y reestableciendo la contraseña de los usuarios que la han olvidado.

Antes de continuar, asegúrese de mezclar todos los cambios en la rama principal:

```
$ git add -A
$ git commit -m "Finish user edit, update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
$ git push
```

También puede desplegar la aplicación y poblar la base de datos de producción con usuarios de ejemplo (utilizando la tarea **pg:reset** para reiniciar la base de datos de producción):

```
$ bundle exec rake test
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
$ heroku restart
```

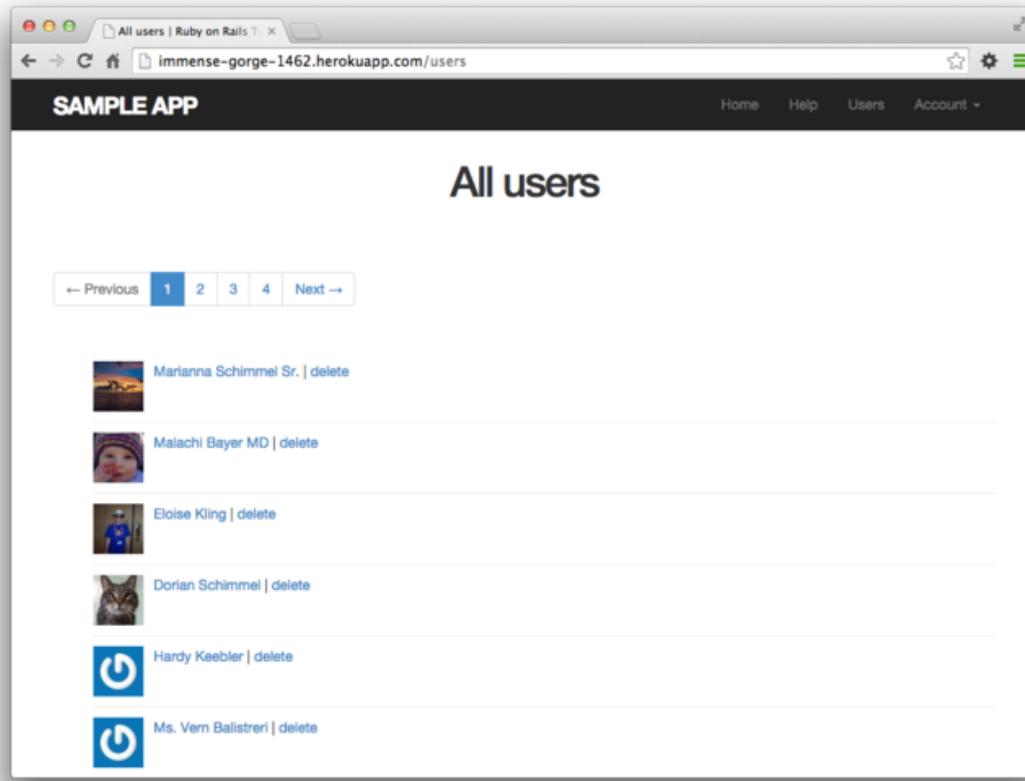


Figura 9.16: Los usuarios de ejemplo en producción.

Por supuesto, en un sitio real usted probablemente no querría poblar con datos de muestra, pero lo incluyo aquí con propósitos ilustrativos ([Figura 9.16](#)). De paso, el orden de los usuarios de ejemplo de la [Figura 9.16](#) puede variar, y en mi sistema no coincide con la versión local de la [Figura 9.11](#); esto es porque no hemos especificado un orden por default para cuando los usuarios son traídos de la base de datos, por lo que el orden en que se muestran es dependiente de la base de datos. Esto no es muy importante para los usuarios, pero lo será para los microposts, y tomaremos nota de este tema en la [Sección 11.1.4](#).

9.5.1 Qué aprendimos en este capítulo

- Los usuarios pueden ser actualizados utilizando un formulario de edición, el cual envía una petición PATCH a la acción `update`.
- La actualización segura através de internet es aplicada usando parámetros fuertes.
- Los filtros que se definen para invocar métodos antes que ciertas acciones del controlador nos permiten ejecutar tales métodos de una forma están-dar.
- Implementamos la autorización utilizando los filtros descritos en el punto anterior.
- Las pruebas de autorización utilizan tanto comandos de bajo nivel para enviar peticiones HTTP particulares directamente a las acciones del con-trolador, como pruebas de integración de alto nivel.
- El redireccionamiento amigable permite dirigir a los usuarios a donde quieren ir luego de iniciar sesión.
- La página del listado de usuarios muestra a todos los usuarios, una página a la vez.
- Rails utiliza el archivo estándar `db/seeds.rb` para alimentar la base de datos con datos de ejemplo usando `rake db:seed`.
- Ejecutar `render @users` automáticamente invoca el parcial `_user.-html.erb` en cada usuario de la colección.
- Un atributo booleano `admin` de usuario automáticamente nos propor-ciona un método booleano `user.admin?`.
- Los administradores pueden eliminar usuarios a través de internet me-diante los enlaces de borrado que emiten peticiones DELETE a la acción `destroy` del controlador Users.

- Podemos crear un gran número de usuarios de prueba utilizando Ruby embebido dentro de los archivos fixtures.

9.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Escriba una prueba que asegure que el redireccionamiento amigable sólo direcciona a la URL dada la primera vez. En inicios de sesión subsiguientes, la URL de redireccionamiento debería volver a ser la predeterminada (es decir, la página del perfil). *Sugerencia:* Agregue a la prueba del Listado 9.26 la verificación del valor correcto de `session[:forward-ing_url]`.
2. Escriba una prueba de integración para todos los enlaces de la estructura de diseño, incluyendo el comportamiento apropiado para los usuarios que están en sesión y los que no lo están. *Sugerencia:* Agregue a la prueba del Listado 5.25 el auxiliar `log_in_as`.
3. Al emitir una petición PATCH directamente al método `update` como se muestra en el Listado 9.59, verifique que el atributo `admin` no es editable através de internet. Para asegurar que su prueba está cubriendo el escenario correcto, su primer paso debería ser *agregar admin* a la lista de parámetros permitidos en `user_params` de forma que la prueba inicial esté en ROJO.
4. Remueva el código duplicado del formulario refactorizando las vistas `new.html.erb` y `edit.html.erb` para utilizar el parcial del Listado 9.60, como se muestra en los Listados 9.61 y 9.62. Observe el uso del método

provide, el cual utilizamos previamente en la Sección 3.4.3 para eliminar duplicados en la estructura de diseño.¹²

Listado 9.59: Probando que el atributo **admin** está restringido.

test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user      = users(:michael)
    @other_user = users(:archer)
  end

  .

  .

  test "should redirect update when logged in as wrong user" do
    log_in_as(@other_user)
    patch :update, id: @user, user: { name: @user.name, email: @user.email }
    assert_redirected_to root_url
  end

  test "should not allow the admin attribute to be edited via the web" do
    log_in_as(@other_user)
    assert_not @other_user.admin?
    patch :update, id: @other_user, user: { password:           FILL_IN,
                                             password_confirmation: FILL_IN,
                                             admin: FILL_IN }
    assert_not @other_user.FILL_IN.admin?
  end

  .
  .
  .
end
```

Listado 9.60: Un parcial para los formularios **new** y **edit**.

app/views/users/_form.html.erb

```
<%= form_for(@user) do |f| %>
  <%= render 'shared/error_messages', object: @user %>
```

¹²Agradezco a José Carlos Montero Gómez por su sugerencia que reduce aún más el código duplicado en los parciales **new** y **edit**.

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

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

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

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

<%= f.submit yield(:button_text), class: "btn btn-primary" %>
<% end %>
```

Listado 9.61: La vista de registro con un parcial.*app/views/users/new.html.erb*

```
<% provide(:title, 'Sign up') %>
<% provide(:button_text, 'Create my account') %>
<h1>Sign up</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
  </div>
</div>
```

Listado 9.62: La vista de edición con un parcial.*app/views/users/edit.html.erb*

```
<% provide(:title, 'Edit user') %>
<% provide(:button_text, 'Save changes') %>
<h1>Update your profile</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="http://gravatar.com/emails" target="_blank">Change</a>
    </div>
  </div>
</div>
```


Capítulo 10

Activación de la cuenta y reinicio de contraseña

En el [Capítulo 9](#), terminamos de elaborar un recurso básico Users (rellenando todas las acciones REST de la [Tabla 7.1](#)), junto con un sistema de autenticación y autorización flexible. En este capítulo, agregaremos los toques finales a este sistema agregando dos funcionalidades relacionados entre sí: la activación de la cuenta (la cual verifica la dirección electrónica de un nuevo usuario) y el reinicio de la contraseña (para los usuarios que olvidaron sus contraseñas). Cada una de estas características involucran la creación de un nuevo recurso, por lo que nos proporcionan la oportunidad de revisar ejemplos adicionales de controladores, ruteos y migraciones de la base de datos. Durante el proceso, también podremos aprender cómo enviar correos electrónicos en Rails, tanto en desarrollo como en producción. Finalmente, las dos características se complementan mutuamente: el reinicio de contraseña requiere enviar un correo electrónico al usuario con el enlace de reinicio, y la validez de la dirección electrónica es confirmada por la activación inicial de la cuenta.¹

¹Técnicamente, un usuario podría actualizar su cuenta con una dirección electrónica equivocada usando la funcionalidad de configuración de la cuenta de la [Sección 9.1](#), pero la implementación actual nos proporciona el mayor beneficio de la verificación de la dirección electrónica sin mucho esfuerzo.

10.1 Activación de la cuenta

En este momento, los usuarios recién registrados inmediatamente tienen acceso total a sus cuentas ([Capítulo 7](#)). En esta sección, implementaremos un paso de activación de la cuenta para verificar que el usuario tiene control sobre la cuenta de correo electrónico que utilizó para registrarse. Esto involucra asociar un token de activación y una digestión con el usuario, enviando al correo electrónico un enlace que incluye el token, y que activa al usuario luego de acceder al enlace.

Nuestra estrategia para manejar la activación de la cuenta se asemeja al inicio de sesión ([Sección 8.2](#)) y especialmente a los usuarios recordados ([Sección 8.4](#)). La secuencia básica es la siguiente:

1. Iniciar a los usuarios en un estado “desactivado”.
2. Cuando un usuario se registra, generar un token de activación y su digestión de activación correspondiente.
3. Guardar la digestión de activación en la base de datos, y luego enviar un correo al usuario con el enlace que contiene el token de activación y la dirección electrónica del usuario.²
4. Cuando el usuario visita el enlace, buscar al usuario mediante su dirección electrónica y luego autenticar el token comparándolo con la digestión de activación.
5. Si el usuario es autenticado, cambiar el status de “desactivado” a “activo”.

Debido a la similitud con las contraseñas y los tokens recordados, podremos reutilizar muchas de las ideas para la activación de la cuenta (así como para el reinicio de la contraseña), incluyendo los métodos `User.digest` y `User.new_token` y una versión modificada del método `user.authenticated?`. La Tabla 10.1

²Podríamos utilizar el id del usuario, puesto que éste ya está expuesto en las URLs de nuestra aplicación, pero utilizar la dirección electrónica es más robusta a futuro en caso de que queramos ofuscar los ids de usuario por alguna razón (tal como prevenir que los competidores sepan cuántos usuarios tiene nuestra aplicación, por ejemplo).

find by	string	digest	authentication
email	password	password_digest	authenticate(password)
id	remember_token	remember_digest	authenticated?(:remember, token)
email	activation_token	activation_digest	authenticated?(:activation, token)
email	reset_token	reset_digest	authenticated?(:reset, token)

Tabla 10.1: La analogía entre login, activación de la cuenta, reinicio de contraseña y las sesiones recordadas.

ilustra la analogía (incluyendo el reinicio de la contraseña de la Sección 10.2). Definiremos la versión generalizada del método `authenticated?` de la Tabla 10.1 en la Sección 10.1.3.

Como es usual, crearemos una rama en el sistema controlador de versiones para la nueva funcionalidad. Como veremos en la Sección 10.3, la activación de la cuenta y el reinicio de la contraseña incluyen alguna configuración común de la dirección electrónica, la cual aplicaremos a ambas funcionalidades antes de mezclar con la rama principal. Como resultado, es conveniente utilizar una rama común:

```
$ git checkout master
$ git checkout -b account-activation-password-reset
```

10.1.1 El recurso para la activación de la cuenta

Como con las sesiones (Sección 8.1), modelaremos las activaciones de la cuenta como un recurso aún cuando no estén asociadas con un modelo de Active Record. En su lugar, incluiremos los datos relevantes (incluyendo la activación del token y el status de activación) en el modelo de usuario. Sin embargo, interactuaremos con las activaciones de la cuenta mediante una URL REST estándar. Como el enlace de activación estará modificando el status de activación del usuario, planeamos utilizar la acción `edit`.³ Esto requiere de un controlador de Acti-

³Podría incluso tener más sentido utilizar la acción `update`, pero el enlace de activación necesita ser enviado en un correo electrónico y de aquí debería involucrar un click regular en el navegador, lo cual emite una petición GET en vez de una petición PATCH requerida por la acción `update`.

vación de Cuenta, el cual podemos generar como sigue:⁴

```
$ rails generate controller AccountActivations --no-test-framework
```

Observe que hemos incluído una bandera para evitar la generación de las pruebas. Esto es porque no necesitamos las pruebas del controlador (preferimos en su lugar utilizar una prueba de integración ([Sección 10.1.4](#))), por lo que es conveniente omitirlas.

El correo de activación incluirá una URL de la forma

```
edit_account_activation_url(activation_token, ...)
```

lo que significa que necesitaremos una ruta nombrada para la acción `edit`. Podemos arreglar esto con la línea `resources` que se muestra en el [Listado 10.1](#).

Listado 10.1: Agregando un recurso para las activaciones de la cuenta.

config/routes.rb

```
Rails.application.routes.draw do
  root      'static_pages#home'
  get    'help'    => 'static_pages#help'
  get    'about'   => 'static_pages#about'
  get    'contact' => 'static_pages#contact'
  get    'signup'   => 'users#new'
  get    'login'    => 'sessions#new'
  post   'login'   => 'sessions#create'
  delete 'logout'  => 'sessions#destroy'
  resources :users
  resources :account_activations, only: [:edit]
end
```

A continuación, necesitamos un token de activación único para activar a los usuarios. La seguridad es menos preocupante con la activación de las cuentas que con las contraseñas, los tokens recordados o (como veremos en la [Sección 10.2](#)) el reinicio de las contraseñas—todo lo cual le daría control total de la

⁴Como estaremos utilizando una acción `edit`, podríamos incluir `edit` en la línea de comandos, pero esto generaría tanto una vista como una prueba, pero no necesitamos ninguna de las dos.

cuenta a un atacante si los obtuviera—pero existen aún algunos escenarios en los que un token de activación no digerido podría comprometer una cuenta.⁵ Entonces, siguiendo el ejemplo de los tokens recordados de la Sección 8.4, crearemos parejas de tokens virtuales expuestos públicamente con su digestión en la base de datos. De esta forma podemos tener acceso al token de activación utilizando

```
user.activation_token
```

y autenticando al usuario con código como

```
user.authenticated?(:activation, token)
```

(Esto requerirá una modificación al método `authenticated?` definido en el Listado 8.33.) También agregaremos un atributo booleano `activated`, el cual nos permitirá determinar si un usuario está activado usando la misma clase de método booleano auto-generado que vimos en la Sección 9.4.1:

```
if user.activated? ...
```

Finalmente, aunque no lo utilizaremos en este tutorial, registraremos la fecha y hora de la activación en caso de que la necesitemos para futura referencia. El modelo de datos completo aparece en la Figura 10.1.

La migración que realizaremos al modelo de datos de la Figura 10.1 agrega los tres atributos en la línea de comandos:

```
$ rails generate migration add_activation_to_users \
> activation_digest:string activated:boolean activated_at:datetime
```

⁵Por ejemplo, un atacante con acceso a la base de datos podría activar inmediatamente las cuentas recién creadas, logrando iniciar sesión como el usuario y luego cambiando la contraseña para obtener control.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string
admin	boolean
activation_digest	string
activated	boolean
activated_at	datetime

Figura 10.1: El modelo **User** con los nuevos atributos para la activación de la cuenta.

(Observe que el > en la segunda línea es un carácter de “continuación de línea” insertado automáticamente por la consola, y no debe ser escrito literalmente.) Como con el atributo **admin** (Listado 9.50), agregaremos un valor booleano **falso** por default al atributo **activated**, como se muestra en el Listado 10.2.

Listado 10.2: Una migración para la activación de la cuenta (con índice incluido).

```
db/migrate/[timestamp]_add_activation_to_users.rb

class AddActivationToUsers < ActiveRecord::Migration
  def change
    add_column :users, :activation_digest, :string
    add_column :users, :activated, :boolean, default: false
    add_column :users, :activated_at, :datetime
  end
end
```

Luego podemos aplicar la migración como es usual:

```
$ bundle exec rake db:migrate
```

Como cada usuario recién registrado requerirá de activación, podemos asignar un token de activación y una digestión a cada objeto usuario antes de que sea creado. Vimos un escenario similar en la Sección 6.2.5, donde necesitamos convertir una dirección electrónica a minúsculas antes de guardar el usuario en la base de datos. En ese caso, utilizamos una invocación a **before_save** junto con el método **downcase** (Listado 6.31). Una invocación a **before_save** es automáticamente realizada antes de que el objeto sea guardado, lo cual incluye tanto la creación del objeto como las actualizaciones, pero en el caso de la digestión de activación sólo queremos que la invocación se realice cuando el usuario es creado. Esto requiere una invocación **before_create**, la cual definiremos como sigue:

```
before_create :create_activation_digest
```

Este código, llamado una *referencia a un método*, se encarga de que Rails busque un método llamado `create_activation_digest` y lo ejecute antes de crear el usuario. (En el Listado 6.31, pasamos `before_save` a un bloque específico, pero la técnica de referencia al método es preferida usualmente.) Como el método `create_activation_digest` es sólo utilizado internamente por el modelo `User`, no necesitamos exponerlo a usuarios externos; como vimos en la Sección 7.3.2, la forma Ruby de implementar esto es utilizando la palabra reservada `private`:

```
private

def create_activation_digest
  # Create the token and digest.
end
```

Todos los métodos definidos en una clase luego de `private` son automáticamente ocultos, como se muestra en esta sesión de consola:

```
$ rails console
>> User.first.create_activation_digest
NoMethodError: private method `create_activation_digest' called for #<User>
```

El propósito de la invocación `before_create` es asignar el token y su digestión correspondiente, lo cual podemos hacer como sigue:

```
self.activation_token = User.new_token
self.activation_digest = User.digest(activation_token)
```

Este código simplemente reutiliza el token y los métodos de digestión utilizados para el token recordado, como puede notar al compararlo con el método `remember` del Listado 8.32:

```
# Remembers a user in the database for use in persistent sessions.
def remember
  self.remember_token = User.new_token
  update_attribute(:remember_digest, User.digest(remember_token))
end
```

La principal diferencia es el uso de `update_attribute` en este último caso. El motivo de la diferencia es que los tokens recordados y las digestiones son creados para usuarios que ya existen en la base de datos, mientras que la invocación a `before_create` sucede *antes* de que el usuario haya sido creado. Como resultado de la invocación, cuando un usuario nuevo es definido con `User.new` (como en el registro de usuarios, [Listado 7.17](#)), éste automáticamente tendrá ambos atributos `activation_token` y `activation_digest`; como éste último está asociado con una columna en la base de datos ([Figura 10.1](#)), será automáticamente escrito cuando el usuario sea guardado.

Juntando los temas discutidos anteriormente obtenemos el modelo `User` que se muestra en el [Listado 10.3](#). Como se requiere por la naturaleza virtual del token de activación, agregaremos un segundo `attr_accessor` a nuestro modelo. Observe también que hemos aprovechado la oportunidad de utilizar una referencia al método para pasar a minúsculas la dirección electrónica.

Listado 10.3: Agregando el código para la activación de la cuenta al modelo `User`.

`User`. VERDE

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessor :remember_token, :activation_token
  before_save   :downcase_email
  before_create :create_activation_digest
  validates :name,  presence: true, length: { maximum: 50 }

  .
  .
  .

  private

    # Converts email to all lower-case.
    def downcase_email
      self.email = email.downcase
    end

    # Creates and assigns the activation token and digest.
    def create_activation_digest
      self.activation_token  = User.new_token
      self.activation_digest = User.digest(activation_token)
    end
end
```

Antes de continuar, deberíamos actualizar nuestros datos semilla y los archivos

fixtures de forma que nuestro ejemplo y los usuarios de prueba estén activados inicialmente, como se muestra en los Listados 10.4 y 10.5. (El método `Time.zone.now` es un método auxiliar preconstruído en Rails que regresa la fecha y hora actuales, tomando en cuenta la zona horaria del servidor.)

Listado 10.4: Activando usuarios semilla por default.

`db/seeds.rb`

```
User.create!(name: "Example User",
            email: "example@railstutorial.org",
            password: "foobar",
            password_confirmation: "foobar",
            admin: true,
            activated: true,
            activated_at: Time.zone.now)

99.times do |n|
  name = Faker::Name.name
  email = "example-#{n+1}@railstutorial.org"
  password = "password"
  User.create!(name: name,
              email: email,
              password: password,
              password_confirmation: password,
              activated: true,
              activated_at: Time.zone.now)
end
```

Listado 10.5: Activando a los usuarios del archivo fixture.

`test/fixtures/users.yml`

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>
  admin: true
  activated: true
  activated_at: <%= Time.zone.now %>

archer:
  name: Sterling Archer
  email: duchess@example.gov
  password_digest: <%= User.digest('password') %>
  activated: true
  activated_at: <%= Time.zone.now %>
```

```
lana:
  name: Lana Kane
  email: hands@example.gov
  password_digest: <%= User.digest('password') %>
  activated: true
  activated_at: <%= Time.zone.now %>

mallory:
  name: Mallory Archer
  email: boss@example.gov
  password_digest: <%= User.digest('password') %>
  activated: true
  activated_at: <%= Time.zone.now %>

<% 30.times do |n| %>
user_<%= n %>:
  name: <%= "User #{n}" %>
  email: <%= "user-#{n}@example.com" %>
  password_digest: <%= User.digest('password') %>
  activated: true
  activated_at: <%= Time.zone.now %>
<% end %>
```

Para aplicar los cambios del [Listado 10.4](#), reinicialice la base de datos para realimentarla con los datos como es usual:

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

10.1.2 Método de envío de correo electrónico para la activación de la cuenta

Con el modelo de datos completo, ahora estamos listos para agregar el código necesario para enviar un correo electrónico para activar la cuenta. El proceso es agregar un *gestor de correo* a **User** utilizando la biblioteca Action Mailer, la cual utilizaremos en la acción **create** del controlador de usuarios para enviar un correo electrónico con un enlace de activación. Los gestores de correo están estructurados de forma muy semejante a las acciones de un controlador, con plantillas de correo definidas como vistas. Nuestra tarea en esta sección es

definir los *gestores de correo* y las vistas con enlaces que contengan el token de activación y la dirección electrónica asociada con la cuenta que se va a activar.

De igual forma que con los modelos y controladores, podemos generar un *gestor de correo* utilizando **rails generate**:

```
$ rails generate mailer UserMailer account_activation password_reset
```

Aquí hemos generado el método **account_activation** al igual que el método **password_reset** que necesitaremos en la Sección 10.2.

Como parte de generar el *gestor de correo*, Rails también genera para cada uno de ellos, dos plantillas para las vistas, una para los correos electrónicos que se envían como texto plano y otro para los correos que usan el formato HTML. El método que envía el correo electrónico para la activación de la cuenta, aparece en los Listados 10.6 y 10.7.

Listado 10.6: La plantilla para la activación de la cuenta en formato texto.

```
app/views/user_mailer/account_activation.text.erb
```

```
UserMailer#account_activation
```

```
<%= @greeting %>, find me in app/views/user_mailer/account_activation.text.erb
```

Listado 10.7: La plantilla para la activación de la cuenta en formato HTML.

```
app/views/user_mailer/account_activation.html.erb
```

```
<h1>UserMailer#account_activation</h1>
```

```
<p>
<%= @greeting %>, find me in app/views/user_mailer/account_activation.html.erb
</p>
```

Echemos un vistazo a los *gestores de correo* generados para tener una idea de cómo funcionan (Listados 10.8 y 10.9). Vemos en el Listado 10.8 que existe una dirección **from** por default, común a todos los *mailers* de la aplicación, y cada método del Listado 10.9 tiene una dirección de destinatario también.

(El Listado 10.8 también utiliza una plantilla de *gestor de correo* correspondiente al formato de correo electrónico; aunque no es relevante en este tutorial, las plantillas de HTML y texto plano resultantes pueden encontrarse en `app/views/layouts`.) El código generado también incluye una variable de instancia (`@greeting`), que está disponible en las vistas del *gestor de correo* de forma muy similar en que las variables de instancia de los controladores están disponibles en las vistas ordinarias.

Listado 10.8: El *gestor de correo* de la aplicación generado.

`app/mailers/application_mailer.rb`

```
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer'
end
```

Listado 10.9: El *gestor de correo* de `User` generado.

`app/mailers/user_mailer.rb`

```
class UserMailer < ApplicationMailer

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.user_mailer.account_activation.subject
  #

  def account_activation
    @greeting = "Hi"

    mail to: "to@example.org"
  end

  # Subject can be set in your I18n file at config/locales/en.yml
  # with the following lookup:
  #
  #   en.user_mailer.password_reset.subject
  #

  def password_reset
    @greeting = "Hi"

    mail to: "to@example.org"
  end
end
```

Para hacer que el correo electrónico de activación funcione, primero personalizaremos la plantilla generada como se muestra en el [Listado 10.10](#). Luego, crearemos una variable de instancia que contenga el usuario (para usarlo en la vista), y después enviaremos por correo electrónico el resultado a `user.email` ([Listado 10.11](#)). Como se muestra en el [Listado 10.11](#), el método `mail` también utiliza una llave `subject`, cuyo valor es utilizado en el asunto del correo electrónico.

Listado 10.10: El *gestor de correo* de la aplicación con una nueva dirección electrónica de remitente por default.

```
app/mailers/application_mailer.rb
```

```
class ApplicationMailer < ActionMailer::Base
  default from: "noreply@example.com"
  layout 'mailer'
end
```

Listado 10.11: Enviando el enlace para la activación de la cuenta.

```
app/mailers/user_mailer.rb
```

```
class UserMailer < ApplicationMailer

  def account_activation(user)
    @user = user
    mail to: user.email, subject: "Account activation"
  end

  def password_reset
    @greeting = "Hi"

    mail to: "to@example.org"
  end
end
```

Como con las vistas ordinarias, podemos utilizar Ruby embebido para personalizar las plantillas de las vistas, en este caso saludando al usuario por su nombre e incluyendo un enlace de activación personalizado. Nuestro plan es buscar al usuario por dirección electrónica y luego autenticar el token de activación, de forma que el enlace necesita incluir tanto la dirección electrónica

como el token. Como estamos modelando activaciones usando el recurso de Activación de Cuenta, el token mismo puede aparecer como argumento de la ruta nombrada que definimos en el [Listado 10.1](#):

```
edit_account_activation_url(@user.activation_token, ...)
```

Recuerde que

```
edit_user_url(user)
```

produce una URL de la forma

```
http://www.example.com/users/1/edit
```

la URL de base correspondiente al enlace de activación de cuenta se verá como esto:

```
http://www.example.com/account_activations/q5lt38hQDc_959PVoo6b7A/edit
```

Aquí **q5lt38hQDc_959PVoo6b7A** es una cadena en base64 segura de utilizar en las URL generada por el método [new_token](#) ([Listado 8.31](#)), y juega el mismo rol que el id del usuario en /users/1/edit. En particular, en la acción **edit** del controlador de Activaciones, el token estará disponible en el arreglo hash **params** de forma análoga al id **params[:id]**.

Con la finalidad de incluir la dirección electrónica también, necesitamos utilizar un *parámetro de consulta*, el cual aparece en una URL como una pareja de llave-valor ubicada después del signo de interrogación:⁶

⁶las URLs pueden contener múltiples parámetros de consulta, que consisten de múltiples parejas de llave-valor separadas por el carácter ‘ampersand’ &, como en `/edit?name=Foo%20Bar&email=foo%40example.com`.

```
account_activations/q5lt38hQDc_959PVoo6b7A/edit?email=foo%40example.com
```

Observe que la ‘@’ en la dirección electrónica aparece como `%40`, es decir, ha sido “escapada” para garantizar que la URL sea válida. La forma de asignar un parámetro de consulta en Rails es incluir un hash en la ruta nombrada:

```
edit_account_activation_url(@user.activation_token, email: @user.email)
```

Cuando utilizamos rutas nombradas de esta forma para definir parámetros de consulta, Rails automáticamente “escapa” cualquier carácter especial. La dirección electrónica resultante también será “des-escapada” automáticamente en el controlador, y estará disponible en `params[:email]`.

Con la variable de instancia `@user` como se definió en el [Listado 10.11](#), podemos crear los enlaces necesarios utilizando la ruta nombrada de edición y Ruby embebido, como se muestra en los [Listados 10.12](#) y [10.13](#). Observe que la plantilla HTML del [Listado 10.13](#) utiliza el método `link_to` para construir un enlace válido.

Listado 10.12: La vista de texto de la activación de la cuenta.

```
app/views/user_mailer/account_activation.text.erb
```

```
Hi <%= @user.name %>,  
Welcome to the Sample App! Click on the link below to activate your account:  
<%= edit_account_activation_url(@user.activation_token, email: @user.email) %>
```

Listado 10.13: La vista HTML de la activación de la cuenta.

```
app/views/user_mailer/account_activation.html.erb
```

```
<h1>Sample App</h1>  
<p>Hi <%= @user.name %>,</p>  
<p>
```

```
Welcome to the Sample App! Click on the link below to activate your account:  
</p>  
  
<%= link_to "Activate", edit_account_activation_url(@user.activation_token,  
                                                 email: @user.email) %>
```

Para ver los resultados de las plantillas definidas en los Listados 10.12 y 10.13, podemos utilizar *vistas previas del correo*, que son URLs especiales expuestas por Rails para permitirnos visualizar cómo se ven nuestros mensajes de correo. Primero, necesitamos agregar algo de configuración a nuestro ambiente de desarrollo de la aplicación, como se muestra en el Listado 10.14.

Listado 10.14: Configuración del correo electrónico en desarrollo.

config/environments/development.rb

```
Rails.application.configure do  
  .  
  .  
  .  
  config.action_mailer.raise_delivery_errors = true  
  config.action_mailer.delivery_method = :test  
  host = 'example.com'  
  config.action_mailer.default_url_options = { host: host }  
  .  
  .  
  .  
end
```

El Listado 10.14 utiliza el nombre de servidor '`example.com`', pero usted puede utilizar el nombre real de su servidor de desarrollo. Por ejemplo, en mi sistema cualquiera de los siguientes funcionan (dependiendo de si estoy utilizando el IDE en la nube o un servidor local):

```
host = 'rails-tutorial-c9-mhartl.c9.io'      # Cloud IDE
```

```
host = 'localhost:3000' # Local server
```

Luego de reiniciar el servidor de desarrollo para activar la configuración del Listado 10.14, necesitamos actualizar el *archivo de vista previa* del gestor de correo, que se generó de forma automática en la Sección 10.1.2, como vemos en el Listado 10.15.

Listado 10.15: Las vistas previas del gestor de correo.

```
test/mailers/previews/user_mailer_preview.rb
```

```
# Preview all emails at http://localhost:3000/rails/mailers/user_mailer
class UserMailerPreview < ActionMailer::Preview

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/account_activation
  def account_activation
    UserMailer.account_activation
  end

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/password_reset
  def password_reset
    UserMailer.password_reset
  end

end
```

Como el método `account_activation` definido en el Listado 10.11 requiere un objeto usuario válido como argumento, el código del Listado 10.15 no funcionará tal como está escrito. Para arreglarlo, definimos una variable `user` igual a la del primer usuario en la base de datos de desarrollo, y luego lo pasamos como argumento a `UserMailer.account_activation` (Listado 10.16). Observe que el Listado 10.16 también asigna un valor a `user.activation_token`, el cual es necesario porque la plantilla de activación de la cuenta de los Listados 10.12 y 10.13 necesita un token de activación de la cuenta. (Como `activation_token` es un atributo virtual (Sección 10.1.1), el usuario que viene de la base de datos no lo trae.)

Listado 10.16: Un método de vista previa funcionando para una activación de cuenta.

```
test/mailers/previews/user_mailer_preview.rb

# Preview all emails at http://localhost:3000/rails/mailers/user_mailer
class UserMailerPreview < ActionMailer::Preview

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/account_activation
  def account_activation
    user = User.first
    user.activation_token = User.new_token
    UserMailer.account_activation(user)
  end

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/password_reset
  def password_reset
    UserMailer.password_reset
  end
end
```

Con el código de vista previa del [Listado 10.16](#), podemos visitar las URLs sugeridas para visualizar los correos de activación de cuenta. (Si usted está utilizando el IDE en la nube, debería reemplazar **localhost:3000** con la URL base correspondiente.) El HTML resultante y los correos de texto aparecen en las Figuras [10.2](#) y [10.3](#).

Como paso final, escribiremos un par de pruebas para volver a verificar los resultados que se muestran en las vistas previas del correo. No es tan difícil como parece, porque Rails ha generado pruebas de ejemplo útiles para nosotros ([Listado 10.17](#)).

Listado 10.17: La prueba del gestor de correo generada por Rails.

```
test/mailers/user_mailer_test.rb

require 'test_helper'

class UserMailerTest < ActionMailer::TestCase

  test "account_activation" do
    mail = UserMailer.account_activation
    assert_equal "Account activation", mail.subject
  end
end
```

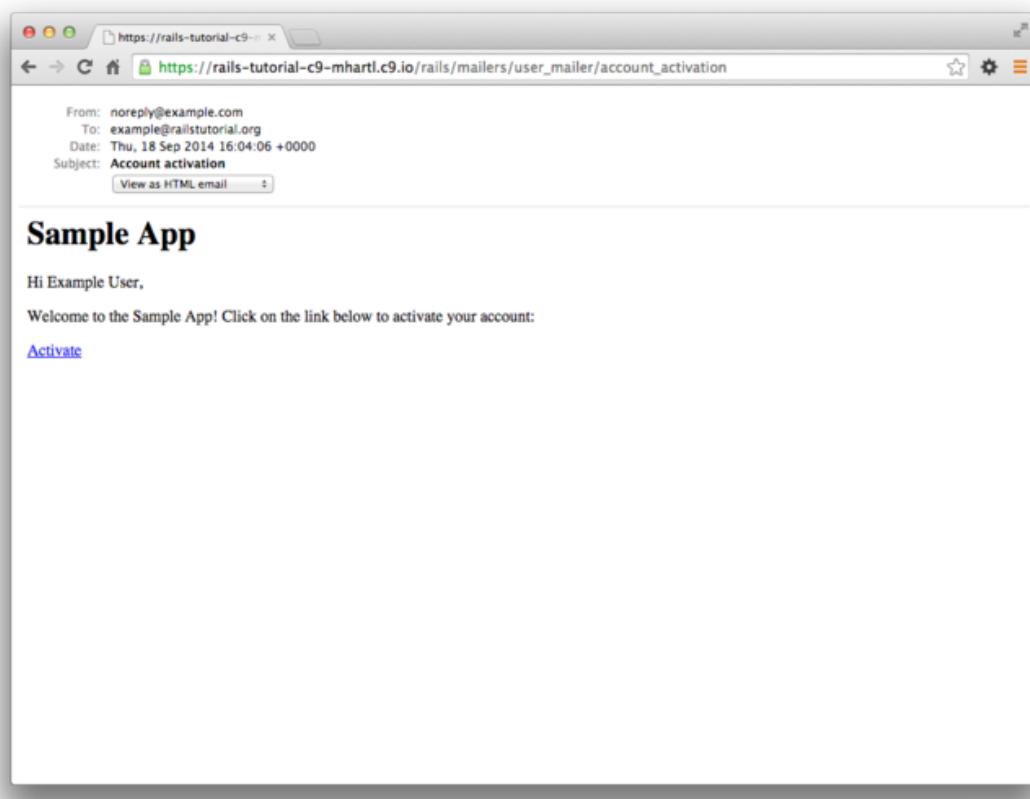


Figura 10.2: Una vista previa de la versión HTML del correo de activación de la cuenta.

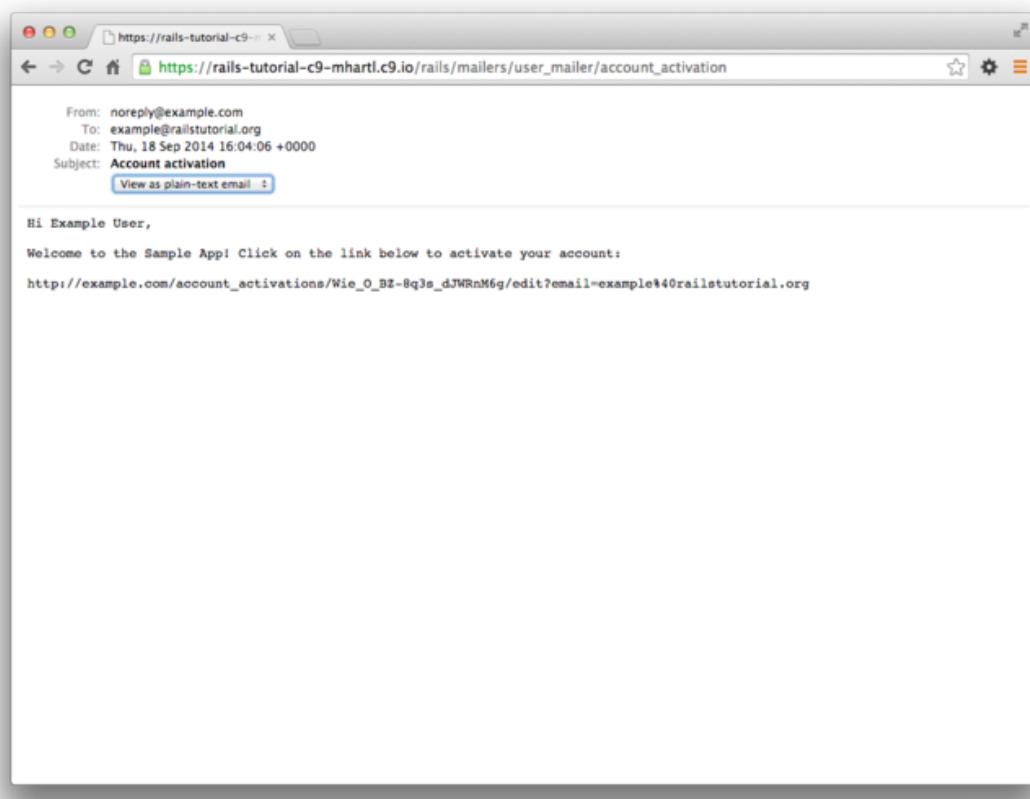


Figura 10.3: Una vista previa de la versión de texto del correo de activación de la cuenta.

```

    assert_equal ["to@example.org"], mail.to
    assert_equal ["from@example.com"], mail.from
    assert_match "Hi", mail.body.encoded
end

test "password_reset" do
  mail = UserMailer.password_reset
  assert_equal "Password reset", mail.subject
  assert_equal ["to@example.org"], mail.to
  assert_equal ["from@example.com"], mail.from
  assert_match "Hi", mail.body.encoded
end
end

```

Las pruebas del Listado 10.17 utilizan el poderoso método `assert_match`, que puede ser utilizado ya sea con una cadena o con una expresión regular:

```

assert_match 'foo', 'foobar'      # true
assert_match 'baz', 'foobar'      # false
assert_match /\w+/, 'foobar'      # true
assert_match /\w+/, '$#!*+@'      # false

```

La prueba del Listado 10.18 usa `assert_match` para verificar que el nombre, el token de activación y la dirección de correo electrónico “escapada” aparecen en el cuerpo del correo. Para éste último, observe el uso de

```
CGI::escape(user.email)
```

para escapar la dirección electrónica del usuario.⁷

Listado 10.18: Una prueba de la implementación actual del correo. **ROJO**
`test/mailers/user_mailer_test.rb`

⁷La forma de aprender cómo hacer algo como esto es buscando en Google “[ruby rails escape url](#)”. Encontrará dos posibilidades principales, `URI::encode(str)` y `CGI::escape(str)`. Probar ambas comprueba que la última funciona. (De hecho, existe una tercera posibilidad también: la biblioteca `ERB::Util` proporciona un método `url_encode` que tiene el mismo efecto.)

```

require 'test_helper'

class UserMailerTest < ActionMailer::TestCase

  test "account_activation" do
    user = users(:michael)
    user.activation_token = User.new_token
    mail = UserMailer.account_activation(user)
    assert_equal "Account activation", mail.subject
    assert_equal [user.email], mail.to
    assert_equal ["noreply@example.com"], mail.from
    assert_match user.name, mail.body.encoded
    assert_match user.activation_token, mail.body.encoded
    assert_match CGI::escape(user.email), mail.body.encoded
  end
end

```

Observe que el Listado 10.18 se encarga de agregar un token de activación al usuario del fixture, que de otra forma estaría en blanco.

Para hacer que la prueba pase, tenemos que configurar nuestro archivo de prueba con el nombre de dominio apropiado, como se muestra en el Listado 10.19.

Listado 10.19: Configurando el nombre de dominio de prueba.

config/environments/test.rb

```

Rails.application.configure do
  .
  .
  .
  config.action_mailer.delivery_method = :test
  config.action_mailer.default_url_options = { host: 'example.com' }
  .
  .
  .
end

```

Con el código anterior, la prueba del gestor de correo debería estar en **VERDE**:

Listado 10.20: **VERDE**

```
$ bundle exec rake test:mailers
```

Para usar el gestor en nuestra aplicación, sólo necesitamos agregar un par de líneas a la acción `create` utilizada para registrar a los usuarios, como se muestra en el [Listado 10.21](#). Observe que el [Listado 10.21](#) ha cambiado el comportamiento de redireccionamiento sobre el registro. Antes, redirigíamos a la página de perfil del usuario ([Sección 7.4](#)), pero esto ya no tiene sentido ahora que estamos requiriendo la activación de la cuenta. En vez de esto, ahora redireccionaremos a la URL raíz.

Listado 10.21: Agregando la activación de la cuenta al registro de usuario.

ROJO

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      UserMailer.account_activation(@user).deliver_now
      flash[:info] = "Please check your email to activate your account."
      redirect_to root_url
    else
      render 'new'
    end
  end
  .
  .
  .
end
```

Como el [Listado 10.21](#) redirecciona a la URL raíz en vez de a la página de perfil y no inicia la sesión del usuario como antes, el conjunto de pruebas está actualmente en **ROJO**, aún cuando la aplicación está funcionando como fué diseñada. Arreglaremos esto comentando temporalmente las líneas que provocan el fallo, como se muestra en el [Listado 10.22](#). Descomentaremos estas líneas y escribiremos pruebas para la activación de la cuenta que pasen en la [Sección 10.1.4](#).

Listado 10.22: Comentando temporalmente nuestras pruebas fallidas. VERDE*test/integration/users_signup_test.rb*

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

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

  test "valid signup information" do
    get signup_path
    assert_difference 'User.count', 1 do
      post_via_redirect users_path, user: { name: "Example User",
                                             email: "user@example.com",
                                             password: "password",
                                             password_confirmation: "password" }
    end
    # assert_template 'users/show'
    # assert_is_logged_in?
  end
end
```

Si usted intenta en este momento registrarse como un nuevo usuario, debería ser redirigido como se muestra en la Figura 10.4, y debería generarse un correo electrónico como el que se muestra en el Listado 10.23. Observe que *no* recibirá un correo electrónico real en un ambiente de desarrollo, pero se mostrará en las bitácoras de su servidor. (Puede que tenga que buscar un poco en ellas para verlo.) La Sección 10.3 trata de cómo enviar realmente un correo electrónico en un ambiente de producción.

Listado 10.23: Un ejemplo de correo de activación de cuenta extraído de la bitácora del servidor.

```

Sent mail to michael@michaelhartl.com (931.6ms)
Date: Wed, 03 Sep 2014 19:47:18 +0000
From: noreply@example.com
To: michael@michaelhartl.com
Message-ID: <540770474e16_61d3fd1914f4cd0300a0@mhartl-rails-tutorial-953753.mail>
Subject: Account activation
Mime-Version: 1.0
Content-Type: multipart/alternative;
  boundary="====_mimepart_5407704656b50_61d3fd1914f4cd02996a";
  charset=UTF-8
Content-Transfer-Encoding: 7bit

=====_mimepart_5407704656b50_61d3fd1914f4cd02996a
Content-Type: text/plain;
  charset=UTF-8
Content-Transfer-Encoding: 7bit

Hi Michael Hartl,

Welcome to the Sample App! Click on the link below to activate your account:

http://rails-tutorial-c9-mhartl.c9.io/account\_activations/fFb\_F94mgQtmlSvRFGsITw/edit?email=michael%40michaelhartl.com
=====_mimepart_5407704656b50_61d3fd1914f4cd02996a
Content-Type: text/html;
  charset=UTF-8
Content-Transfer-Encoding: 7bit

<h1>Sample App</h1>

<p>Hi Michael Hartl,</p>

<p>
  Welcome to the Sample App! Click on the link below to activate your account:
</p>

<a href="http://rails-tutorial-c9-mhartl.c9.io/account_activations/fFb_F94mgQtmlSvRFGsITw/edit?email=michael%40michaelhartl.com">Activate</a>
=====_mimepart_5407704656b50_61d3fd1914f4cd02996a--
```

10.1.3 Activando la cuenta

Ahora que tenemos un correo electrónico generado correctamente como en el Listado 10.23, necesitamos escribir una acción **edit** en el controlador de activaciones de cuenta para activar al usuario. Recuerde que comentamos en

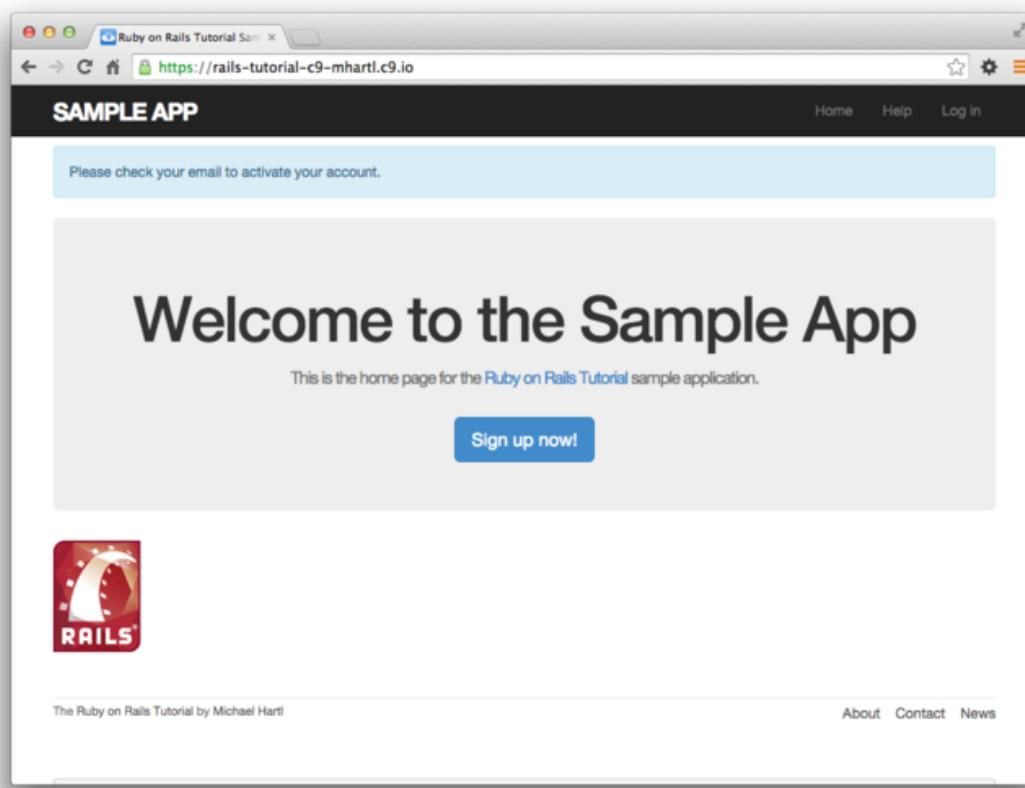


Figura 10.4: La página principal con un mensaje de activación luego de un registro exitoso.

la Sección 10.1.2 que el token de activación y la dirección electrónica están disponibles en `params[:id]` y `params[:email]`, respectivamente. Siguiendo el modelo de las contraseñas (Listado 8.5) y tokens recordados (Listado 8.36), podemos planear buscar y autenticar al usuario con un código como el que sigue:

```
user = User.find_by(email: params[:email])
if user && user.authenticated?(:activation, params[:id])
```

(Como veremos en un momento, habrá un booleano extra en la expresión anterior. Vea si puede adivinar cuál será.)

El código anterior utiliza el método `authenticated?` para probar si la digestión de la activación de la cuenta coincide con el token dado, pero en este momento no va a funcionar porque ése método está especializado en el token recordado (Listado 8.33):

```
# Returns true if the given token matches the digest.
def authenticated?(remember_token)
  return false if remember_digest.nil?
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

Aquí `remember_digest` es un atributo del modelo de usuario, y dentro del modelo podemos reescribirlo como sigue:

```
self.remember_digest
```

De alguna forma, queremos ser capaces de crear esta *variable*, de forma que podamos invocar

```
self.activation_token
```

en vez de pasar el parámetro apropiado a `authenticated?`.

La solución involucra nuestro primer ejemplo de *metaprogramación*, que es esencialmente un programa que escribe un programa. (Metaprogramación es una de las características más fuertes de Ruby y muchas de las características “mágicas” de Rails se deben al uso de la metaprogramación de Ruby.) La clave en este caso es el poderoso método **send**, que nos permite llamar a un método con el nombre de nuestra elección al “enviar un mensaje” al objeto dado. Por ejemplo, en esta sesión de consola estamos utilizando **send** sobre un objeto Ruby nativo para encontrar la longitud de un arreglo:

```
$ rails console
>> a = [1, 2, 3]
>> a.length
=> 3
>> a.send(:length)
=> 3
>> a.send('length')
=> 3
```

Aquí vemos que pasando el símbolo **:length** o la cadena **'length'** a **send** es equivalente a invocar el método **length** sobre un objeto dado. Como segundo ejemplo, accedaremos al atributo **activation_digest** del primer usuario en la base de datos:

```
>> user = User.first
>> user.activation_digest
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> user.send(:activation_digest)
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> user.send('activation_digest')
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> attribute = :activation
>> user.send("#{attribute}_digest")
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
```

Observe en el último ejemplo que hemos definido una variable **attribute** igual al símbolo **:activation** y usamos la interpolación de cadenas para construir el argumento apropiado para **send**. Esto también funcionará con la cadena **'activation'**, pero utilizar un símbolo es más convencional, y en cualquier caso

```
"#{attribute}_digest"
```

se convertirá en

```
"activation_digest"
```

una vez que la cadena sea interpolada. (Ya vimos cómo los símbolos son interpolados como cadenas en la Sección 7.4.2.)

Basándonos en esta discusión acerca de `send`, podemos reescribir el método `authenticated?` como sigue:

```
def authenticated?(remember_token)
  digest = self.send('remember_digest')
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(remember_token)
end
```

Con esta plantilla en su lugar, podemos generalizar el método agregando un argumento a la función con el nombre de la digestión, y luego utilizar la interpolación de cadenas como en el ejemplo anterior:

```
def authenticated?(attribute, token)
  digest = self.send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

(Aquí hemos renombrado el segundo argumento `token` para enfatizar que ahora es genérico.) Como estamos dentro del modelo de usuario, también podemos omitir `self`, lo cual nos lleva a la versión más idiomáticamente correcta:

```
def authenticated?(attribute, token)
  digest = send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

Ahora podemos reproducir el comportamiento anterior de **authenticated?** al invocarlo como sigue:

```
user.authenticated?(:remember, remember_token)
```

Aplicando este razonamiento al modelo **User** obtenemos el método generalizado **authenticated?** que se muestra en el [Listado 10.24](#).

Listado 10.24: Un método **authenticated?** generalizado. **ROJO**
app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns true if the given token matches the digest.
  def authenticated?(attribute, token)
    digest = send("#{attribute}_digest")
    return false if digest.nil?
    BCrypt::Password.new(digest).is_password?(token)
  end
  .
  .
  .
end
```

La leyenda del [Listado 10.24](#) indica que tenemos un conjunto de pruebas en **ROJO** :

Listado 10.25: **ROJO**

```
$ bundle exec rake test
```

El motivo del fallo es que el método **current_user** ([Listado 8.36](#)) y la prueba para las digestiones en **nil** ([Listado 8.43](#)) utilizan ambos la versión anterior de **authenticated?**, la cual espera un argumento en vez de dos. Para arreglar esto, simplemente actualizamos los dos casos en que se utiliza el método generalizado, como se muestra en los [Listados 10.26](#) y [10.27](#).

Listado 10.26: Usando el método `authenticated?` generalizado en `current_user`.

```
app/helpers/sessions_helper.rb

module SessionsHelper
  .
  .
  .

  # Returns the current logged-in user (if any).
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(:remember, cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

Listado 10.27: Usando el método `authenticated?` generalizado en la prueba de usuario. VERDE

```
test/models/user_test.rb
```

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

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

  test "authenticated? should return false for a user with nil digest" do
    assert_not @user.authenticated?(:remember, '')
  end
end
```

En este momento, las pruebas deberían de estar en VERDE:

Listado 10.28: VERDE

```
$ bundle exec rake test
```

Refactorizando el código como recién se indicó es increíblemente más propenso a errores sin un conjunto de pruebas sólido, que es por lo que pasamos tantos problemas al escribir buenas pruebas en las Secciones 8.4.2 y 8.4.6.

Con el método `authenticated?` como en el Listado 10.24, ahora estamos listos para escribir una acción `edit` que autentique al usuario que corresponde a la dirección de correo del arreglo hash `params`. Nuestra prueba para validar se verá como sigue:

```
if user && !user.activated? && user.authenticated?(:activation, params[:id])
```

Observe la presencia de `!user.activated?`, que es el booleano extra que mencionamos anteriormente. Esto evita que nuestro código active usuarios que ya han sido activados previamente, lo cual es importante porque estaremos iniciando la sesión de los usuarios que confirmen, y no queremos permitir a usuarios maliciosos que consigan el enlace de activación, que inicien sesión como si fueran el usuario legítimo.

Si el usuario es autenticado de acuerdo a los booleanos anteriores, necesitamos activar el usuario y actualizar la fecha y hora `activated_at`:⁸

```
user.update_attribute(:activated, true)
user.update_attribute(:activated_at, Time.zone.now)
```

Esto nos lleva a la acción `edit` que se muestra en el Listado 10.29. Observe también que el Listado 10.29 maneja el caso de un token de activación no válido; esto raramente debería suceder, pero es suficientemente fácil redirigir en estos casos a la URL raíz.

⁸Aquí utilizamos dos llamadas a `update_attribute` en vez de una sola llamada a `updated_attributes` porque (vea la Sección 6.1.5) ésta última ejecutaría las validaciones. Como en este caso no tenemos la contraseña del usuario, dichas validaciones fallarían.

Listado 10.29: Una acción `edit` para activar cuentas.

`app/controllers/account_activations_controller.rb`

```
class AccountActivationsController < ApplicationController

  def edit
    user = User.find_by(email: params[:email])
    if user && !user.activated? && user.authenticated?(:activation, params[:id])
      user.update_attribute(:activated, true)
      user.update_attribute(:activated_at, Time.zone.now)
      log_in user
      flash[:success] = "Account activated!"
      redirect_to user
    else
      flash[:danger] = "Invalid activation link"
      redirect_to root_url
    end
  end
end
```

Con el código del Listado 10.29, usted debería poder utilizar la URL del Listado 10.23 para activar al usuario correspondiente. Por ejemplo, en mi sistema yo visité la URL

```
http://rails-tutorial-c9-mhartl.c9.io/account_activations/
fFb_F94mgQtmlSvRFGsITw/edit?email=michael%40michaelhartl.com
```

y obtuve el resultado que se muestra en la Figura 10.5.

Por supuesto, actualmente la activación de usuario no *hace* realmente nada, porque no hemos cambiado la forma en que los usuarios inician sesión. Para que la activación de la cuenta signifique algo, necesitamos permitir a los usuarios iniciar sesión sólo si han sido activados. Como se muestra en el Listado 10.30, la forma de lograr esto es iniciando la sesión del usuario como es usual: si `user.activated?` es verdadero; en cualquier otro caso, redirigimos a la URL raíz con un mensaje `warning` (Figura 10.6).

Listado 10.30: Evitando que los usuarios desactivados inicien sesión.

`app/controllers/sessions_controller.rb`

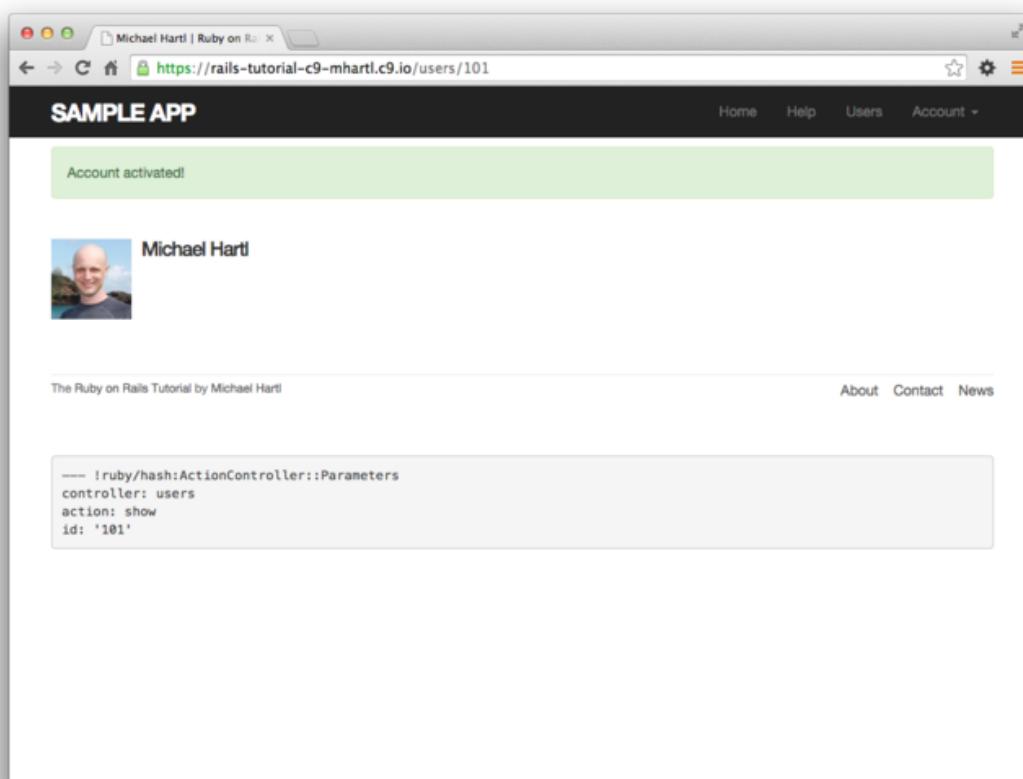


Figura 10.5: La página del perfil luego de una activación exitosa.

```

class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      if user.activated?
        log_in user
        params[:session][:remember_me] == '1' ? remember(user) : forget(user)
        redirect_back_or user
      else
        message = "Account not activated."
        message += "Check your email for the activation link."
        flash[:warning] = message
        redirect_to root_url
      end
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end

```

Con esto, aparte de pulir algún detalle, la funcionalidad básica de la activación de usuario está terminada. (El detalle es evitar que usuarios que no han sido activados sean mostrados, lo cual se deja como ejercicio ([Sección 10.5](#)).) En la [Sección 10.1.4](#), completaremos el proceso agregando algunas pruebas y luego haciendo un poco de refactorización.

10.1.4 Prueba de activación y refactorización

En esta Sección, agregaremos una prueba de integración para la activación de la cuenta. Como ya tenemos una prueba para el registro en el caso de información válida, le agregaremos los pasos que desarrollamos en la [Sección 7.4.4](#) ([Listado 7.26](#)). Hay algunos pasos, pero la mayoría son directos; vea si usted puede seguirlos en el [Listado 10.31](#).

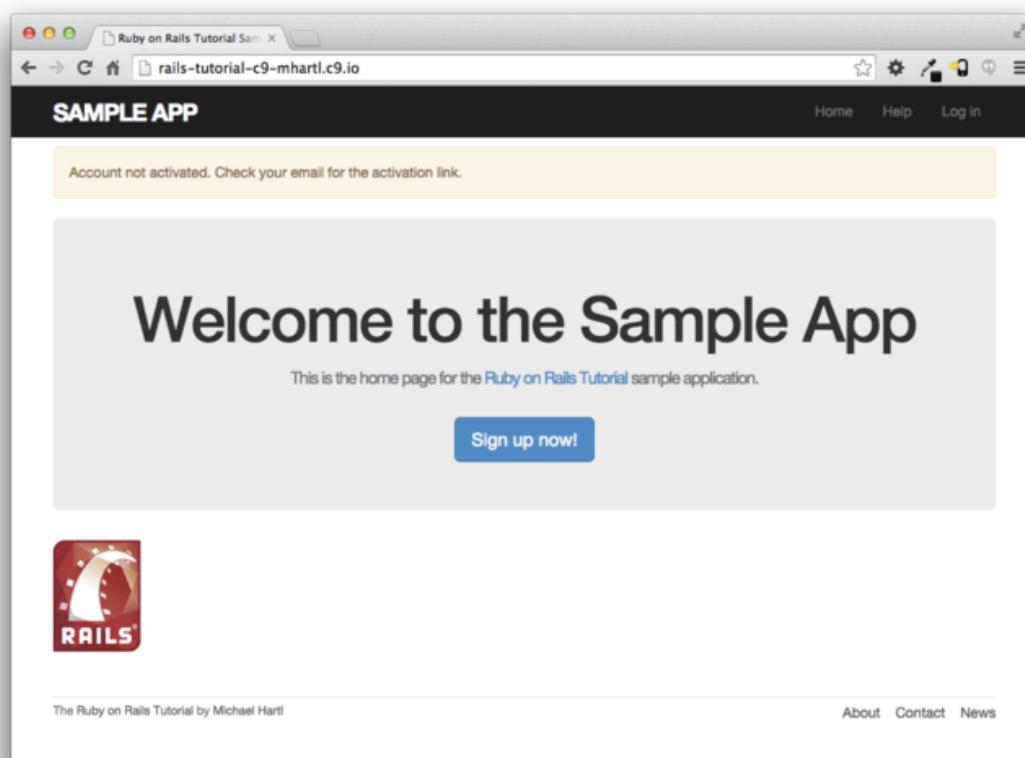


Figura 10.6: El mensaje de advertencia para un usuario que aún no ha sido activado.

Listado 10.31: Agregando la activación de la cuenta a la prueba de registro de usuario. [VERDE](#)

```
test/integration/users_signup_test.rb

require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

  def setup
    ActionMailer::Base.deliveries.clear
  end

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

  test "valid signup information with account activation" do
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, user: { name: "Example User",
                               email: "user@example.com",
                               password: "password",
                               password_confirmation: "password" }
    end
    assert_equal 1, ActionMailer::Base.deliveries.size
    user = assigns(:user)
    assert_not user.activated?
    # Try to log in before activation.
    log_in_as(user)
    assert_not is_logged_in?
    # Invalid activation token
    get edit_account_activation_path("invalid token")
    assert_not is_logged_in?
    # Valid token, wrong email
    get edit_account_activation_path(user.activation_token, email: 'wrong')
    assert_not is_logged_in?
    # Valid activation token
    get edit_account_activation_path(user.activation_token, email: user.email)
    assert user.reload.activated?
    follow_redirect!
    assert_template 'users/show'
```

```
    assert_is_logged_in?  
  end  
end
```

Hay mucho código en el [Listado 10.31](#), pero el único código completamente nuevo está en la línea

```
assert_equal 1, ActionMailer::Base.deliveries.size
```

Este código verifica que exactamente un mensaje sea entregado. Como el arreglo **deliveries** es global, necesitamos reiniciarlo en el método **setup** para evitar que nuestro código falle si cualquier otra prueba envía correo (como será el caso en la [Sección 10.2.5](#)). El [Listado 10.31](#) también utiliza el método **assigns** por primera vez en el tutorial; como explicamos en el ejercicio del Capítulo 8 ([Sección 8.6](#)), **assigns** nos permite el acceso a variables de instancia en la acción correspondiente. Por ejemplo, la acción **create** del controlador de usuarios define una variable **@user** ([Listado 10.21](#)), de forma que podemos tener acceso a ella en la prueba utilizando **assigns(:user)**. Finalmente, observe que el [Listado 10.31](#) restaura las líneas que comentamos en el [Listado 10.22](#).

En este momento, el conjunto de pruebas debería estar en **VERDE**:

Listado 10.32: **VERDE**

```
$ bundle exec rake test
```

Con la prueba del [Listado 10.31](#), estamos listos para refactorizar un poco sacando algo de la manipulación de usuario del controlador y pasándolo al modelo. En particular, crearemos un método **activate** para actualizar los atributos de activación del usuario y un **send_activation_email** para enviar el correo de activación. Los métodos extra aparecen en el [Listado 10.33](#), y el código refactorizado de la aplicación se muestra en los [Listados 10.34](#) y [10.35](#).

602 CAPÍTULO 10. ACTIVACIÓN DE LA CUENTA Y REINICIO DE CONTRASEÑA

Listado 10.33: Agregando los métodos de activación de usuario al modelo eUser.

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Activates an account.
  def activate
    update_attribute(:activated, true)
    update_attribute(:activated_at, Time.zone.now)
  end

  # Sends activation email.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  private
  .
  .
  .
end
```

Listado 10.34: Enviando correo electrónico mediante el objeto del modelo de usuario.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .

  def create
    @user = User.new(user_params)
    if @user.save
      @user.send_activation_email
      flash[:info] = "Please check your email to activate your account."
      redirect_to root_url
    else
      render 'new'
    end
  end
  .
  .
  .
end
```

Listado 10.35: Activación de la cuenta mediante el objeto de modelo de usuario.

```
app/controllers/account_activations_controller.rb

class AccountActivationsController < ApplicationController

  def edit
    user = User.find_by(email: params[:email])
    if user && !user.activated? && user.authenticated?(:activation, params[:id])
      user.activate
      log_in user
      flash[:success] = "Account activated!"
      redirect_to user
    else
      flash[:danger] = "Invalid activation link"
      redirect_to root_url
    end
  end
end
```

Observe que el [Listado 10.33](#) elimina el uso de `user.`, el cual fallaría dentro del modelo `User` porque no existe tal variable: modelo User

```
-user.update_attribute(:activated, true)
-user.update_attribute(:activated_at, Time.zone.now)
+update_attribute(:activated, true)
+update_attribute(:activated_at, Time.zone.now)
```

(Pudimos haber reemplazado `user` por `self`, pero recuerde que en la [Sección 6.2.5](#) vimos que `self` es opcional dentro del modelo.) También cambiamos `@user` por `self` en la llamada al gestor de correo:

```
-UserMailer.account_activation(@user).deliver_now
+UserMailer.account_activation(self).deliver_now
```

Estos son *exactamente* la clase de detalles que son fáciles de omitir aún en refactorizaciones simples pero que son detectados por un buen conjunto de pruebas. Hablando de éste, el conjunto de pruebas debería seguir en [VERDE](#):

Listado 10.36: VERDE

```
$ bundle exec rake test
```

La activación de la cuenta ahora está completa, lo cual representa una funcionalidad que vale la pena agrupar en un commit:

```
$ git add -A
$ git commit -m "Add account activations"
```

10.2 Reinicio de Contraseña

Habiendo completado la activación de la cuenta (y por lo tanto verificado la dirección electrónica del usuario), estamos ahora en buena posición para manejar el caso común de los usuarios que olvidan sus contraseñas. Como veremos, muchos de estos pasos son similares, y tendremos varias oportunidades de aplicar las lecciones aprendidas en la [Sección 10.1](#). Aunque el comienzo es diferente; a diferencia de la activación de la cuenta, la implementación del reinicio de contraseña requiere tanto de cambiar una de nuestras vistas, como de crear dos nuevos formularios (para manejar la dirección electrónica y el envío de la nueva contraseña).

Antes de escribir algún código, bosquejemos la secuencia esperada para el reinicio de contraseñas. Empezaremos agregando un enlace “forgot password” al formulario de inicio de sesión de la aplicación de ejemplo ([Figura 10.7](#)). El enlace “forgot password” nos llevará a una página con un formulario que recibe una dirección electrónica y envía un correo que contiene el enlace para el reinicio de la contraseña ([Figura 10.8](#)). El enlace de reinicio conducirá a un formulario para el reinicio de la contraseña del usuario (y su respectiva confirmación) ([Figura 10.9](#)).

Análogamente a las activaciones de cuenta, nuestro plan general es crear un recurso para el reinicio de contraseñas, que consista de un token de reinicio y su correspondiente digestión, para cada reinicio solicitado. La secuencia principal es la siguiente:

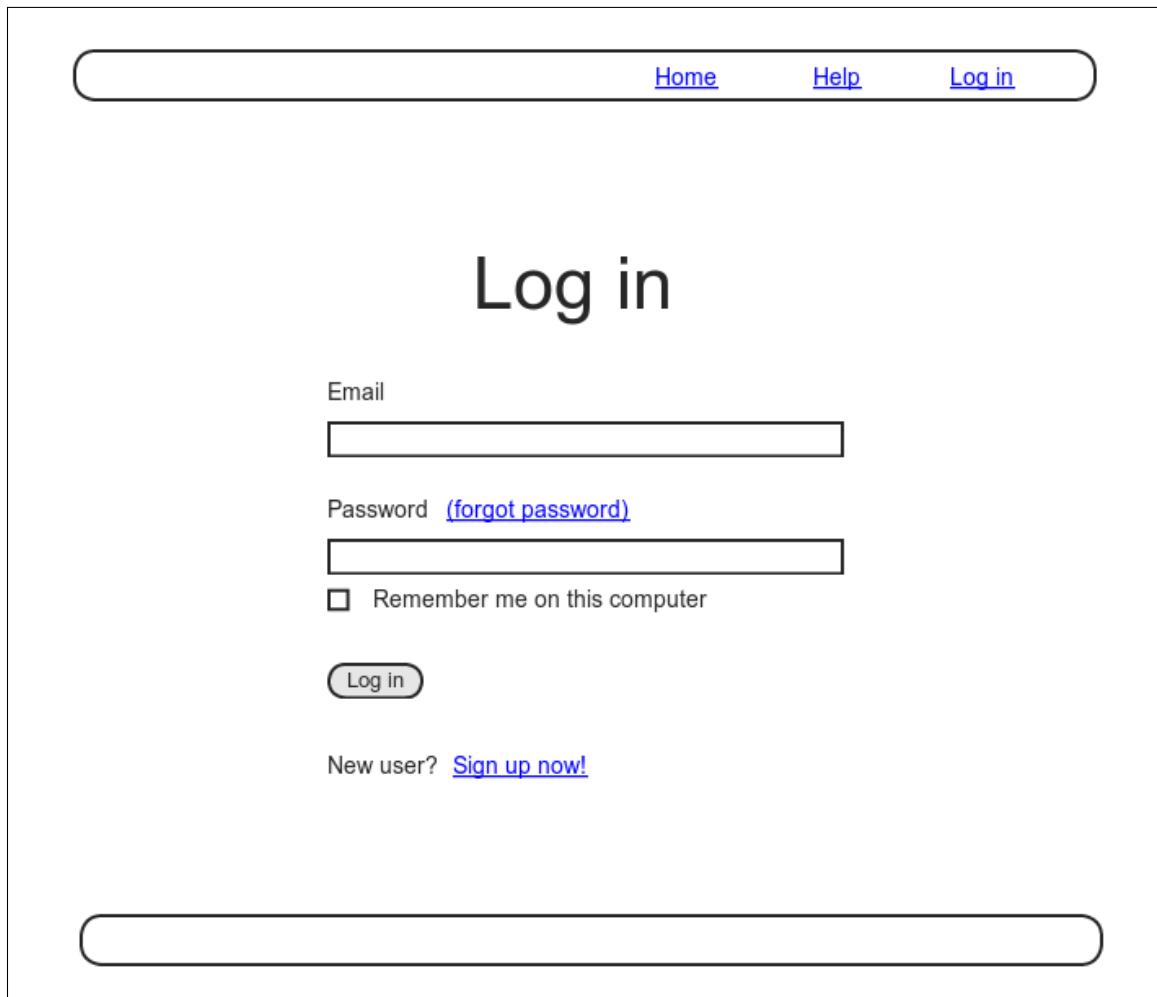
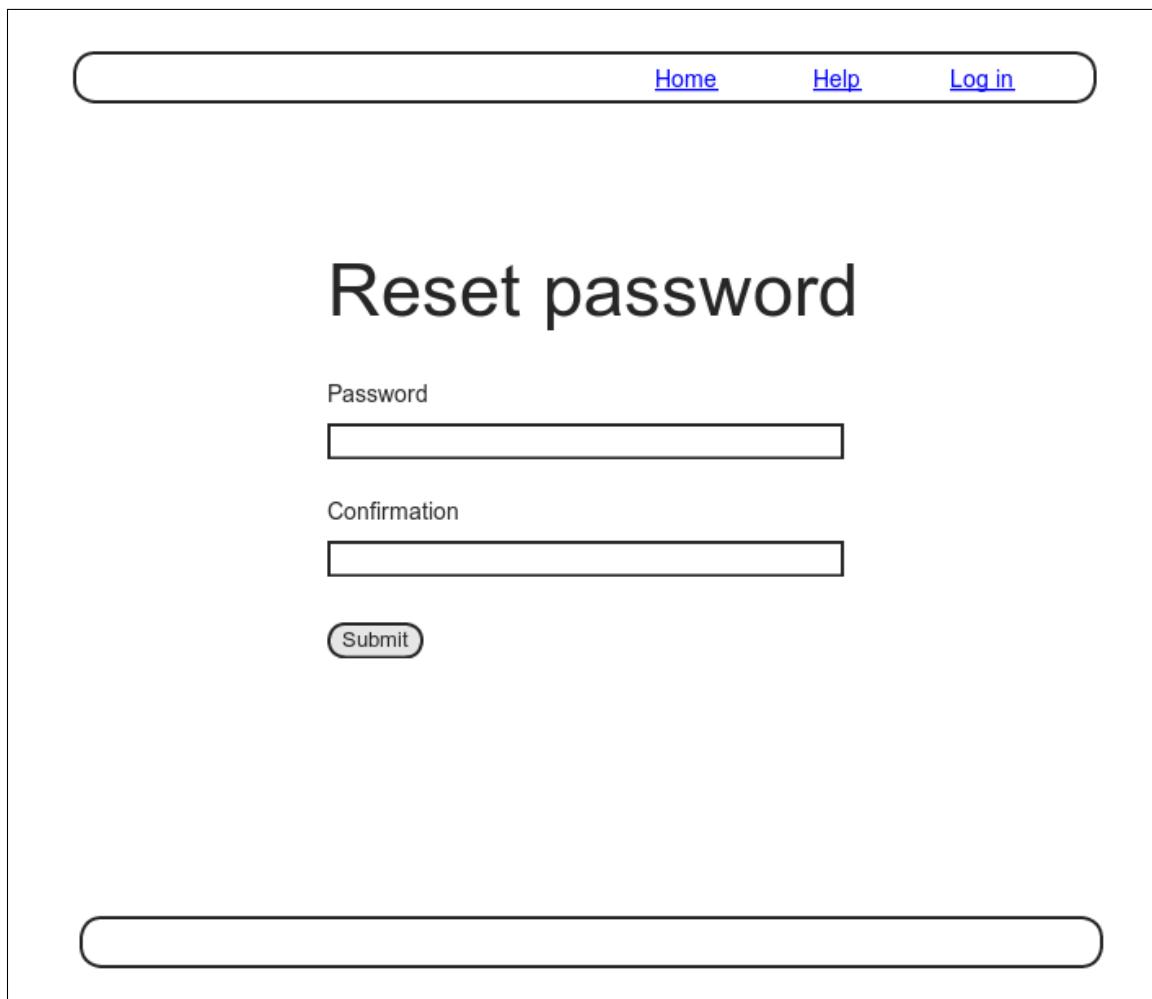


Figura 10.7: Un bosquejo del enlace “forgot password”.

The image shows a wireframe sketch of a web page titled "Forgot password". At the top, there is a horizontal navigation bar with three items: "Home", "Help", and "Log in". Below the title, there is a label "Email" followed by a rectangular input field. Underneath the input field is a "Submit" button. The entire page is enclosed in a large rectangular frame.

Figura 10.8: Un bosquejo del formulario “forgot password”.



Un bosquejo de un formulario de reinicio de contraseña. El formulario se encierra en un cuadro rectangular con un efecto de sombra. En la parte superior, hay un menú horizontal con tres ítems: "Home", "Help" y "Log in", todos en azul y subrayados. En el centro del formulario, el título "Reset password" aparece en un fuente grande y negrita. Debajo de él, hay dos campos de texto: uno para la "Password" y otro para la "Confirmation", ambos representados por cuadros vacíos. A continuación, hay un botón "Submit" rodeado por un círculo. La parte inferior del formulario tiene un efecto de sombra.

Figura 10.9: Un bosquejo del formulario de reinicio de contraseña.

1. Cuando un usuario requiera un reinicio de contraseña, buscar el usuario por la dirección electrónica enviada.
2. Si la dirección electrónica existe en la base de datos, generar un token de reinicio y su correspondiente digestión.
3. Guardar la digestión del token en la base de datos, y enviar un correo electrónico al usuario con el enlace que contiene el token de reinicio y la dirección electrónica del usuario.
4. Cuando el usuario visite el enlace, buscar al usuario por la dirección electrónica y autenticar el token comparándolo con la digestión del mismo.
5. Si es autenticado, presentarle el formulario para el cambio de contraseña.

10.2.1 El recurso de reinicio de contraseñas

Como con las activaciones de cuenta ([Sección 10.1.1](#)), nuestro primer paso es generar un controlador para nuestro nuevo recurso:

```
$ rails generate controller PasswordResets new edit --no-test-framework
```

De igual forma que en la [Sección 10.1.1](#), hemos incluído una bandera para evitar la generación de pruebas y en su lugar realizarlas en las pruebas de integración de la [Sección 10.1.4](#).

Como necesitaremos formularios tanto para crear reinicios de contraseñas ([Figura 10.8](#)) como para actualizarlos cambiando la contraseña en el modelo **User** ([Figura 10.9](#)), necesitamos rutas para **new**, **create**, **edit** y **update**. Podemos encargarnos de esto con la línea **resources** que se muestra en el [Listado 10.37](#).

Listado 10.37: Agregando un recurso para reinicio de contraseña.
config/routes.rb

Petición HTTP	URL	Acción	Ruta nombrada
GET	/password_resets/new	new	new_password_reset_path
POST	/password_resets	create	password_resets_path
GET	/password_resets/<token>/edit	edit	edit_password_reset_path(token)
PATCH	/password_resets/<token>	update	password_reset_path(token)

Tabla 10.2: Rutas RESTful proporcionadas por el recurso **Password Resets** del Listado 10.37.

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help' => 'static_pages#help'
  get 'about' => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup' => 'users#new'
  get 'login' => 'sessions#new'
  post 'login' => 'sessions#create'
  delete 'logout' => 'sessions#destroy'
  resources :users
  resources :account_activations, only: [:edit]
  resources :password_resets, only: [:new, :create, :edit, :update]
end
```

El código del Listado 10.37 se encarga de las rutas RESTful que se muestran en la Tabla 10.2. En particular, la primer ruta de la Tabla 10.2 nos proporciona un enlace al formulario “forgot password” mediante

```
new_password_reset_path
```

como vimos en el Listado 10.38 y en la Figura 10.10.

Listado 10.38: Agregando un enlace al reinicio de contraseñas.

```
app/views/sessions/new.html.erb
```

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session, url: login_path) do |f| %>
```

```

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

<%= f.label :password %>
<%= link_to "(forgot password)", new_password_reset_path %>
<%= f.password_field :password, class: 'form-control' %>

<%= f.label :remember_me, class: "checkbox inline" do %>
  <%= f.check_box :remember_me %>
  <span>Remember me on this computer</span>
<% end %>

  <%= f.submit "Log in", class: "btn btn-primary" %>
<% end %>

<p>New user? <%= link_to "Sign up now!", signup_path %></p>
</div>
</div>

```

El modelo de datos para el reinicio de contraseñas es similar al utilizado para la activación de la cuenta (Figura 10.1). Siguiendo el patrón establecido por los tokens recordados (Sección 8.4) y los tokens de activación de cuenta (Sección 10.1), el reinicio de contraseña asociará un token virtual para usarlo en el correo de reinicio junto con su correspondiente digestión en la base de datos. Si en vez de esto guardáramos el token sin digerir, un usuario malicioso con acceso a la base de datos podría enviar una petición de reinicio a la cuenta de correo del usuario y luego utilizar el token y el correo para visitar el enlace de reinicio de contraseñas, y por tanto, obtener el control de la cuenta. Entonces, el uso de las digestiones de los tokens para reiniciar contraseñas es esencial. Como medida de seguridad adicional, planeamos que el enlace de reinicio de contraseña *expire* luego de un par de horas, lo cual requiere que se guarde la hora en la que el reinicio de contraseña fue enviado. Los atributos **reset_digest** y **reset_sent_at** resultantes aparecen en la Figura 10.11.

La migración para agregar los atributos de la Figura 10.11 aparece como sigue:

```

$ rails generate migration add_reset_to_users reset_digest:string \
> reset_sent_at:datetime

```

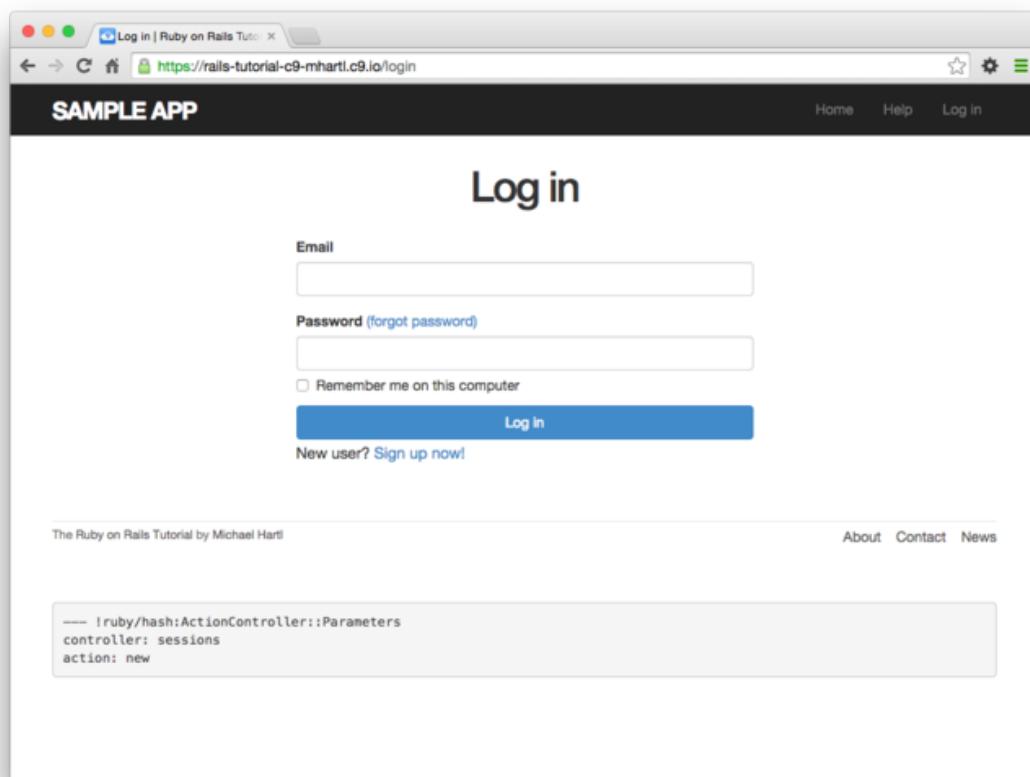


Figura 10.10: La página de inicio de sesión con un enlace “forgot password”.

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string
admin	boolean
activation_digest	string
activated	boolean
activated_at	datetime
reset_digest	string
reset_sent_at	datetime

Figura 10.11: El modelo **User** con los nuevos atributos para el reinicio de contraseña.

(Como vimos anteriormente, el > en la segunda línea es un carácter de “continuación de línea” insertado automáticamente por la consola, y no debe ser escrito literalmente.) Luego migramos como es usual:

```
$ bundle exec rake db:migrate
```

10.2.2 Controlador y formulario de reinicio de contraseñas

Para crear la nueva vista para el reinicio de contraseñas, trabajaremos de forma análoga al formulario anterior creando un recurso que no es de tipo Active Record, digamos, el formulario de inicio de sesión (Listado 8.2) para crear una sesión nueva, que mostramos nuevamente en el Listado 10.39 como referencia.

Listado 10.39: Revisando el código del formulario de inicio de sesión.

app/views/sessions/new.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:session, url: login_path) do |f| %>

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

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

      <%= f.label :remember_me, class: "checkbox inline" do %>
        <%= f.check_box :remember_me %>
        <span>Remember me on this computer</span>
      <% end %>

      <%= f.submit "Log in", class: "btn btn-primary" %>
    <% end %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

614CAPÍTULO 10. ACTIVACIÓN DE LA CUENTA Y REINICIO DE CONTRASEÑA

El nuevo formulario para reinicio de contraseñas tiene mucho en común con el [Listado 10.39](#); las diferencias más importantes son el uso de un recurso diferente, una URL en la llamada a `form_for` y la omisión del atributo de contraseña. El resultado aparece en el [Listado 10.40](#) y en la [Figura 10.12](#).

Listado 10.40: Una nueva vista del reinicio de contraseña.

`app/views/password_resets/new.html.erb`

```
<% provide(:title, "Forgot password") %>
<h1>Forgot password</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(:password_reset, url: password_resets_path) do |f| %>
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

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

Al enviar el formulario de la [Figura 10.12](#), necesitamos buscar al usuario mediante su dirección electrónica y actualizar sus atributos con el token de reinicio de la contraseña y una estampilla de tiempo del momento en que se envía. Luego redireccionamos a la URL raíz con un mensaje flash informativo. De igual forma que con el inicio de sesión ([Listado 8.9](#)), en el caso de enviar datos inválidos, volveremos a desplegar la página `new` con un mensaje `flash.now`. Los resultados se muestran en el [Listado 10.41](#).

Listado 10.41: Una acción `create` para el reinicio de contraseñas.

`app/controllers/password_resets_controller.rb`

```
class PasswordResetsController < ApplicationController

  def new
  end

  def create
    @user = User.find_by(email: params[:password_reset][:email].downcase)
    if @user
```

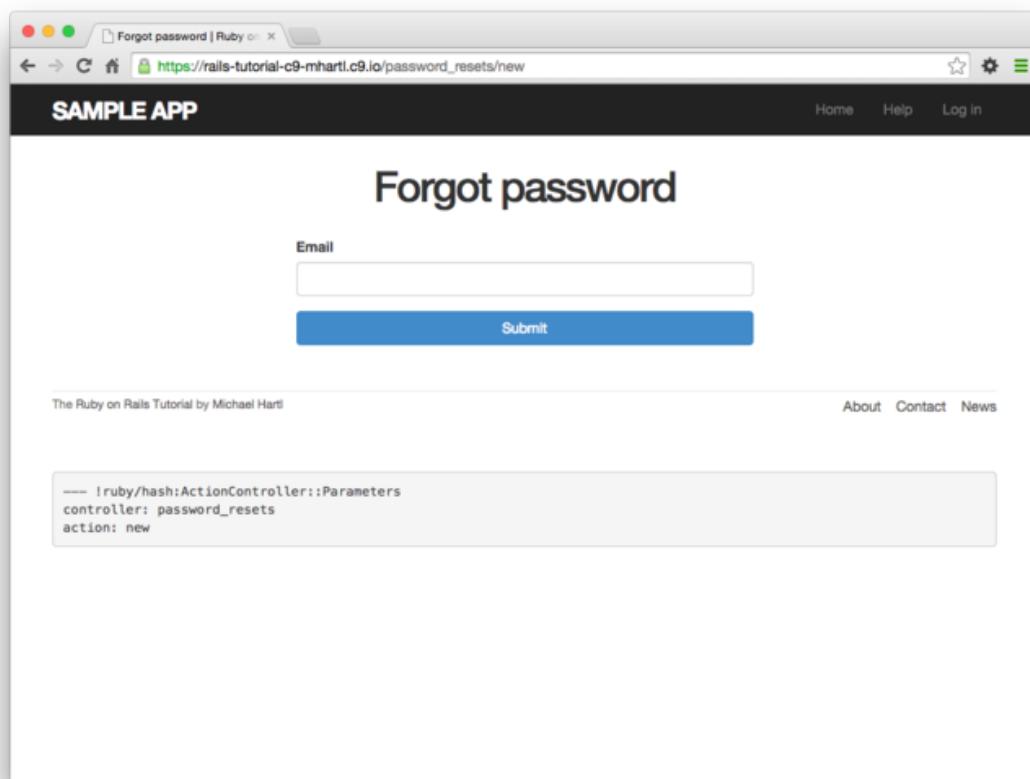


Figura 10.12: El formulario “forgot password”.

```

    @user.create_reset_digest
    @user.send_password_reset_email
    flash[:info] = "Email sent with password reset instructions"
    redirect_to root_url
  else
    flash.now[:danger] = "Email address not found"
    render 'new'
  end
end

def edit
end
end

```

El código en el modelo `User` es similar al del método `create_activation_digest` utilizado en el llamado `before_create` (Listado 10.3), como se muestra en el Listado 10.42.

Listado 10.42: Agregando métodos de reinicio de contraseña al modelo `User`.

`app/models/user.rb`

```

class User < ActiveRecord::Base
  attr_accessor :remember_token, :activation_token, :reset_token
  before_save   :downcase_email
  before_create :create_activation_digest
  .

  .

  .

  # Activates an account.
  def activate
    update_attribute(:activated,     true)
    update_attribute(:activated_at, Time.zone.now)
  end

  # Sends activation email.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  # Sets the password reset attributes.
  def create_reset_digest
    self.reset_token = User.new_token
    update_attribute(:reset_digest,  User.digest(reset_token))
    update_attribute(:reset_sent_at, Time.zone.now)
  end

```

```
# Sends password reset email.
def send_password_reset_email
  UserMailer.password_reset(self).deliver_now
end

private

# Converts email to all lower-case.
def downcase_email
  self.email = email.downcase
end

# Creates and assigns the activation token and digest.
def create_activation_digest
  self.activation_token = User.new_token
  self.activation_digest = User.digest(activation_token)
end
end
```

Como se muestra en la [Figura 10.13](#), en este momento el comportamiento de la aplicación en caso de direcciones electrónicas no válidas ya está funcionando. Para hacer que la aplicación funcione al enviar una dirección electrónica válida, necesitamos definir un método gestor de reinicio de contraseña.

10.2.3 Método gestor de reinicio de contraseña

El código para enviar el correo de reinicio de contraseña se muestra en el [Listado 10.42](#) como sigue:

```
UserMailer.password_reset(self).deliver_now
```

El método gestor de reinicio de contraseña necesario para hacer que esto funcione es casi idéntico al gestor de la activación de la cuenta desarrollado en la [Sección 10.1.2](#). Primero creamos un método `password_reset` en el gestor de correo del usuario ([Listado 10.43](#)), y luego definimos las plantillas de las vistas tanto para los correos en texto plano ([Listado 10.44](#)) como para los que van en formato HTML ([Listado 10.45](#)).

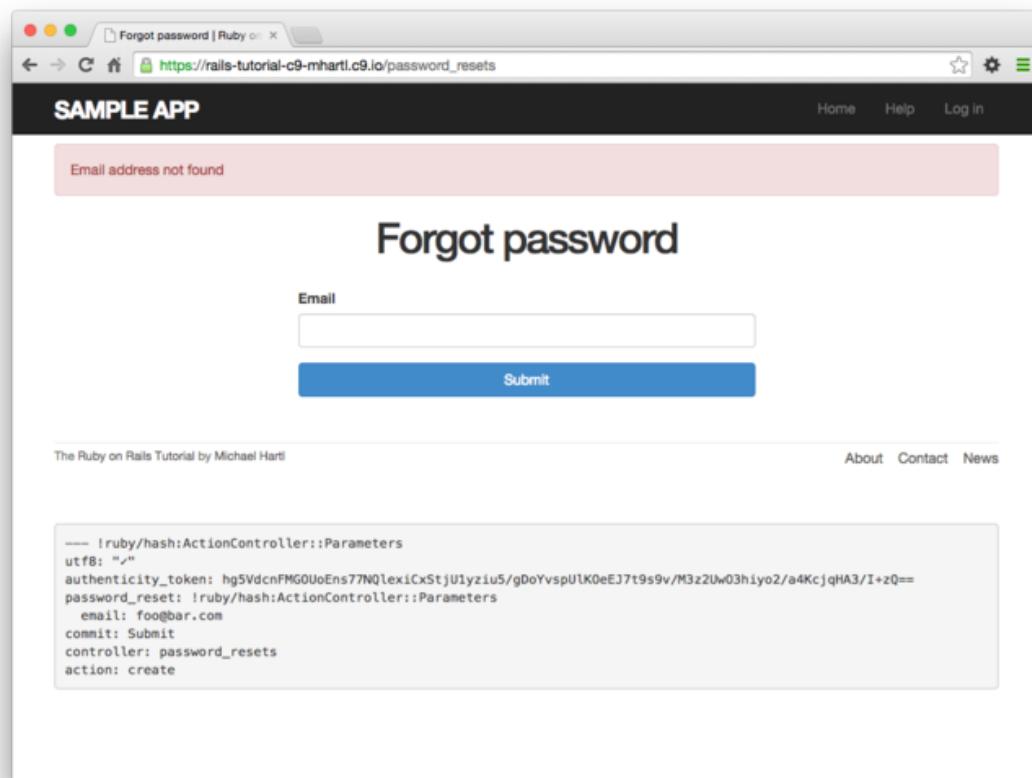


Figura 10.13: El formulario “forgot password” para una dirección electrónica no válida.

Listado 10.43: Enviando por correo electrónico el enlace de reinicio de contraseña.

app/mailers/user_mailer.rb

```
class UserMailer < ApplicationMailer

  def account_activation(user)
    @user = user
    mail to: user.email, subject: "Account activation"
  end

  def password_reset(user)
    @user = user
    mail to: user.email, subject: "Password reset"
  end
end
```

Listado 10.44: La plantilla para el reinicio de contraseña en formato texto.

app/views/user_mailer/password_reset.text.erb

To reset your password click the link below:

`<%= edit_password_reset_url(@user.reset_token, email: @user.email) %>`

This link will expire in two hours.

If you did not request your password to be reset, please ignore this email and your password will stay as it is.

Listado 10.45: La plantilla para el reinicio de contraseña en formato HTML.

app/views/user_mailer/password_reset.html.erb

```
<h1>Password reset</h1>

<p>To reset your password click the link below:</p>

<%= link_to "Reset password", edit_password_reset_url(@user.reset_token,
                                                       email: @user.email) %>

<p>This link will expire in two hours.</p>

<p>
  If you did not request your password to be reset, please ignore this email and
  your password will stay as it is.
</p>
```

Análogamente a los correos de activación de cuenta (Sección 10.1.2), podemos visualizar una versión preliminar de los correos de reinicio de contraseña utilizando el visor de correo electrónico de Rails. El código es totalmente análogo al del Listado 10.16, como se puede observar en el Listado 10.46.

Listado 10.46: Un método funcional para la vista previa del reinicio de contraseña.

```
test/mailers/previews/user_mailer_preview.rb

# Preview all emails at http://localhost:3000/rails/mailers/user_mailer
class UserMailerPreview < ActionMailer::Preview

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/account_activation
  def account_activation
    user = User.first
    user.activation_token = User.new_token
    UserMailer.account_activation(user)
  end

  # Preview this email at
  # http://localhost:3000/rails/mailers/user_mailer/password_reset
  def password_reset
    user = User.first
    user.reset_token = User.new_token
    UserMailer.password_reset(user)
  end
end
```

Con el código del Listado 10.46, las vistas previas de los correos electrónicos en formato HTML y texto plano se muestran en las Figuras 10.14 y 10.15.

Análogamente a la prueba del método de activación de cuenta (Listado 10.18), escribiré una pequeña prueba para el método de reinicio de contraseña, como se muestra en el Listado 10.47. Observe que necesitamos crear un token para el reinicio de la contraseña en las vistas; a diferencia del token de activación, el cual es creado para cada usuario por el llamado `before_create` (Listado 10.3), el token de reinicio de contraseña es creado únicamente cuando un usuario envía de forma satisfactoria el formulario “forgot password”. Esto ocurrirá de forma natural en una prueba de integración (Listado 10.54), pero en el contexto actual necesitamos crear uno manualmente.

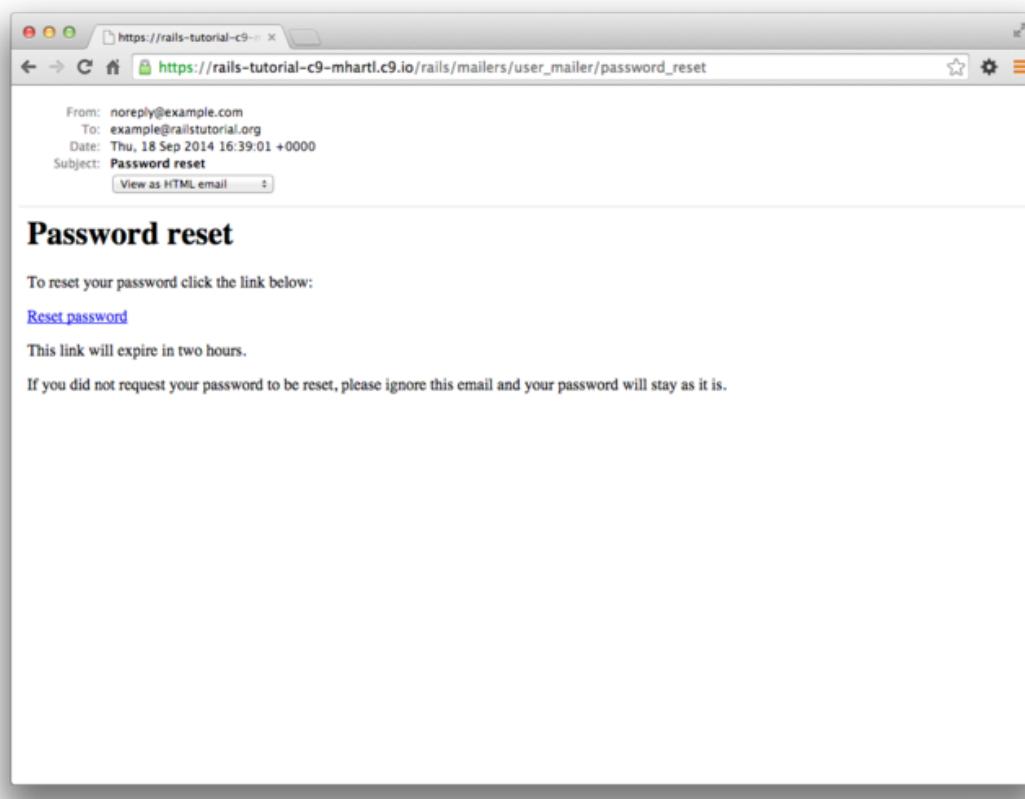


Figura 10.14: Una vista previa de la versión HTML del correo de reinicio de contraseña.

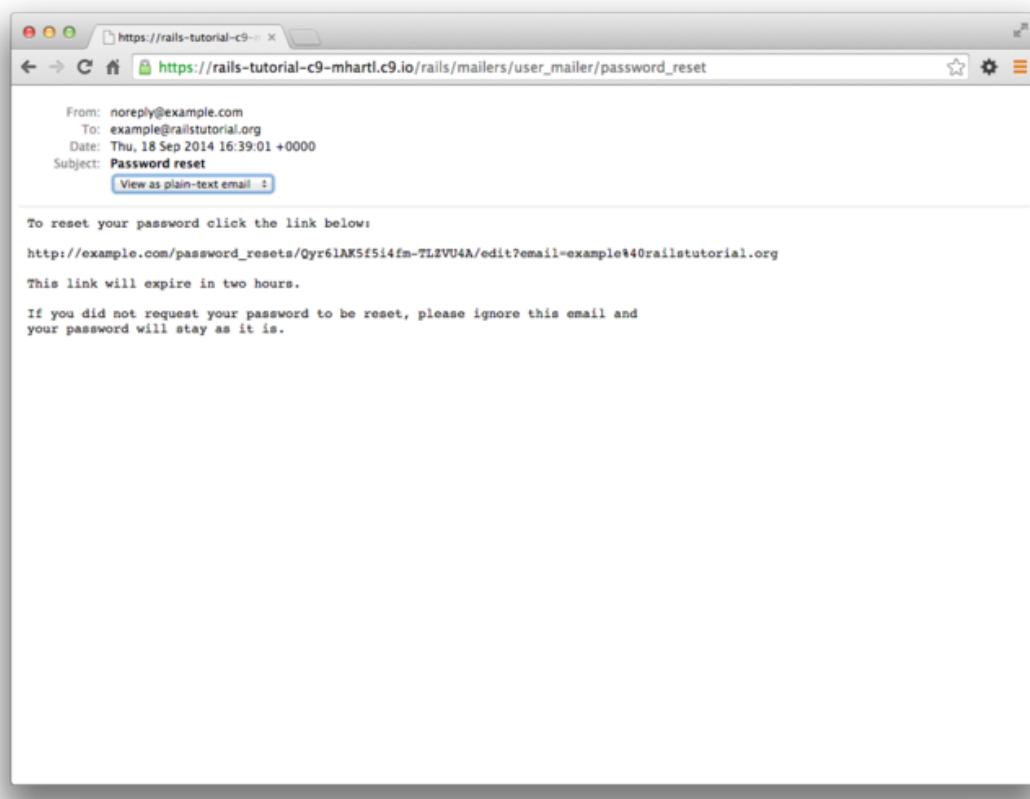


Figura 10.15: Una vista previa de la versión de texto del correo de reinicio de contraseña.

Listado 10.47: Agregando una prueba al método gestor de correos del reinicio de contraseña. **VERDE**

```
test/mailers/user_mailer_test.rb

require 'test_helper'

class UserMailerTest < ActionMailer::TestCase

  test "account_activation" do
    user = users(:michael)
    user.activation_token = User.new_token
    mail = UserMailer.account_activation(user)
    assert_equal "Account activation", mail.subject
    assert_equal [user.email], mail.to
    assert_equal ["noreply@example.com"], mail.from
    assert_match user.name, mail.body.encoded
    assert_match user.activation_token, mail.body.encoded
    assert_match CGI::escape(user.email), mail.body.encoded
  end

  test "password_reset" do
    user = users(:michael)
    user.reset_token = User.new_token
    mail = UserMailer.password_reset(user)
    assert_equal "Password reset", mail.subject
    assert_equal [user.email], mail.to
    assert_equal ["noreply@example.com"], mail.from
    assert_match user.reset_token, mail.body.encoded
    assert_match CGI::escape(user.email), mail.body.encoded
  end
end
```

En este momento, el conjunto de pruebas debería estar en **VERDE**:

Listado 10.48: **VERDE**

```
$ bundle exec rake test
```

Con el código de los Listados 10.43, 10.44, y 10.45, el envío de una dirección electrónica válida se muestra en la Figura 10.16. El correo correspondiente se muestra en la bitácora del servidor y debe verse de forma similar al Listado 10.49.

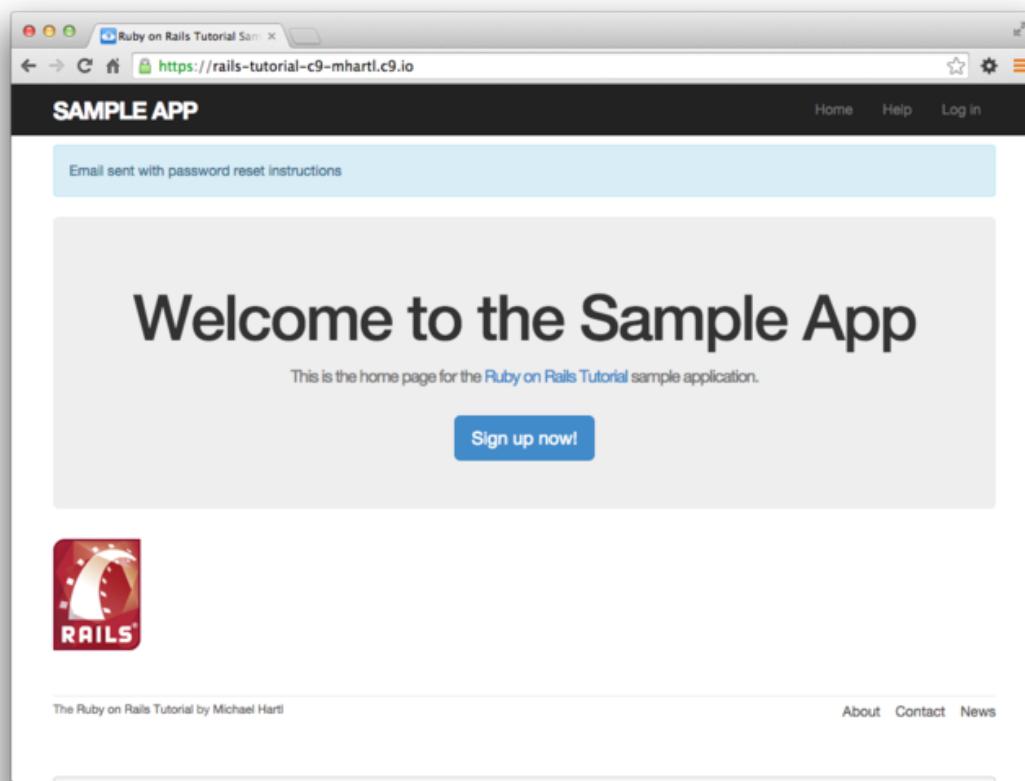


Figura 10.16: El resultado de enviar una dirección electrónica válida.

Listado 10.49: Un ejemplo de correo electrónico para el reinicio de contraseña extraído de la bitácora del servidor.

```
Sent mail to michael@michaelhartl.com (66.8ms)
Date: Thu, 04 Sep 2014 01:04:59 +0000
From: noreply@example.com
To: michael@michaelhartl.com
Message-ID: <5407babbe139_8722b257d04576a@mhartl-rails-tutorial-953753.mail>
Subject: Password reset
Mime-Version: 1.0
Content-Type: multipart/alternative;
boundary="====_mimepart_5407babbe3505_8722b257d045617";
charset=UTF-8
Content-Transfer-Encoding: 7bit

=====_mimepart_5407babbe3505_8722b257d045617
Content-Type: text/plain;
charset=UTF-8
Content-Transfer-Encoding: 7bit

To reset your password click the link below:

http://rails-tutorial-c9-mhartl.c9.io/password\_resets/3BdBrXeQZSWqFIDRN8cxHA/
edit?email=michael%40michaelhartl.com

This link will expire in two hours.

If you did not request your password to be reset, please ignore this email and
your password will stay as it is.
=====_mimepart_5407babbe3505_8722b257d045617
Content-Type: text/html;
charset=UTF-8
Content-Transfer-Encoding: 7bit

<h1>Password reset</h1>

<p>To reset your password click the link below:</p>

<a href="http://rails-tutorial-c9-mhartl.c9.io/
password_resets/3BdBrXeQZSWqFIDRN8cxHA/
edit?email=michael%40michaelhartl.com">Reset password</a>

<p>This link will expire in two hours.</p>

<p>
If you did not request your password to be reset, please ignore this email and
your password will stay as it is.
</p>
=====_mimepart_5407babbe3505_8722b257d045617--
```

10.2.4 Reiniciando la contraseña

Para hacer que los enlaces de la forma

```
http://example.com/password_resets/3BdBrXeQZSWqFIDRN8cxHA/edit?email=foo%40bar.com
```

funcionen, necesitamos un formulario para reinicio de contraseñas. La tarea es similar a la actualización de usuarios mediante la vista de edición de usuario ([Listado 9.2](#)), pero en este caso únicamente con los campos de contraseña y confirmación. Aunque hay una dificultad adicional: esperamos encontrar al usuario mediante su dirección electrónica, lo cual significa que necesitamos este valor tanto en las acciones `edit` como en `update`. El correo estará automáticamente disponible en la acción `edit` debido a su presencia en el enlace anterior, pero luego de que enviamos el formulario, se perderá su valor. La solución es utilizar un *campo oculto* para guardar (pero sin mostrar) la dirección electrónica en la página, y luego enviarla junto con el resto de la información del formulario. El resultado se muestra en el [Listado 10.50](#).

Listado 10.50: El formulario de reinicio de contraseña.

```
app/views/password_resets/edit.html.erb
```

```
<% provide(:title, 'Reset password') %>
<h1>Reset password</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user, url: password_reset_path(params[:id])) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= hidden_field_tag :email, @user.email %>

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

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

      <%= f.submit "Update password", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Observe que el [Listado 10.50](#) utiliza el auxiliar de etiquetas de formulario

```
hidden_field_tag :email, @user.email
```

en vez de

```
f.hidden_field :email, @user.email
```

porque el primero coloca el correo en `params[:email]`, mientras que éste último lo coloca en `params[:user][:email]`.

Para hacer que el formulario se despliegue, necesitamos definir una variable `@user` en la acción `edit` del controlador de reinicio de contraseñas. Como con la activación de la cuenta ([Listado 10.29](#)), esto implica encontrar al usuario que corresponde a la dirección electrónica en `params[:email]`. A continuación necesitamos verificar que el usuario es válido, es decir: que existe, está activado y está autenticado de acuerdo al token de reinicio de `params[:id]` (usando el método generalizado `authenticated?` definido en el [Listado 10.24](#)). Debido a que la existencia de un `@user` válido es necesaria tanto en la acción `edit` como en la acción `update`, pondremos el código para buscarlo y validararlo en un par de filtros previos, como se muestra en el [Listado 10.51](#).

Listado 10.51: La acción `edit` para el reinicio de contraseña.

`app/controllers/password_resets_controller.rb`

```
class PasswordResetsController < ApplicationController
  before_action :get_user,   only: [:edit, :update]
  before_action :valid_user, only: [:edit, :update]

  def edit
  end

  private

  def get_user
    @user = User.find_by(email: params[:email])
  end
```

```
# Confirms a valid user.
def valid_user
  unless (@user && @user.activated? &&
         @user.authenticated?(:reset, params[:id]))
    redirect_to root_url
  end
end
end
```

En el [Listado 10.51](#), compare el uso de

```
authenticated?(:reset, params[:id])
```

con

```
authenticated?(:remember, cookies[:remember_token])
```

del [Listado 10.26](#) y con

```
authenticated?(:activation, params[:id])
```

del [Listado 10.29](#). Juntos, estos tres casos completan los métodos de autenticación que se muestran en la [Tabla 10.1](#).

Con el código anterior, al seguir el enlace del [Listado 10.49](#) se debería mostrar un formulario de reinicio de contraseña. El resultado se observa en la [Figura 10.17](#).

Para definir la acción **update** correspondiente a la acción **edit** del [Listado 10.51](#), necesitamos considerar cuatro casos: un reinicio de contraseña expirado, una actualización satisfactoria, una actualización fallida (debido a una contraseña no válida), y una actualización fallida (que inicialmente parece “exitosa”) debido a una contraseña y su confirmación en blanco. El primer caso aplica tanto para la acción **edit** como para **update**, y por tanto pertenece de forma lógica a un filtro previo ([Listado 10.52](#)). Los siguientes dos casos corresponden a los dos bloques **if** del enunciado principal que se muestra en el

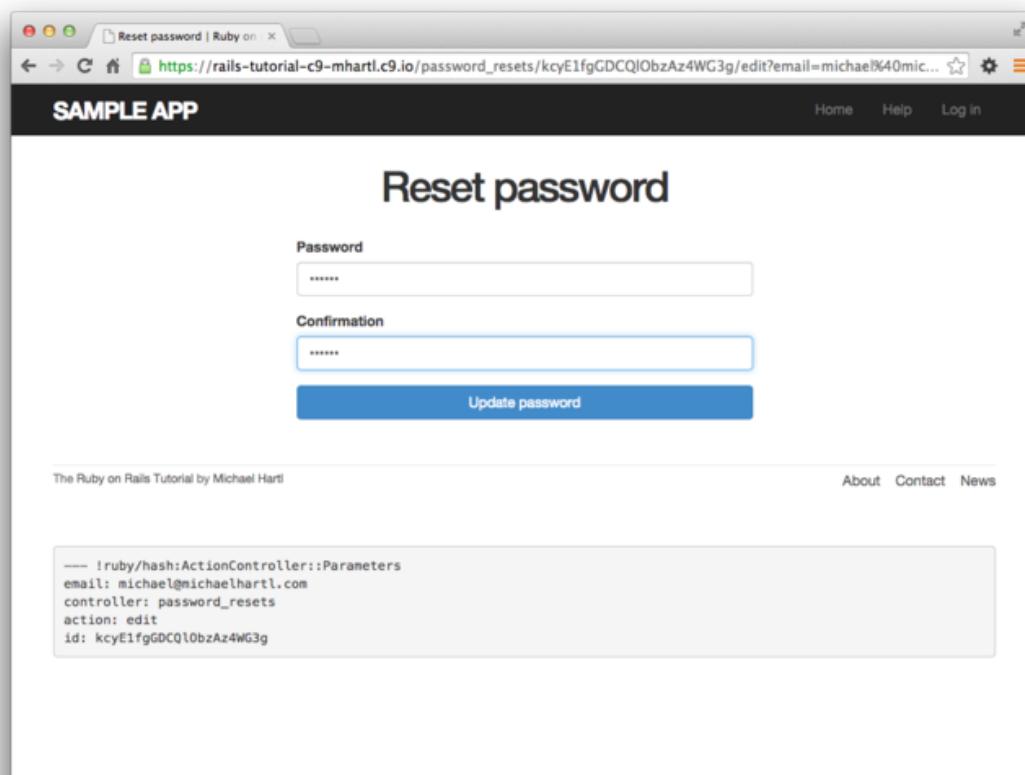


Figura 10.17: El formulario de reinicio de contraseña.

Listado 10.52. Como el formulario de edición está modificando un objeto de un modelo Active Record (es decir, un usuario), podemos depender del parcial compartido que aparece en el Listado 10.50 para mostrar mensajes de error. La única excepción es el caso en el que la contraseña está vacía, lo cual, en este momento, es permitido por nuestro modelo **User** (Listado 9.10) y por tanto necesita ser atrapada y manejada explícitamente.⁹ Nuestro método en este caso es agregar un error directamente a los mensajes de error del objeto **@user**:

```
@user.errors.add(:password, "can't be empty")
```

Listado 10.52: La acción **update** para el reinicio de contraseña.

app/controllers/password_resets_controller.rb

```
class PasswordResetsController < ApplicationController
  before_action :get_user,           only: [:edit, :update]
  before_action :valid_user,         only: [:edit, :update]
  before_action :check_expiration, only: [:edit, :update]

  def new
  end

  def create
    @user = User.find_by(email: params[:password_reset][:email].downcase)
    if @user
      @user.create_reset_digest
      @user.send_password_reset_email
      flash[:info] = "Email sent with password reset instructions"
      redirect_to root_url
    else
      flash.now[:danger] = "Email address not found"
      render 'new'
    end
  end

  def edit
  end

  def update
    if params[:user][:password].empty?
      @user.errors.add(:password, "can't be empty")
    end
  end
end
```

⁹Sólo necesitamos encargarnos del caso en el que la contraseña es vacía, porque si la confirmación es vacía, la validación de la confirmación (la cual no se ejecuta si la contraseña es vacía) atrapará el problema y proporcionará un mensaje de error apropiado.

```
render 'edit'
elsif @user.update_attributes(user_params)
  log_in @user
  flash[:success] = "Password has been reset."
  redirect_to @user
else
  render 'edit'
end
end

private

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

# Before filters

def get_user
  @user = User.find_by(email: params[:email])
end

# Confirms a valid user.
def valid_user
  unless (@user && @user.activated? &&
         @user.authenticated?(:reset, params[:id]))
    redirect_to root_url
  end
end

# Checks expiration of reset token.
def check_expiration
  if @user.password_reset_expired?
    flash[:danger] = "Password reset has expired."
    redirect_to new_password_reset_url
  end
end
end
```

La implementación del Listado 10.52 delega la prueba para la expiración del reinicio de contraseña al modelo **User** mediante el código

```
@user.password_reset_expired?
```

Para hacer que esto funcione, necesitamos definir el método **password_reset_expired?**. Como se indica en las plantillas de correo electrónico de la Sección 10.2.3, con-

sideraremos un reinicio de contraseña que expire a las dos horas de haber sido enviado, lo cual podemos expresar en Ruby como sigue:

```
reset_sent_at < 2.hours.ago
```

Esto puede ser confuso si usted lee `<` como “menor que”, porque entonces suena como “El reinicio de contraseña se envió hace menos de dos horas,” que es lo opuesto de lo que queremos. En este contexto, es mejor leer `<` como “hace más de”, lo cual nos da algo como “El reinicio de la contraseña fue enviado hace más de dos horas.” Esto *es* lo que queremos, y nos lleva al método `password_reset_expired?` del Listado 10.53. (Para una demostración formal de que la comparación es correcta, vea la prueba de la Sección 10.6.)

Listado 10.53: Agregando los métodos para el reinicio de la contraseña al modelo `User`.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns true if a password reset has expired.
  def password_reset_expired?
    reset_sent_at < 2.hours.ago
  end

  private
  .
  .
  .
end
```

Con el código del Listado 10.53, la acción `update` del Listado 10.52 debería estar funcionando. Los resultados para envío de datos válidos e inválidos se muestra en las Figuras 10.18 y 10.19. (Careciendo de paciencia para esperar un par de horas, cubriremos la tercera rama en una prueba, lo cual se deja como ejercicio (Sección 10.5).)

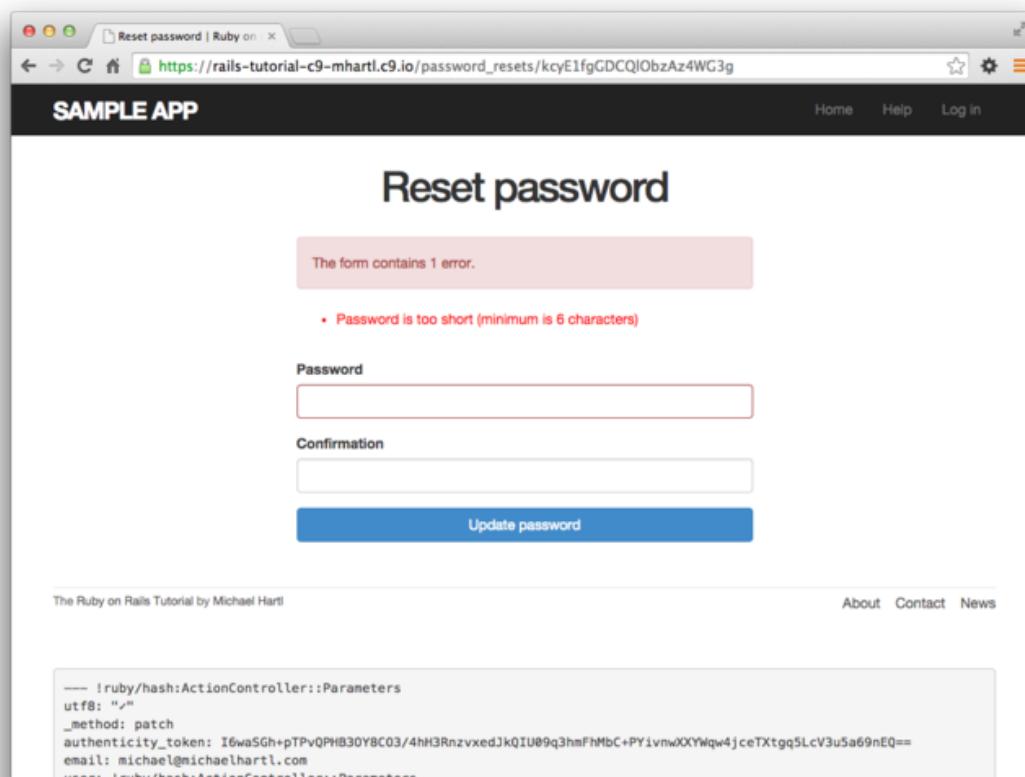


Figura 10.18: Un reinicio de contraseña fallido.

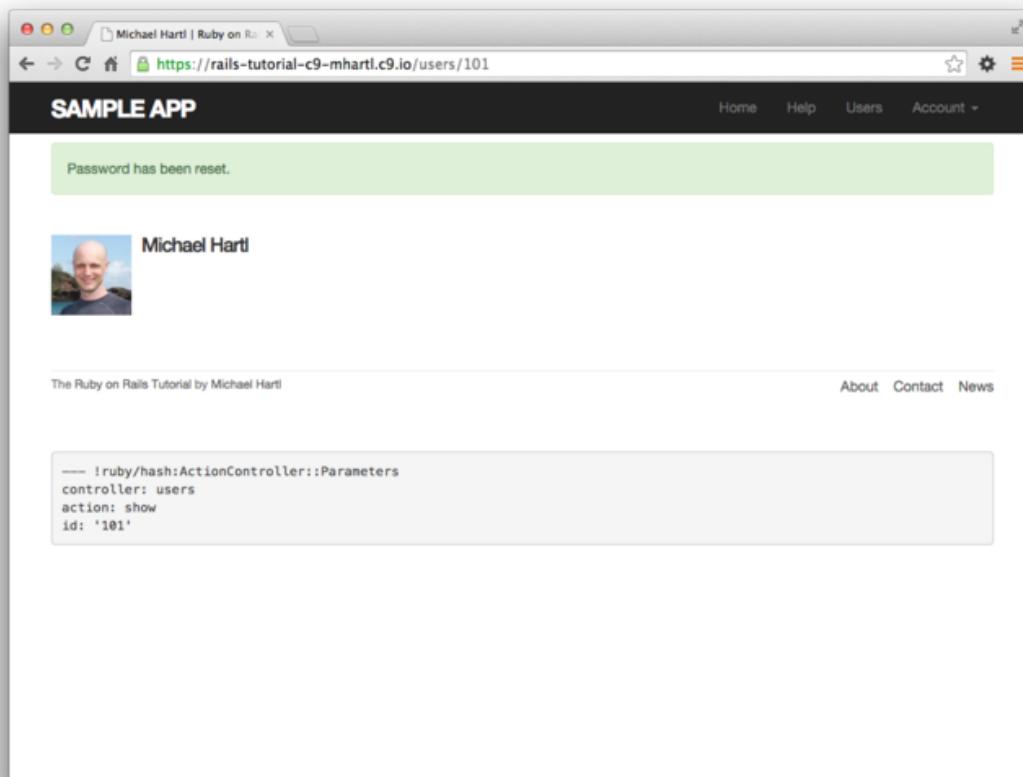


Figura 10.19: Un reinicio de contraseña exitoso.

10.2.5 Prueba de reinicio de contraseña

En esta Sección, escribiremos una prueba de integración que cubra dos de las tres ramas del [Listado 10.52](#), envío de datos válidos e inválidos. (Como se hizo notar antes, la prueba de la tercera rama se deja como ejercicio.) Empezaremos por generar un archivo de prueba para el reinicio de contraseñas:

```
$ rails generate integration_test password_resets
  invoke test_unit
  create test/integration/password_resets_test.rb
```

Los pasos para probar el reinicio de contraseña son muy similares a los de la prueba para la activación de la cuenta del [Listado 10.31](#), aunque existe una diferencia en la salida: primero visitamos el formulario “forgot password” y luego enviamos direcciones electrónicas válidas e inválidas, en el primero de estos casos, creamos un token para reinicio de contraseña y enviamos un correo. Luego visitamos el enlace del correo y nuevamente enviamos información tanto válida como inválida, verificando que el comportamiento sea el correcto en cada caso. La prueba resultante, se muestra en el [Listado 10.54](#), cuya lectura es un excelente ejercicio de lectura de código.

Listado 10.54: Una prueba de integración para el reinicio de contraseñas.

test/integration/password_resets_test.rb

```
require 'test_helper'

class PasswordResetsTest < ActionDispatch::IntegrationTest
  def setup
    ActionMailer::Base.deliveries.clear
    @user = users(:michael)
  end

  test "password resets" do
    get new_password_reset_path
    assert_template 'password_resets/new'
    # Invalid email
    post password_resets_path, password_reset: { email: "" }
    assert_not flash.empty?
    assert_template 'password_resets/new'
    # Valid email
  end
end
```

```

post password_resets_path, password_reset: { email: @user.email }
assert_not_equal @user.reset_digest, @user.reload.reset_digest
assert_equal 1, ActionMailer::Base.deliveries.size
assert_not flash.empty?
assert_redirected_to root_url
# Password reset form
user = assigns(:user)
# Wrong email
get edit_password_reset_path(user.reset_token, email: "")
assert_redirected_to root_url
# Inactive user
user.toggle!(:activated)
get edit_password_reset_path(user.reset_token, email: user.email)
assert_redirected_to root_url
user.toggle!(:activated)
# Right email, wrong token
get edit_password_reset_path('wrong token', email: user.email)
assert_redirected_to root_url
# Right email, right token
get edit_password_reset_path(user.reset_token, email: user.email)
assert_template 'password_resets/edit'
assert_select "input[name=email][type=hidden][value=?]", user.email
# Invalid password & confirmation
patch password_reset_path(user.reset_token),
      email: user.email,
      user: { password: "foobaz",
               password_confirmation: "barquux" }
assert_select 'div#error_explanation'
# Empty password
patch password_reset_path(user.reset_token),
      email: user.email,
      user: { password: "",
               password_confirmation: "" }
assert_select 'div#error_explanation'
# Valid password & confirmation
patch password_reset_path(user.reset_token),
      email: user.email,
      user: { password: "foobaz",
               password_confirmation: "foobaz" }
assert_is_logged_in?
assert_not flash.empty?
assert_redirected_to user
end
end

```

La mayoría de las ideas del Listado 10.54 han aparecido previamente en este tutorial; el único elemento realmente nuevo es la prueba de la etiqueta **input**:

```
assert_select "input[name=email][type=hidden][value=?]", user.email
```

Esto se asegura de que existe una etiqueta `input` con el nombre correcto, de tipo oculto, y una dirección electrónica:

```
<input id="email" name="email" type="hidden" value="michael@example.com" />
```

Con el código del Listado 10.54, nuestro conjunto de pruebas debería estar en **VERDE**:

Listado 10.55: **VERDE**

```
$ bundle exec rake test
```

10.3 Correo electrónico en producción

Como un objetivo de nuestro trabajo en los recordatorios de activación de la cuenta y contraseña, en esta Sección configuraremos nuestra aplicación de forma que realmente pueda enviar correo en producción. Primero pondremos en marcha un servicio gratuito para enviar correo, y luego configuraremos y desplegaremos nuestra aplicación.

Para enviar correo en producción, utilizaremos SendGrid, el cual está disponible como complemento en Heroku para cuentas verificadas. (Esto requiere que agregue información de una tarjeta de crédito a su cuenta de Heroku, pero no hay cargos cuando verifican una cuenta.) Para nuestros propósitos, el nivel “starter” es suficiente (al momento de escribir esto, éste nivel está limitado a 400 correos electrónicos diarios pero no cuesta nada). Podemos agregarlo a nuestra aplicación como sigue:

```
$ heroku addons:create sendgrid:starter
```

(Esto podría fallar en sistemas que tienen una versión vieja de la interfaz de línea de comandos de Heroku. En este caso, puede [actualizar a la última versión de herramientas de Heroku](#) o utilizar la sintaxis anterior **heroku addons:add sendgrid:starter**.)

Para configurar nuestra aplicación para que utilice SendGrid, necesitamos proporcionar la configuración de **SMTP** para nuestro ambiente productivo. Como se muestra en el [Listado 10.56](#), también tendrá que definir una variable **host** con la dirección de su sitio web de producción.

Listado 10.56: Configurando Rails para utilizar SendGrid en producción.

config/environments/production.rb

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

  config.action_mailer.raise_delivery_errors = true
  config.action_mailer.delivery_method = :smtp
  host = '<your heroku app>.herokuapp.com'
  config.action_mailer.default_url_options = { host: host }
  ActionMailer::Base.smtp_settings = {
    :address      => 'smtp.sendgrid.net',
    :port         => '587',
    :authentication => :plain,
    :user_name    => ENV['SENDGRID_USERNAME'],
    :password     => ENV['SENDGRID_PASSWORD'],
    :domain       => 'heroku.com',
    :enable_starttls_auto => true
  }
  .
  .
  .

end
```

La configuración de correo electrónico del [Listado 10.56](#) incluye el **user_name** y **password** de la cuenta SendGrid, pero observe que estos datos están configurados como variables de ambiente y son accesados mediante la variable **ENV** en vez de estar en código duro. Esto es una buena práctica para aplicaciones productivas, las cuales por motivos de seguridad nunca deberían de exponer información sensible como contraseñas planas en el código fuente. En este caso, estas variables son configuradas automáticamente mediante el complemento

SendGrid, pero veremos un ejemplo en la Sección 11.4.4 donde tendremos que definirlas nosotros mismos. En caso de que tenga curiosidad, puede ver las variables de ambiente utilizadas en el Listado 10.56 como sigue:

```
$ heroku config:get SENDGRID_USERNAME  
$ heroku config:get SENDGRID_PASSWORD
```

En este momento, debería mezclar sus cambios en la rama principal:

```
$ bundle exec rake test  
$ git add -A  
$ git commit -m "Add password resets & email configuration"  
$ git checkout master  
$ git merge account-activation-password-reset
```

Luego súbalos al repositorio remoto y despliegue en Heroku:

```
$ bundle exec rake test  
$ git push  
$ git push heroku  
$ heroku run rake db:migrate
```

Una vez que el despliegue en Heroku ha terminado, intente registrarse en la aplicación de ejemplo en producción utilizando una cuenta de correo electrónica que usted controle. Debería obtener un correo electrónico de activación como el implementado en la Sección 10.1.1 (Figura 10.20). Si usted olvida (o pretende que olvida) su contraseña, puede reiniciarla como se desarrolló en la Sección 10.2 (Figura 10.21).

10.4 Conclusión

Con la activación de la cuenta y reinicio de contraseña agregados a nuestra aplicación, la maquinaria para el registro, inicio y cierre de sesión está completa y es de grado profesional. El resto del *Tutorial de Ruby on Rails* construirá sobre esta base para crear un sitio con micromensajes estilo Twitter (Capítulo 11) y un

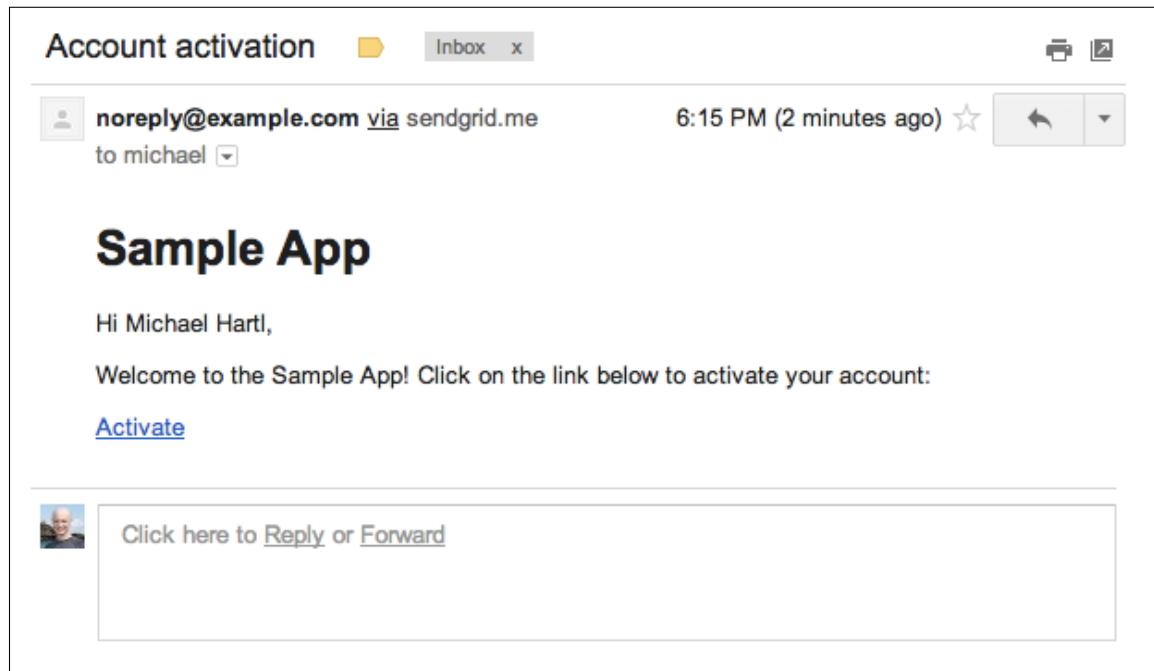


Figura 10.20: Un correo electrónico de activación de cuenta enviado desde producción.

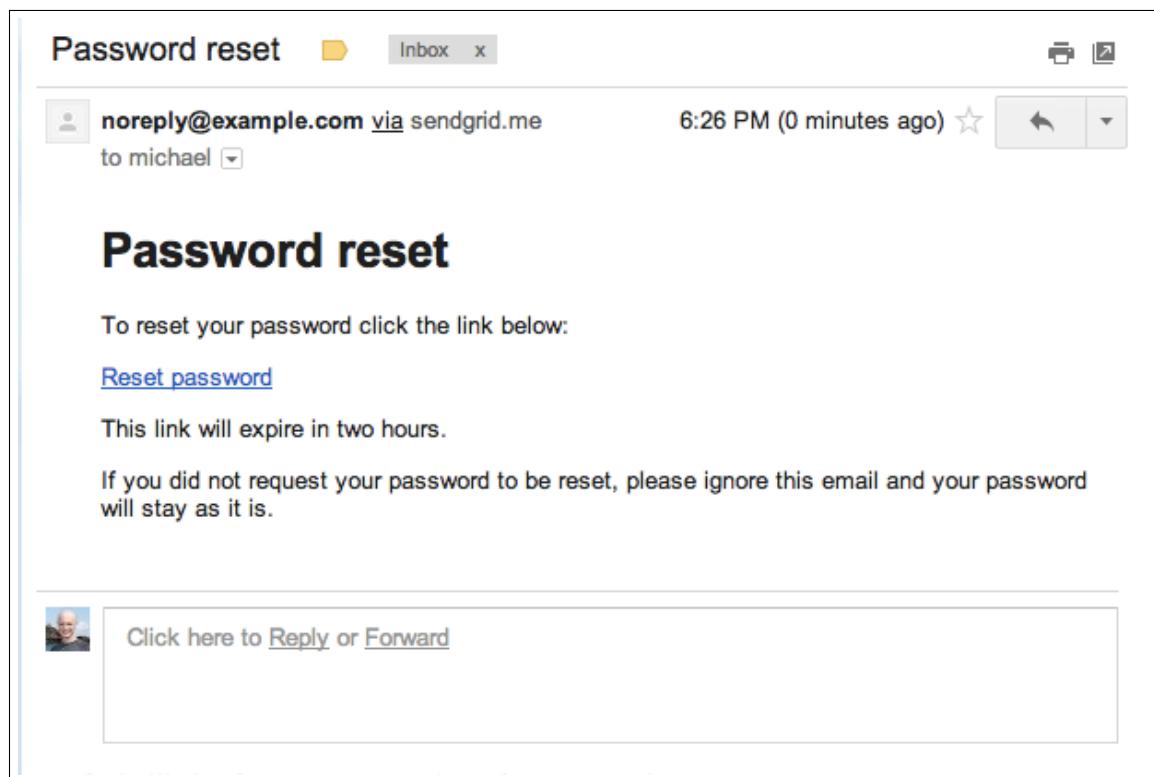


Figura 10.21: Un correo de reinicio de contraseña enviado desde producción.

status de mensajes publicados por los usuarios a los que se siguen ([Capítulo 12](#)). En el proceso, aprenderemos algunas de las características más poderosas de Rails, incluyendo la funcionalidad para subir imágenes, hacer consultas personalizadas a la base de datos, y modelado de datos avanzado con `has_many` y `has_many :through`.

10.4.1 Qué aprendimos en este capítulo

- Como en las sesiones, las activaciones de cuenta pueden ser modeladas como un recurso a pesar de no ser objetos Active Record.
- Rails puede generar acciones y vistas de Active Mailer para enviar correo.
- Action Mailer puede enviar correo en formato de texto plano y HTML.
- Como con acciones y vistas regulares, las variables de instancia definidas en las acciones para enviar correo, están disponibles en las vistas para enviar correo.
- Como en las sesiones y las activaciones de cuenta, los reinicios de contraseña puede ser modelados como un recurso a pesar de no ser objetos Active Record.
- Las activaciones de cuenta y los reinicios de contraseña utilizan un token generado para crear una URL única para activar usuarios o reiniciar contraseñas, respectivamente.
- Tanto las pruebas de correo como las de integración son útiles para verificar el comportamiento del gestor de correo.
- Podemos enviar correo en producción utilizando SendGrid.

10.5 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en

cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Escriba una prueba de integración para el reinicio de contraseña expirado que se muestra en el Listado 10.52 rellenando la plantilla que se muestra en el Listado 10.57. (Este código introduce `response.body`, el cual obtiene el cuerpo HTML completo de la página.) Hay muchas formas de probar el resultado de una expiración, pero el método que se sugiere en el Listado 10.57 es verificar (sin importar mayúsculas o minúsculas) que el cuerpo de la respuesta incluya la palabra “expired”.
2. En este momento, *todos* los usuarios son mostrados en la página del listado en `/users` y son visibles mediante la URL `/users/:id`, pero tiene sentido mostrar usuarios sólo si están activados. Corrija este comportamiento rellenando la plantilla del Listado 10.58.¹⁰ (Esto usa el método `where` de Active Record, del cual aprenderemos más en la Sección 11.3.3.) *Crédito extra:* Escriba las pruebas de integración tanto para `/users` como para `/users/:id`.
3. En el Listado 10.42, los métodos `activate` y `create_reset_digest` realizan dos llamados a `update_attribute`, cada uno de los cuales requiere una transacción a la base de datos por separado. Al llenar la plantilla del Listado 10.59, reemplace cada par de llamados a `update_attribute` con una sola llamada a `update_columns`, lo cual accesa la base de datos en una sola ocasión. Luego de realizar estos cambios, verifique que el conjunto de pruebas está aún en VERDE.

¹⁰Observe que el Listado 10.58 utiliza `and` en vez de `&&`. Los dos son casi idénticos, pero el último operador tiene una *precedencia* mayor, lo cual, en este caso, lo relaciona fuertemente con la URL raíz. Podemos arreglar el problema poniendo la URL raíz entre paréntesis, pero la forma idiomáticamente correcta de hacerlo es utilizando `and`.

Listado 10.57: Una prueba para el reinicio de contraseña expirado. VERDE

test/integration/password_resets_test.rb

```
require 'test_helper'

class PasswordResetsTest < ActionDispatch::IntegrationTest

  def setup
    ActionMailer::Base.deliveries.clear
    @user = users(:michael)
  end

  .

  .

  .

  test "expired token" do
    get new_password_reset_path
    post password_resets_path, password_reset: { email: @user.email }

    @user = assigns(:user)
    @user.update_attribute(:reset_sent_at, 3.hours.ago)
    patch password_reset_path(@user.reset_token),
          email: @user.email,
          user: { password: "foobar",
                    password_confirmation: "foobar" }
    assert_response :redirect
    follow_redirect!
    assert_match /FILL_IN/i, response.body
  end
end
end
```

Listado 10.58: Una plantilla para el código que muestra sólo los usuarios activos.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .

  .

  .

  def index
    @users = User.where(activated: FILL_IN).paginate(page: params[:page])
  end

  def show
    @user = User.find(params[:id])
    redirect_to root_url and return unless FILL_IN
  end
  .
end
```

```
•  
•  
end
```

Listado 10.59: Una plantilla para utilizar `update_columns`.

app/models/user.rb

```
class User < ActiveRecord::Base  
  attr_accessor :remember_token, :activation_token, :reset_token  
  before_save  :downcase_email  
  before_create :create_activation_digest  
  •  
  •  
  •  
  # Activates an account.  
  def activate  
    update_columns(activated: FILL_IN, activated_at: FILL_IN)  
  end  
  
  # Sends activation email.  
  def send_activation_email  
    UserMailer.account_activation(self).deliver_now  
  end  
  
  # Sets the password reset attributes.  
  def create_reset_digest  
    self.reset_token = User.new_token  
    update_columns(reset_digest: FILL_IN,  
                  reset_sent_at: FILL_IN)  
  end  
  
  # Sends password reset email.  
  def send_password_reset_email  
    UserMailer.password_reset(self).deliver_now  
  end  
  
  private  
  
  # Converts email to all lower-case.  
  def downcase_email  
    self.email = email.downcase  
  end  
  
  # Creates and assigns the activation token and digest.  
  def create_activation_digest  
    self.activation_token  = User.new_token  
    self.activation_digest = User.digest(activation_token)  
  end  
end
```

10.6 Comparación de la prueba de expiración

En esta Sección, probaremos que la comparación para la expiración de la contraseña de la [Sección 10.2.4](#) es correcta. Empezamos por definir dos intervalos de tiempo. Sea Δt_r el intervalo de tiempo desde que enviamos el reinicio de la contraseña y sea Δt_e el tiempo límite de expiración (por ejemplo, dos horas). Un reinicio de contraseña ha expirado si el intervalo de tiempo desde que el reinicio fue enviado es mayor que el tiempo límite de expiración:

$$\Delta t_r > \Delta t_e. \quad (10.1)$$

Si escribimos la hora actual como t_N , la hora en que se envió el reinicio de la contraseña como t_r , y el tiempo de expiración como t_e (digamos, hace un par de horas), entonces tenemos

$$\Delta t_r = t_N - t_r \quad (10.2)$$

y

$$\Delta t_e = t_N - t_e. \quad (10.3)$$

Conectando las ecuaciones (10.2) y (10.3) en (10.1) nos da

$$\begin{aligned} \Delta t_r &> \Delta t_e \\ t_N - t_r &> t_N - t_e \\ -t_r &> -t_e, \end{aligned}$$

las cuales al ser multiplicadas por -1 nos dan

$$t_r < t_e. \quad (10.4)$$

Convirtiendo (10.4) en código con el valor $t_e = 2$ hours ago nos da el método `password_reset_expired?` que se muestra en el [Listado 10.53](#):

```
def password_reset_expired?
    reset_sent_at < 2.hours.ago
end
```

Como observamos en la [Sección 10.2.4](#), si leemos `<` como “hace más de” en vez de “menor que”, este código tiene sentido como en la sentencia “El reinicio de la contraseña fue enviado hace más de dos horas.”

Capítulo 11

Micromensajes de usuario

En el transcurso de desarrollar la parte principal de la aplicación de ejemplo, nos hemos encontrado con cuatro recursos—usuarios, sesiones, activaciones de cuenta y reinicio de contraseñas—pero sólo el primero de éstos es un modelo de Active Record con una tabla en la base de datos. Ha llegado la hora de agregar un segundo recurso de este tipo: los *micromensajes* de usuario, que son mensajes cortos asociados con un usuario en particular.¹ Vimos por primera vez los micromensajes en una forma muy rústica en el Capítulo 2, y en este capítulo crearemos una versión robusta del bosquejo de la Sección 2.3 al construir el modelo de datos **Micropost**, asociándolo con el modelo **User** usando los métodos **has_many** y **belongs_to**, y luego creando los formularios y los parciales necesarios para manipular y desplegar los resultados (incluyendo, en la Sección 11.4, la carga de imágenes). En el Capítulo 12, completaremos nuestro mini clon de Twitter al agregar la noción de *seguir* usuarios con la finalidad de recibir un avance de sus micromensajes.

11.1 Un modelo de micromensaje

Empezamos con el recurso Microposts al crear un modelo **Micropost**, que capture las características esenciales de los micromensajes. Lo que sigue va

¹El nombre está motivado por la descripción común de Twitter como un *microblog*; puesto que los blogs tienen publicaciones, los microblogs deberían tener micropublicaciones.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime

Figura 11.1: El modelo de datos del micromensaje.

construyendo sobre el trabajo de la [Sección 2.3](#); de igual forma que con el modelo de esa sección, nuestro nuevo modelo de micromensajes incluirá validaciones de datos y una asociación con el modelo **User**. A diferencia de ése modelo, éste será totalmente probado y tendrá un *ordenamiento* por default, así como una *destrucción* automática si su usuario padre es destruido.

Si usted está utilizando Git para el control de versiones, le sugiero que cree una rama en este momento:

```
$ git checkout master
$ git checkout -b user-microposts
```

11.1.1 El modelo básico

El modelo **Micropost** necesita sólo de dos atributos: un atributo **content** que contendrá el contenido del micromensaje y un **user_id** para asociar el micromensaje con un usuario en particular. El resultado es un modelo **Micropost** con la estructura que se muestra en la [Figura 11.1](#).

Vale la pena observar que el modelo de la [Figura 11.1](#) utiliza el tipo de dato **text** para el contenido del micromensaje (en vez de **string**), el cual es capaz de almacenar una cantidad arbitraria de texto. Aún cuando el contenido será restringido a menos de 140 caracteres ([Sección 11.1.2](#)) y por tanto puede caber

dentro de un dato cuyo tipo sea una cadena de 255 caracteres, usar `text` expresa mejor la naturaleza de los micromensajes, los cuales son considerados de forma más natural como bloques de texto. Efectivamente, en la Sección 11.3.2 utilizaremos un *área* de texto en vez de un campo de texto para enviar los micromensajes. Además, al utilizar `text` tenemos mayor flexibilidad en caso de que en el futuro deseemos incrementar el límite de la longitud (como parte de la internacionalización, por ejemplo). Finalmente, utilizar el tipo `text` implica que *no hay diferencia en el desempeño* en producción,² por lo que no nos cuesta nada utilizarlo aquí.

Como con el caso del modelo `User` (Listado 6.1), generamos el modelo `Micropost` usando `generate model`:

```
$ rails generate model Micropost content:text user:references
```

El comando `generate` produce una migración para crear una tabla `microposts` en la base de datos (Listado 11.1); compárela con la migración análoga de la tabla `users` del Listado 6.2. La principal diferencia es el uso de `references`, las cuales automáticamente agregan una columna `user_id` (junto con un índice y una referencia a una llave foránea)³ para usarlas en la asociación usuario/micromensaje. Como con el modelo `User`, la migración del modelo `Micropost` automáticamente incluye la línea `t.timestamps`, la cual (como mencionamos en la Sección 6.1.1) agrega las columnas mágicas `created_at` y `updated_at` que se muestran en la Figura 11.1. (Pondremos la columna `created_at` a trabajar en las Secciones 11.1.4 y 11.2.1.)

Listado 11.1: La migración del micromensaje con el índice añadido.

```
db/migrate/[timestamp]_create_microposts.rb
```

²<http://www.postgresql.org/docs/9.1/static/datatype-character.html>

³La referencia a una llave foránea es una restricción a nivel base de datos que indica que el id del usuario en la tabla `microposts` se refiere a la columna `id` de la tabla `users`. Este detalle no será importante en este tutorial, y la restricción de la llave foránea no es soportada en todas las bases de datos. (Es soportada por PostgreSQL, la cual utilizamos en producción, pero no por el adaptador de base de datos de SQLite que usamos en desarrollo.) Aprenderemos más sobre llaves foráneas en la Sección 12.1.2.

```

class CreateMicroposts < ActiveRecord::Migration
  def change
    create_table :microposts do |t|
      t.text :content
      t.references :user, index: true, foreign_key: true

      t.timestamps null: false
    end
    add_index :microposts, [:user_id, :created_at]
  end
end

```

Como esperamos recuperar todos los micromensajes asociados con un id de usuario dado, en orden inverso a su creación, el [Listado 11.1](#) agrega un índice (Recuadro 6.2) sobre las columnas `user_id` y `created_at`:

```
add_index :microposts, [:user_id, :created_at]
```

Al incluir tanto la columna `user_id` como `created_at` como un arreglo, nos encargamos de que Rails cree un *índice de llave múltiple*, lo que significa que Active Record utiliza *ambas* llaves al mismo tiempo.

Con la migración del [Listado 11.1](#), podemos actualizar la base de datos como es usual:

```
$ bundle exec rake db:migrate
```

11.1.2 Validaciones de micromensajes

Ahora que hemos creado el modelo básico, agregaremos algunas validaciones para cumplir con las restricciones de diseño deseadas. Uno de los aspectos necesarios del modelo **Micropost** es la presencia de un id de usuario para indicar cuál usuario creó el micromensaje. La forma idiomáticamente correcta de hacer esto es utilizar *asociaciones* de Active Record, las cuales implementaremos en la [Sección 11.1.3](#), pero por ahora trabajaremos con el modelo **Micropost** directamente.

Las pruebas iniciales de los micromensajes son similares a las del modelo **User** (Listado 6.7). En el bloque **setup**, creamos un nuevo micromensaje que asociamos con un usuario válido del archivo fixtures, y luego verificamos que el resultado es válido. Como todo micromensaje debería tener un id de usuario, agregaremos una prueba que valide la presencia de **user_id**. Juntando estos elementos obtenemos la prueba del Listado 11.2.

Listado 11.2: Pruebas para la validez de un nuevo micromensaje. ROJO*test/models/micropost_test.rb*

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
    # This code is not idiomatically correct.
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
  end

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

  test "user id should be present" do
    @micropost.user_id = nil
    assert_not @micropost.valid?
  end
end
```

Como se indica en el comentario del método **setup**, el código para crear el micromensaje no es idiomáticamente correcto, y lo arreglaremos en la Sección 11.1.3.

La prueba de validación ya está en VERDE, pero la prueba que valida la presencia del id de usuario debería estar en ROJO porque actualmente no hay ninguna validación en el modelo **Micropost**:

Listado 11.3: ROJO

```
$ bundle exec rake test:models
```

Para arreglar esto, sólo necesitamos agregar dicha validación como se muestra en el Listado 11.4. (Observe la línea `belongs_to` del Listado 11.4, la cual es generada automáticamente por la migración del Listado 11.1. La Sección 11.1.3 revisa los efectos de esta línea con mayor profundidad.)

Listado 11.4: Una validación para el `user_id` de los micromensajes. VERDE
`app/models/micropost.rb`

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :user_id, presence: true
end
```

Las pruebas del modelo deberían estar en este momento en VERDE:

Listado 11.5: VERDE

```
$ bundle exec rake test:models
```

A continuación, agregaremos validaciones para el atributo `content` de los micromensajes (siguiendo el ejemplo de la Sección 2.3.2). Como con el `user_id`, el atributo `content` debe estar presente, y además está restringido a que su longitud no exceda los 140 caracteres, haciéndolo un verdadero *micromensaje*. Primero escribiremos algunas pruebas simples, las cuales generalmente siguen los ejemplos de las pruebas de validación del modelo `User` de la Sección 6.2, como se muestra en el Listado 11.6.

Listado 11.6: Pruebas de validación para el modelo `Micropost`. ROJO

`test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
  end
```

```

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

test "user id should be present" do
  @micropost.user_id = nil
  assert_not @micropost.valid?
end

test "content should be present" do
  @micropost.content = " "
  assert_not @micropost.valid?
end

test "content should be at most 140 characters" do
  @micropost.content = "a" * 141
  assert_not @micropost.valid?
end
end

```

Como en la Sección 6.2, el código del Listado 11.6 utiliza la multiplicación de cadena para la prueba que valida la longitud del micromensaje:

```

$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

```

El código correspondiente es virtualmente idéntico al de la validación del nombre para usuarios (Listado 6.16), como se muestra en el Listado 11.7.

Listado 11.7: Las validaciones del modelo **Micropost**. VERDE
app/models/micropost.rb

```

class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end

```

En este momento, el conjunto de pruebas debería estar en **VERDE**:

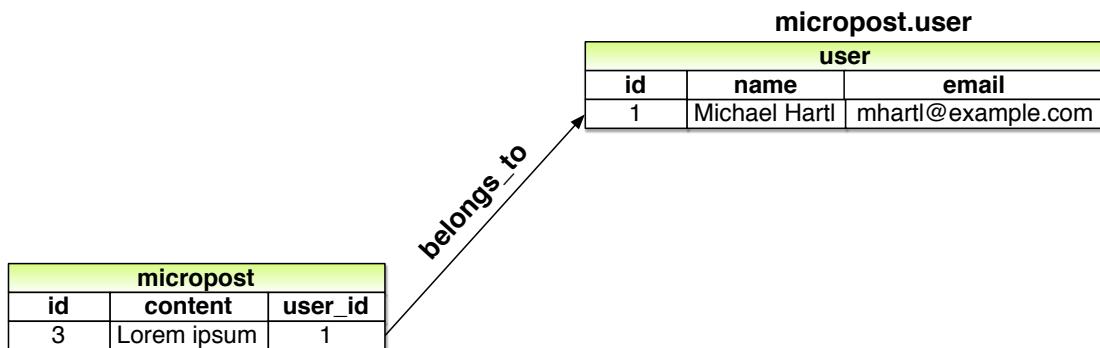


Figura 11.2: La relación `belongs_to` entre un micromensaje y su usuario asociado.

Listado 11.8: VERDE

```
$ bundle exec rake test
```

11.1.3 Asociaciones Usuario / Micromensaje

Cuando construimos modelos de datos para aplicaciones web, es esencial ser capaz de establecer *asociaciones* entre modelos individuales. En este caso, cada micromensaje es asociado con un usuario, y cada usuario es asociado con (potencialmente) muchos micromensajes—una relación que vimos brevemente en la Sección 2.3.3 y que se muestra de forma esquemática en las Figuras 11.2 y 11.3. Como parte de implementar estas asociaciones, escribiremos pruebas para el modelo `Micropost` y un par de pruebas para el modelo `User`.

Usando la asociación `belongs_to` / `has_many` definida en esta sección, Rails construye los métodos que se muestran en la Tabla 11.1. Observe en dicha tabla, que en vez de

```
Micropost.create
Micropost.create!
Micropost.new
```

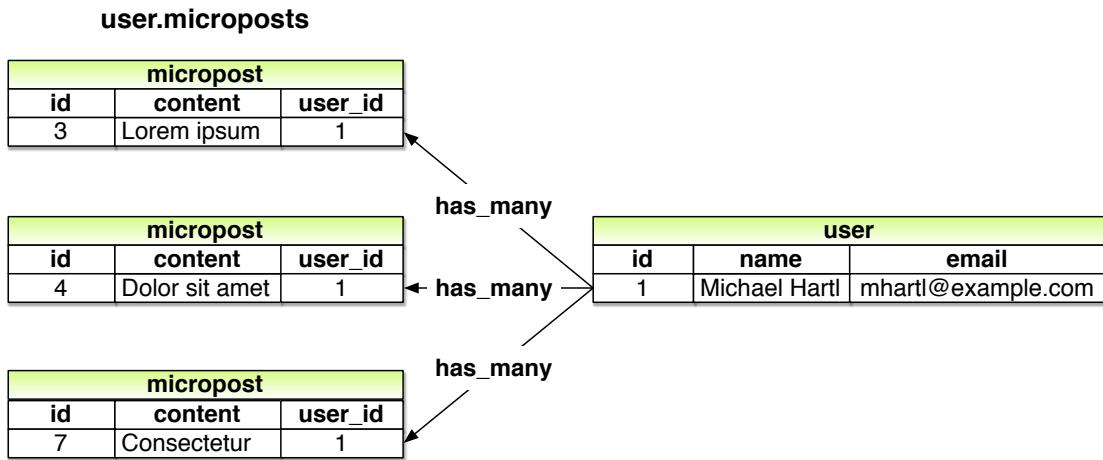


Figura 11.3: La relación **has_many** entre un usuario y sus micromensajes.

tenemos

```

user.microposts.create
user.microposts.create!
user.microposts.build
  
```

Estos últimos métodos constituyen la forma idiomáticamente correcta de crear un micromensaje, digamos, *mediante* su asociación con un usuario. Cuando un nuevo micromensaje se crea de esta forma, su **user_id** automáticamente es asociado con el valor correcto. En particular, podemos reemplazar el código

```

@user = users(:michael)
# This code is not idiomatically correct.
@micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
  
```

del Listado 11.2 con esto:

```

@user = users(:michael)
@micropost = @user.microposts.build(content: "Lorem ipsum")
  
```

Método	Propósito
<code>micropost.user</code>	Regresa el objeto <code>User</code> asociado con el micromensaje
<code>user.microposts</code>	Regresa una colección de micromensajes del usuario
<code>user.microposts.create(arg)</code>	Crea un micromensaje asociado con <code>user</code>
<code>user.microposts.create!(arg)</code>	Crea un micromensaje asociado con <code>user</code> (en caso de fallo arroja una excepción)
<code>user.microposts.build(arg)</code>	Regresa un nuevo objeto <code>Micropost</code> asociado con <code>user</code>
<code>user.microposts.find_by(id: 1)</code>	Busca el micromensaje con id <code>1</code> y <code>user_id</code> igual a <code>user.id</code>

Tabla 11.1: Un resumen de los métodos que asocian usuario / micromensaje.

(Como con `new`, `build` regresa un objeto en memoria pero no modifica la base de datos.) Una vez que definimos las asociaciones apropiadas, la variable `@micropost` resultante, automáticamente tendrá un atributo `user_id` que corresponde al id de su usuario.

Para que el código como `@user.microposts.build` funcione, necesitamos actualizar los modelos `User` y `Micropost` con el código que los relaciona entre sí. La primera parte de la asociación fue incluída automáticamente por la migración del Listado 11.1 mediante `belongs_to :user`, como se muestra en el Listado 11.9. La segunda parte de la asociación, `has_many :microposts`, necesita agregarse manualmente, como se muestra en el Listado 11.10.

Listado 11.9: Un micromensaje pertenece a un usuario. VERDE

`app/models/micropost.rb`

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

Listado 11.10: Un usuario tiene muchos micromensajes. VERDE

`app/models/user.rb`

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

Con la asociación hecha de esta forma, podemos actualizar el método `setup` del Listado 11.2 con la forma idiomáticamente correcta de construir un nuevo micromensaje, como se muestra en el Listado 11.11.

Listado 11.11: Usando código idiomáticamente correcto para construir un micromensaje. VERDE

```
test/models/micropost_test.rb

require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
    @micropost = @user.microposts.build(content: "Lorem ipsum")
  end

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

  test "user id should be present" do
    @micropost.user_id = nil
    assert_not @micropost.valid?
  end

  .
  .
  .

end
```

Por supuesto, luego de esta pequeña refactorización el conjunto de pruebas debería estar aún en VERDE:

Listado 11.12: VERDE

```
$ bundle exec rake test
```

11.1.4 Refinamiento de Micromensajes

En esta sección, agregaremos un par de refinamientos a la asociación usuario / micromensaje. En particular, nos encargaremos de recuperar los micromensajes de un usuario en un *orden* específico, y también crearemos micromensajes

dependientes de usuarios de forma que sean automáticamente destruidos si su usuario asociado es destruido.

Alcance por default

Por default, el método `user.microposts` no garantiza el orden de los mensajes, pero (siguiendo la convención de los blogs y Twitter) queremos extraerlos en orden inverso a como fueron creados, de forma que el más reciente sea el primero.⁴ Nos encargaremos de que esto suceda utilizando un *alcance por default*.

Esta es exactamente la clase de característica que podría fácilmente llevarnos a una prueba exitosa pero defectuosa (es decir, una prueba que pasa aún cuando el código de la aplicación es incorrecto), por lo que procederemos a utilizar el desarrollo orientado a pruebas para asegurarnos de que estamos probando lo correcto. En particular, escribiremos una prueba para verificar que el primer micromensaje en la base de datos es el mismo que el micromensaje del fixture que llamaremos `most_recent`, como se muestra en el Listado 11.13.

Listado 11.13: Probando el orden de los micromensajes. ROJO

`test/models/micropost_test.rb`

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase
  .
  .
  .
  test "order should be most recent first" do
    assert_equal microposts(:most_recent), Micropost.first
  end
end
```

El Listado 11.13 cuenta con que tenemos algunos micromensajes en el archivo fixtures, los cuales definimos como se muestra en el Listado 11.14.

⁴Encontramos una situación ligeramente similar en la Sección 9.5 en el contexto del listado de usuarios.

Listado 11.14: Fixtures de micromensajes.

```
test/fixtures/microposts.yml
```

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>

tau_manifesto:
  content: "Check out the @tauday site by @mhartl: http://tauday.com"
  created_at: <%= 3.years.ago %>

cat_video:
  content: "Sad cats are sad: http://youtu.be/PKffm2uI4dk"
  created_at: <%= 2.hours.ago %>

most_recent:
  content: "Writing a short test"
  created_at: <%= Time.zone.now %>
```

Observe que hemos establecido explícitamente la columna `created_at` usando Ruby embebido. Como es una columna “mágica” automáticamente actualizada por Rails, usualmente no es posible darle valor de forma manual, usualmente no es posible, pero en los fixtures sí lo es. En la práctica esto podría no ser necesario, y de hecho en muchos sistemas los fixtures son creados en orden. En este caso, el fixture al final del archivo es creado al último (y por tanto es el más reciente), pero sería tonto depender de este comportamiento, ya que es frágil y probablemente dependiente de sistema.

Con el código de los Listados 11.13 y 11.14, el conjunto de pruebas debería estar en ROJO:

Listado 11.15: ROJO

```
$ bundle exec rake test TEST=test/models/micropost_test.rb \
>           TESTOPTS="--name test_order_should_be_most_recent_first"
```

Haremos que la prueba pase usando un método de Rails llamado `default_scope`, el cual, entre otras cosas puede ser utilizado para establecer el orden por default en el que los elementos son recuperados de la base de datos. Para imponer un orden en particular, incluiremos el argumento `order` en `default_scope`, lo que nos permite ordenar por la columna `created_at` como sigue:

```
order(:created_at)
```

Desafortunadamente, esto ordena los resultados en orden *ascendente*, lo que significa que los micromensajes más viejos vienen primero. Para extraerlos en el orden inverso, podemos ir hacia un nivel más profundo e incluir una cadena con algo de SQL plano:

```
order('created_at DESC')
```

Aquí **DESC** es la instrucción SQL para “descendente”, es decir, en orden descendente de los más nuevos a los más viejos.⁵ En versiones anteriores de Rails, usar esta instrucción plana de SQL hubiera sido la única opción para conseguir el comportamiento deseado, pero desde Rails 4.0 también podemos utilizar una sintaxis de Ruby que resulta más natural:

```
order(created_at: :desc)
```

Agregar esto en un alcance por default para el modelo de micromensajes nos lleva al [Listado 11.16](#).

Listado 11.16: Ordenando los micromensajes con **default_scope**. VERDE
app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

El Listado 11.16 introduce la sintaxis “lambda literal” para un objeto llamado *Proc* (procedimiento) o *lambda*, que es una *función anónima* (una función

⁵SQL no es sensible a mayúsculas y minúsculas, pero es una convención escribir las palabras reservadas de SQL (tal como **DESC**) en mayúsculas.

creada sin nombre). La lambda literal `->` toma un bloque (Sección 4.3.2) y regresa un *Proc*, el cual puede luego ser evaluado con el método `call`. Podemos ver cómo funciona esto en la consola:

```
>> -> { puts "foo" }
=> #<Proc:0x007fab938d0108@(irb):1 (lambda)>
>> -> { puts "foo" }.call
foo
=> nil
```

(Este es un tema de Ruby algo avanzado, por lo que no se preocupe si de momento no tiene sentido para usted.)

Con el código del Listado 11.16, las pruebas deberían estar en VERDE:

Listado 11.17: VERDE

```
$ bundle exec rake test
```

Dependencia: destruir

A parte del ordenamiento apropiado, hay un segundo refinamiento que nos gustaría agregar a los micromensajes. Recuerde de la Sección 9.4 que los administradores del sitio tienen el poder de *destruir* usuarios. Es lógico pensar que, si un usuario es destruido, sus micromensajes deberían ser destruidos también.

Podemos implementar este comportamiento pasando una opción al método `has_many`, como se muestra en el Listado 11.18.

Listado 11.18: Asegurándonos que los micromensajes de un usuario son destruidos junto con él.

app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  .
  .
  .
end
```

Aquí la opción **dependent: :destroy** se encarga de que los micromensajes dependientes sean destruídos cuando el usuario mismo es destruido. Esto evita que haya micromensajes sin usuario colgados en la base de datos cuando los administradores decidan remover usuarios del sistema.

Podemos verificar que el [Listado 11.18](#) está funcionando con una prueba del modelo **User**. Todo lo que necesitamos hacer es guardar al usuario (de forma que obtenga un id) y crearle un micromensaje asociado. Luego verificamos que cuando destruimos el usuario, los micromensajes decrementan en 1. El resultado aparece en el [Listado 11.19](#). (Compare con la prueba de integración para los enlaces de borrado del [Listado 9.57](#).)

Listado 11.19: Una prueba para **dependent: :destroy**. **VERDE**
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "associated microposts should be destroyed" do
    @user.save
    @user.microposts.create!(content: "Lorem ipsum")
    assert_difference 'Micropost.count', -1 do
      @user.destroy
    end
  end
end
end
```

Si el código del [Listado 11.18](#) está funcionando correctamente, el conjunto de pruebas debería estar aún en **VERDE**:

Listado 11.20: **VERDE**

```
$ bundle exec rake test
```

11.2 Mostrando los micromensajes

Aunque aún no tenemos una forma de crear micromensajes a través de la web—eso viene en la [Sección 11.3.2](#)—esto no nos impide desplegarlos (y probar el despliegue). Siguiendo la idea de Twitter, no planeamos desplegar los micromensajes del usuario en una página de listado diferente sino directamente en la página que muestra al usuario, como se bosqueja en la [Figura 11.4](#). Empezaremos con plantillas bastante simples de Ruby embebido para agregar el despliegue de micromensajes al perfil del usuario, y luego agregaremos micromensajes para alimentar los datos de la [Sección 9.3.2](#) de forma que tengamos algo que mostrar.

11.2.1 Desplegando micromensajes

Nuestro plan es desplegar los micromensajes para cada usuario en sus respectivas páginas de perfil (`show.html.erb`), junto con un contador que indique cuántos micromensajes han creado. Como veremos, muchas de estas ideas son similares al trabajo que realizamos en la [Sección 9.3](#) cuando mostramos todos los usuarios.

Aunque no necesitaremos un controlador para los micromensajes sino hasta la [Sección 11.3](#), necesitaremos el directorio de las vistas en un momento más, por lo que generaremos ahora el controlador:

```
$ rails generate controller Microposts
```

Nuestro principal propósito en esta sección es mostrar todos los micromensajes de cada usuario. Como vimos en la [Sección 9.3.5](#), el código

```
<ul class="users">
  <%= render @users %>
</ul>
```

automáticamente despliega cada uno de los usuarios en la variable `@users` usando el parcial `_user.html.erb`. Definiremos un parcial análogo



Figura 11.4: Un bosquejo de una página de perfil con micromensajes.

_**micropost.html.erb** de forma que podamos utilizar la misma técnica en una colección de micromensajes como sigue:

```
<ol class="microposts">
  <%= render @microposts %>
</ol>
```

Observe que hemos utilizado la etiqueta *ordered list* **ol** (al contrario de las listas sin orden **ul**) porque los micromensajes están listados en un orden particular (cronológicamente inverso). El parcial correspondiente aparece en el [Listado 11.21](#).

Listado 11.21: Un parcial que muestra un solo micromensaje.

app/views/microposts/_micropost.html.erb

```
<li id="micropost-<%= micropost.id %>">
  <%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
  <span class="user"><%= link_to micropost.user.name, micropost.user %></span>
  <span class="content"><%= micropost.content %></span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
  </span>
</li>
```

Esto utiliza el increíble método auxiliar `time_ago_in_words`, cuyo significado es probablemente claro y cuyo efecto veremos en la Sección 11.2.2. El Listado 11.21 también agrega un id en el CSS para cada micromensaje usando

```
<li id="micropost-<%= micropost.id %>">
```

Esta es una buena práctica en general, puesto que ofrece la posibilidad de manipular los micromensajes individualmente en un futuro (usando JavaScript, por ejemplo).

El siguiente paso es ocuparnos de la problemática de mostrar un número potencialmente grande de micromensajes. Resolveremos este problema de la misma forma que hicimos con los usuarios en la Sección 9.3.3, digamos, usando paginación. De la misma forma que antes, utilizaremos el método `will_paginate`:

```
<%= will_paginate @microposts %>
```

Si usted compara esto con la línea análoga de la página del listado de usuarios, del Listado 9.41, podrá ver que sólo teníamos

```
<%= will_paginate %>
```

Esto funcionó porque, en el contexto del controlador de usuarios, `will_paginate` *supone* la existencia de una variable de instancia llamada `@users` (la cual, como

vimos en la Sección 9.3.3, debería pertenecer a la clase `ActiveRecord::Relation`). En este caso, puesto que aún estamos en el controlador de usuarios, pero queremos paginar *micromensajes*, debemos pasar la variable `@microposts` explícitamente a `will_paginate`. Por supuesto, esto significa que debemos definir tal variable en la acción `show` del usuario (Listado 11.22).

Listado 11.22: Agregando una variable de instancia `@microposts` a la acción `show` de usuario.

```
app/controllers/users_controller.rb

class UsersController < ApplicationController
  .
  .
  .
  def show
    @user = User.find(params[:id])
    @microposts = @user.microposts.paginate(page: params[:page])
  end
  .
  .
  .
end
```

Observe qué inteligente es `paginate` —funciona aún *a través* de la relación de micromensajes, alcanzando la tabla `microposts` y extrayendo la página deseada de micromensajes.

Nuestra tarea final es mostrar el número de micromensajes de cada usuario, lo cual podemos lograr con el método `count`:

```
user.microposts.count
```

De igual forma que con `paginate`, podemos utilizar el método `count` a través de la relación. En particular, `count` *no* extrae todos los micromensajes de la base de datos y luego cuenta el número de elementos del arreglo resultante, puesto que esto sería ineficiente conforme el número de micromensajes se incrementa. En vez de esto, realiza un cálculo directamente en la base de datos, pidiéndole que cuente los micromensajes con el `user_id` dado (una operación

para la cual todas las bases de datos están altamente optimizadas). (En el rarísimo caso de que este conteo sea un cuello de botella en su aplicación, podemos calcularlo rápidamente utilizando un [contador en caché](#).)

Juntando todos los elementos anteriores, estamos en posición de agregar micromensajes a la página del perfil, como se muestra en el [Listado 11.23](#). Observe el uso de `if @user.microposts.any?` (una construcción que vimos anteriormente en el [Listado 7.19](#)), la cual se asegura de que una lista vacía no se mostrará si el usuario no tiene micromensajes.

Listado 11.23: Agregando micromensajes a la página que muestra el usuario.
`app/views/users/show.html.erb`

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


<aside class="col-md-4">
    <section class="user_info">
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
  </aside>
  <div class="col-md-8">
    <% if @user.microposts.any? %>
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
  </div>
</div>


```

En este momento, podemos echar un vistazo a nuestra página de perfil actualizada de la [Figura 11.5](#). Es más bien ...*decepcionante*. Por supuesto, esto es porque no hay ningún micromensaje en este momento. Es hora de cambiar eso.

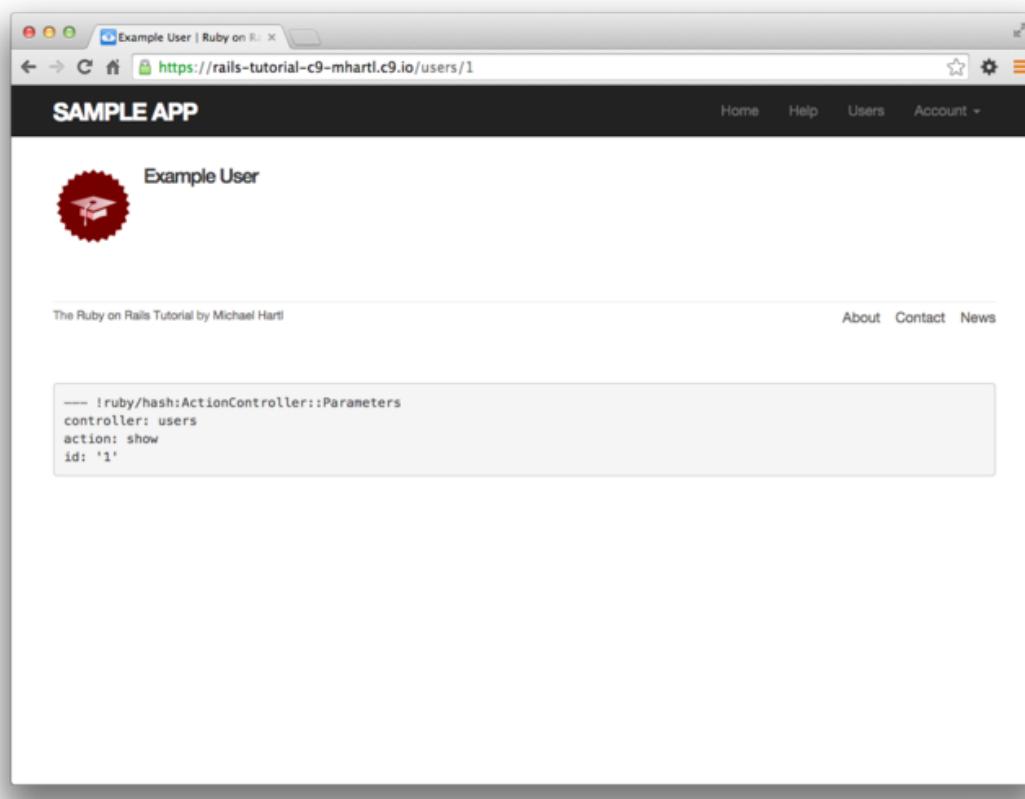


Figura 11.5: La página de perfil de usuario con el código para los micromensajes—pero sin ellos.

11.2.2 Micromensajes de ejemplo

Con todo el trabajo realizado en la Sección 11.2.1 creando las plantillas para los micromensajes del usuario, el resultado fue algo decepcionante. Podemos remediar esta triste situación agregando micromensajes a los datos semilla de la Sección 9.3.2.

Agregar micromensajes de ejemplo para *todos* los usuarios nos va a tomar un tiempo considerable, por lo que primero seleccionaremos los primeros seis usuarios (es decir, los cinco usuarios con gravatares personalizados y uno con el de default) usando el método `take`:

```
User.order(:created_at).take(6)
```

La llamada a `order` nos asegura que buscamos los primeros seis usuarios que fueron creados.

Para cada uno de los usuarios seleccionados, crearemos 50 micromensajes (suficientes para desbordar el límite de la paginación de 30). Para generar contenido de ejemplo para cada micromensaje, utilizaremos el método `Lorem.sentence` de la útil gema Faker.⁶ El resultado es el nuevo método para crear datos que se muestra en el Listado 11.24. (El motivo de realizar los ciclos como en el Listado 11.24 es entremezclar los micromensajes que vamos a usar en el status de avance (Sección 12.3). Si sólo realizamos el ciclo con los primeros usuarios, obtendremos avances de muchos micromensajes para el mismo usuario, lo cual es visualmente poco atractivo.)

Listado 11.24: Agregando micromensajes a los datos de ejemplo.

```
db/seeds.rb

.
.
.

users = User.order(:created_at).take(6)
50.times do
  content = Faker::Lorem.sentence(5)
  users.each { |user| user.microposts.create!(content: content) }
end
```

⁶`Faker::Lorem.sentence` regresa texto *lorem ipsum*; como observamos en el Capítulo 6, *lorem ipsum* tiene una fascinante historia.

En este momento, podemos realimentar la base de datos de desarrollo como es usual:

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

También debería de reiniciar el servidor de desarrollo de Rails.

Con esto, estamos en posición de disfrutar el resultado de nuestro trabajo de la Sección 11.2.1 al desplegar información para cada micromensaje.⁷ Los resultados preliminares aparecen en la Figura 11.6.

La página que se muestra en la Figura 11.6 no tiene ningún estilo para los micromensajes, por lo que agregremos algo de estilo (Listado 11.25) y echaremos un vistazo a las páginas resultantes.⁸

Listado 11.25: El CSS para micromensajes (incluyendo todo el CSS del capítulo).

app/assets/stylesheets/custom.css.scss

```
.*  
.*  
.*  
/* microposts */  
  
.microposts {  
  list-style: none;  
  padding: 0;  
  li {  
    padding: 10px 0;  
    border-top: 1px solid #e8e8e8;  
  }  
  .user {  
    margin-top: 5em;  
    padding-top: 0;  
  }  
  .content {  
    display: block;  
    margin-left: 60px;  
    img {
```

⁷Por diseño, el texto *lorem ipsum* de la gema Faker es generado aleatoriamente, por lo que el contenido de sus micromensajes de ejemplo será diferente para cada uno.

⁸Por conveniencia, el Listado 11.25 realmente tiene *todo* el CSS necesario para este capítulo.

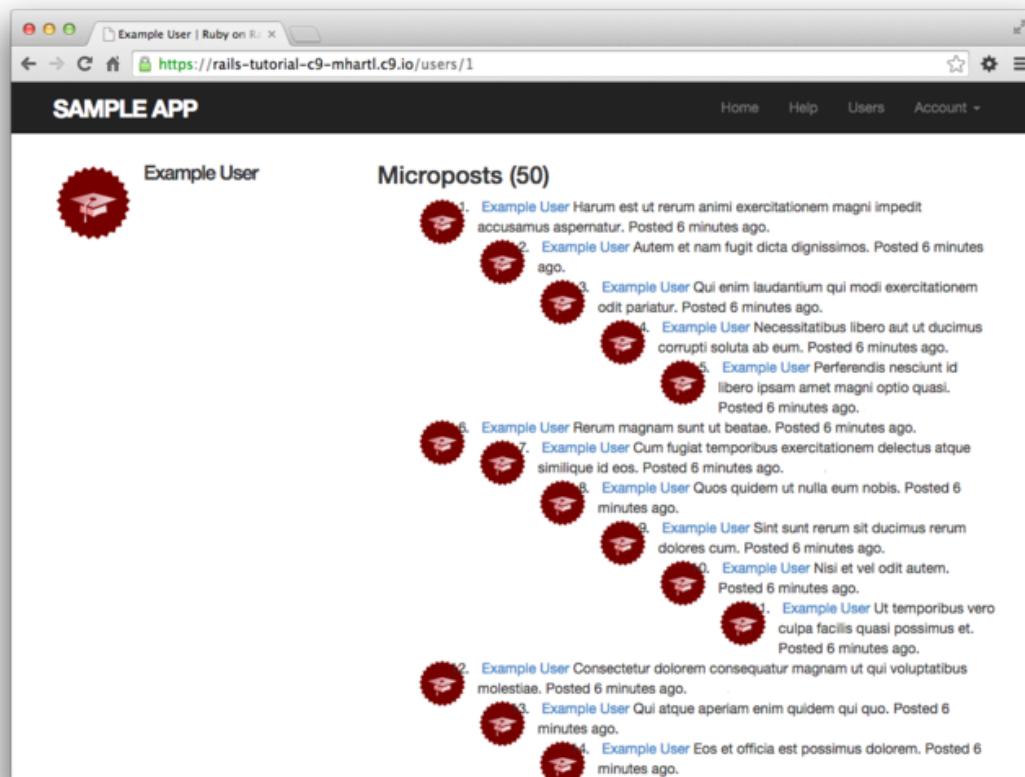


Figura 11.6: El perfil del usuario con micromensajes sin estilo.

```
        display: block;
        padding: 5px 0;
    }
}
.timestamp {
    color: $gray-light;
    display: block;
    margin-left: 60px;
}
.gravatar {
    float: left;
    margin-right: 10px;
    margin-top: 5px;
}
aside {
    textarea {
        height: 100px;
        margin-bottom: 5px;
    }
}

span.picture {
    margin-top: 10px;
    input {
        border: 0;
    }
}
```

La Figura 11.7 muestra la página de perfil del usuario para el primer usuario, mientras que la Figura 11.8 muestra el perfil del segundo usuario. Finalmente, la Figura 11.9 muestra la *segunda* página de micromensajes del primer usuario, junto con los enlaces de la paginación en la parte inferior de la pantalla. En los tres casos, observe que cada micromensaje mostrado indica el tiempo que ha transcurrido desde que fue creado (por ejemplo, “Publicado hace 1 minuto.”); este es el trabajo del método `time_ago_in_words` del Listado 11.21. Si usted espera un par de minutos y refresca la página, verá cómo el texto se actualiza automáticamente con respecto a la hora actual.

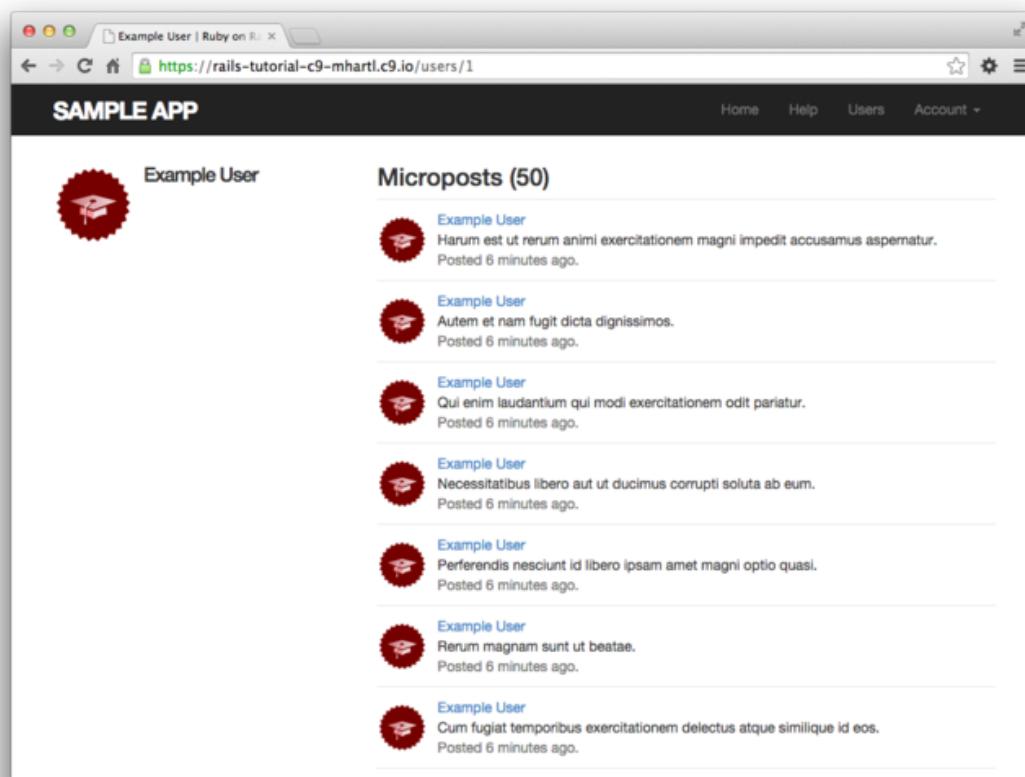


Figura 11.7: El perfil del usuario con micromensajes ([/users/1](#)).

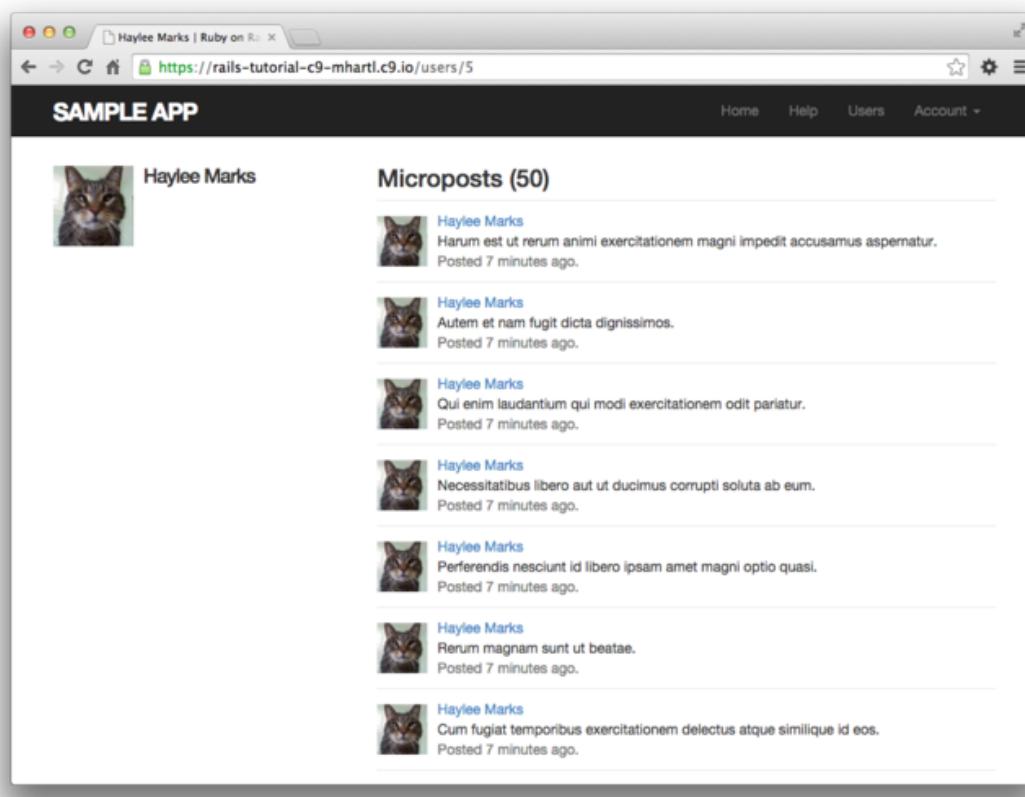


Figura 11.8: El perfil de un usuario diferente, también con micromensajes ([/users/5](#)).

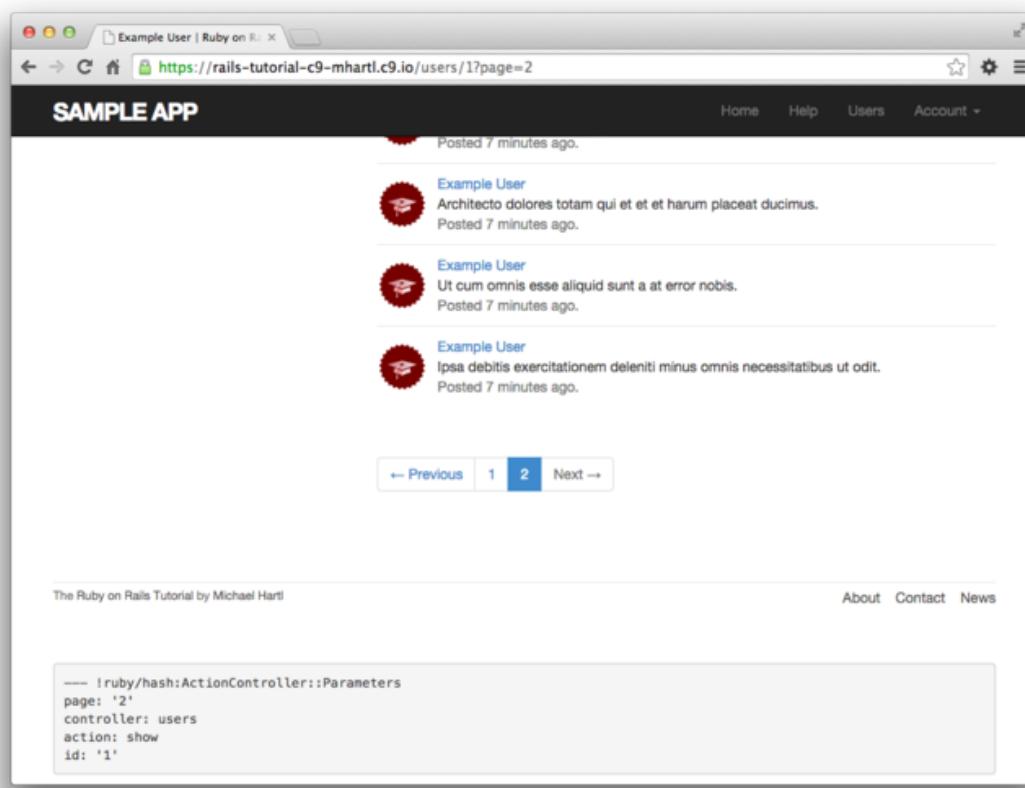


Figura 11.9: Los enlaces de paginación de los micromensajes ([/users/1?page=2](#)).

11.2.3 Pruebas a los micromensajes del perfil

Como los usuarios recién activados son redirigidos a sus páginas de perfil, ya tenemos una prueba de que la página del perfil se muestra correctamente ([Listado 10.31](#)). En esta sección, escribiremos una pequeña prueba de integración para algunos elementos de la página de perfil, incluyendo el trabajo de esta sección. Empezaremos generando una prueba de integración para los perfiles de los usuarios de nuestro sitio:

```
$ rails generate integration_test users_profile
  invoke  test_unit
  create    test/integration/users_profile_test.rb
```

Para probar el despliegue de los micromensajes en el perfil, necesitamos asociar los micromensajes del archivo fixture con un usuario. Rails proporciona una forma conveniente de construir asociaciones en los fixtures, como ésta:

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

Al identificar al usuario como `michael`, le decimos a Rails que asocie este micromensaje con el usuario correspondiente en el archivo fixture de usuarios:

```
michael:
  name: Michael Example
  email: michael@example.com
  .
  .
  .
```

Para probar la paginación de los micromensajes, generaremos algunos archivos fixture para micromensajes adicionales usando la misma técnica de Ruby embedido que utilizamos para crear usuarios adicionales en el [Listado 9.43](#):

```
<% 30.times do |n| %>
micropost_<%= n %>:
  content: <%= Faker::Lorem.sentence(5) %>
  created_at: <%= 42.days.ago %>
  user: michael
<% end %>
```

Juntando todo esto obtenemos la última versión del archivo fixtures para los micromensajes que se muestra en el [Listado 11.26](#).

Listado 11.26: Archivo fixtures de micromensajes con las asociaciones de usuario.

test/fixtures/microposts.yml

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael

tau_manifesto:
  content: "Check out the @tauday site by @mhartl: http://tauday.com"
  created_at: <%= 3.years.ago %>
  user: michael

cat_video:
  content: "Sad cats are sad: http://youtu.be/PKffm2uI4dk"
  created_at: <%= 2.hours.ago %>
  user: michael

most_recent:
  content: "Writing a short test"
  created_at: <%= Time.zone.now %>
  user: michael

<% 30.times do |n| %>
micropost_<%= n %>:
  content: <%= Faker::Lorem.sentence(5) %>
  created_at: <%= 42.days.ago %>
  user: michael
<% end %>
```

Con los datos que hemos preparado, la prueba misma es bastante sencilla: visitaremos la página de perfil del usuario y verificaremos el título de la misma, el nombre del usuario, el gravatar, el conteo de micromensajes y la paginación

de éstos. El resultado aparece en el Listado 11.27. Observe el uso del auxiliar `full_title` del Listado 4.2 para probar el título de la página, al cual tenemos acceso por haber incluído el módulo auxiliar de la aplicación en la prueba.⁹

Listado 11.27: Una prueba para el perfil del usuario. VERDE

```
test/integration/users_profile_test.rb

require 'test_helper'

class UsersController < ActionDispatch::IntegrationTest
  include ApplicationHelper

  def setup
    @user = users(:michael)
  end

  test "profile display" do
    get user_path(@user)
    assert_template 'users/show'
    assert_select 'title', full_title(@user.name)
    assert_select 'h1', text: @user.name
    assert_select 'h1>img.gravatar'
    assert_match @user.microposts.count.to_s, response.body
    assert_select 'div.pagination'
    @user.microposts.paginate(page: 1).each do |micropost|
      assert_match micropost.content, response.body
    end
  end
end
```

La afirmación sobre el conteo de micromensajes del Listado 11.27 utiliza `response.body`, el cual vimos brevemente en los ejercicios del Capítulo 10 (Sección 10.5). A pesar de su nombre, `response.body` contiene el código fuente completo de la página HTML (y no sólo el cuerpo de la página). Esto significa que si lo único que nos interesa es el número de micromensajes que aparece *en algún lugar* de la página, podemos buscarlo como se indica a continuación:

⁹Si desea refactorizar otras pruebas para utilizar `full_title` (como las del Listado 3.38), entonces debe incluir el auxiliar de la aplicación en `test_helper.rb`.

```
assert_match @user.microposts.count.to_s, response.body
```

Esta es una afirmación mucho menos específica que `assert_select`; en particular, a diferencia de `assert_select`, usar `assert_match` en este contexto no requiere que indiquemos en qué etiqueta HTML estamos buscando.

El Listado 11.27 también nos presenta una sintaxis anidada para `assert_select`:

```
assert_select 'h1>img.gravatar'
```

Esto verifica que exista una etiqueta `img` con clase `gravatar` dentro de una etiqueta (`h1`) en un nivel más alto.

Como el código de la aplicación estaba funcionando, el conjunto de pruebas debería estar en **VERDE**:

Listado 11.28: **VERDE**

```
$ bundle exec rake test
```

11.3 Manipulando micromensajes

Habiendo terminado tanto el modelado de datos como las plantillas para mostrar micromensajes, prestaremos atención ahora a la interfaz para crear estos micromensajes a través de la web. En esta sección, también veremos nuestra primera sugerencia de un *status de avance*—una noción que alcanzará su plenitud en el Capítulo 12. Finalmente, de igual forma que con los usuarios, haremos posible la destrucción de los micromensajes a través de la web.

Hay una cuestión diferente con respecto a las convenciones anteriores que vale la pena hacer notar: la interfaz del recurso de micromensajes se ejecutará principalmente a través de las páginas principal y la de Perfil, por lo que no necesitaremos acciones como `new` o `edit` en el controlador Microposts; sólo necesitaremos `create` y `destroy`. Esto nos lleva a las rutas para el recurso

Petición HTTP	URL	Action	Propósito
POST	/microposts	<code>create</code>	crear un nuevo micromensaje
DELETE	/microposts/1	<code>destroy</code>	eliminar un micromensaje con id 1

Tabla 11.2: Rutas RESTful proporcionadas por el recurso Microposts del Listado 11.29.

Microposts que se muestran en el Listado 11.29. El código del Listado 11.29 nos lleva a su vez a las rutas RESTful que se observan en la Tabla 11.2, las cuales constituyen un pequeño subconjunto de todo el conjunto de rutas que se muestran en la Tabla 2.3. Por supuesto, esta simplicidad es signo de estar *más* avanzados, no menos—hemos recorrido un largo camino desde nuestra dependencia en la creación de estructuras temporales que vimos en el Capítulo 2, y no necesitamos ya de la mayor parte de su complejidad.

Listado 11.29: Rutas del recurso Microposts.

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help' => 'static_pages#help'
  get 'about' => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup' => 'users#new'
  get 'login' => 'sessions#new'
  post 'login' => 'sessions#create'
  delete 'logout' => 'sessions#destroy'
  resources :users
  resources :account_activations, only: [:edit]
  resources :password_resets, only: [:new, :create, :edit, :update]
  resources :microposts, only: [:create, :destroy]
end
```

11.3.1 Control de acceso de micromensajes

Empezamos nuestro desarrollo del recurso Microposts con algo de control de acceso en el controlador Microposts. En particular, como accesamos los mi-

cromensajes a través de sus usuarios asociados, tanto las acciones `create` como `destroy` deben requerir que los usuarios hayan iniciado sesión.

Las pruebas para corroborar el status de sesión iniciada son similares a las que usamos en el controlador de usuarios (Listados 9.17 y 9.56). Simplemente emitimos la petición correcta a cada acción y confirmamos que el conteo de micromensajes no cambia y que el resultado es redirigido a la URL de inicio de sesión, como se muestra en el Listado 11.30.

Listado 11.30: Pruebas de autorización para el controlador Microposts. ROJO

`test/controllers/microposts_controller_test.rb`

```
require 'test_helper'

class MicropostsControllerTest < ActionController::TestCase

  def setup
    @micropost = microposts(:orange)
  end

  test "should redirect create when not logged in" do
    assert_no_difference 'Micropost.count' do
      post :create, micropost: { content: "Lorem ipsum" }
    end
    assert_redirected_to login_url
  end

  test "should redirect destroy when not logged in" do
    assert_no_difference 'Micropost.count' do
      delete :destroy, id: @micropost
    end
    assert_redirected_to login_url
  end
end
```

Escribir el código necesario para hacer que las pruebas del Listado 11.30 pasen requiere una pequeña refactorización primero. Recuerde que en la Sección 9.2.1 requerimos que se iniciara sesión utilizando un filtro previo que invocaba el método `logged_in_user` (Listado 9.12). En ese momento, necesitábamos ese método sólo en el controlador de usuarios, pero ahora lo necesitamos en el controlador de micromensajes también, por lo que lo desplazaremos al controlador de la aplicación, que es la clase base de todos los controladores (Sección 4.4.4). El resultado se muestra en el Listado 11.31.

Listado 11.31: Desplazando el método `logged_in_user` al controlador de la aplicación.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper

  private

    # Confirms a logged-in user.
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end
end
```

Para evitar la repetición de código, también debemos remover ahora `logged_in_user` del controlador de usuarios.

Con el código del Listado 11.31, el método `logged_in_user` está disponible en el controlador de micromensajes, lo que significa que podemos agregar las acciones `create` y `destroy` y luego restringir el acceso a ellas mediante un filtro previo, como se muestra en el Listado 11.32.

Listado 11.32: Agregando autorización a las acciones del controlador de micromensajes. **VERDE**

```
app/controllers/microposts_controller.rb

class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]

  def create
  end

  def destroy
  end
end
```

En este momento, las pruebas deberían pasar:

Listado 11.33: **VERDE**

```
$ bundle exec rake test
```

11.3.2 Creando micromensajes

En el [Capítulo 7](#), implementamos el registro de usuarios creando un formulario HTML que emitía una petición HTTP POST a la acción `create` del controlador Users. La implementación de la creación de micromensajes es similar; la principal diferencia es que, en vez de utilizar una página separada en /microposts/new, colocaremos el formulario en la página Home (es decir, en la ruta raíz /), como se bosqueja en la [Figura 11.10](#).

La última vez que vimos la página Home, se veía como en la [Figura 5.6](#)—es decir, tenía un botón “¡Regístrese ahora!” en medio. Puesto que la creación de micromensajes tiene sentido únicamente en el contexto de un usuario que ha iniciado sesión, uno de los objetivos de esta sección será entregar diferentes versiones de la página Home, dependiendo del status de sesión de quien la visita. Implementaremos esto en el [Listado 11.35](#) siguiente.

Empezaremos creando una acción `create` para los micromensajes, la cual es similar a su análoga de usuarios ([Listado 7.23](#)); la principal diferencia reside

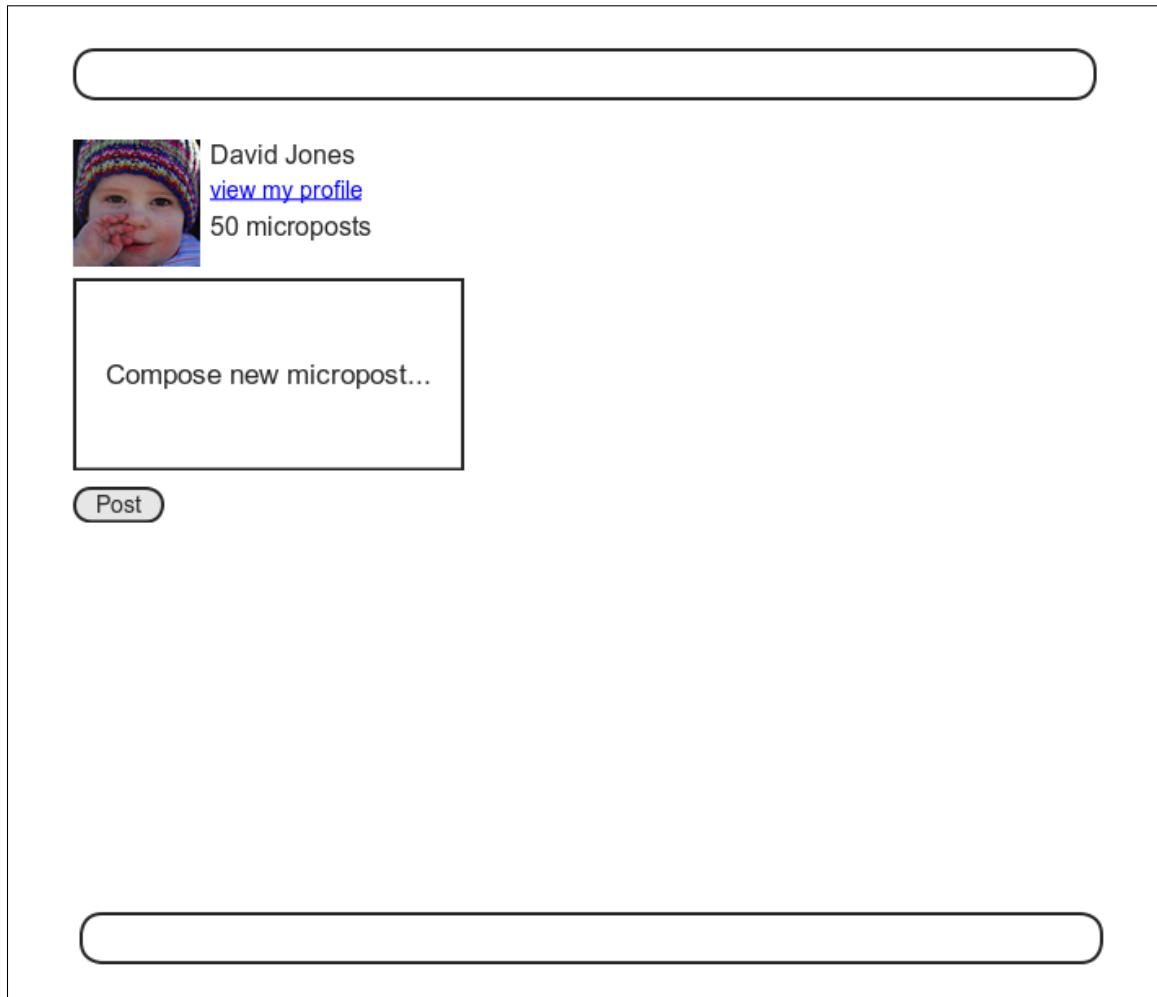


Figura 11.10: Un bosquejo de la página Home con un formulario para crear micromensajes.

en utilizar la asociación user/micropost para construir el nuevo micromensaje, como se observa en el Listado 11.34. Observe el uso de parámetros fuertes mediante **micropost_params**, lo cual permite que el atributo **content** del micromensaje sea modificado únicamente a través de la web.

Listado 11.34: La acción **create** del controlador Microposts.

app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]

  def create
    @micropost = current_user.microposts.build(micropost_params)
    if @micropost.save
      flash[:success] = "Micropost created!"
      redirect_to root_url
    else
      render 'static_pages/home'
    end
  end

  def destroy
  end

  private

  def micropost_params
    params.require(:micropost).permit(:content)
  end
end
```

Para construir un formulario que cree micromensajes, usamos el código del Listado 11.35, que entrega diferentes HTML dependiendo de si el visitante del sitio está o no en sesión.

Listado 11.35: Agregando la creación de micromensajes a la página Home ([/](#)).

app/views/static_pages/home.html.erb

```
<% if logged_in? %>
<div class="row">
  <aside class="col-md-4">
    <section class="user_info">
```

```

<%= render 'shared/user_info' %>
</section>
<section class="micropost_form">
  <%= render 'shared/micropost_form' %>
</section>
</aside>
</div>
<% else %>
<div class="center jumbotron">
  <h1>Welcome to the Sample App</h1>

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

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

<%= link_to image_tag("rails.png", alt: "Rails logo"),
           'http://rubyonrails.org/' %>
<% end %>

```

(Tener tanto código en cada bloque **if-else** es un poco desaliñado, por lo que limpiarlo utilizando parciales se deja como ejercicio ([Sección 11.6](#)).

Para hacer que funcione la página definida en el [Listado 11.35](#), necesitamos crear y dar contenido a un par de parciales. El primero es la nueva barra lateral de la página Home, que vemos en el [Listado 11.36](#).

Listado 11.36: El parcial para la barra lateral de información de usuario.

app/views/shared/_user_info.html.erb

```

<%= link_to gravatar_for(current_user, size: 50), current_user %>
<h1><%= current_user.name %></h1>
<span><%= link_to "view my profile", current_user %></span>
<span><%= pluralize(current_user.microposts.count, "micropost") %></span>

```

Observe que, como en la barra lateral del perfil ([Listado 11.23](#)), la información del usuario en el [Listado 11.36](#) muestra el número total de micromensajes del usuario. Aunque existe una ligera diferencia en el despliegue; en la barra lateral del perfil, “Microposts” es una etiqueta, y mostrar “Microposts (1)” tiene

sentido. Aunque en este caso, decir “1 micromensajes” es gramaticalmente incorrecto, por lo que nos encargaremos de mostrar “1 micromensaje” y “2 micromensajes” usando el método **pluralize** que vimos en la Sección 7.3.3.

A continuación definimos el formulario para crear micromensajes (Listado 11.37), el cual es similar al formulario de registro que vimos en el Listado 7.13.

Listado 11.37: El parcial del formulario para crear micromensajes.

app/views/shared/_micropost_form.html.erb

```
<%= form_for(@micropost) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
<% end %>
```

Necesitamos hacer un par de cambios antes de que el formulario del Listado 11.37 funcione. Primero, necesitamos definir la variable **@micropost**, la cual (igual que antes) definimos por asociación:

```
@micropost = current_user.microposts.build
```

El resultado se muestra en el Listado 11.38.

Listado 11.38: Agregando una variable de instancia **micropost** a la acción **home**.

app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
    @micropost = current_user.microposts.build if logged_in?
  end

  def help
  end

  def about
  end
end
```

```
end

def contact
end
end
```

Por supuesto, `current_user` existe únicamente si el usuario está en sesión, por lo que la variable `@micropost` debería estar definida sólo en este caso.

El segundo cambio que necesitamos hacer para que el [Listado 11.37](#) funcione, es redefinir el parcial de mensajes de error de forma que el siguiente código del [Listado 11.37](#) funcione:

```
<%= render 'shared/error_messages', object: f.object %>
```

Puede ser que usted recuerde del [Listado 7.18](#) que el parcial de mensajes de error hace referencia a la variable `@user` explícitamente, pero en este caso tenemos una variable `@micropost`. Para unificar estos casos, podemos pasar la variable del formulario `f` al parcial y accesar el objeto asociado mediante `f.object`, de forma que

```
form_for(@user) do |f|
```

`f.object` es `@user`, y en

```
form_for(@micropost) do |f|
```

`f.object` es `@micropost`, etc.

Para pasar el objeto al parcial, usamos un arreglo hash con valor igual al objeto y llave igual al nombre deseado de la variable en el parcial, que es lo que hace la segunda línea del [Listado 11.37](#). En otras palabras, `object: f.object` crea una variable llamada `object` en el parcial `error_messages`, y podemos usarla para construir un mensaje de error personalizado, como se observa en el [Listado 11.39](#).

Listado 11.39: Mensajes de error que funcionan con otros objetos. ROJO

app/views/shared/_error_messages.html.erb

```
<% if object.errors.any? %>
  <div id="error_explanation">
    <div class="alert alert-danger">
      The form contains <%= pluralize(object.errors.count, "error") %>.
    </div>
    <ul>
      <% object.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

En este momento, debería verificar que el conjunto de pruebas está en ROJO:

Listado 11.40: ROJO

\$ bundle exec rake test

Esta es una sugerencia de que necesitamos actualizar las otras ocurrencias del parcial de mensajes de error, el cual utilizamos cuando registramos usuarios ([Listado 7.18](#)), cuando reiniciamos contraseñas ([Listado 10.50](#)), y cuando editamos usuarios ([Listado 9.2](#)). Las versiones actualizadas se muestran en los [Listados 11.41, 11.43 y 11.42](#).

Listado 11.41: Actualizando el despliegue de errores de registro de usuario.

app/views/users/new.html.erb

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

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>
      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>
```

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

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

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

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

Listado 11.42: Actualizando los errores en la edición de usuarios.

app/views/users/edit.html.erb

```
<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>

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

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

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

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

      <%= f.submit "Save changes", class: "btn btn-primary" %>
    <% end %>

    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="http://gravatar.com/emails">change</a>
    </div>
  </div>
</div>
```

Listado 11.43: Actualizando los errores en el reinicio de contraseña.

app/views/password_resets/edit.html.erb

```
<% provide(:title, 'Reset password') %>
<h1>Password reset</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_for(@user, url: password_reset_path(params[:id])) do |f| %>
      <%= render 'shared/error_messages', object: f.object %>

      <%= hidden_field_tag :email, @user.email %>

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

```
<%= f.password_field :password, class: 'form-control' %>  
  
<%= f.label :password_confirmation, "Confirmation" %>  
<%= f.password_field :password_confirmation, class: 'form-control' %>  
  
<%= f.submit "Update password", class: "btn btn-primary" %>  
  <% end %>  
</div>  
</div>
```

En este momento, todas las pruebas deberían estar en **VERDE**:

```
$ bundle exec rake test
```

Adicionalmente, todo el HTML de esta sección debería desplegarse apropiadamente, mostrando el formulario como en la Figura 11.11, y un formulario con un error al enviar datos como en la Figura 11.12.

11.3.3 Un avance prototipo

Aunque el formulario de micromensajes está funcionando en este momento, los usuarios no pueden ver inmediatamente el resultado de un envío de datos exitoso porque la página principal en este momento no muestra ningún micromensaje. Si usted así lo desea, puede verificar que el formulario que se observa en la Figura 11.11 está funcionando enviando datos válidos y luego navegando a la [página del perfil](#) para ver el mensaje, pero esto es un poco engorroso. Sería mucho mejor tener un *avance* de micromensajes que incluya los mensajes del propio usuario, como se bosqueja en la Figura 11.13. (En el Capítulo 12, generalizaremos este avance incluyendo los micromensajes de usuarios que son *seguidos* por el usuario actual.)

Puesto que cada usuario debería tener un avance, naturalmente consideramos un método **feed** en el modelo **User**, el cual inicialmente seleccionará los micromensajes que pertenecen al usuario actual. Lograremos esto usando el método **where** en el modelo **Micropost** (que vimos brevemente en la Sección 10.5), como se observa en el Listado 11.44.¹⁰

¹⁰Revise la Guía de Rails en [Active Record Query Interface](#) para mayor detalle sobre el método **where** y otros

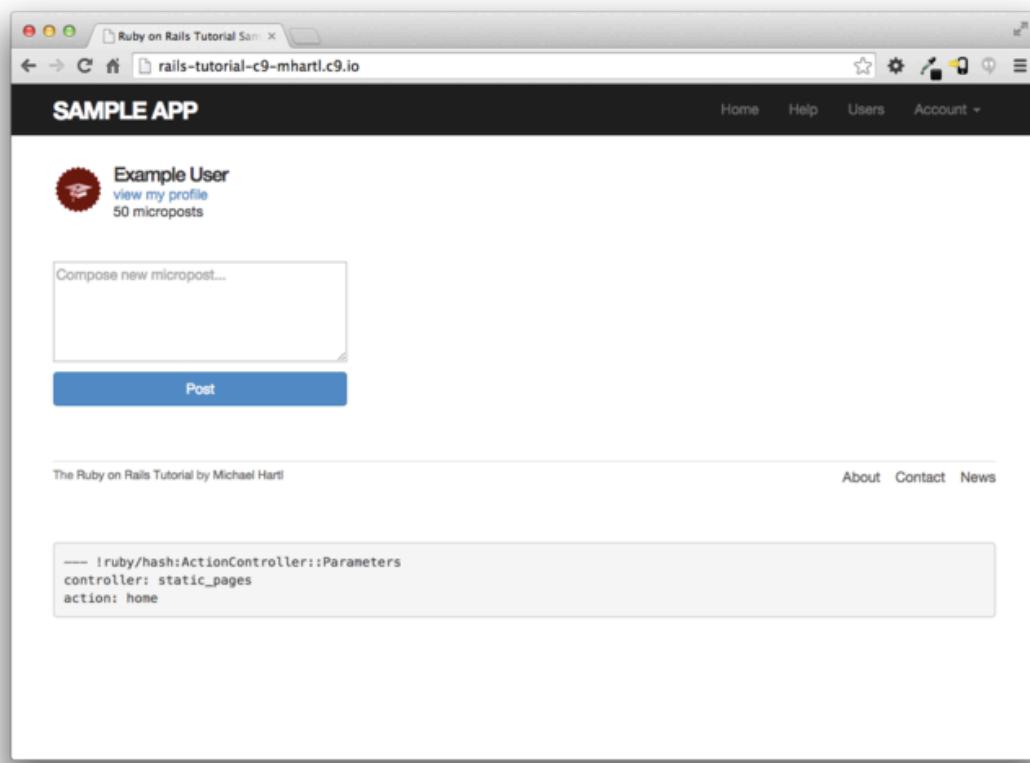


Figura 11.11: La página principal con un formulario para crear un micromensaje.

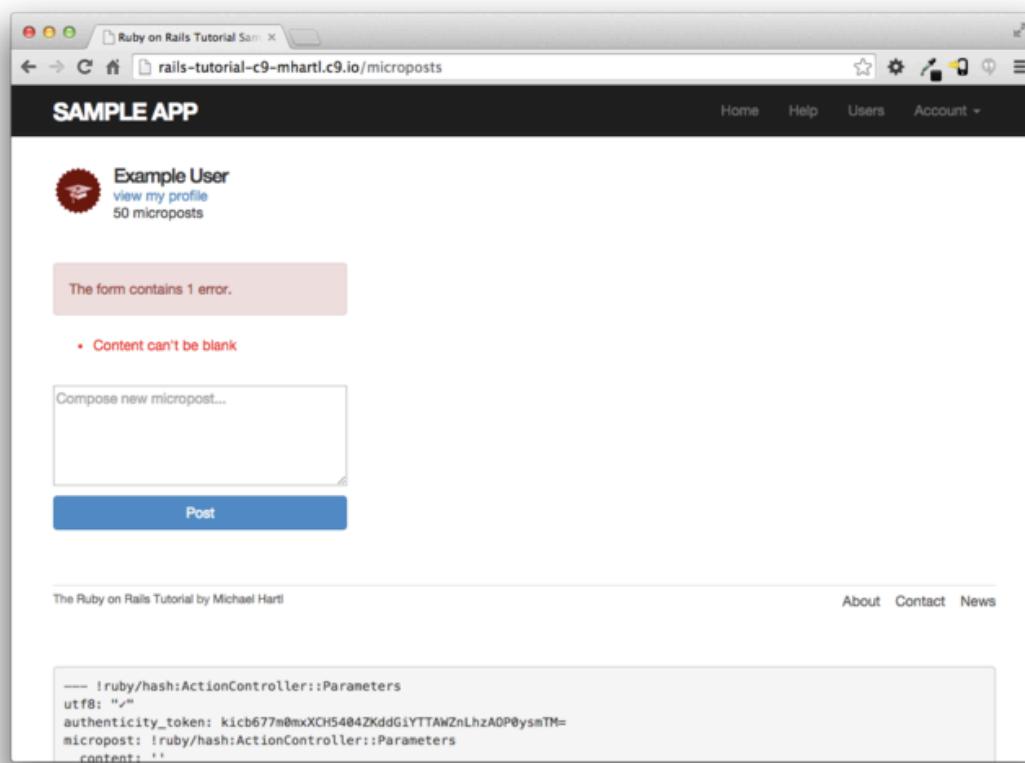


Figura 11.12: La página principal con un error en el formulario.

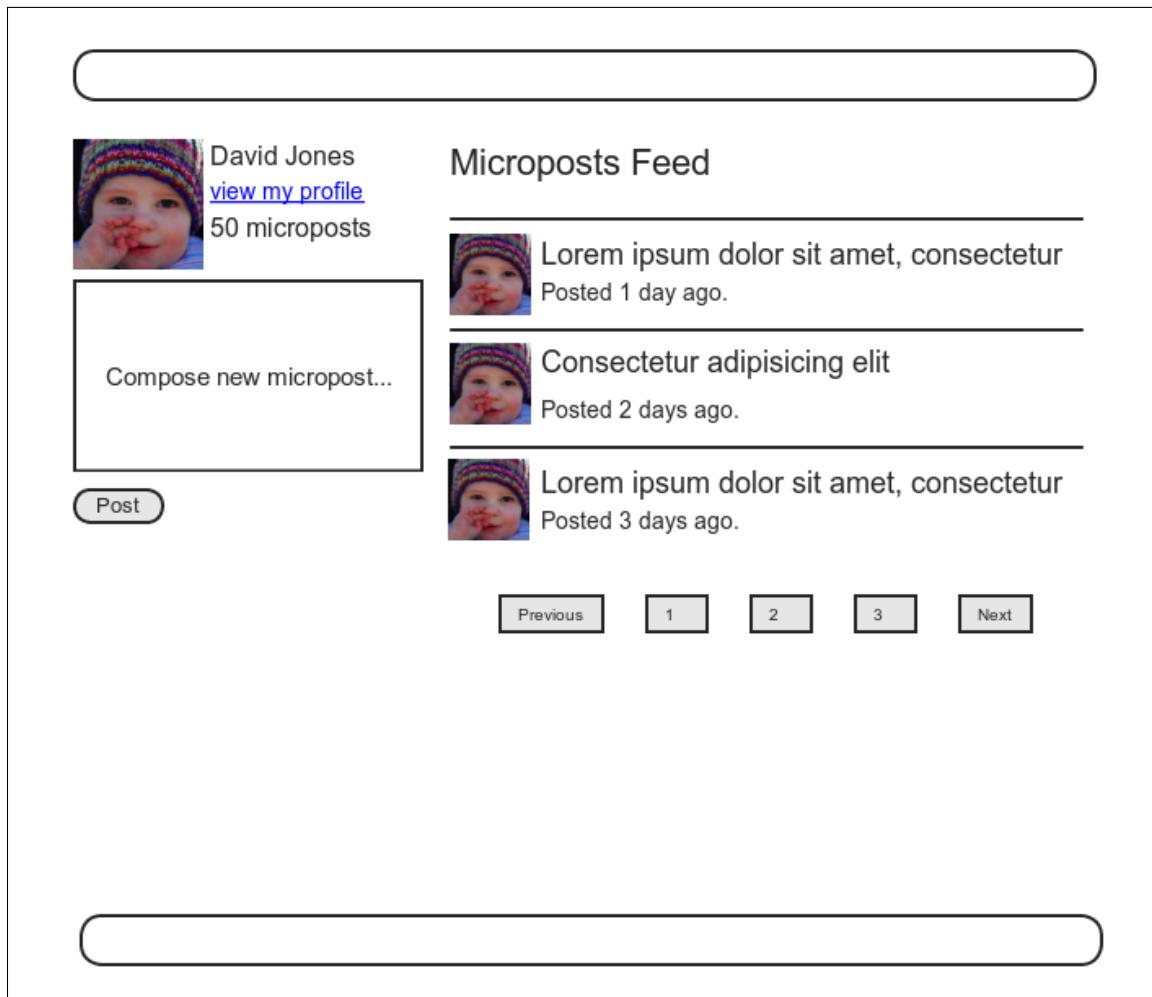


Figura 11.13: Un bosquejo de la página principal con un avance prototípico.

Listado 11.44: Una implementación preliminar para el status de avance de los micromensajes.

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Defines a proto-feed.
  # See "Following users" for the full implementation.
  def feed
    Micropost.where("user_id = ?", id)
  end

  private
  .
  .
  .
end
```

El signo de interrogación en

```
Micropost.where("user_id = ?", id)
```

nos asegura que **id** es *escapado* apropiadamente antes de ser incluído en la consulta SQL subyacente, evitando de esta forma un serio hueco de seguridad llamado *inyección SQL*. El atributo **id** aquí es sólo un número entero (es decir, **self.id**, el ID único del usuario), por lo que no hay peligro de inyectar SQL en este caso, pero escapar *siempre* las variables que se inyectan en sentencias SQL es un buen hábito que se debe cultivar.

Los lectores atentos pueden observar en este punto que el código del [Listado 11.44](#) es esencialmente equivalente a escribir

```
def feed
  microposts
end
```

métodos relacionados.

Hemos usado el código del [Listado 11.44](#) en su lugar porque generaliza de forma mucho más natural el status de avance necesario en el [Capítulo 12](#).

Para usar el avance en la aplicación de ejemplo, agregamos una variable de instancia `@feed_items` para el avance (paginado) del usuario actual, como en el [Listado 11.45](#), y luego agregamos un parcial para el status de avance ([Listado 11.46](#)) a la página Home ([Listado 11.47](#)). Observe que, ahora hay dos líneas que necesitan ejecutarse cuando el usuario está en sesión: el [Listado 11.45](#) cambia

```
@micropost = current_user.microposts.build if logged_in?
```

del [Listado 11.38](#) por

```
if logged_in?
  @micropost = current_user.microposts.build
  @feed_items = current_user.feed.paginate(page: params[:page])
end
```

de este modo, moviendo la condición del final de la línea a un bloque if-end.

Listado 11.45: Agregando una variable de instancia para el avance de la acción `home`.

```
app/controllers/static_pages_controller.rb

class StaticPagesController < ApplicationController

  def home
    if logged_in?
      @micropost = current_user.microposts.build
      @feed_items = current_user.feed.paginate(page: params[:page])
    end
  end

  def help
  end

  def about
  end

  def contact
  end
end
```

Listado 11.46: El parcial de status de avance.

app/views/shared/_feed.html.erb

```
<% if @feed_items.any? %>
  <ol class="microposts">
    <%= render @feed_items %>
  </ol>
  <%= will_paginate @feed_items %>
<% end %>
```

El parcial de status de avance delega el despliegue al parcial de micromensajes definido en el [Listado 11.21](#):

```
<%= render @feed_items %>
```

Aquí Rails sabe cómo invocar al parcial de micromensajes porque cada elemento de `@feed_items` tiene clase **Micropost**. Esto hace que Rails busque un parcial con el nombre correspondiente en el directorio de las vistas de un recurso dado:

app/views/microposts/_micropost.html.erb

Podemos agregar el avance a la página Home mostrando el parcial respectivo como es usual ([Listado 11.47](#)). El resultado es un despliegue del avance en la página Home, como es requerido ([Figura 11.14](#)).

Listado 11.47: Agregando un status de avance a la página Home.

app/views/static_pages/home.html.erb

```
<% if logged_in? %>
  <div class="row">
    <aside class="col-md-4">
      <section class="user_info">
        <%= render 'shared/user_info' %>
      </section>
      <section class="micropost_form">
        <%= render 'shared/micropost_form' %>
      </section>
    </aside>
    <div class="col-md-8">
      <h1>Welcome, <%= current_user.name %>!</h1>
      <h2>Share your thoughts with the world!</h2>
      <ol class="microposts">
        <li><%= render @feed_items %></li>
      </ol>
      <%= will_paginate @feed_items %>
    </div>
  </div>
<% end %>
```

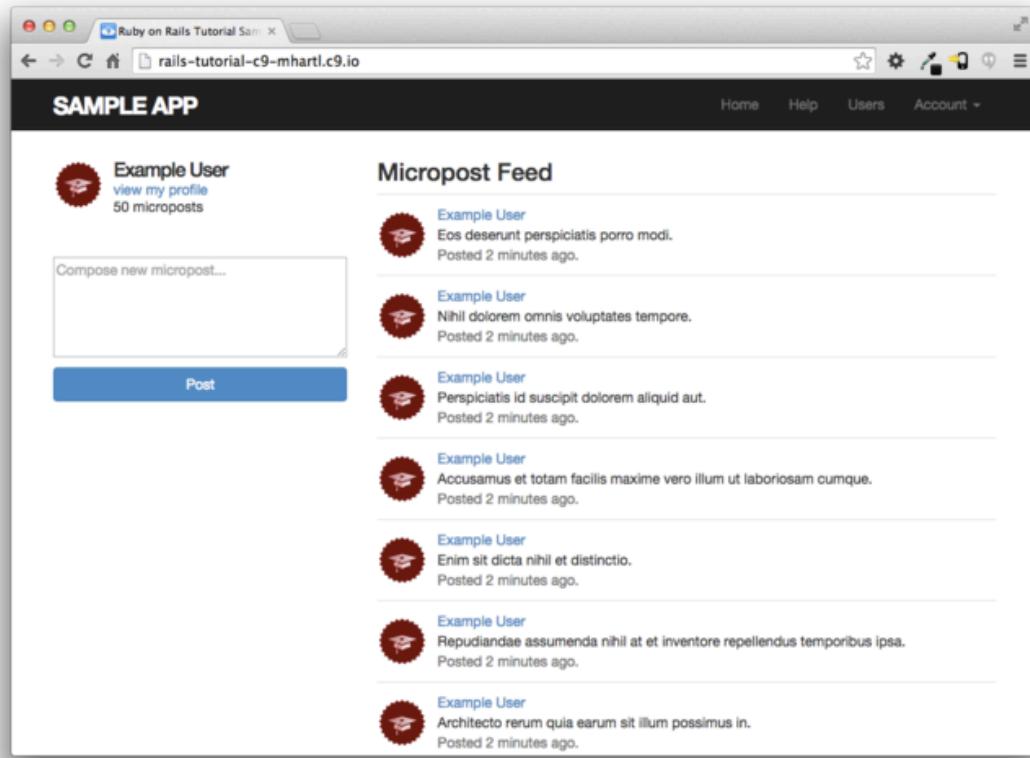


Figura 11.14: La página Home con un avance prototípico.

```
</section>
</aside>


<h3>Micropost Feed</h3>
  <%= render 'shared/feed' %>


```

En este momento, la creación de un nuevo micromensaje funciona como es esperado, observe la Figura 11.15. Aunque hay una sutileza: si el envío

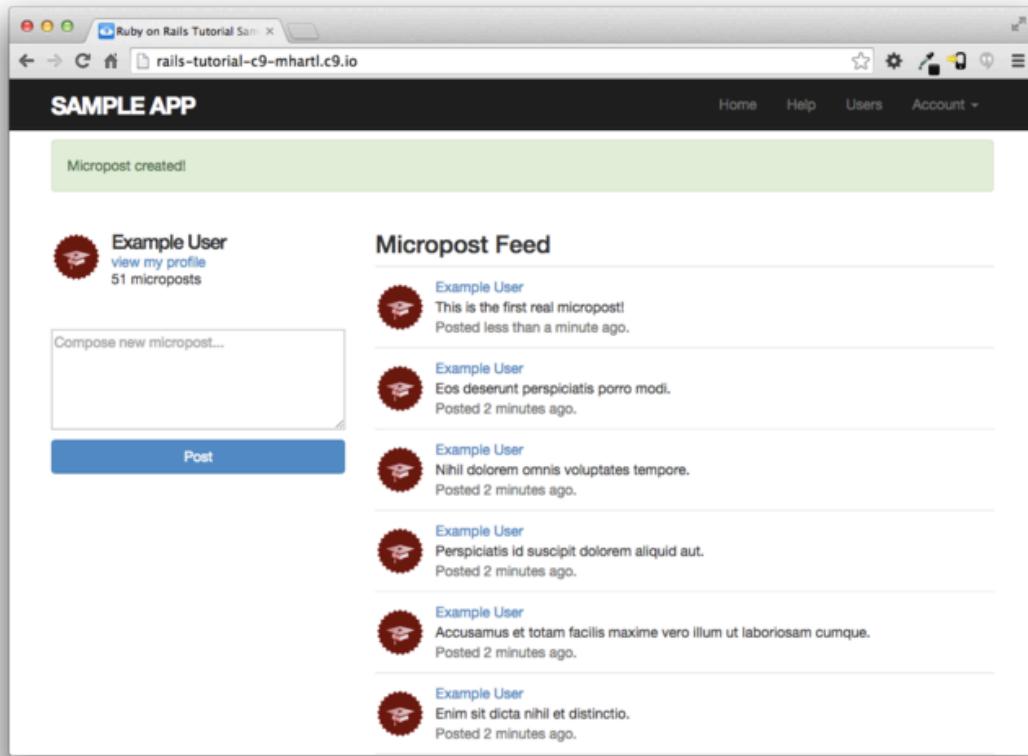


Figura 11.15: La página Home luego de haber creado un nuevo micromensaje.

de datos del micromensaje *falla*, la página Home espera una variable de instancia `@feed_items`, por lo que los envíos fallidos en este momento truenan. La solución más sencilla es suprimir el avance por completo asignando un arreglo vacío, como vemos en el [Listado 11.48](#). (Desafortunadamente, regresar un avance paginado no funciona en este caso. Impleméntelo y visite un enlace de paginación para ver porqué.)

Listado 11.48: Agregando una variable de instancia `@feed_items` (vacía) a la acción `create`.

```
app/controllers/microposts_controller.rb  
class MicropostsController < ApplicationController
```

```
before_action :logged_in_user, only: [:create, :destroy]

def create
  @micropost = current_user.microposts.build(micropost_params)
  if @micropost.save
    flash[:success] = "Micropost created!"
    redirect_to root_url
  else
    @feed_items = []
    render 'static_pages/home'
  end
end

def destroy
end

private

  def micropost_params
    params.require(:micropost).permit(:content)
  end
end
```

11.3.4 Destruyendo micromensajes

La última funcionalidad que resta por agregar al recurso Microposts es la de destruir micromensajes. De igual forma que con el borrado de usuarios ([Sección 9.4.2](#)), realizaremos esto con enlaces de borrado, como se bosqueja en la Figura 11.16. A diferencia de ese caso, que restringíamos la destrucción de usuarios a usuarios administradores, los enlaces de borrado funcionarán sólo para micromensajes que el usuario actual haya creado.

Nuestro primer paso es agregar un enlace de borrado al parcial de micromensajes como se muestra en el [Listado 11.21](#). El resultado se muestra en el [Listado 11.49](#).

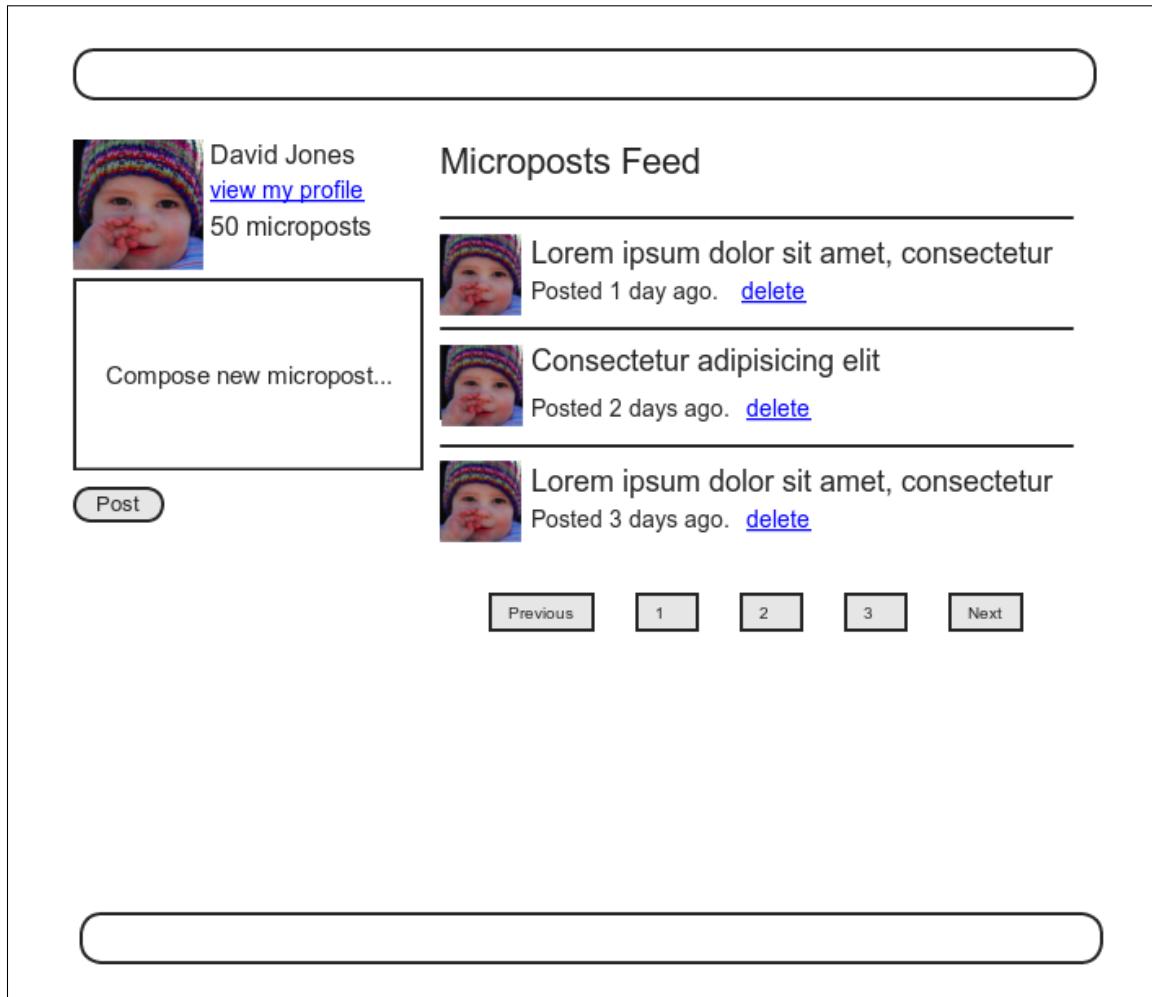


Figura 11.16: Un bosquejo del avance prototípico con enlaces de borrado de micromensajes.

Listado 11.49: Agregando un enlace de borrado al parcial de micromensajes.

app/views/microposts/_micropost.html.erb

```
<li id="<%= micropost.id %>">
  <%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
  <span class="user"><%= link_to micropost.user.name, micropost.user %></span>
  <span class="content"><%= micropost.content %></span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
    <% if current_user?(micropost.user) %>
      <%= link_to "delete", micropost, method: :delete,
                  data: { confirm: "You sure?" } %>
    <% end %>
  </span>
</li>
```

El siguiente paso es definir una acción **destroy** en el controlador Microposts, que es análoga a la del usuario del [Listado 9.54](#). La principal diferencia es que, en vez de utilizar una variable **@user** con un filtro previo **admin_user**, buscaremos los micromensajes por asociación, lo cual fallará de forma automática si un usuario intenta borrar un micromensaje de otro usuario. Pondremos el resultado de la búsqueda dentro de un filtro previo **correct_user**, el cual verifica que el usuario actual realmente tiene un micromensaje con el id dado. El resultado aparece en el [Listado 11.50](#).

Listado 11.50: La acción **destroy** del controlador Microposts.

app/controllers/microposts_controller.rb

```
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]
  before_action :correct_user,   only: [:destroy]

  def destroy
    @micropost.destroy
    flash[:success] = "Micropost deleted"
    redirect_to request.referrer || root_url
  end

  private

    def micropost_params
```

```

params.require(:micropost).permit(:content)
end

def correct_user
  @micropost = current_user.microposts.find_by(id: params[:id])
  redirect_to root_url if @micropost.nil?
end
end

```

Observe que el método **destroy** del Listado 11.50 redirecciona a la URL

```
request.referrer || root_url
```

Esto utiliza el método **request.referrer**,¹¹ que está estrechamente relacionado con la variable **request.url** utilizada en el redireccionamiento amigable (Sección 9.2.3), y es justo la URL anterior (en este caso, la página Home).¹² Esto es conveniente porque los micromensajes aparecen tanto en la página principal como en la del perfil del usuario, por lo que con el uso de **request.referrer** nos encargamos de redireccionar de regreso a la página que solicitó el borrado en ambos casos. Si la página de referencia es **nil** (como sucede dentro de algunas pruebas), el Listado 11.50 le asigna por default el valor de la URL raíz usando el operador **||**. (Compare con las opciones por default definidas en el Listado 8.50.)

Con el código anterior, el resultado de destruir el segundo mensaje más reciente se muestra en la Figura 11.17.

11.3.5 Pruebas de los micromensajes

Con el código de la Sección 11.3.4, el modelo **Micropost** y la interfaz están completos. Todo lo que resta es escribir una pequeña prueba para el controlador Microposts que verifique la autorización y una prueba de integración de micromensajes que reúna todo.

¹¹Este corresponde a **HTTP_REFERER**, como está definido en la especificación de HTTP. Observe que “referrer” no es un error tipográfico—la palabra está realmente mal escrita en la especificación. Rails corrige el error escribiendo “referrer”.

¹²De repente no recordaba cómo obtener esta URL dentro de una aplicación Rails, por lo que busqué en Google “rails request previous url” y encontré una entrada en Stack Overflow con la respuesta.

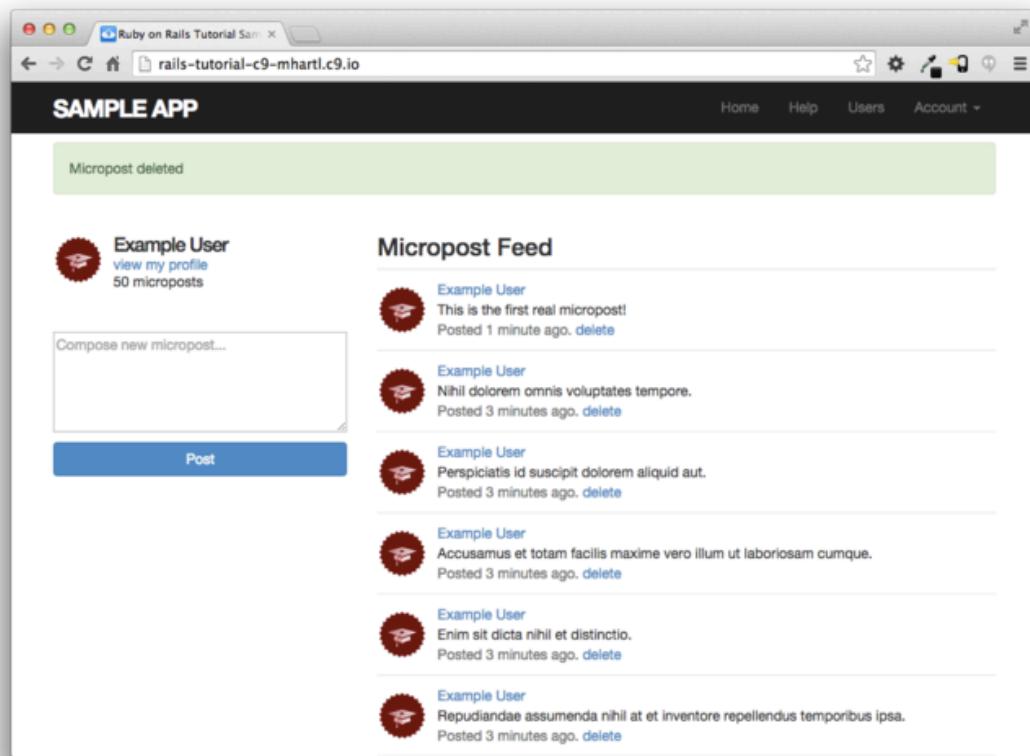


Figura 11.17: La página Home luego de borrar el segundo micromensaje más reciente.

Empezaremos por agregar unos cuantos micromensajes que pertenezcan a usuarios diferentes en los archivos fixtures respectivos, como se muestra en el [Listado 11.51](#). (Estaremos utilizando sólo uno por ahora, pero pondremos los otros para futuras referencias.)

Listado 11.51: Agregando un micromensaje que pertenece a un usuario diferente.

```
test/fixtures/microposts.yml

.
.
.

ants:
  content: "Oh, is that what you want? Because that's how you get ants!"
  created_at: <%= 2.years.ago %>
  user: archer

zone:
  content: "Danger zone!"
  created_at: <%= 3.days.ago %>
  user: archer

tone:
  content: "I'm sorry. Your words made sense, but your sarcastic tone did not."
  created_at: <%= 10.minutes.ago %>
  user: lana

van:
  content: "Dude, this van's, like, rolling probable cause."
  created_at: <%= 4.hours.ago %>
  user: lana
```

A continuación escribiremos una pequeña prueba para asegurarnos de que un usuario no puede borrar micromensajes de otro usuario, y verificaremos también el redireccionamiento correcto, como se muestra en el [Listado 11.52](#).

Listado 11.52: Probando el borrado de micromensajes con un usuario que no corresponde. **VERDE**

```
test/controllers/microposts_controller_test.rb

require 'test_helper'

class MicropostsControllerTest < ActionController::TestCase
```

```

def setup
  @micropost = microposts(:orange)
end

test "should redirect create when not logged in" do
  assert_no_difference 'Micropost.count' do
    post :create, micropost: { content: "Lorem ipsum" }
  end
  assert_redirected_to login_url
end

test "should redirect destroy when not logged in" do
  assert_no_difference 'Micropost.count' do
    delete :destroy, id: @micropost
  end
  assert_redirected_to login_url
end

test "should redirect destroy for wrong micropost" do
  log_in_as(users(:michael))
  micropost = microposts(:ants)
  assert_no_difference 'Micropost.count' do
    delete :destroy, id: micropost
  end
  assert_redirected_to root_url
end
end

```

Finalmente, escribiremos una prueba de integración para iniciar sesión, verificar la paginación de los micromensajes, enviar datos tanto válidos como inválidos, borrar un mensaje y luego visitar la página de otro usuario para asegurarnos que no hay enlaces de borrado. Empezaremos generando la prueba como es usual:

```

$ rails generate integration_test microposts_interface
  invoke test_unit
  create test/integration/microposts_interface_test.rb

```

La prueba se muestra en el [Listado 11.53](#). Vea si puede relacionar las líneas del [Listado 11.11](#) con los pasos mencionados anteriormente. (El [Listado 11.53](#) utiliza `post` seguido de `follow_redirect!` en vez de su equivalente `post_via_redirect` anticipándose a la prueba para subir imágenes que viene en los ejercicios ([Listado 11.69](#)).)

Listado 11.53: Una prueba de integración para la interfaz de micromensajes.

VERDE

```
test/integration/microposts_interface_test.rb

require 'test_helper'

class MicropostsInterfaceTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "micropost interface" do
    log_in_as(@user)
    get root_path
    assert_select 'div.pagination'
    # Invalid submission
    assert_no_difference 'Micropost.count' do
      post microposts_path, micropost: { content: "" }
    end
    assert_select 'div#error_explanation'
    # Valid submission
    content = "This micropost really ties the room together"
    assert_difference 'Micropost.count', 1 do
      post microposts_path, micropost: { content: content }
    end
    assert_redirected_to root_url
    follow_redirect!
    assert_match content, response.body
    # Delete a post.
    assert_select 'a', text: 'delete'
    first_micropost = @user.microposts.paginate(page: 1).first
    assert_difference 'Micropost.count', -1 do
      delete micropost_path(first_micropost)
    end
    # Visit a different user.
    get user_path(users(:archer))
    assert_select 'a', text: 'delete', count: 0
  end
end
```

Como escribimos el código funcional de la aplicación primero, el conjunto de pruebas debería estar en VERDE:

Listado 11.54: VERDE

```
$ bundle exec rake test
```

11.4 Micromensajes de imágenes

Ahora que hemos agregado soporte para todas las acciones de micromensajes relevantes, en esta sección habilitaremos la posibilidad de que los micromensajes incluyan tanto imágenes como texto. Empezaremos con una versión básica suficientemente buena para utilizarla en desarrollo, y luego agregaremos una serie de mejoras para que quede lista para subir imágenes en producción.

Agregar imágenes involucra principalmente dos elementos visibles: un campo en el formulario para subir la imagen y las imágenes mismas en los micromensajes. Un bosquejo del botón “Upload image” y una foto en el micromensaje se muestran en la Figura 11.18.¹³

11.4.1 Carga de imágenes básica

Para manejar la carga de imágenes y asociarla con el modelo **Micropost**, utilizaremos el cargador de imágenes **CarrierWave**. Para empezar, necesitamos incluir la gema **carrierwave** en el archivo **Gemfile** (Listado 11.55). Por completez, el Listado 11.55 también incluye las gemas **mini_magick** y **fog** necesarias para modificar el tamaño de las imágenes (Sección 11.4.3) y para cargar las imágenes en producción (Sección 11.4.4).

Listado 11.55: Agregando CarrierWave al archivo **Gemfile**.

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

gem 'rails',                  '4.2.2'
gem 'bcrypt',                  '3.1.7'
gem 'faker',                   '1.4.2'
gem 'carrierwave',             '0.10.0'
```

¹³Foto de playa de <https://www.flickr.com/photos/grungepunk/14026922186>

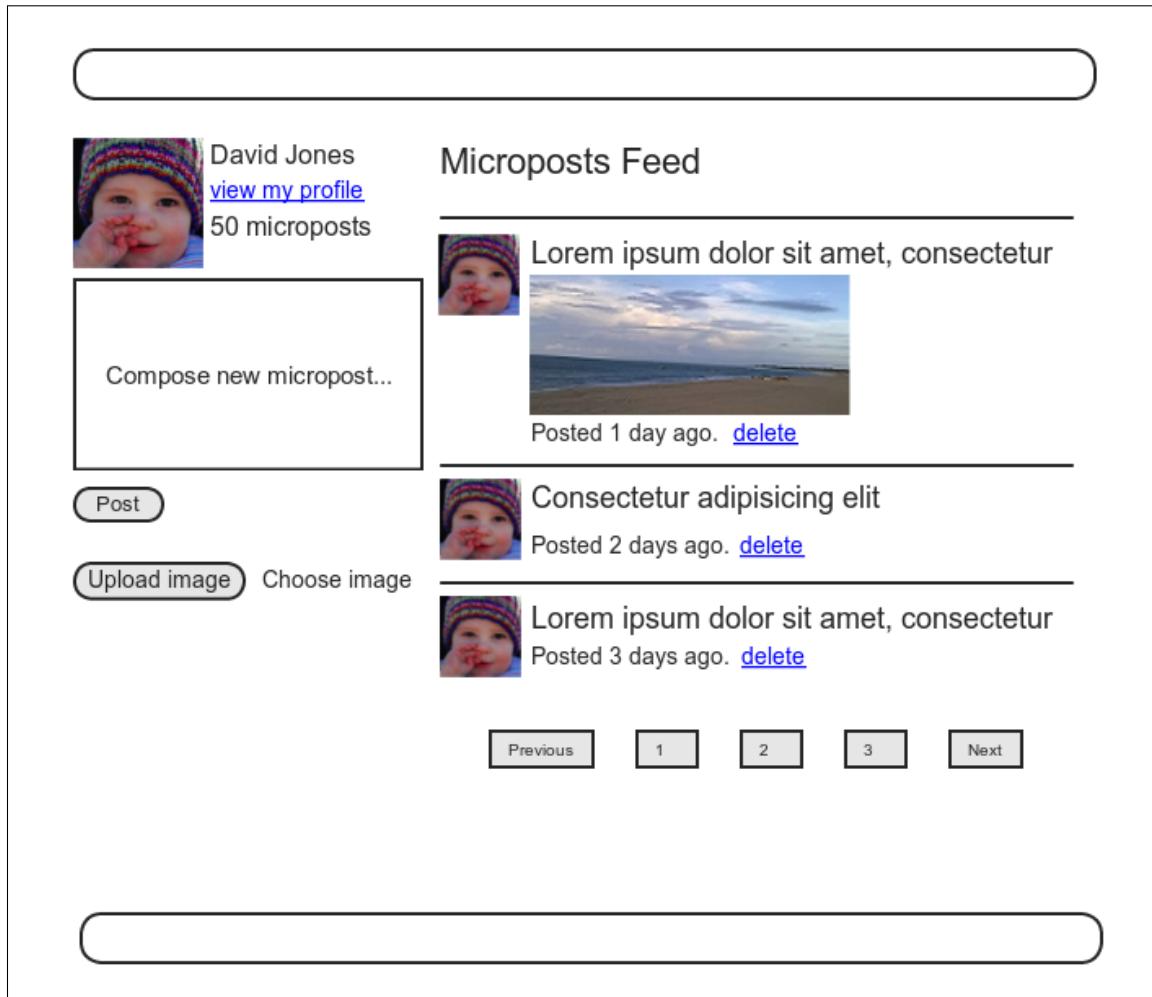


Figura 11.18: Un bosquejo de la carga de imagen para un micromensaje (con una imagen cargada).

```
gem 'mini_magick',           '3.8.0'
gem 'fog',                   '1.26.0'
gem 'will_paginate',         '3.0.7'
gem 'bootstrap-will_paginate', '0.0.10'
.
.
.
```

Luego las instalamos como es usual:

```
$ bundle install
```

Carrierwave agrega un generador Rails para crear un cargador de imágenes, el cual utilizaremos para definir uno que se llame **picture**:¹⁴

```
$ rails generate uploader Picture
```

Las imágenes cargadas con **carrierwave** deberían estar asociadas con un atributo correspondiente en un modelo Active Record, el cual simplemente contiene el nombre del archivo de la imagen en un campo de tipo cadena. El modelo de datos de micromensajes resultante se muestra en la Figura 11.19.

Para agregar el atributo requerido **picture** al modelo **Micropost**, generamos una migración y migramos la base de datos de desarrollo:

```
$ rails generate migration add_picture_to_microposts picture:string
$ bundle exec rake db:migrate
```

La forma de decirle a **carrierwave** que asocie la imagen con el modelo es utilizando el método **mount_uploader**, que toma como argumentos un símbolo que representa el atributo y el nombre de la clase del cargador generado:

¹⁴Inicialmente, nombré el nuevo atributo **image**, pero ese nombre es tan genérico que terminé confundido.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime
picture	string

Figura 11.19: El modelo de datos **Micropost** con un atributo **picture**.

```
mount_uploader :picture, PictureUploader
```

(Aquí **PictureUploader** está definido en el archivo **picture_uploader.rb**, el cual empezaremos a editar en la Sección 11.4.2, pero por ahora, el generado por default está bien.) Agregar el cargador al modelo **Micropost** nos produce el código que se muestra en el Listado 11.56.

Listado 11.56: Agregando una imagen al modelo **Micropost**.

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order(created_at: :desc) }
  mount_uploader :picture, PictureUploader
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

En algunos sistemas, puede que sea necesario que reinice el servidor de Rails en este momento para mantener el conjunto de pruebas en **VERDE**. (Si usted está usando Guard como describimos en la Sección 3.7.3, puede que necesite

reiniciarlo también, incluso podría ser necesario cerrar la sesión de la terminal y correr `Guard` en una nueva.)

Para incluir el cargador en la página Home como en la Figura 11.18, necesitamos incluir una etiqueta `file_field` en el formulario de los micromensajes, como se muestra en el Listado 11.57.

Listado 11.57: Agregando la carga de imágenes al formulario que crea micromensajes.

`app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost, html: { multipart: true }) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
  <span class="picture">
    <%= f.file_field :picture %>
  </span>
<% end %>
```

Observe la inclusión de

```
html: { multipart: true }
```

en los argumentos de `form_for`, lo cual es necesario para subir los archivos.

Finalmente, necesitamos agregar `picture` a la lista de atributos que se permite sean modificados a través de la web. Esto implica editar el método `micropost_params`, como se muestra en el Listado 11.58.

Listado 11.58: Agregando `picture` a la lista de atributos permitidos.

`app/controllers/microposts_controller.rb`

```
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]
  before_action :correct_user,   only: :destroy
  .
  .
```

```

    .
  private

    def micropost_params
      params.require(:micropost).permit(:content, :picture)
    end

    def correct_user
      @micropost = current_user.microposts.find_by(id: params[:id])
      redirect_to root_url if @micropost.nil?
    end
  end

```

Una vez que la imagen ha sido enviada al servidor, podemos mostrarla usando el auxiliar `image_tag` en el parcial de micromensajes, como se muestra en el [Listado 11.59](#). Observe el uso del método booleano `picture?` para evitar desplegar una etiqueta imagen cuando no hay una imagen qué mostrar. Este método es creado automáticamente por `carrierwave` con base en el nombre del atributo de la imagen. El resultado de realizar un envío de datos manual exitoso se muestra en la [Figura 11.20](#). Escribir una prueba automatizada para el envío de imágenes al servidor se deja como ejercicio ([Sección 11.6](#)).

Listado 11.59: Agregando el despliegue de la imagen a los micromensajes.

`app/views/microposts/_micropost.html.erb`

```

<li id="micropost-<%= micropost.id %>">
  <%= link_to gravatar_for(micropost.user, size: 50), micropost.user %>
  <span class="user"><%= link_to micropost.user.name, micropost.user %></span>
  <span class="content">
    <%= micropost.content %>
    <%= image_tag micropost.picture.url if micropost.picture? %>
  </span>
  <span class="timestamp">
    Posted <%= time_ago_in_words(micropost.created_at) %> ago.
    <% if current_user?(micropost.user) %>
      <%= link_to "delete", micropost, method: :delete,
                  data: { confirm: "You sure?" } %>
    <% end %>
  </span>
</li>

```

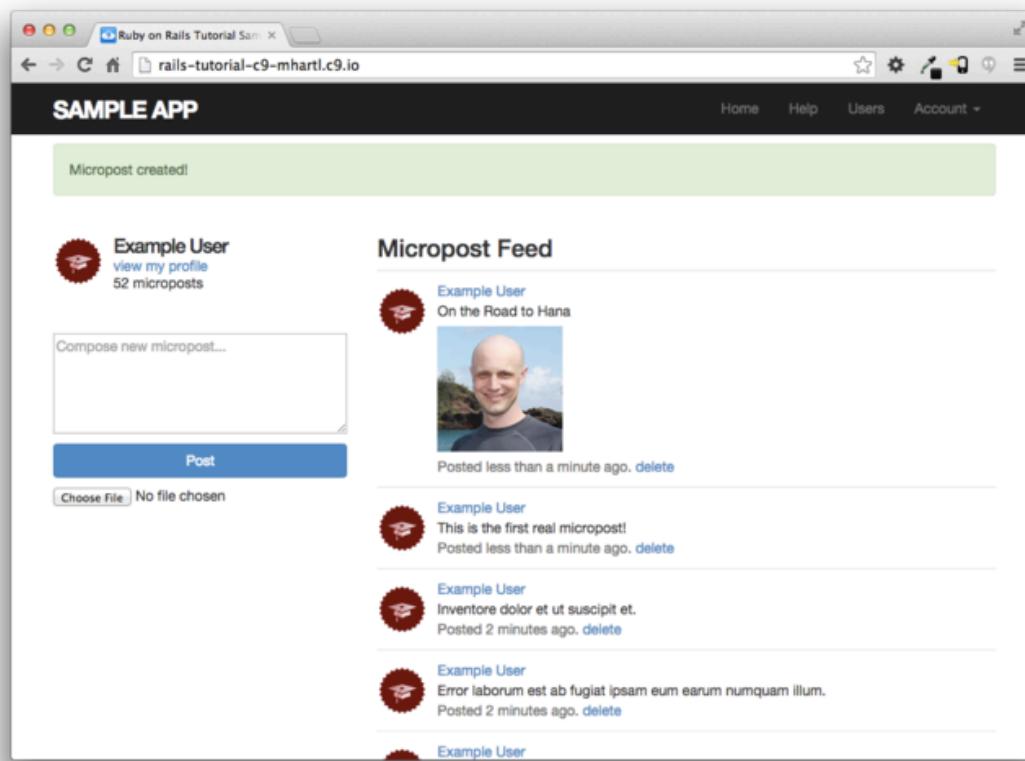


Figura 11.20: El resultado de enviar un micromensaje con una imagen.

11.4.2 Validación de imagen

El cargador de la [Sección 11.4.1](#) es un buen comienzo, pero tiene limitaciones significativas. En particular, no realiza ninguna validación sobre el archivo a subir, lo cual puede causar problemas si los usuarios intentan subir archivos grandes o de un tipo inválido. Para remediar este defecto, agregaremos validaciones para el tamaño de la imagen y el formato, ambas tanto en el servidor como en el cliente (es decir, en el navegador).

La primera validación de imagen, que restringe la carga de imágenes a tipos válidos, aparece en el cargador `carrierwave` mismo. El código resultante (que aparece como una sugerencia comentada en el cargador generado) verifica que el nombre del archivo termine con una extensión de imagen válida (PNG, GIF, y ambas variantes de JPEG), como se muestra en el [Listado 11.60](#).

Listado 11.60: La validación del formato de la imagen.

app/uploaders/picture_uploader.rb

```
class PictureUploader < CarrierWave::Uploader::Base
  storage :file

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounted:
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Add a white list of extensions which are allowed to be uploaded.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

La segunda validación, que controla el tamaño de la imagen, aparece en el modelo `Micropost`. A diferencia de las validaciones previas al modelo, la validación del tamaño del archivo no corresponde a un validador de Rails pre-construído. Como resultado, la validación de imagen requiere de una validación personalizada, a la que llamaremos `picture_size` y la definiremos como se muestra en el [Listado 11.61](#). Observe el uso de `validate` (a diferencia de `validates`) para invocar una validación personalizada.

Listado 11.61: Agregando validaciones a las imágenes.

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order(created_at: :desc) }
  mount_uploader :picture, PictureUploader
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
  validate :picture_size

  private

    # Validates the size of an uploaded picture.
    def picture_size
      if picture.size > 5.megabytes
        errors.add(:picture, "should be less than 5MB")
      end
    end
end
```

Esta validación personalizada se encarga de invocar el método correspondiente al símbolo dado (`:picture_size`). En `picture_size`, agregamos un mensaje de error personalizado a la colección `errors` (que vimos brevemente en la Sección 6.2.2), en este caso un límite de 5 megabytes (usando la sintaxis que vimos en el Recuadro 8.2).

Para continuar con las validaciones de los Listados 11.60 y 11.61, agregaremos dos validaciones del lado del cliente para la imagen a cargar. Primero reflejaremos la validación del formato usando el parámetro `accept` en la etiqueta de entrada `file_field`:

```
<%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
```

Los formatos válidos consisten de **tipos MIME** aceptados por la validación del Listado 11.60.

A continuación, incluiremos un poco de JavaScript (o más específicamente, [jQuery](#)) para emitir una alerta si el usuario intenta cargar una imagen demasiado grande (lo que evita cargas accidentales que podrían consumir mucho tiempo en el servidor, y que por tanto, aligeran su carga):

```
$('#micropost_picture').bind('change', function() {
  var size_in_megabytes = this.files[0].size/1024/1024;
  if (size_in_megabytes > 5) {
    alert('Maximum file size is 5MB. Please choose a smaller file.');
  }
});
```

Aunque jQuery no es el objetivo de este libro, usted puede darse cuenta de que el código anterior monitorea el elemento de la página que contiene el id de CSS `micropost_picture` (como indica el signo de número `#`), que es el id del formulario de micromensajes del [Listado 11.57](#). (La manera de descubrir esto es presionando Ctrl-click y utilizando el inspector de su navegador web.) Cuando el elemento con ese id CSS cambia, la función de jQuery se activa y emite la alerta si el archivo es demasiado grande.¹⁵

El resultado de agregar estas validaciones adicionales se muestra en el [Listado 11.62](#).

Listado 11.62: Verificando el tamaño de archivo con jQuery.

`app/views/shared/_micropost_form.html.erb`

```
<%= form_for(@micropost, html: { multipart: true }) do |f| %>
  <%= render 'shared/error_messages', object: f.object %>
  <div class="field">
    <%= f.text_area :content, placeholder: "Compose new micropost..." %>
  </div>
  <%= f.submit "Post", class: "btn btn-primary" %>
  <span class="picture">
    <%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
  </span>
<% end %>

<script type="text/javascript">
  $('#micropost_picture').bind('change', function() {
    var size_in_megabytes = this.files[0].size/1024/1024;
    if (size_in_megabytes > 5) {
      alert('Maximum file size is 5MB. Please choose a smaller file.');
    }
  });
</script>
```

¹⁵Para aprender a hacer cosas como éstas, usted puede hacer lo que yo hice: busque en Google por cosas como “javascript tamaño de archivo máximo” hasta que encuentre algo en Stack Overflow.

Es importante entender que código como el del Listado 11.62 no puede evitar realmente que un usuario suba un archivo demasiado grande. Aún si nuestro código evita que tales archivos sean enviados a través de la web, siempre puede editarse el JavaScript con un inspector web o emitir una petición POST directa, usando por ejemplo, `curl`. Para evitar que los usuarios suban archivos de imágenes arbitrariamente grandes, es esencial incluir la validación del lado del servidor, como se muestra en el Listado 11.61.

11.4.3 Modificando el tamaño de la imagen

Las validaciones al tamaño de la imagen de la Sección 11.4.2 son un buen comienzo, pero aún permiten subir imágenes suficientemente grandes como para descomponer la estructura de diseño de nuestro sitio, algunas veces con resultados espantosos (Figura 11.21). Entonces, aunque es conveniente permitir a los usuarios que seleccionen imágenes bastante grandes de su disco local, también es buena idea modificar el tamaño de la imagen antes de mostrarla.¹⁶

Estaremos modificando el tamaño de las imágenes usando el programa para manipulación de imágenes [ImageMagick](#), el cual necesitamos instalar en nuestro ambiente de desarrollo. (Como veremos en la Sección 11.4.4, cuando usamos Heroku para el despliegue, ImageMagick viene pre-instalado en producción.) En el IDE en la nube, podemos hacer lo siguiente:¹⁷

```
$ sudo apt-get update  
$ sudo apt-get install imagemagick --fix-missing
```

A continuación, necesitamos incluir la interfaz [MiniMagick](#) para ImageMagick de `carrierwave`, junto con un comando para el cambio de tamaño. Para este comando, existen varias posibilidades enlistadas en la [documentación de MiniMagick](#), pero la que queremos es `resize_to_limit: [400, 400]`,

¹⁶Es posible restringir el tamaño que se *despliega* con CSS, pero esto no cambia el tamaño de la imagen. En particular, las imágenes grandes aún tomarían algún tiempo en cargar. (Probablemente usted ha visitado sitios web donde imágenes “pequeñas” parecen tomar una eternidad en cargar. Ésta es la razón.)

¹⁷Obtuve esto de la [documentación oficial de Ubuntu](#). Si usted no está usando el IDE en la nube o un sistema Linux equivalente, realice una búsqueda en Google con “imagemagick <su plataforma>”. En OS X, `brew install imagemagick` debería funcionar si usted tiene [Homebrew](#) instalado.

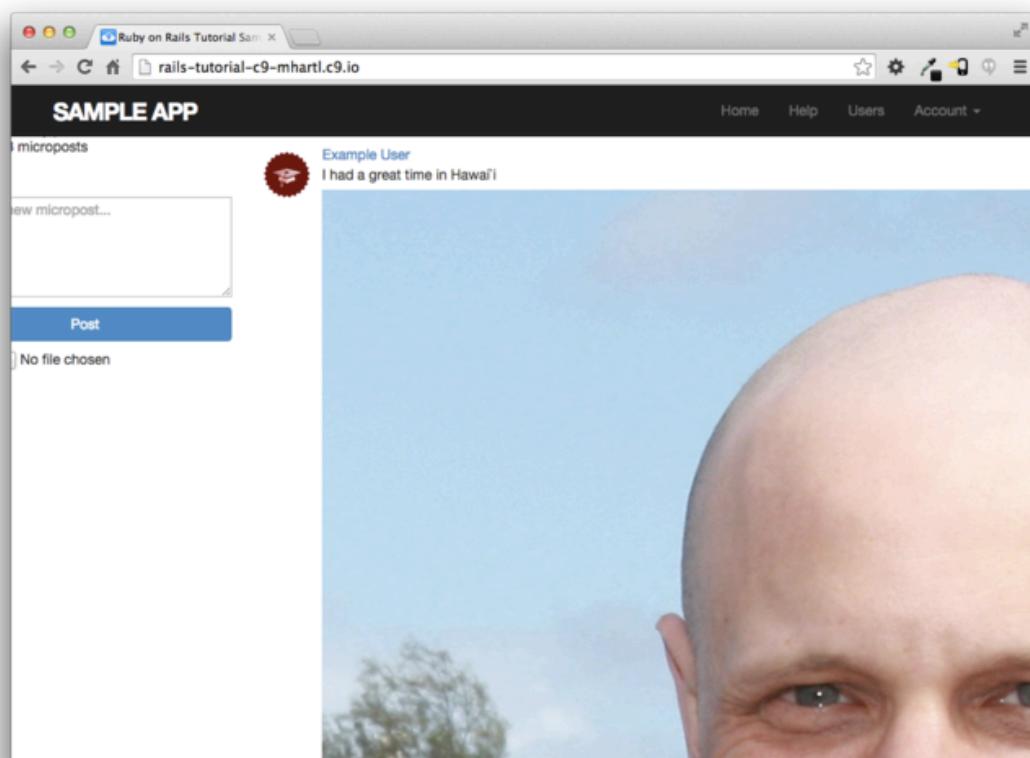


Figura 11.21: Una carga de imagen espantosamente grande.

que cambia el tamaño de imágenes grandes de forma que no sean mayores a 400px en cualquier dimensión, mientras que las imágenes pequeñas las conserva sin cambios. (Las otras posibilidades principales enlistadas en la documentación de CarrierWave acerca de MiniMagick *estiran* las imágenes si son muy pequeñas, que no es lo que deseamos en estos casos.) Con el código que se muestra en el Listado 11.63, las imágenes grandes cambian de tamaño de forma amigable (Figura 11.22).

Listado 11.63: Configurando el cargador de imágenes para el cambio de tamaño de éstas.

```
app/uploaders/picture_uploader.rb

class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
  process resize_to_limit: [400, 400]

  storage :file

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounted:
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Add a white list of extensions which are allowed to be uploaded.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

11.4.4 Carga de imágenes en producción

El cargador de imágenes que desarrollamos en la Sección 11.4.3 es suficientemente bueno para desarrollo, pero (como vimos en la línea `storage :file` del Listado 11.63) utiliza el sistema de archivos locales para almacenar las imágenes, lo cual no es una buena práctica en producción.¹⁸ En vez de esto, uti-

¹⁸Entre otras cosas, el almacenamiento en archivos en Heroku es temporal, por lo que las imágenes subidas serán eliminadas cada vez que usted haga un despliegue.

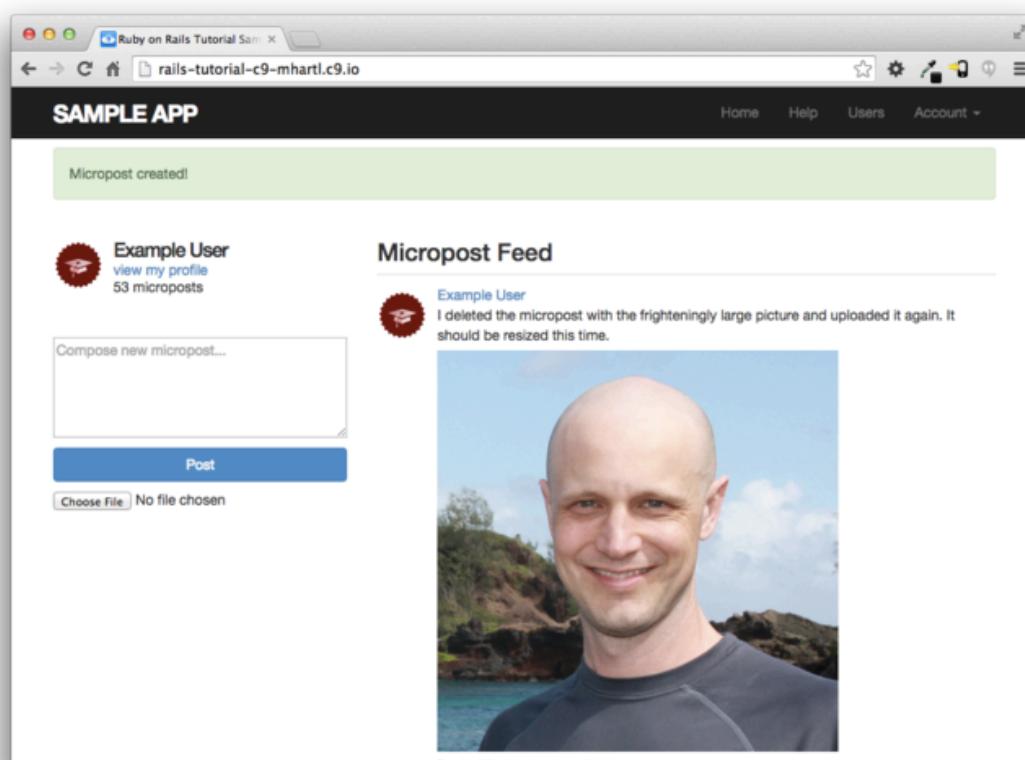


Figura 11.22: Una imagen a la que se le cambió el tamaño de forma amigable.

lizaremos un servicio de almacenamiento en la nube para guardar de forma separada las imágenes de nuestra aplicación.¹⁹

Para configurar nuestra aplicación en producción, de forma que use el almacenamiento en la nube, usaremos la gema `fog`, como se muestra en el [Listado 11.64](#).

Listado 11.64: Configurando el cargador de imágenes en producción.

app/uploaders/picture_uploader.rb

```
class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
  process resize_to_limit: [400, 400]

  if Rails.env.production?
    storage :fog
  else
    storage :file
  end

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounted:
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Add a white list of extensions which are allowed to be uploaded.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

El [Listado 11.64](#) utiliza el método booleano `production?` del Recuadro 7.1 para cambiar el método de almacenamiento basado en el ambiente:

```
if Rails.env.production?
  storage :fog
else
  storage :file
end
```

¹⁹ Esta es una sección desafiante y puede saltarla sin pérdida de continuidad.

Existen muchas opciones para almacenamiento en la nube, pero usaremos una de las más populares y mejor soportadas: el [Servicio de Almacenamiento Simple \(S3\)](#) de Amazon.²⁰ Aquí están los pasos esenciales para empezar:

1. Regístrese para obtener una cuenta de [Amazon Web Services](#).
2. Cree un usuario mediante el [Administrador de Identidad y Acceso AWS \(IAM\)](#) y guarde la llave de acceso y la llave secreta.
3. Cree un cubo S3 (con el nombre que usted elija) usando la [Consola AWS](#), y luego otorgue permisos de lectura y escritura al usuario creado en el paso anterior.

Para mayores detalles sobre estos pasos, consulte la [documentación S3](#)²¹ (y si es necesario, Google o Stack Overflow).

Una vez que haya creado y configurado su cuenta S3, debería crear y llenar el archivo de configuración de `carrierwave` como se muestra en el Listado 11.65.

Listado 11.65: Configurando CarrierWave para usar S3.

`config/initializers/carrier_wave.rb`

```
if Rails.env.production?
  CarrierWave.configure do |config|
    config.fog_credentials = {
      # Configuration for Amazon S3
      :provider          => 'AWS',
      :aws_access_key_id => ENV['S3_ACCESS_KEY'],
      :aws_secret_access_key => ENV['S3_SECRET_KEY']
    }
    config.fog_directory     = ENV['S3_BUCKET']
  end
end
```

²⁰S3 es un servicio de pago, pero el almacenamiento que necesitamos para preparar y probar la aplicación de ejemplo del Tutorial de Rails cuesta menos de un centavo por mes.

²¹<http://aws.amazon.com/documentation/s3/>

Como con la configuración del correo electrónico en producción ([Listado 10.56](#)), el [Listado 11.65](#) utiliza las variables de ambiente de Heroku para evitar almacenar información sensible en código duro. En la [Sección 10.3](#), estas variables fueron definidas automáticamente por el complemento SendGrid, pero en este caso necesitamos definirlas explícitamente, lo cual podemos hacer usando **heroku config:set** como sigue:

```
$ heroku config:set S3_ACCESS_KEY=<access key>
$ heroku config:set S3_SECRET_KEY=<secret key>
$ heroku config:set S3_BUCKET=<bucket name>
```

Con la configuración anterior, estamos listos para subir nuestros cambios y desplegar. Le recomiendo actualizar su archivo **.gitignore** como se muestra en el [Listado 11.66](#) de forma que el directorio de las imágenes sea ignorado.

Listado 11.66: Agregando el directorio de las imágenes al archivo **.gitignore**.

```
# See https://help.github.com/articles/ignoring-files for more about ignoring
# files.
#
# If you find yourself ignoring temporary files generated by your text editor
# or operating system, you probably want to add a global ignore instead:
#   git config --global core.excludesfile '~/.gitignore_global'

# Ignore bundler config.
.bundle

# Ignore the default SQLite database.
/db/*.sqlite3
/db/*.sqlite3-journal

# Ignore all logfiles and tempfiles.
/log/*.log
/tmp

# Ignore Spring files.
/spring/*.pid

# Ignore uploaded test images.
/public/uploads
```

Ahora estamos listos para subir nuestros cambios de la rama local y mezclarlos con la rama principal:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

Luego desplegamos, reiniciamos la base de datos, y re-alimentamos los datos de ejemplo:

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

Como Heroku viene con una instalación de ImageMagick, el resultado es una carga de imágenes y un cambio de tamaño de imagen exitosos en producción, como se muestra en la [Figura 11.23](#).

11.5 Conclusión

Con la adición del recurso Microposts, estamos casi terminando nuestra aplicación de ejemplo. Todo lo que falta es agregar una capa social para permitir a los usuarios seguirse entre sí. Aprenderemos cómo modelar tales relaciones de usuario, y ver las implicaciones para el avance de micromensajes, en el [Capítulo 12](#).

Si usted se saltó la [Sección 11.4.4](#), asegúrese de guardar y mezclar sus cambios:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

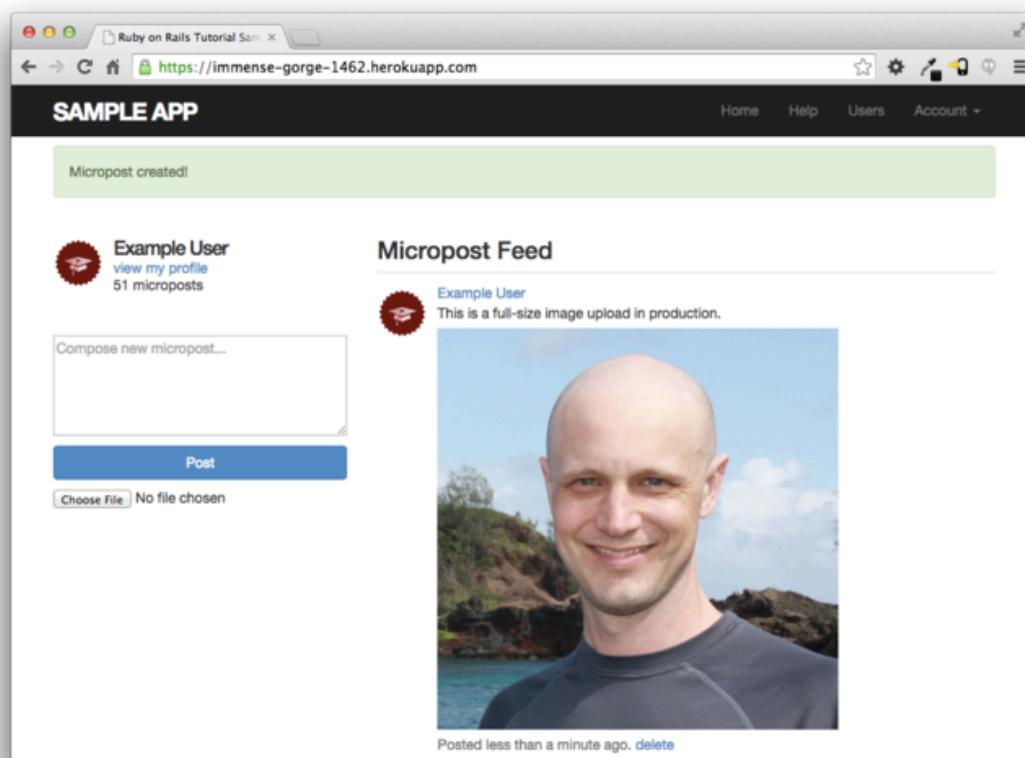


Figura 11.23: Carga de imágenes en producción.

Luego despliegue en producción:

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

Vale la pena observar que en este capítulo realizamos las últimas instalaciones de gemas necesarias. Como referencia, la versión final del archivo **Gemfile** se muestra en el [Listado 11.67](#).

Listado 11.67: La versión final del archivo **Gemfile** para la aplicación de ejemplo.

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

gem 'rails',                      '4.2.2'
gem 'bcrypt',                       '3.1.7'
gem 'faker',                        '1.4.2'
gem 'carrierwave',                 '0.10.0'
gem 'mini_magick',                 '3.8.0'
gem 'fog',                          '1.26.0'
gem 'will_paginate',                '3.0.7'
gem 'bootstrap-will_paginate',      '0.0.10'
gem 'bootstrap-sass',                '3.2.0.0'
gem 'sass-rails',                   '5.0.2'
gem 'uglifier',                     '2.5.3'
gem 'coffee-rails',                  '4.1.0'
gem 'jquery-rails',                  '4.0.3'
gem 'turbolinks',                   '2.3.0'
gem 'jbuilder',                      '2.2.3'
gem 'sdoc',                         '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',                    '1.3.9'
  gem 'byebug',                     '3.4.0'
  gem 'web-console',                 '2.0.0.beta3'
  gem 'spring',                      '1.1.3'
end

group :test do
  gem 'minitest-reporters', '1.0.5'
  gem 'mini_backtrace',           '0.1.3'
  gem 'guard-minitest',           '2.3.1'
end
```

```
group :production do
  gem 'pg',           '0.17.1'
  gem 'rails_12factor', '0.0.2'
  gem 'puma',          '2.11.1'
end
```

11.5.1 Qué aprendimos en este capítulo

- Los micromensajes, como los usuarios, son modelados como un recurso respaldado por un modelo Active Record.
- Rails soporta índices de llave múltiple.
- Podemos modelar un usuario que tiene muchos micromensajes usando los métodos `has_many` y `belongs_to` en los modelos `User` y `Micropost`, respectivamente.
- La combinación `has_many/belongs_to` da lugar a métodos que funcionan a través de la relación.
- El código `user.microposts.build(...)` regresa un objeto de un nuevo `Micropost` automáticamente asociado con el usuario dado.
- Rails soporta el ordenamiento por default mediante `default_scope`.
- Los alcances toman funciones anónimas como argumentos.
- La opción `dependent: :destroy` causa que los objetos sean destruidos al mismo tiempo que sus objetos asociados.
- La paginación y el conteo de objetos pueden ser realizados mediante asociaciones, lo que nos conduce automáticamente a código eficiente.
- Los archivos Fixtures soportan la creación de asociaciones.
- Es posible pasar variables a los parciales de Rails.

- El método `where` puede ser usado para realizar selecciones de Active Record.
- Podemos implementar operaciones seguras al crear y destruir siempre objetos dependientes a través de su asociación.
- Podemos subir y cambiar el tamaño de las imágenes usando `carrierwave`.

11.6 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Refactorice la página Home para que utilice parciales separados para los dos bloques de código `if-else`.
2. Agregue pruebas para el conteo de micromensajes de la barra lateral (incluyendo la adecuada pluralización). El Listado 11.68 le ayudará a empezar.
3. Siguiendo la plantilla del Listado 11.69, escriba una prueba del cargador de imágenes de la Sección 11.4. Como preparación, debería agregar una imagen al directorio fixtures (usando por ejemplo, `cp app/assets/-images/rails.png test/fixtures/`). Para evitar errores, también necesitará configurar `carrierwave` para que no realice el cambio de tamaño de imágenes en las pruebas creando un archivo de inicialización como se muestra en el Listado 11.70. Las afirmaciones adicionales del Listado 11.69 verifican tanto el campo para subir archivos en la página Home como un atributo de imagen válido en el micromensaje resultante

de un envío de datos exitoso. Observe el uso del método especial `fixture_file_upload` para subir archivos como fixtures dentro de las pruebas.²²

Sugerencia: Para verificar que el atributo `picture` sea válido, utilice el método `assigns` que se mencionó en la Sección 10.1.4 para accesar el micromensaje en la acción `create` luego de un envío de datos válido.

²²Los usuarios de Windows deben agregar un parámetro `:binary`: `fixture_file_upload(file, type, :binary)`.

Listado 11.68: Una plantilla para la prueba del conteo de micromensajes de la barra lateral.

test/integration/microposts_interface_test.rb

```
require 'test_helper'

class MicropostInterfaceTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  .

  .

  .

  test "micropost sidebar count" do
    log_in_as(@user)
    get root_path
    assert_match "#{FILL_IN} microposts", response.body
    # User with zero microposts
    other_user = users(:mallory)
    log_in_as(other_user)
    get root_path
    assert_match "0 microposts", response.body
    other_user.microposts.create!(content: "A micropost")
    get root_path
    assert_match FILL_IN, response.body
  end
end
```

Listado 11.69: Una plantilla para probar la carga de imágenes.

test/integration/microposts_interface_test.rb

```
require 'test_helper'

class MicropostInterfaceTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "micropost interface" do
    log_in_as(@user)
    get root_path
    assert_select 'div.pagination'
    assert_select 'input[type=FILL_IN]'
    # Invalid submission
    post microposts_path, micropost: { content: "" }
```

```
assert_select 'div#error_explanation'
# Valid submission
content = "This micropost really ties the room together"
picture = fixture_file_upload('test/fixtures/rails.png', 'image/png')
assert_difference 'Micropost.count', 1 do
  post microposts_path, micropost: { content: content, picture: FILL_IN }
end
assert FILL_IN.picture?
follow_redirect!
assert_match content, response.body
# Delete a post.
assert_select 'a', 'delete'
first_micropost = @user.microposts.paginate(page: 1).first
assert_difference 'Micropost.count', -1 do
  delete micropost_path(first_micropost)
end
# Visit a different user.
get user_path(users(:archer))
assert_select 'a', { text: 'delete', count: 0 }
end
.
.
.
end
```

Listado 11.70: Un archivo de inicialización para evitar el cambio de tamaño de imágenes en las pruebas.

config/initializers/skip_image_resizing.rb

```
if Rails.env.test?
  CarrierWave.configure do |config|
    config.enable_processing = false
  end
end
```

Capítulo 12

Siguiendo usuarios

En este capítulo, completaremos la aplicación de ejemplo agregando una capa social que permita a los usuarios seguir (y dejar de seguir) a otros usuarios, resultando en que la página Home de cada usuario muestre un status del avance de los micromensajes de los usuarios seguidos. Empezaremos aprendiendo cómo modelar relaciones entre usuarios en la [Sección 12.1](#), y luego construiremos la interfaz web correspondiente en la [Sección 12.2](#) (incluyendo una introducción a Ajax). Terminaremos desarrollando un status de avance totalmente funcional en la [Sección 12.3](#).

Este capítulo final contiene parte del material más desafiante del tutorial, incluyendo algunos trucos de Ruby/SQL para crear el status de avance. A través de estos ejemplos, usted verá cómo Rails puede manejar aún los más intrincados modelos de datos, lo cual debería serle de gran utilidad conforme usted desarrolle sus propias aplicaciones con sus requerimientos específicos. Para ayudarle en la transición del tutorial al desarrollo independiente, la [Sección 12.4](#) ofrece algunas sugerencias a recursos más avanzados.

Como el material de este capítulo es particularmente retador, antes de escribir cualquier código haremos una pausa momentánea y echaremos un vistazo por la interfaz. Como en capítulos anteriores, en esta etapa temprana representaremos las páginas utilizando bosquejos.¹ El flujo completo de las páginas será como sigue: un usuario (John Calvin) inicia en su página de perfil

¹Las nuevas fotografías de los bosquejos provienen de http://www.flickr.com/photos/john_lustig/2518452221/ y <https://www.flickr.com/photos/renemensen/9187111340>.

(Figura 12.1) y navega a la página de usuarios (Figura 12.2) para seleccionar un usuario a quién seguir. Calvin navega al perfil de un segundo usuario, Thomas Hobbes (Figura 12.3), da click en el botón “Follow” para seguir a ese usuario. Esto cambia el botón “Follow” a “Unfollow” e incrementa el número de “followers” (seguidores) de Hobbes en uno (Figura 12.4). Navegando a su página home, Calvin ahora ve incrementado su conteo de “following” (siguiendo) y encuentra los micromensajes de Hobbes en su status de avance (Figura 12.5). El resto de este capítulo está dedicado a hacer que este flujo de páginas realmente funcione.

12.1 El modelo relación

Nuestro primer paso hacia la implementación de seguir usuarios es construir un modelo de datos, que no es tan directo como parece. Ingenuamente, parece que una relación **has_many** funcionaría: un usuario tiene muchos usuarios a los que sigue y también tiene muchos usuarios que lo siguen. Como veremos, existe un problema con este modelado, y aprenderemos cómo arreglarlo usando **has_many :through**.

Como es usual, los usuarios de Git deben crear una nueva rama:

```
$ git checkout master
$ git checkout -b following-users
```

12.1.1 Un problema con el modelo de datos (y una solución)

Como primer paso hacia la construcción de un modelo de datos para seguir usuarios, examinemos un caso típico. Por ejemplo, considere un usuario que sigue a un segundo usuario: podríamos decir, por ejemplo, que Calvin está siguiendo a Hobbes, y por tanto, que Hobbes está siendo seguido por Calvin, de forma que Calvin es el *seguidor* y Hobbes es el *seguido*. Usando la convención por default de Rails para la pluralización, el conjunto de todos los usuarios siguiendo a un usuario dado es: los *seguidores* del usuario, y **hobbes.followers** es un arreglo de tales usuarios. En el caso contrario, adoptaremos la convención



 John Calvin **Microposts (10)**

 Lorem ipsum dolor sit amet, consectetur
Posted 1 day ago. [delete](#)

 Consectetur adipisicing elit
Posted 2 days ago. [delete](#)

 Lorem ipsum dolor sit amet, consectetur
Posted 3 days ago. [delete](#)

[Previous](#) [1](#) [2](#) [3](#) [Next](#)



Figura 12.1: El perfil de usuario actual.

All users

[Previous](#) [1](#) [2](#) [3](#) [Next](#)

 [Thomas Hobbes](#)

 [Sasha Smith](#)

 [Hippo Potamus](#)

 [David Jones](#)

[Previous](#) [1](#) [2](#) [3](#) [Next](#)

Figura 12.2: Buscando un usuario a quien seguir.



Thomas Hobbes [Follow](#)

23 144
following followers

Microposts (42)

 Also poor, nasty, brutish, and short.
Posted 1 day ago.

 Life of man in a state of nature is solitary.
Posted 2 days ago.

 Lex naturalis is found out by reason.
Posted 2 days ago.

[Previous](#) [1](#) [2](#) [3](#) [Next](#)

Figura 12.3: El perfil del usuario a seguir, con un botón para seguirlo.



Thomas Hobbes [Unfollow](#)

[23](#) [145](#)
[following](#) [followers](#)

Microposts (42)

 Also poor, nasty, brutish, and short.
Posted 1 day ago.

 Life of man in a state of nature is solitary.
Posted 2 days ago.

 Lex naturalis is found out by reason.
Posted 2 days ago.

[Previous](#) [1](#) [2](#) [3](#) [Next](#)

Figura 12.4: Un perfil con un botón para dejar de seguir al usuario y un conteo de seguidores incrementado.

John Calvin
[view my profile](#)

17 microposts

51 77

[following](#) [followers](#)

Compose new micropost...

[Post](#)

[Upload image](#) [Choose image](#)

Micropost Feed

 [Thomas Hobbes](#) Also poor, nasty, brutish, and short.
Posted 1 day ago.

 [Sasha Smith](#) Lorem ipsum dolor sit amet, consectetur.
Posted 2 days ago.

 [Thomas Hobbes](#) Life of man in a state of nature is solitary
Posted 2 days ago.

 [John Calvin](#) Excepteur sint occaecat
Posted 3 days ago. [delete](#)

[Previous](#) [1](#) [2](#) [3](#) [Next](#)

Figura 12.5: La página Home con un status de avance y el conteo de usuarios a quienes se sigue, incrementado.

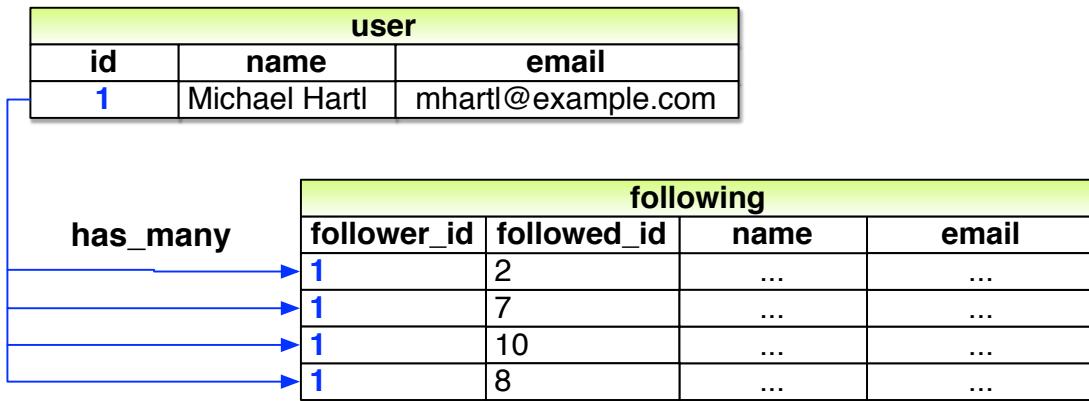


Figura 12.6: Una implementación ingenua del seguimiento de usuarios.

de Twitter y les llamaremos a tales usuarios *seguidos*, (como en “50 seguidos, 75 seguidores”), con un arreglo correspondiente `calvin.following`.

Esta discusión sugiere modelar a los usuarios seguidos como en la Figura 12.6, con una tabla **seguidos** y una asociación **has_many**. Puesto que `user.following` debería ser una colección de usuarios, cada renglón de la tabla **seguidos** debería ser un usuario, identificado por `followed_id`, junto con el `follower_id` para establecer la asociación.² Adicionalmente, puesto que cada renglón es un usuario, necesitaríamos incluir los atributos del otro usuario, incluyendo el nombre, dirección electrónica, contraseña, etc.

El problema con el modelo de datos de la Figura 12.6 es que es terriblemente redundante: cada renglón contiene no sólo cada id de usuario seguido, sino también toda la información restante—todo lo cual *ya* está en la tabla **users**. Aún peor, para modelar a los *seguidores* necesitaríamos una tabla separada, análogamente redundante. Finalmente, el mantenimiento del modelo de datos sería una pesadilla: cada vez que un usuario cambiara, digamos su nombre, necesitaríamos actualizar no sólo su registro en la tabla **users** sino que también *cada renglón que contuviera ese usuario tanto en la tabla de seguidores como en la de seguidos*.

El problema aquí es que estamos omitiendo una abstracción subyacente.

²Por simplicidad, la Figura 12.6 omite la columna `id` de la tabla `following`.

Una forma de encontrar el modelo apropiado es considerar cómo podemos implementar el acto de *seguir* en una aplicación web. Recuerde de la Sección 7.1.2 que la arquitectura REST involucra *recursos* que son creados y destruidos. Esto nos lleva a hacernos un par de preguntas: Cuando un usuario sigue a otro, ¿qué es lo que será creado? Cuando un usuario *deja* de seguir a otro usuario, ¿qué es lo que será destruido? Después de reflexionar, vemos que en estos casos la aplicación debería crear o destruir una *relación* entre dos usuarios. Así, un usuario tiene muchas relaciones y tiene muchos **seguidos** (o **seguidores**) *a través* de estas relaciones.

Existe un detalle adicional que necesitamos considerar independientemente de nuestro modelo de datos en esta aplicación: a diferencia de las amistades simétricas estilo Facebook, que siempre son recíprocas (al menos a nivel de modelo de datos), al estilo Twitter las relaciones de seguimiento son potencialmente *asimétricas*—Calvin puede seguir a Hobbes sin que Hobbes siga a Calvin. Para distinguir entre estos dos casos, adoptaremos la terminología de relaciones *activas* y *pasivas*: si Calvin está siguiendo a Hobbes pero no al revés, Calvin tiene una relación activa con Hobbes y Hobbes tiene una relación pasiva con Calvin.³

Nos enfocaremos ahora en utilizar relaciones activas para generar una lista de usuarios seguidos, y consideraremos el caso pasivo en la Sección 12.1.5. La Figura 12.6 sugiere cómo implementarlo: puesto que cada usuario seguido está identificado de forma única por **followed_id**, podríamos convertir **seguir** en una tabla de **relaciones_activas**, omitimos los detalles del usuario, y usamos el **followed_id** para recuperar el usuario seguido de la tabla **users**. Un diagrama del modelo de datos se muestra en la Figura 12.7.

Como terminaremos usando la misma tabla de la base de datos tanto para las relaciones activas como para las pasivas, usaremos el término genérico *relación* para el nombre de la tabla, con un modelo correspondiente. El resultado es el modelo de datos **Relationship** que se muestra en la Figura 12.8. Empezaremos a ver en la Sección 12.1.4 cómo usar el modelo de relación para simular tanto los modelos de relaciones activas como pasivas.

Para empezar la implementación, primero generaremos una migración co-

³Agradezco al lector Paul Fioravanti por sugerir esta terminología.

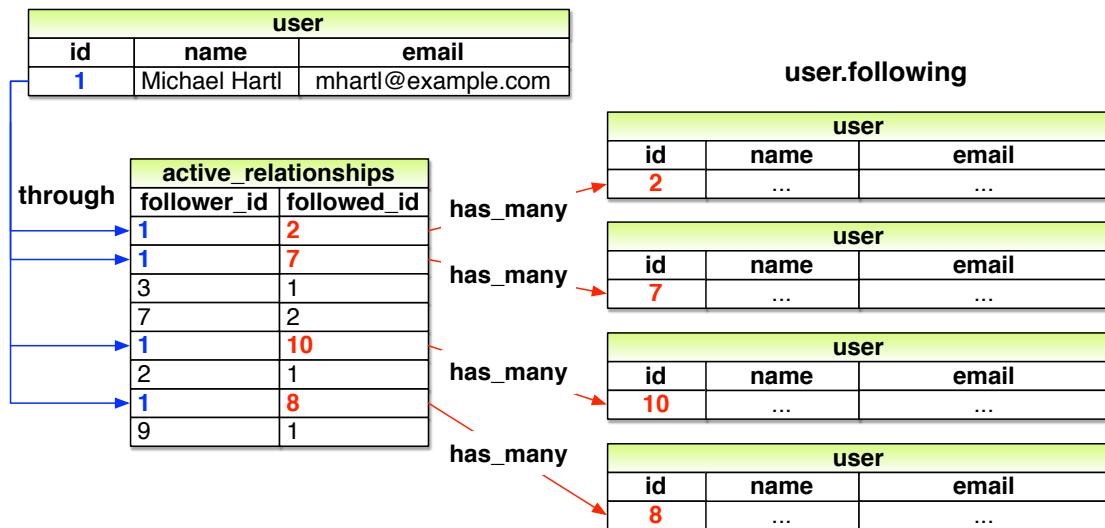


Figura 12.7: Un modelo de usuarios seguidos mediante relaciones activas.

relationships	
id	integer
follower_id	integer
followed_id	integer
created_at	datetime
updated_at	datetime

Figura 12.8: El modelo de datos **Relationship**.

rrespondiente a la Figura 12.8:

```
$ rails generate model Relationship follower_id:integer followed_id:integer
```

Como estaremos buscando las relaciones por `follower_id` y por `followed_id`, deberíamos agregar, por eficiencia, un índice a cada columna, como se muestra en el Listado 12.1.

Listado 12.1: Agregando índices a la tabla `relationships`.

`db/migrate/[timestamp]_create_relationships.rb`

```
class CreateRelationships < ActiveRecord::Migration
  def change
    create_table :relationships do |t|
      t.integer :follower_id
      t.integer :followed_id

      t.timestamps null: false
    end
    add_index :relationships, :follower_id
    add_index :relationships, :followed_id
    add_index :relationships, [:follower_id, :followed_id], unique: true
  end
end
```

El Listado 12.1 también incluye un índice de llave múltiple que asegura la unicidad de las parejas (`follower_id`, `followed_id`), de forma que un usuario no puede seguir a otro más de una vez. (Compare con el índice de unicidad del correo electrónico del Listado 6.28 y el índice de llave múltiple del Listado 11.1.) Como veremos empezando la Sección 12.1.4, nuestra interfaz de usuario no permitirá que esto suceda, pero agregar un índice único se encarga de arrojar un error si un usuario intenta crear una relación duplicada de alguna forma (por ejemplo, usando una herramienta de línea de comandos tal como `curl`).

Para crear la tabla `relationships`, migramos la base de datos como es usual:

```
$ bundle exec rake db:migrate
```

12.1.2 Asociaciones Usuario / relación

Antes de implementar el seguimiento de usuarios, primero necesitamos establecer la asociación entre usuarios y relaciones. Un usuario tiene muchas relaciones, y—como las relaciones involucran a *dos* usuarios—una relación pertenece tanto al usuario seguido como al seguidor.

Como con los micromensajes de la Sección 11.1.3, crearemos nuevas relaciones usando la asociación de usuario, con código del estilo

```
user.active_relationships.build(followed_id: ...)
```

En este momento, usted podría esperar código como el de la Sección 11.1.3, y es similar, pero existen dos diferencias importantes.

Primero, en el caso de la asociación usuario / micromensaje podíamos escribir

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

Esto funciona porque por convención Rails busca un modelo **Micropost** correspondiente a los símbolos **:microposts**.⁴ En nuestro caso actual, queremos escribir

⁴Técnicamente, Rails convierte el argumento de **has_many** a un nombre de clase utilizando el método **classify**, el cual convierte "**foo_bars**" en "**FooBar**".

```
has_many :active_relationships
```

aún cuando el modelo subyacente sea llamado **Relationship**. Entonces tendríamos que decirle a Rails el nombre de la clase modelo que debe buscar.

Segundo, antes escribimos

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

en el modelo **Micropost**. Esto funciona porque la tabla **microposts** tiene un atributo **user_id** para identificar al usuario (Sección 11.1.1). Un id utilizado de esta forma para conectar dos tablas de la base de datos se conoce como *llave foránea*, y cuando la llave foránea de un objeto que pertenece al modelo **User** es **user_id**, permite a Rails inferir la asociación automáticamente: por default, Rails espera una llave foránea de la forma **<class>_id**, donde **<class>** es la versión en minúsculas del nombre de la clase.⁵ En este caso, aunque aún estamos tratando con usuarios, el usuario que sigue a otro ahora es identificado con la llave foránea **follower_id**, por lo que necesitamos decirle eso a Rails.

El resultado del razonamiento anterior es la asociación usuario / relación que se muestra en los Listados 12.2 y 12.3.

Listado 12.2: Implementando la asociación de las relaciones activas.

app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  has_many :active_relationships, class_name: "Relationship",
    foreign_key: "follower_id",
    dependent: :destroy
```

⁵Técnicamente, Rails utiliza el método **underscore** para convertir el nombre de la clase en un id. Por ejemplo, **"FooBar".underscore** es **"foo_bar"**, por lo que la llave foránea de un objeto **FooBar** debería ser **foo_bar_id**.

Método	Propósito
<code>active_relationship.follower</code>	Regresa al seguidor
<code>active_relationship.followed</code>	Regresa al usuario seguido
<code>user.active_relationships.create(followed_id: user.id)</code>	Crea una relación activa asociada con <code>user</code>
<code>user.active_relationships.create!(followed_id: user.id)</code>	Crea una relación activa asociada con <code>user</code> (y arroja una excepción en caso de fallo)
<code>user.active_relationships.build(followed_id: user.id)</code>	Regresa un objeto de nueva relación asociado con <code>user</code>

Tabla 12.1: Un resumen de los métodos de la asociación usuario / relación activa.

```

  .
  .
  .
end
```

(Puesto que destruir a un usuario también debería destruir sus relaciones, hemos agregado `dependent::destroy` a la asociación.)

Listado 12.3: Agregando la asociación `belongs_to` del seguidor al modelo `Relationship`.

`app/models/relationship.rb`

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end
```

La asociación `followed` no es realmente necesaria sino hasta la Sección 12.1.4, pero la estructura paralela seguidor / seguido es más clara si las implementamos al mismo tiempo.

Las relaciones de los Listados 12.2 y 12.3 dan lugar a métodos análogos a los que vimos en la Tabla 11.1, como se observa en la Tabla 12.1.

12.1.3 Validaciones de relación

Antes de continuar, agregaremos un par de validaciones al modelo de relación por completez. Las pruebas ([Listado 12.4](#)) y el código de la aplicación ([Listado 12.5](#)) son directos. Como con el archivo fixture generado para el usuario ([Listado 6.29](#)), el archivo fixture generado para la relación también viola la restricción de unicidad impuesta por la migración correspondiente ([Listado 12.1](#)). La solución del [Listado 12.6](#) es equivalente a remover el contenido del archivo fixture, como en el [Listado 6.30](#).

Listado 12.4: Probando las validaciones al modelo `Relationship`.

`test/models/relationship_test.rb`

```
require 'test_helper'

class RelationshipTest < ActiveSupport::TestCase

  def setup
    @relationship = Relationship.new(follower_id: 1, followed_id: 2)
  end

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

  test "should require a follower_id" do
    @relationship.follower_id = nil
    assert_not @relationship.valid?
  end

  test "should require a followed_id" do
    @relationship.followed_id = nil
    assert_not @relationship.valid?
  end
end
```

Listado 12.5: Agregando las validaciones al modelo `Relationship`.

`app/models/relationship.rb`

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
  validates :follower_id, presence: true
```

```
validates :followed_id, presence: true
end
```

Listado 12.6: Removiendo los contenidos del archivo fixture para la relación.
test/fixtures/relationships.yml

```
# empty
```

En este momento, las pruebas deberían estar en **VERDE**:

Listado 12.7: **VERDE**

```
$ bundle exec rake test
```

12.1.4 Usuarios seguidos

Llegamos ahora al corazón de las asociaciones de relación: seguir y ser seguido. Aquí usaremos **has_many :through** por primera vez: un usuario tiene muchas relaciones de usuarios a los que sigue, como se muestra en la [Figura 12.7](#). Por default, en una asociación **has_many :through** Rails busca una llave foránea correspondiente a la versión en singular de la asociación. En otras palabras, con código como

```
has_many :followeds, through: :active_relationships
```

Rails vería “followeds” y utilizaría el singular “followed”, ensamblando una colección que usa el **followed_id** en la tabla **relationships**. Pero, como observamos en la [Sección 12.1.1](#), **user.followeds** es más bien extraña, por lo que escribiremos mejor **user.following**. Naturalmente, Rails nos permite sobreescribir el default, en este caso usando el parámetro **source** (como se muestra en el [Listado 12.8](#)), el cual explícitamente le dice a Rails que la fuente del arreglo **following** es el conjunto de ids **followed**.

Listado 12.8: Agregando la asociación **following** al modelo **User**.

app/models/user.rb

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  has_many :active_relationships, class_name: "Relationship",
    foreign_key: "follower_id",
    dependent: :destroy
  has_many :following, through: :active_relationships, source: :followed
  .
  .
  .
end
```

La asociación definida en el [Listado 12.8](#) nos lleva a una combinación poderosa de Active Record y a un comportamiento tipo arreglo. Por ejemplo, podemos verificar si la colección de usuarios seguidos incluye otro usuario con el método **include?** ([Sección 4.3.1](#)), ó busca objetos a través de la asociación:

```
user.following.include?(other_user)
user.following.find(other_user)
```

Aunque en muchos contextos podemos tratar efectivamente **following** como un arreglo, Rails es inteligente acerca de cómo maneja las cosas internamente. Por ejemplo, código como

```
following.include?(other_user)
```

parece que pudiera traer todos los usuarios seguidos de la base de datos para aplicarles el método **include?** pero de hecho por eficiencia Rails se encarga de que la comparación suceda directamente en la base de datos. (Compare con el código de la [Sección 11.2.1](#), donde vimos que

```
user.microposts.count
```

realiza el conteo directamente en la base de datos.)

Para manipular las relaciones de seguimiento, introduciremos métodos de utilería `follow` y `unfollow` de forma que podamos escribir, por ejemplo, `user.follow(other_user)`. También agregaremos un método booleano asociado `following?` para probar si un usuario está siguiendo a otro.⁶

Esta es exactamente la clase de situación donde me gusta escribir algunas pruebas primero. La razón es que estamos algo lejos de escribir una interfaz web que funcione para seguir usuarios, pero es difícil proceder sin alguna clase de *cliente* para el código que estamos desarrollando. En este caso, es fácil escribir una pequeña prueba para el modelo `User`, en la que usemos `following?` para asegurarnos de que el usuario no está siguiendo a otro usuario, usamos `follow` para seguir a otro usuario y usamos `following?` para verificar que la operación fue exitosa. Finalmente invocamos `unfollow` y verificamos que funcionó. El resultado aparece en el [Listado 12.9](#).

Listado 12.9: Pruebas para algunos métodos de utilería para el “seguimiento”.

ROJO

```
test/models/user_test.rb

require 'test_helper'

class UserTest < ActiveSupport::TestCase
  .
  .
  .
  test "should follow and unfollow a user" do
    michael = users(:michael)
    archer = users(:archer)
    assert_not michael.following?(archer)
    michael.follow(archer)
    assert michael.following?(archer)
    michael.unfollow(archer)
    assert_not michael.following?(archer)
  end
end
```

⁶Una vez que usted tenga mucha experiencia modelando un dominio en particular, a menudo podrá adivinar tales métodos de utilería por adelantado, y aún cuando no pueda, a menudo se encontrará escribiéndolos para hacer las pruebas más limpias. En este caso, está bien si usted no los adivinó. El desarrollo de software es usualmente un proceso iterativo—usted escribe código hasta que empieza a verse feo, luego lo refactoriza—pero por brevedad del tutorial, lo agilizamos un poco.

Al referirnos a los métodos de la Tabla 12.1, podemos escribir los métodos **follow**, **unfollow**, y **following?** usando la asociación con **following**, como se muestra en el Listado 12.10. (Observe que hemos omitido la variable de usuario **self** cada vez que es posible.)

Listado 12.10: Métodos de utilería para seguir. VERDE*app/models/user.rb*

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

  # Follows a user.
  def follow(other_user)
    active_relationships.create(followed_id: other_user.id)
  end

  # Unfollows a user.
  def unfollow(other_user)
    active_relationships.find_by(followed_id: other_user.id).destroy
  end

  # Returns true if the current user is following the other user.
  def following?(other_user)
    following.include?(other_user)
  end

  private
  .
  .
  .
end
```

Con el código del Listado 12.10, las pruebas deberían estar en VERDE:

Listado 12.11: VERDE

```
$ bundle exec rake test
```

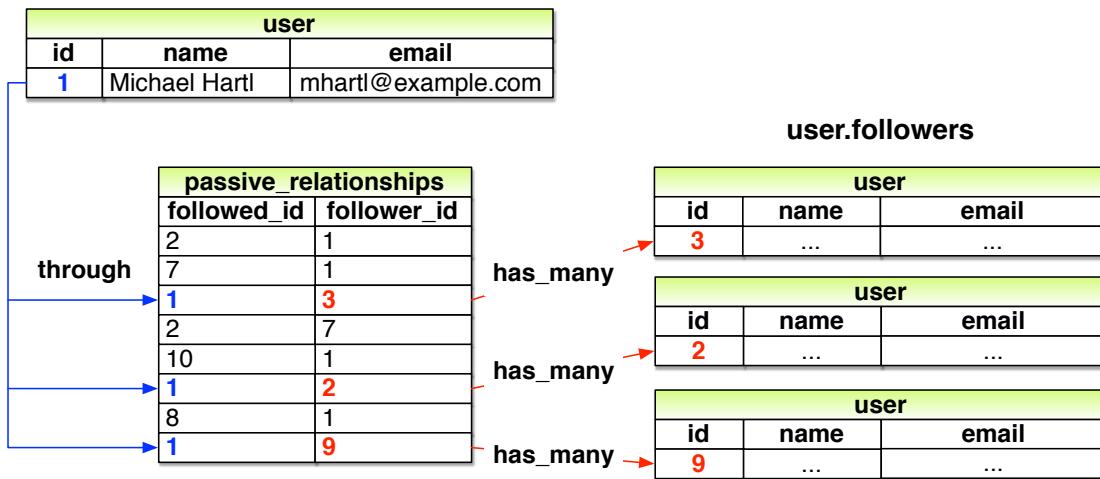


Figura 12.9: Un modelo para seguidores de usuarios a través de relaciones pasivas.

12.1.5 Seguidores

La pieza final del rompecabezas de relaciones es agregar un método `user.followers` para acompañar a `user.following`. Puede que usted haya notado en la Figura 12.7 que toda la información necesaria para extraer un arreglo de seguidores está presente en la tabla `relationships` (la cual estamos tratando como la tabla `active_relationships` mediante el código del Listado 12.2). De hecho, la técnica es exactamente la misma que para los usuarios seguidos, con los roles de `follower_id` y `followed_id` al revés, y con `passive_relationships` en vez de `active_relationships`. El modelo de datos aparece entonces como en la Figura 12.9.

La implementación del modelo de datos de la Figura 12.9 es análoga a la del Listado 12.8, como puede observarse en el Listado 12.12.

Listado 12.12: Implementando `user.followers` mediante relaciones pasivas.

`app/models/user.rb`

```
class User < ActiveRecord::Base
```

```

has_many :microposts, dependent: :destroy
has_many :active_relationships, class_name: "Relationship",
                                 foreign_key: "follower_id",
                                 dependent: :destroy
has_many :passive_relationships, class_name: "Relationship",
                                 foreign_key: "followed_id",
                                 dependent: :destroy
has_many :following, through: :active_relationships, source: :followed
has_many :followers, through: :passive_relationships, source: :follower
.
.
.
end

```

Vale la pena notar que pudimos haber omitido la llave `:source` para `followers` en el Listado 12.12, usando simplemente

```
has_many :followers, through: :passive_relationships
```

Esto es porque, en el caso de un atributo `:followers`, Rails hará singular la palabra “followers” y automáticamente buscará la llave foránea `follower_id` en este caso. El Listado 12.8 mantiene la llave `:source` para enfatizar la estructura análoga con la asociación `has_many :following`.

Podemos probar de forma conveniente el modelo de datos anterior usando el método `followers.include?` como se muestra en el Listado 12.13. (El Listado 12.13 pudo haber utilizado un método `followed_by?` para complementar el método `following?` pero sucede que no lo necesitaremos en nuestra aplicación.)

Listado 12.13: Una prueba para `followers`. VERDE

`test/models/user_test.rb`

```

require 'test_helper'

class UserTest < ActiveSupport::TestCase
.
.
.
test "should follow and unfollow a user" do
  michael = users(:michael)

```

```

archer = users(:archer)
assert_not michael.following?(archer)
michael.follow(archer)
assert michael.following?(archer)
assert archer.followers.include?(michael)
michael.unfollow(archer)
assert_not michael.following?(archer)
end
end

```

El [Listado 12.13](#) agrega únicamente una línea a la prueba del [Listado 12.9](#), pero hay tantas cosas que deben estar correctas para que pase que la convierte en una prueba muy sensible derivada del código del [Listado 12.12](#).

En este momento, el conjunto de pruebas completo debería estar en [VERDE](#):

```
$ bundle exec rake test
```

12.2 Una interfaz web para seguir usuarios

La [Sección 12.1](#) requirió fuertes habilidades en el modelado de datos por nuestra parte, y está bien si nos demoramos un poco en asimilarlo. De hecho, una de las mejores formas de entender las asociaciones es usándolas en la interfaz web.

En la introducción de este capítulo, vimos una versión preliminar del flujo de la página para el seguimiento de usuarios. En esta sección, implementaremos la interfaz básica y la funcionalidad de seguimiento y de dejar de seguir a un usuario que se mostró en esos bosquejos. También crearemos páginas separadas para mostrar los usuarios seguidos y los seguidores. En la [Sección 12.3](#), completaremos nuestra aplicación de ejemplo agregando el status de avance del usuario.

12.2.1 Datos de ejemplo de seguimiento

Como en capítulos anteriores, será conveniente utilizar la tarea Rake para alimentar la base de datos con relaciones de ejemplo. Esto nos permitirá diseñar la

apariencia de las páginas web primero, postergando la funcionalidad del servidor para después en esta sección.

El código que alimenta las siguientes relaciones, se muestra en el [Listado 12.14](#). Aquí nos encargamos, de forma algo arbitraria, de que el primer usuario siga a los usuarios 3 al 51, y luego tenemos a los usuarios 4 al 41 para que sigan a su vez al primero. Las relaciones resultantes serán suficientes para desarrollar la interfaz de la aplicación.

Listado 12.14: Agregando relaciones de seguimiento a los datos de ejemplo.

db/seeds.rb

```
# Users
User.create!(name: "Example User",
            email: "example@railstutorial.org",
            password: "foobar",
            password_confirmation: "foobar",
            admin: true,
            activated: true,
            activated_at: Time.zone.now)

99.times do |n|
  name = Faker::Name.name
  email = "example-#{n+1}@railstutorial.org"
  password = "password"
  User.create!(name: name,
              email: email,
              password: password,
              password_confirmation: password,
              activated: true,
              activated_at: Time.zone.now)
end

# Microposts
users = User.order(:created_at).take(6)
50.times do
  content = Faker::Lorem.sentence(5)
  users.each { |user| user.microposts.create!(content: content) }
end

# Following relationships
users = User.all
user = users.first
following = users[2..50]
followers = users[3..40]
following.each { |followed| user.follow(followed) }
followers.each { |follower| follower.follow(user) }
```



Figura 12.10: Un bosquejo del parcial de estadísticas.

Para ejecutar el código del [Listado 12.14](#), realimentaremos la base de datos como es usual:

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

12.2.2 Estadísticas y el formulario de seguimiento

Ahora que nuestros usuarios de ejemplo siguen a otros y tienen seguidores, necesitamos actualizar las páginas del perfil y la principal para reflejar esto. Empezaremos creando un parcial que muestre las estadísticas de los usuarios seguidos y los seguidores en las páginas mencionadas. A continuación agregaremos un formulario para seguir o dejar de hacerlo, y luego crearemos páginas dedicadas a mostrar “el seguimiento”: usuarios seguidos y seguidores.

Como se observó en la [Sección 12.1.1](#), adoptaremos la convención de Twitter de utilizar “siguiendo” como etiqueta de los usuarios seguidos, como en “50 siguiendo”. Este uso se ve reflejado en la secuencia inicial del bosquejo que se muestra en la [Figura 12.1](#) y del cual se hace un acercamiento en la [Figura 12.10](#).

Las estadísticas de la [Figura 12.10](#) consisten del número de usuarios que el usuario en sesión está siguiendo y el número de seguidores, cada uno de los cuales debería ser un enlace a su respectiva página dedicada a desplegar el detalle. En el [Capítulo 5](#), reemplazamos temporalmente estos enlaces con el texto '#', pero eso fue antes de que tuviéramos más experiencia con las rutas. Aunque esta vez, diferiremos las páginas reales a la [Sección 12.2.3](#), y crearemos las rutas ahora, como se muestra en el [Listado 12.15](#). Este código utiliza el método `:member` dentro de un *bloque resources*, que no hemos visto antes, pero veremos si usted puede adivinar lo que éste hace.

Listado 12.15: Agregando acciones `following` y `followers` al controlador de usuarios.

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help'      => 'static_pages#help'
  get 'about'     => 'static_pages#about'
  get 'contact'   => 'static_pages#contact'
  get 'signup'    => 'users#new'
  get 'login'     => 'sessions#new'
  post 'login'    => 'sessions#create'
  delete 'logout'  => 'sessions#destroy'
  resources :users do
    member do
      get :following, :followers
    end
  end
  resources :account_activations, only: [:edit]
  resources :password_resets,      only: [:new, :create, :edit, :update]
  resources :microposts,           only: [:create, :destroy]
end
```

Puede ser que usted intuya que las URLs para seguidores y seguidos serán como `/users/1/following` y `/users/1/followers`, y eso es exactamente de lo que se encarga el código del Listado 12.15. Puesto que ambas páginas *mostrarán* datos, el verbo HTTP propio para la petición es GET, por lo que utilizaremos el método `get` para encargarnos de que las URLs respondan apropiadamente. Mientras tanto, el método `member` se encarga de que las rutas respondan a las URLs que contienen el `id` del usuario. La otra posibilidad, `collection`, funciona sin el `id`, de modo que

```
resources :users do
  collection do
    get :tigers
  end
end
```

respondería a la URL `/users/tigers` (probablemente para mostrar todos los tigres de nuestra aplicación).⁷

⁷Para más detalle acerca de las opciones de ruteo, vea el artículo de la Guía de Rails llamada “Rails Routing

Petición HTTP	URL	Acción	Ruta nombrada
GET	/users/1/following	<code>following</code>	<code>following_user_path(1)</code>
GET	/users/1/followers	<code>followers</code>	<code>followers_user_path(1)</code>

Tabla 12.2: Rutas RESTful proporcionadas por las reglas personalizadas del recurso del [Listado 12.15](#).

Una tabla de rutas generada por el [Listado 12.15](#) se muestra en la [Tabla 12.2](#). Observe las rutas nombradas para las páginas de usuarios seguidos y seguidores, las cuales pondremos a trabajar muy pronto.

Con las rutas definidas, estamos en posición de definir el parcial de estadísticas, el cual involucra un par de enlaces dentro de un `div`, como se muestra en el [Listado 12.16](#).

Listado 12.16: Un parcial para mostrar las estadísticas de los seguidores.

`app/views/shared/_stats.html.erb`

```
<% @user ||= current_user %>


<strong id="following" class="stat">
        <%= @user.following.count %>
    </strong>
    following


    <strong id="followers" class="stat">
        <%= @user.followers.count %>
    </strong>
    followers


```

Puesto que estaremos incluyendo las estadísticas tanto en las páginas que muestran al usuario como en la página principal, la primer línea del [Listado 12.16](#) elige la correcta usando

from the Outside In”.

```
<% @user ||= current_user %>
```

Como revisamos en el Recuadro 8.1, esto no hace nada cuando `@user` es diferente de `nil` (como en la página del perfil), pero cuando lo es (como en la página principal) establece como valor de la variable `@user` el usuario actual. Observe también que el conteo de seguidos / seguidores está calculado mediante las asociaciones usando

```
@user.following.count
```

y

```
@user.followers.count
```

Compare esto con el conteo de micromensajes del Listado 11.23, donde escribimos

```
@user.microposts.count
```

para contar los micromensajes. Como en ese caso, Rails calcula el conteo directamente en la base de datos por eficiencia.

Un detalle final que vale la pena notar es la presencia de id's CSS en algunos elementos, como en

```
<strong id="following" class="stat">
...
</strong>
```

Esto se aprovecha en la implementación de Ajax de la Sección 12.2.5, la cual identifica los elementos de la página utilizando sus `id`'s únicos.

Con el parcial a mano, incluir las estadísticas en la página principal es fácil, como se muestra en el Listado 12.17.

Listado 12.17: Agregando las estadísticas de los seguidores a la página principal.

app/views/static_pages/home.html.erb

```
<% if logged_in? %>
  <div class="row">
    <aside class="col-md-4">
      <section class="user_info">
        <%= render 'shared/user_info' %>
      </section>
      <section class="stats">
        <%= render 'shared/stats' %>
      </section>
      <section class="micropost_form">
        <%= render 'shared/micropost_form' %>
      </section>
    </aside>
    <div class="col-md-8">
      <h3>Micropost Feed</h3>
      <%= render 'shared/feed' %>
    </div>
  </div>
<% else %>
  .
  .
  .
<% end %>
```

Para dar estilo a las estadísticas, agregaremos algo de SCSS, como se muestra en el [Listado 12.18](#) (el cual contiene todo el código de estilo necesario en este capítulo). La página de inicio resultante se muestra en la [Figura 12.11](#).

Listado 12.18: SCSS para la barra lateral de la página principal.

app/assets/stylesheets/custom.css.scss

```
.
.
.
/*
 sidebar */
.
.
.
.gravatar {
  float: left;
  margin-right: 10px;
```

```
}

.gravatar_edit {
  margin-top: 15px;
}

.stats {
  overflow: auto;
  margin-top: 0;
  padding: 0;
  a {
    float: left;
    padding: 0 10px;
    border-left: 1px solid $gray-lighter;
    color: gray;
    &:first-child {
      padding-left: 0;
      border: 0;
    }
    &:hover {
      text-decoration: none;
      color: blue;
    }
  }
  strong {
    display: block;
  }
}

.user_avatars {
  overflow: auto;
  margin-top: 10px;
  .gravatar {
    margin: 1px 1px;
  }
  a {
    padding: 0;
  }
}

.users.follow {
  padding: 0;
}

/* forms */
.
```

En un momento, desplegaremos el parcial de estadísticas en la página de

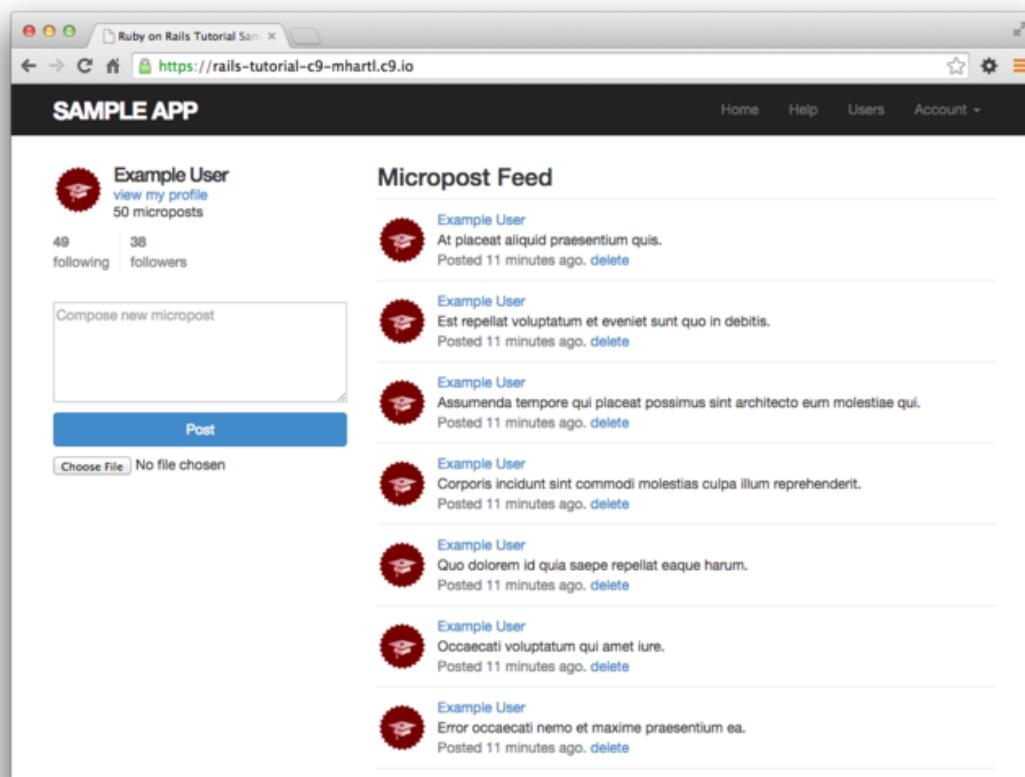


Figura 12.11: La página principal con las estadísticas de seguimiento.

perfil, pero primero crearemos un parcial para el botón de seguir / dejar de seguir, como se muestra en el Listado 12.19.

Listado 12.19: Un parcial para el formulario seguir / dejar de seguir.

app/views/users/_follow_form.html.erb

```
<% unless current_user?(@user) %>
  <div id="follow_form">
    <% if current_user.following?(@user) %>
      <%= render 'unfollow' %>
    <% else %>
      <%= render 'follow' %>
    <% end %>
  </div>
<% end %>
```

Esto no hace más que diferir el trabajo real a los parciales **follow** y **unfollow**, los cuales necesitan nuevas rutas para el recurso **Relationships**, el cual sigue el ejemplo del recurso **Microposts** (Listado 11.29), como vemos en el Listado 12.20.

Listado 12.20: Agregando las rutas para las relaciones del usuario.

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get 'help' => 'static_pages#help'
  get 'about' => 'static_pages#about'
  get 'contact' => 'static_pages#contact'
  get 'signup' => 'users#new'
  get 'login' => 'sessions#new'
  post 'login' => 'sessions#create'
  delete 'logout' => 'sessions#destroy'
  resources :users do
    member do
      get :following, :followers
    end
  end
  resources :account_activations, only: [:edit]
  resources :password_resets, only: [:new, :create, :edit, :update]
  resources :microposts, only: [:create, :destroy]
  resources :relationships, only: [:create, :destroy]
end
```

Los parciales para seguir / dejar de seguir se muestran en los Listados 12.21 y 12.22.

Listado 12.21: Un formulario para seguir a un usuario.

app/views/users/_follow.html.erb

```
<%= form_for(current_user.active_relationshipships.build) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

Listado 12.22: Un formulario para dejar de seguir a un usuario.

app/views/users/_unfollow.html.erb

```
<%= form_for(current_user.active_relationshipships.find_by(followed_id: @user.id),
             html: { method: :delete }) do |f| %>
  <%= f.submit "Unfollow", class: "btn" %>
<% end %>
```

Ambos formularios utilizan `form_for` para manipular el objeto del modelo **Relationship**; la principal diferencia entre ellos es que el Listado 12.21 crea una *nueva* relación, mientras que el Listado 12.22 destruye la relación existente. Naturalmente, el primero envía una petición POST al controlador Relationships para **crear** una relación, mientras que el segundo envía una petición DELETE para **destruir** una relación. (Escribiremos estas acciones en la Sección 12.2.4.) Finalmente, usted notará que el formulario de seguimiento no contiene nada más que el botón, pero aún así necesita enviar el `followed_id` al controlador. Logramos esto con el método `hidden_field_tag` del Listado 12.21, que produce un código HTML de la forma

```
<input id="followed_id" name="followed_id" type="hidden" value="3" />
```

Como vimos en la Sección 10.2.4 (Listado 10.50), la etiqueta oculta `input` almacena información relevante en la página sin mostrarla en el navegador.

Ahora podemos incluir el formulario y las estadísticas en la página de perfil del usuario con sólo desplegar los parciales, como se observa en el [Listado 12.23](#). Los perfiles con botones para seguir o dejar de seguir, respectivamente, aparecen en las Figuras 12.12 y 12.13.

Listado 12.23: Agregando el formulario para seguir y las estadísticas de seguimiento a la página de perfil del usuario.

app/views/users/show.html.erb

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


<aside class="col-md-4">
    <section>
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
    <section class="stats">
      <%= render 'shared/stats' %>
    </section>
  </aside>
  <div class="col-md-8">
    <%= render 'follow_form' if logged_in? %>
    <% if @user.microposts.any? %>
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
  </div>
</div>


```

Haremos que estos botones funcionen pronto—de hecho, lo haremos de dos formas, la forma estándar ([Sección 12.2.4](#)) y usando Ajax ([Sección 12.2.5](#))—pero primero terminaremos la interfaz HTML creando las páginas de seguidos y seguidores.

12.2.3 Páginas de seguidos y seguidores

Las páginas para mostrar los usuarios seguidos y seguidores se asemejan a un híbrido de la página de perfil del usuario y la página del listado de usuarios

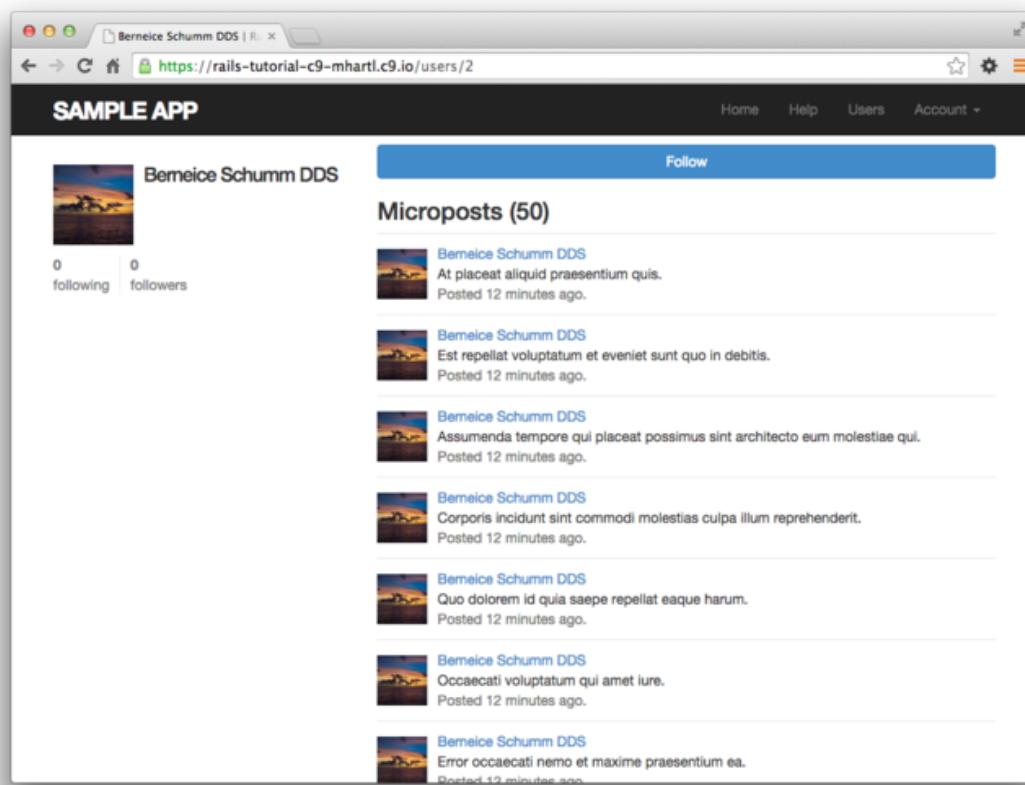


Figura 12.12: Un perfil de usuario con el botón de seguimiento ([/users/2](#)).

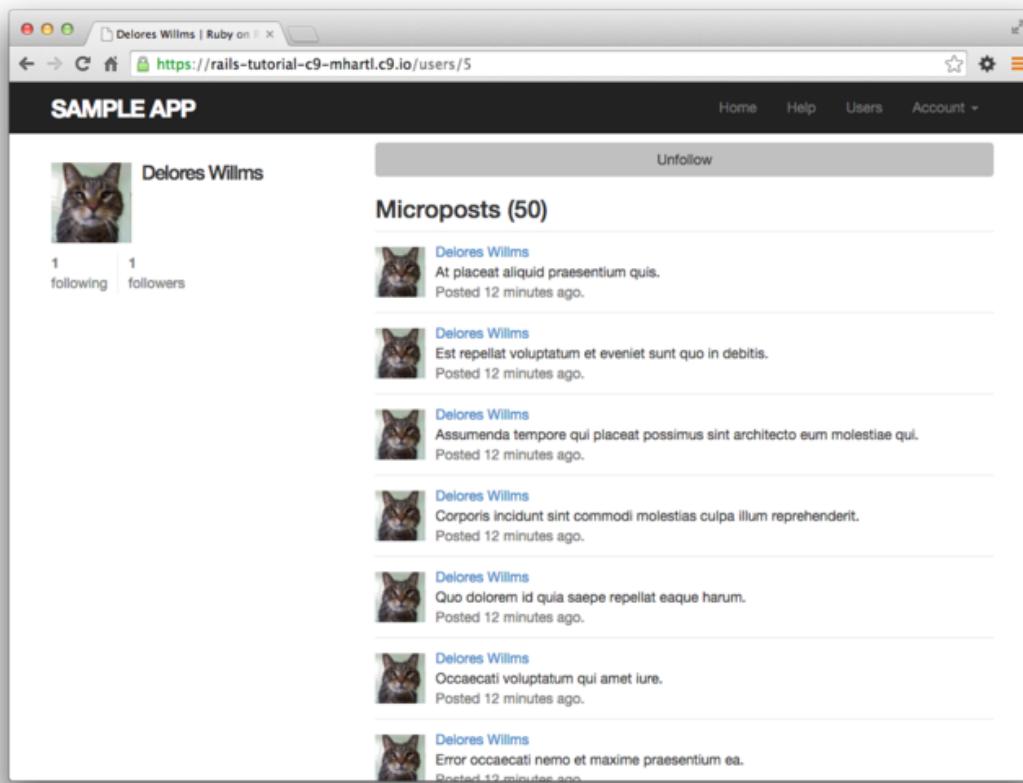


Figura 12.13: Un perfil de usuario con el botón para dejar de seguir ([/users/5](#)).

(Sección 9.3.1), con una barra lateral de información de usuario (incluyendo las estadísticas de seguimiento) y una lista de usuarios. Adicionalmente, incluiremos una serie de pequeñas imágenes con enlaces a las páginas de perfil de los usuarios en la barra lateral. Los bosquejos que cubren estos requerimientos se muestran en las Figuras 12.14 (seguidos) y 12.15 (seguidores).

Nuestro primer paso es hacer que los enlaces a los seguidos y seguidores funcionen. Siguiendo la filosofía de Twitter, haremos que ambas páginas requieran que el usuario haya iniciado sesión. Como con la mayoría de los ejemplos anteriores de control de acceso, escribiremos las pruebas primero, como se muestra en el Listado 12.24.

Listado 12.24: Pruebas para la autorización de las páginas de seguidores y seguidos. ROJO

```
test/controllers/users_controller_test.rb

require 'test_helper'

class UsersControllerTest < ActionController::TestCase

  def setup
    @user = users(:michael)
    @other_user = users(:archer)
  end

  .

  .

  test "should redirect following when not logged in" do
    get :following, id: @user
    assert_redirected_to login_url
  end

  test "should redirect followers when not logged in" do
    get :followers, id: @user
    assert_redirected_to login_url
  end
end
```

La única parte truculenta de la implementación es darnos cuenta de que necesitamos agregar dos nuevas acciones al controlador de usuarios. Basándonos en las rutas definidas en el Listado 12.15, necesitamos nombrarlas **following** y **followers**. Cada acción necesita asignar un título, buscar al usuario, recu-

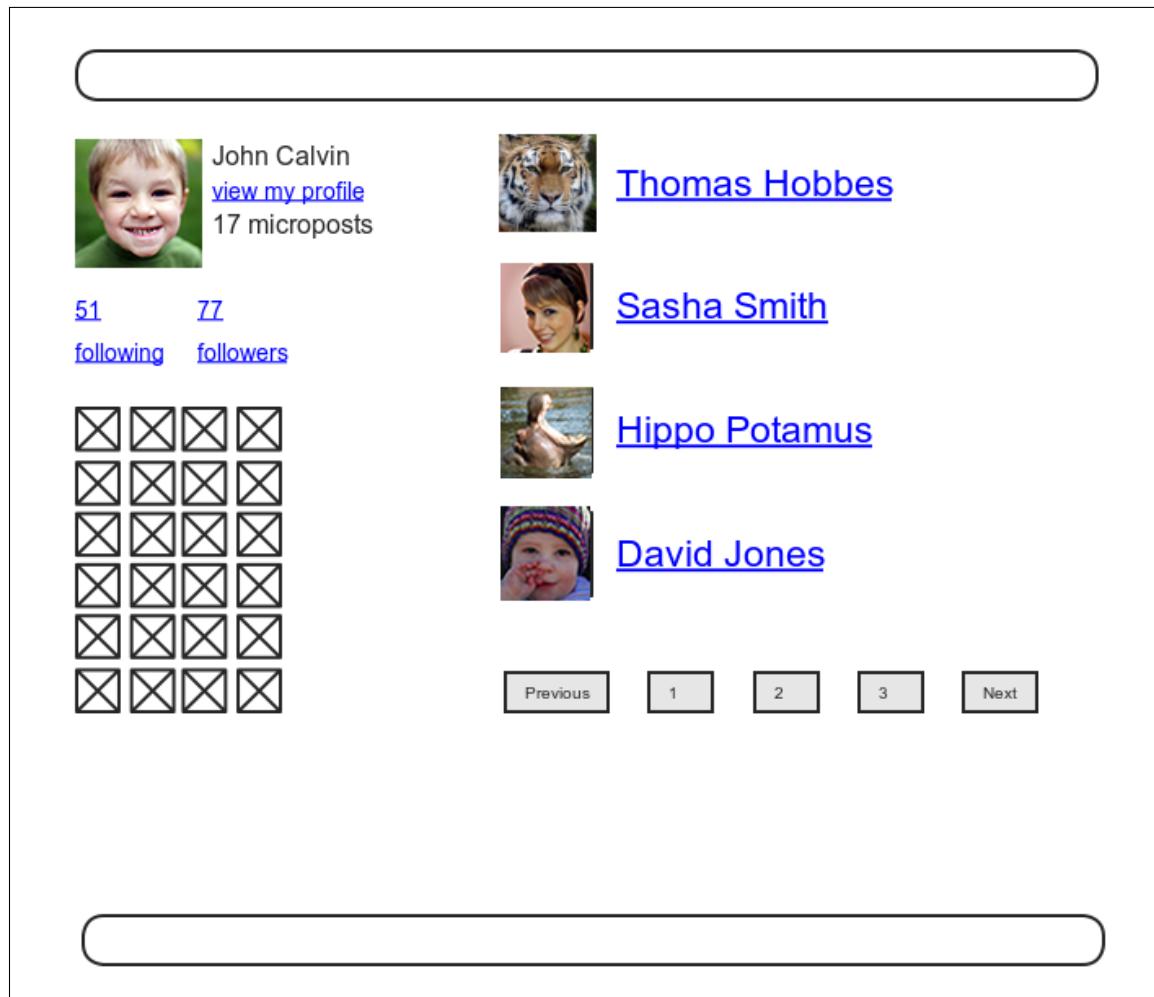


Figura 12.14: Un bosquejo de la página de usuario seguido.

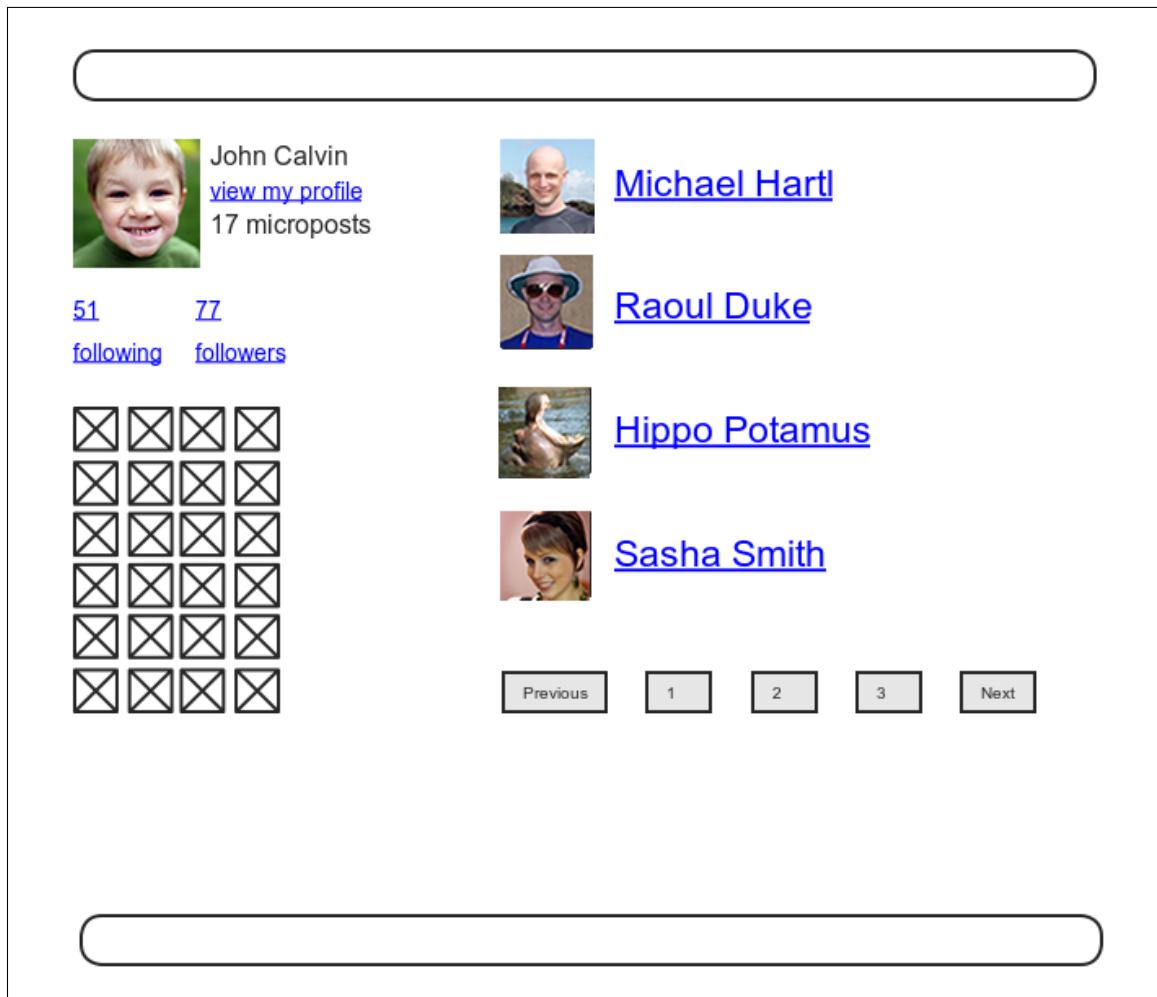


Figura 12.15: Un bosquejo de la página de usuario seguidor.

perar ya sea `@user.following` o `@user.followers` (de forma paginada), y luego desplegar la página. El resultado aparece en el [Listado 12.25](#).

Listado 12.25: Las acciones `following` y `followers`.

`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update, :destroy,
                                         :following, :followers]
  .
  .
  .
  def following
    @title = "Following"
    @user = User.find(params[:id])
    @users = @user.following.paginate(page: params[:page])
    render 'show_follow'
  end

  def followers
    @title = "Followers"
    @user = User.find(params[:id])
    @users = @user.followers.paginate(page: params[:page])
    render 'show_follow'
  end

  private
  .
  .
  .
end
```

Como hemos visto a lo largo de este tutorial, la convención de Rails usual es mostrar implícitamente la plantilla correspondiente a una acción, tal como sucede con `show.html.erb` al final de la acción `show`. Por el contrario, ambas acciones del [Listado 12.25](#) realizan una llamada *explícita* a `render`, en este caso mostrando una vista llamada `show_follow`, la cual debemos crear. El motivo de tener una vista común es que el ERb es casi idéntico en ambos casos, y el [Listado 12.26](#) cubre los dos.

Listado 12.26: La vista `show_follow` utilizada para mostrar seguidos y seguidores.

`app/views/users/show_follow.html.erb`

```

<%
      :title  @title  %>
div class="row"
  aside class="col-md-4"
    section class="user_info"
      <%=          @user %>
      h1 <%= @user.      %> h1
      span <%=      "view my profile"  @user %>  span
      span b           b <%= @user.      .      %>  span
    section
    section class="stats"
      <%=        'shared/stats' %>
      <% if @users.      %>
        div class="user_avatars"
          <% @users.      do | | %>
            <%=                                size 30      %>
          <% end %>
        div
      <% end %>
    section
  aside
div class="col-md-8"
  h3 <%= @title %> h3
  <% if @users.      %>
    ul class="users_follow"
      <%=          @users %>
    ul
    <%=      %>
  <% end %>
  div
div

```

Las acciones del Listado 12.25 muestran la vista del Listado 12.26 en dos contextos, “seguidos” y “seguidores”, con los resultados que se muestran en las Figuras 12.16 y 12.17. Observe que en ninguna parte del código anterior se utiliza al usuario actual, por lo que los mismos enlaces funcionan para otros usuarios, como se muestra en la Figura 12.18.

Ahora que hemos trabajado en las páginas de seguidores y seguidos, escribiremos un par de pruebas de integración pequeñas para verificar su comportamiento. Estas pruebas están diseñadas para una validación simple, no para que sean exhaustivas. De hecho, como observamos en la Sección 5.3.4, las pruebas exhaustivas de cosas como estructuras HTML que son propensas a cambios, son contraproducentes. Nuestro plan en el caso de las páginas de seguidores / seguidos es verificar que el número se muestre correctamente y que los enlaces

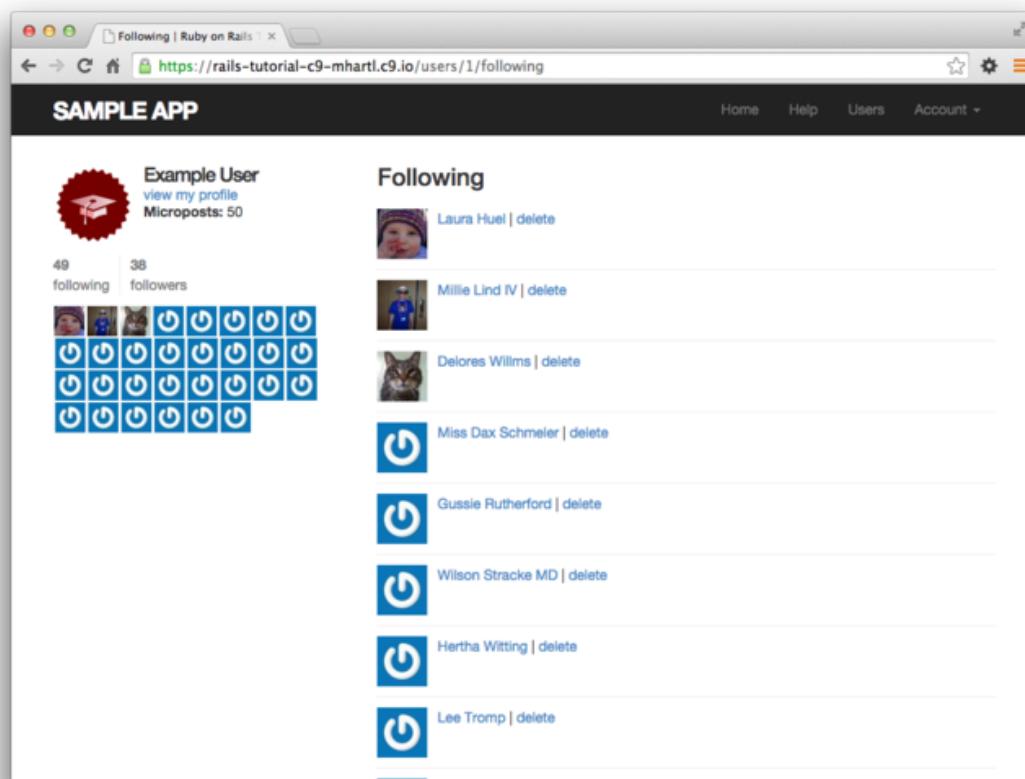


Figura 12.16: Mostrando los usuarios que el usuario actual está siguiendo.

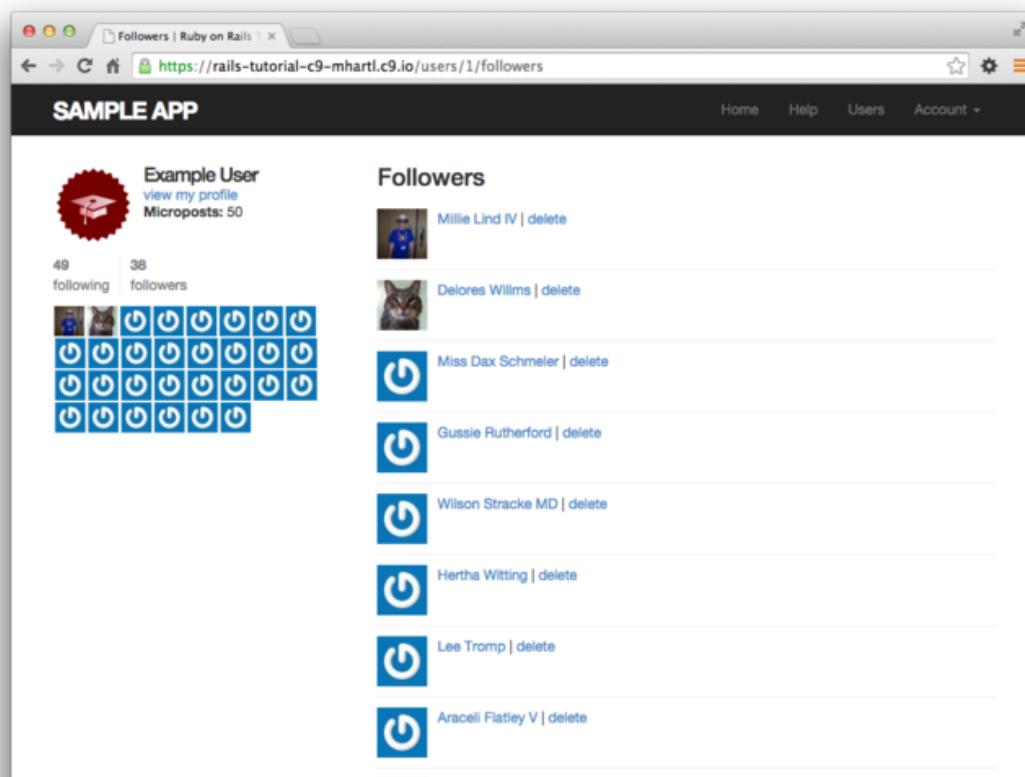


Figura 12.17: Mostrando los seguidores del usuario actual.

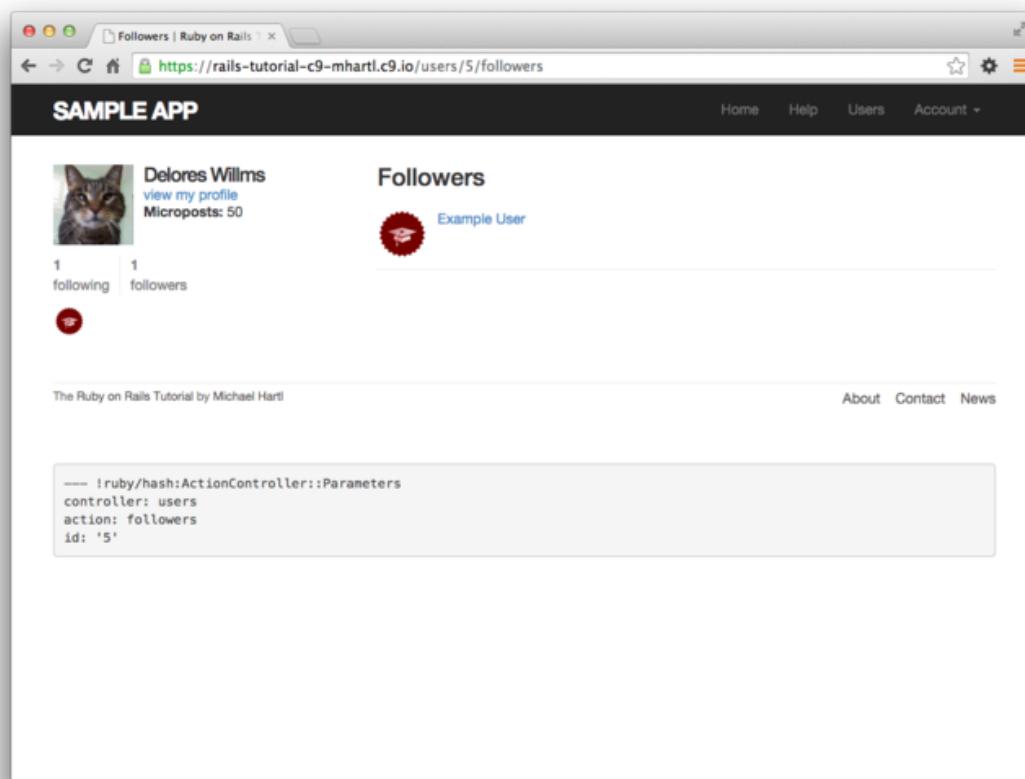


Figura 12.18: Mostrando los seguidores de un usuario diferente.

con las URLs correctas aparezcan en la página.

Para empezar, generaremos la prueba de integración como es usual:

```
$ rails generate integration_test following
  invoke  test_unit
  create    test/integration/following_test.rb
```

A continuación, necesitamos preparar algunos datos de prueba, lo cual podemos hacer agregando algunas relaciones a los fixtures para crear relaciones de seguidos / seguidores. Recuerde de la [Sección 11.2.3](#) que podemos utilizar código como

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

para asociar un micromensaje con un usuario dado. En particular, podemos escribir

```
user: michael
```

en vez de

```
user_id: 1
```

Aplicando esta idea a los fixtures de relaciones obtenemos las asociaciones del [Listado 12.27](#).

Listado 12.27: Fixtures de relaciones para utilizar en las pruebas de seguidos / seguidores.

```
test/fixtures/relationships.yml
```

```

one:
  follower: michael
  followed: lana

two:
  follower: michael
  followed: mallory

three:
  follower: lana
  followed: michael

four:
  follower: archer
  followed: michael

```

El fixtures del [Listado 12.27](#) primero se encarga de que Michael siga a Lana y Mallory, y luego se encarga de que Michael sea seguido por Lana y Archer. Para probar que el conteo es correcto, podemos usar el mismo método **assert_match** que utilizamos en el [Listado 11.27](#) para probar que se muestra el número de micromensajes en la página de perfil del usuario. Al agregar las afirmaciones para los enlaces correctos, obtenemos las pruebas que se muestran en el [Listado 12.28](#).

Listado 12.28: Pruebas para las páginas de seguidos / seguidores. [VERDE](#)

```

test/integration/following_test.rb

require 'test_helper'

class FollowingTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
    log_in_as(@user)
  end

  test "following page" do
    get following_user_path(@user)
    assert_not @user.following.empty?
    assert_match @user.following.count.to_s, response.body
    @user.following.each do |user|
      assert_select "a[href=?]", user_path(user)
    end
  end
end

```

```
test "followers page" do
  get followers_user_path(@user)
  assert_not @user.followers.empty?
  assert_match @user.followers.count.to_s, response.body
  @user.followers.each do |user|
    assert_select "a[href=?]", user_path(user)
  end
end
end
```

En el [Listado 12.28](#), observe que utilizamos la afirmación

```
assert_not @user.following.empty?
```

la cual se incluye para asegurarnos de que

```
@user.following.each do |user|
  assert_select "a[href=?]", user_path(user)
end
```

no es una **verdad vacía** (y análogamente hacemos para **followers**).

El conjunto de pruebas debería estar en **VERDE**:

Listado 12.29: VERDE

```
$ bundle exec rake test
```

12.2.4 Un botón de seguimiento funcionando de forma estándar

Ahora que nuestras vistas están en orden, es hora de hacer que los botones de seguir / dejar de seguir funcionen. Como el seguimiento involucra la creación y destrucción de relaciones, necesitamos un controlador **Relationships**, el cual generamos como es usual

```
$ rails generate controller Relationships
```

Como veremos en el [Listado 12.31](#), aplicar el control de acceso en las acciones del controlador de relaciones no es muy importante, pero podemos seguir nuestra práctica previa de aplicar el modelo de seguridad tan pronto como sea posible. En particular, verificaremos que los intentos de accesar las acciones del controlador de relaciones, requieran que el usuario esté en sesión (y por tanto, redireccionen a la página de inicio de sesión), y que al mismo tiempo no cambie el contador de relaciones, como se muestra en el [Listado 12.30](#).

Listado 12.30: Pruebas de control de acceso básicas para las relaciones. **ROJO**
test/controllers/relationships_controller_test.rb

```
require 'test_helper'

class RelationshipsControllerTest < ActionController::TestCase

  test "create should require logged-in user" do
    assert_no_difference 'Relationship.count' do
      post :create
    end
    assert_redirected_to login_url
  end

  test "destroy should require logged-in user" do
    assert_no_difference 'Relationship.count' do
      delete :destroy, id: relationships(:one)
    end
    assert_redirected_to login_url
  end
end
```

Podemos hacer que las pruebas del [Listado 12.30](#) pasen agregando el filtro previo **logged_in_user** ([Listado 12.31](#)).

Listado 12.31: Control de acceso para las relaciones. **VERDE**

app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user
```

```

def create
end

def destroy
end
end

```

Para hacer que los botones de seguir y dejar de seguir funcionen, todo lo que necesitamos hacer es encontrar el usuario asociado con el `followed_id` en el formulario correspondiente (es decir, Listado 12.21 o Listado 12.22), y luego utilizar el método apropiado `follow` o `unfollow` del Listado 12.10. La implementación completa se muestra en el Listado 12.32.

Listado 12.32: El controlador Relationships.

app/controllers/relationships_controller.rb

```

class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
    user = User.find(params[:followed_id])
    current_user.follow(user)
    redirect_to user
  end

  def destroy
    user = Relationship.find(params[:id]).followed
    current_user.unfollow(user)
    redirect_to user
  end
end

```

Podemos ver del Listado 12.32 porqué el problema de seguridad que mencionamos anteriormente es menor: si un usuario que no ha iniciado sesión tuviera acceso a la acción directamente (por ejemplo, digamos que utiliza una herramienta de línea de comandos como `curl`), `current_user` sería `nil`, y en ambos casos la segunda línea de la acción arrojaría una excepción, resultando en un error que no daña la aplicación o sus datos. Sin embargo, es mejor no depender de esto, por lo que hemos dado un paso adelante y hemos agregado la capa de seguridad adicional.

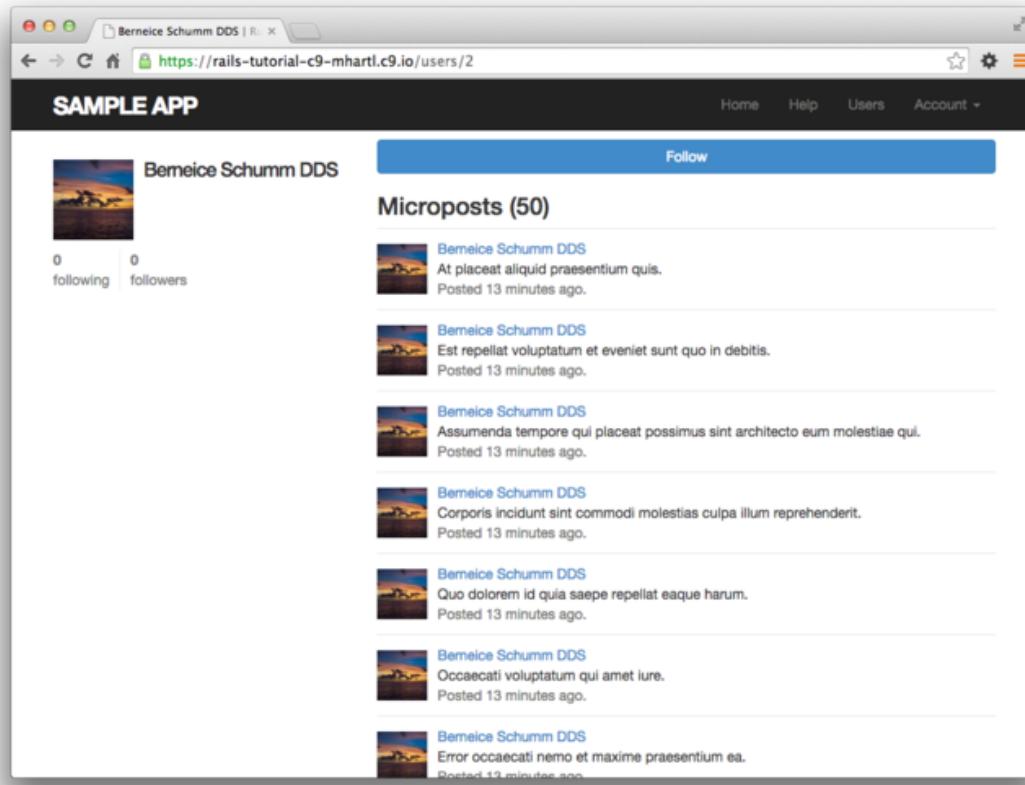


Figura 12.19: Un usuario no seguido.

Con esto, la funcionalidad principal de seguir / dejar de seguir está completa, y cualquier usuario puede seguir o dejar de seguir a cualquier otro usuario, como puede verificar presionando los botones correspondientes en su navegador. (Escribiremos pruebas de integración para verificar este comportamiento en la Sección 12.2.6.) El resultado de seguir al usuario #2 se muestra en las Figuras 12.19 y 12.20.

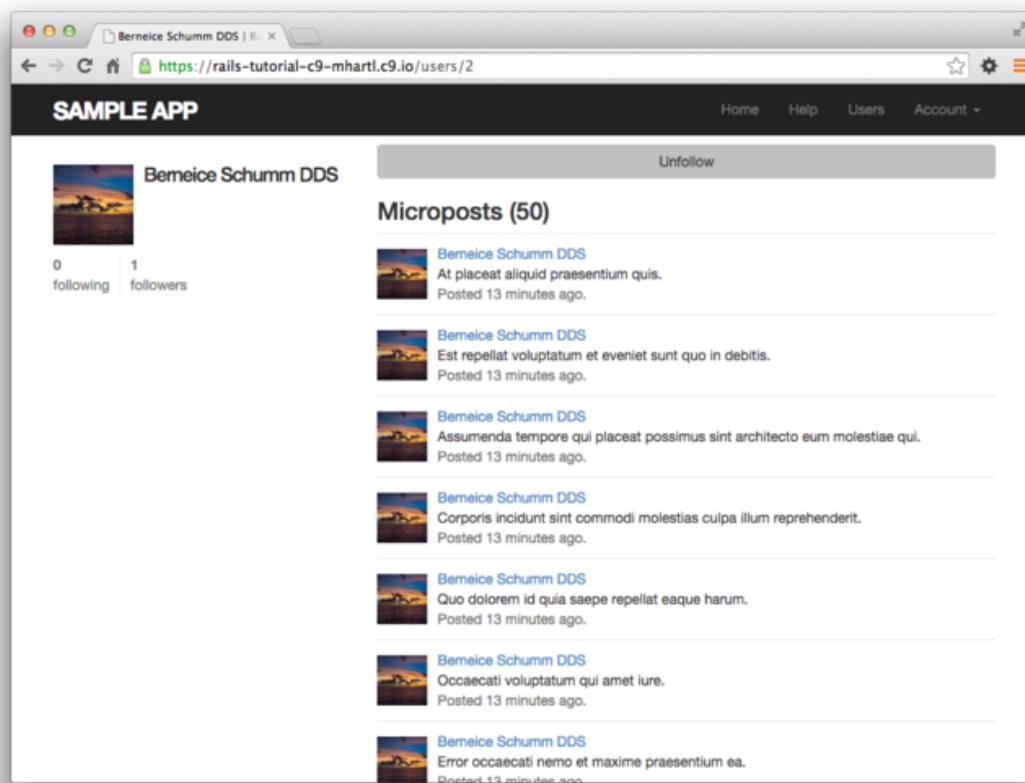


Figura 12.20: El resultado de seguir a un usuario no seguido.

12.2.5 Un botón de seguimiento funcionando con Ajax

Aunque nuestra implementación de seguir usuarios está completa en este momento, podemos pulirla un poco antes de empezar a trabajar en el status de avance. Puede haber notado que en la Sección 12.2.4 las acciones `create` y `destroy` en el controlador de relaciones simplemente redireccionan *de regreso* al perfil original. En otras palabras, un usuario empieza en la página de perfil de otro usuario, sigue a ese otro usuario, y luego es inmediatamente redirigido a la página original. Es razonable preguntar porqué el usuario necesita abandonar esa página en absoluto.

Este es exactamente el problema que se resuelve con *Ajax*, el cual permite que las páginas web envíen peticiones asíncronamente al servidor sin abandonar la página.⁸ Como agregar Ajax a los formularios web es una práctica común, Rails facilita su implementación. De hecho, actualizar los parciales con los formularios de seguir / dejar de seguir es trivial: sólo cambie

```
form_for
```

por

```
form_for ..., remote: true
```

y Rails [automágicamente](#) utilizará Ajax. Los parciales actualizados se muestran en los Listados 12.33 y 12.34.

Listado 12.33: Un formulario para seguir a un usuario utilizando Ajax.

app/views/users/_follow.html.erb

```
<%= form_for(current_user.active_relationships.build, remote: true) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

⁸Como nominalmente es un acrónimo de Javascript Asíncrona y XML (en inglés *asynchronous JavaScript and XML*), Ajax algunas veces es escrito como “AJAX”, aún cuando el [artículo original de Ajax](#) lo escribe como “Ajax” en todas partes.

Listado 12.34: Un formulario para dejar de seguir a un usuario utilizando Ajax.

```
app/views/users/_unfollow.html.erb

<%= form_for(current_user.active_relationships.find_by(followed_id: @user.id),
              html: { method: :delete },
              remote: true) do |f| %>
  <%= f.submit "Unfollow", class: "btn" %>
<% end %>
```

El código HTML generado por este ERb no es particularmente relevante, pero por curiosidad, podemos echar un vistazo a la vista generada (los detalles podrían variar):

```
<form action="/relationships/117" class="edit_relationship" data-remote="true"
      id="edit_relationship_117" method="post">
  .
  .
  .
</form>
```

Esto asigna la variable **data-remote="true"** dentro de la etiqueta **form**, lo que le indica a Rails que permita que el formulario sea manipulado por JavaScript. Al utilizar una propiedad HTML simple en vez de insertar el código JavaScript completo (como en versiones de Rails anteriores), Rails sigue la filosofía de *JavaScript no obstructiva*.

Habiendo actualizado el formulario, ahora necesitamos encargarnos de que el controlador de relaciones responda a las peticiones Ajax. Podemos realizar esto utilizando el método **respond_to**, respondiendo adecuadamente dependiendo del tipo de petición. El patrón general se muestra como éste:

```
respond_to do |format|
  format.html { redirect_to user }
  format.js
end
```

La sintaxis es potencialmente confusa, y es importante entender que en el código anterior sólo *una* de las líneas será ejecutada. (En este sentido, **respond_to**

es más como una sentencia if-then-else que una serie de líneas secuenciales.) Adaptar el controlador de relaciones para que responda a Ajax implica agregar `respond_to` como se indicó anteriormente en las acciones `create` y `destroy` del Listado 12.32. El resultado se muestra en el Listado 12.35. Observe el cambio de la variable local `user`, a la variable de instancia `@user`; en el Listado 12.32 no había necesidad de una variable de instancia, pero ahora tal variable es necesaria para utilizarla en los Listados 12.33 y 12.34.

Listado 12.35: Respondiendo a las peticiones Ajax en el controlador Relationships.

app/controllers/relationships_controller.rb

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
    @user = User.find(params[:followed_id])
    current_user.follow(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end

  def destroy
    @user = Relationship.find(params[:id]).followed
    current_user.unfollow(@user)
    respond_to do |format|
      format.html { redirect_to @user }
      format.js
    end
  end
end
```

Las acciones del Listado 12.35 degradan con gracia, lo que significa que funcionan bien en navegadores que tienen JavaScript deshabilitado (aunque se requiere un poco de configuración, como se observa en el Listado 12.36).

Listado 12.36: La configuración necesaria para la degradación del envío del formulario.

config/application.rb

```

require File.expand_path('../boot', __FILE__)

#
#
#
module SampleApp
  class Application < Rails::Application
    #
    #
    #

    # Include the authenticity token in remote forms.
    config.action_view.embed_authenticity_token_in_remote_forms = true
  end
end

```

Por otra parte, aún debemos responder adecuadamente cuando JavaScript está habilitado. En el caso de una petición Ajax, Rails automáticamente invoca al archivo *JavaScript Ruby embebido (.js.erb)* que lleva el mismo nombre que la acción, es decir, **create.js.erb** o **destroy.js.erb**. Como puede suponer, tales archivos nos permiten mezclar JavaScript con Ruby embebido para realizar acciones en la página actual. Son estos archivos los que necesitamos crear y editar con la finalidad de actualizar la página de perfil del usuario al ser seguido o dejar de serlo.

Dentro de un archivo JS-ERb, Rails automáticamente proporciona los auxiliares de **jQuery** JavaScript para manipular la página utilizando el **Modelo de objetos del documento (DOM, por sus siglas en inglés Document Object Model)**. La biblioteca jQuery (que vimos brevemente en la [Sección 11.4.2](#)) proporciona una gran cantidad de métodos para manipular el DOM, pero aquí sólo necesitaremos dos. Primero, necesitaremos conocer la sintaxis del signo de dólares para tener acceso a un elemento del DOM basándonos en su id único de CSS. Por ejemplo, para manipular el elemento **follow_form**, utilizaremos la sintaxis

```

$("#follow_form")

```

(Recuerde del [Listado 12.19](#) que esto es un **div** que envuelve al formulario, no es el formulario como tal.) Esta sintaxis, inspirada por CSS, utiliza el símbolo **#** para indicar que se trata de un id de CSS. Como puede suponer, jQuery, de igual forma que CSS, utiliza un punto **.** para manipular las clases CSS.

El segundo método que necesitaremos es `html`, el cual actualiza el HTML de un elemento relevante, con el contenido que se le pasa como argumento. Por ejemplo, para reemplazar el formulario completo de seguimiento con la cadena "`foobar`", escribiríamos

```
$("#follow_form").html("foobar")
```

A diferencia de los archivos JavaScript planos, los archivos JS-ERb nos permiten utilizar Ruby embebido, el cual aplicamos en el archivo `create.js.erb` para actualizar el formulario de seguimiento con el parcial `unfollow` (que es lo que debemos mostrar luego de un seguimiento exitoso) y actualizar el contador de seguidores. El resultado se muestra en el [Listado 12.37](#). Esto ocupa el método `escape_javascript`, el cual es necesario para escapar el resultado cuando insertamos HTML en un archivo JavaScript.

Listado 12.37: El JavaScript con Ruby embebido para crear una relación de seguimiento.

`app/views/relationships/create.js.erb`

```
$("#follow_form").html("<%= escape_javascript(render('users/unfollow')) %>");  
$("#followers").html('<%= @user.followers.count %>');
```

Observe la presencia de los caracteres *punto y coma* al final de la línea, que son característicos de lenguajes cuya sintaxis desciende de [ALGOL](#).

El archivo `destroy.js.erb` es análogo ([Listado 12.38](#)).

Listado 12.38: El JavaScript con Ruby embebido para destruir una relación de seguimiento.

`app/views/relationships/destroy.js.erb`

```
$("#follow_form").html("<%= escape_javascript(render('users/follow')) %>");  
$("#followers").html('<%= @user.followers.count %>');
```

Con esto, podría usted navegar a la página de perfil del usuario y verificar que puede seguir y dejar de hacerlo sin que la página se refresque.

12.2.6 Pruebas de seguimiento

Ahora que los botones de seguimiento están funcionando, escribiremos algunas pruebas sencillas para evitar regresiones. Para seguir a un usuario, enviaremos una petición POST a la ruta de relaciones y verificaremos que el número de usuarios seguidos se incrementa en 1:

```
assert_difference '@user.following.count', 1 do
  post relationships_path, followed_id: @other.id
end
```

Esto prueba la implementación estándar, pero probar la versión Ajax es casi idéntico, con `xhr :post` en vez de simplemente `post`:

```
assert_difference '@user.following.count', 1 do
  xhr :post, relationships_path, followed_id: @other.id
end
```

Esto utiliza el método `xhr` (de XMLHttpRequest) para emitir una petición Ajax, lo que causa que el bloque `respond_to` del [Listado 12.35](#) ejecute el método JavaScript adecuado.

La misma estructura paralela aplica para el borrado de usuarios, con `delete` en vez de `post`. Aquí verificamos que el conteo de usuarios seguidos se decremente en 1 e incluye la relación y el id del usuario seguido:

```
assert_difference '@user.following.count', -1 do
  delete relationship_path(relationship),
    relationship: relationship.id
end
```

y

```
assert_difference '@user.following.count', -1 do
  xhr :delete, relationship_path(relationship),
    relationship: relationship.id
end
```

Juntando estos dos casos obtenemos las pruebas del Listado 12.39.

Listado 12.39: Pruebas para los botones de seguir y de dejar de seguir. **VERDE**
test/integration/following_test.rb

```
require 'test_helper'

class FollowingTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
    @other = users(:archer)
    log_in_as(@user)
  end

  .
  .
  .

  test "should follow a user the standard way" do
    assert_difference '@user.following.count', 1 do
      post relationships_path, followed_id: @other.id
    end
  end

  test "should follow a user with Ajax" do
    assert_difference '@user.following.count', 1 do
      xhr :post, relationships_path, followed_id: @other.id
    end
  end

  test "should unfollow a user the standard way" do
    @user.follow(@other)
    relationship = @user.active_relationships.find_by(followed_id: @other.id)
    assert_difference '@user.following.count', -1 do
      delete relationship_path(relationship)
    end
  end

  test "should unfollow a user with Ajax" do
    @user.follow(@other)
    relationship = @user.active_relationships.find_by(followed_id: @other.id)
    assert_difference '@user.following.count', -1 do
      xhr :delete, relationship_path(relationship)
    end
  end
end
```

En este momento, las pruebas deberían estar en **VERDE**:

Listado 12.40: VERDE

```
$ bundle exec rake test
```

12.3 El status de avance

Llegamos ahora a la cumbre de nuestra aplicación de ejemplo: el status de avance de los micromensajes. De forma oportuna, esta sección contiene algo del material más avanzado de todo el tutorial. El status de avance completo se basa en el avance de la [Sección 11.3.3](#) mediante el ensamble de un arreglo de micromensajes de usuarios que son seguidos por el usuario actual, junto con los micromensajes propios del usuario actual. A lo largo de esta sección, procederemos con una serie de implementaciones de status con sofisticación progresiva. Para lograr esto, necesitamos algo de Rails avanzado, Ruby y algunas técnicas de programación SQL.

Debido a la cuesta que estamos por escalar, es importante analizar hacia donde vamos. Un resumen del status de avance final, se muestra en la [Figura 12.5](#), y aparece nuevamente en la [Figura 12.21](#).

12.3.1 Motivación y estrategia

La idea básica detrás del status de avance es simple. La [Figura 12.22](#) muestra una tabla de la base de datos **microposts** de ejemplo y su status de avance resultante. El propósito del status es extraer los micromensajes cuyos **id** de usuario corresponden a los usuarios que son seguidos por el usuario en sesión y los de él mismo, como indican las flechas del diagrama.

Aunque no sabemos todavía cómo implementar el avance, las pruebas son relativamente directas, por lo que (siguiendo las directrices del Recuadro 3.3) las escribiremos primero. La clave es verificar los tres requerimientos para el avance: micromensajes tanto para los usuarios seguidos como para el usuario mismo deben incluirse en el avance, pero un mensaje de un usuario *que no es seguido*, no debe incluirse. Con base en el archivo fixtures de los Listados 9.43

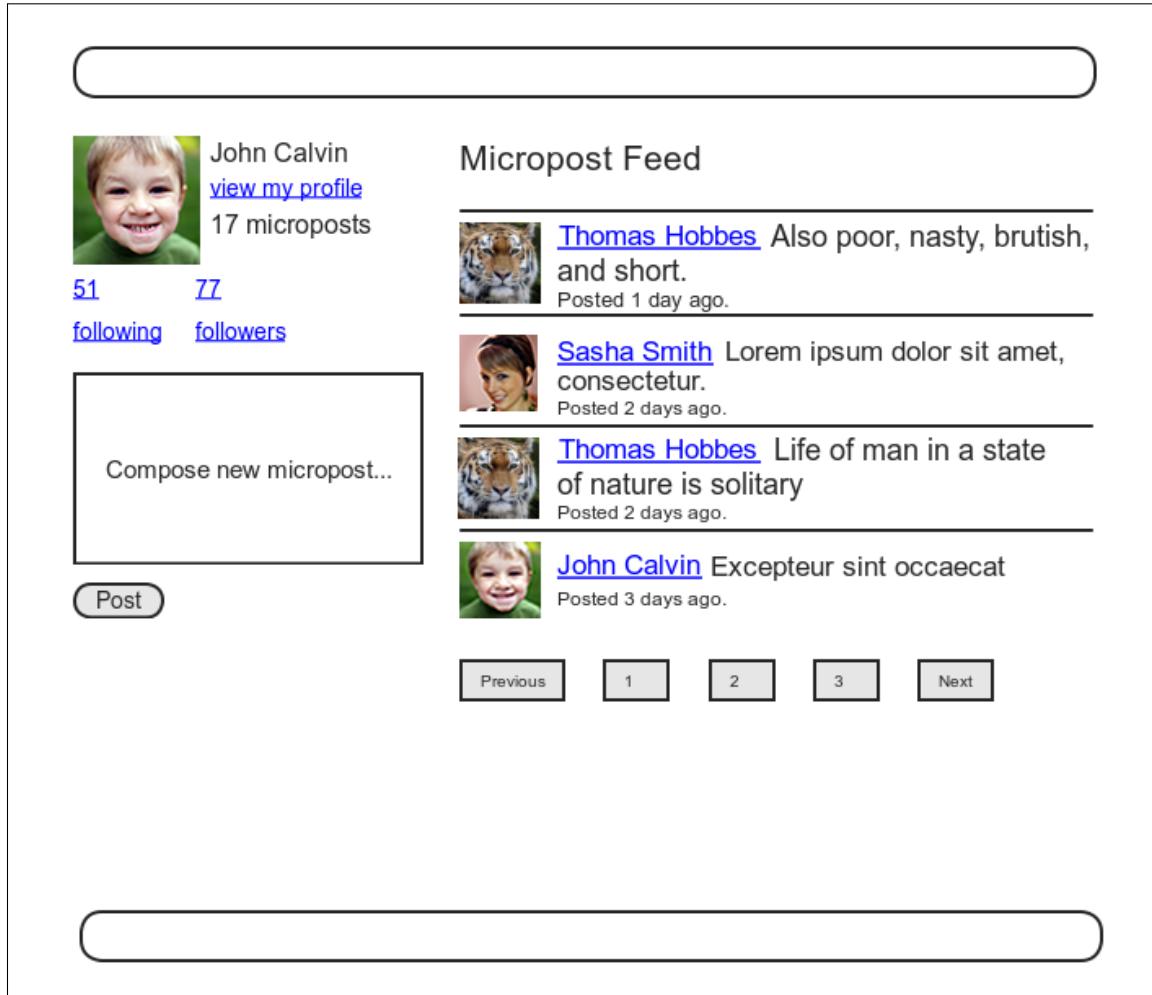


Figura 12.21: Un bosquejo de la página principal del usuario con un status de avance.

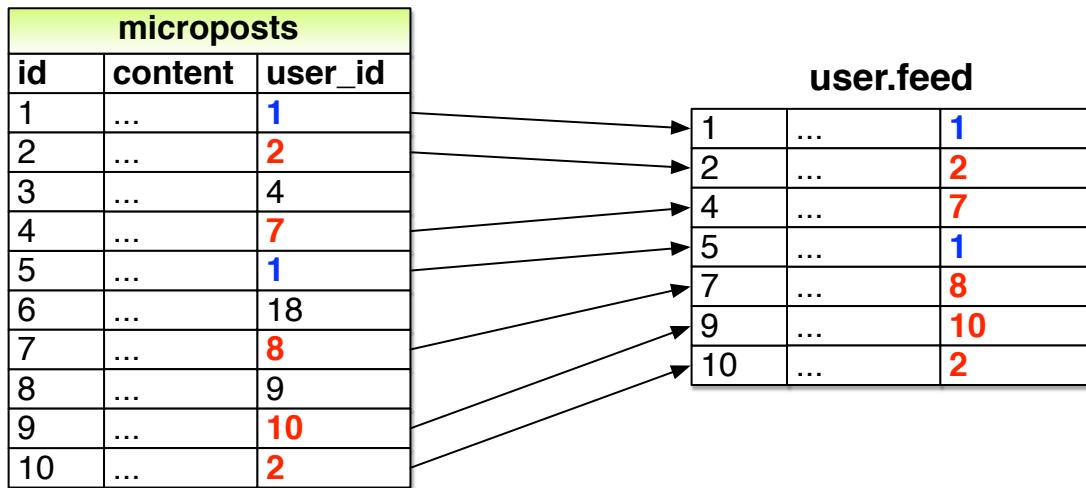


Figura 12.22: El avance de un usuario (id 1) que sigue a los usuarios con ids 2, 7, 8, y 10.

y 11.51, tenemos que Michael debería ver los mensajes de Lana y los suyos propios, pero no los de Archer. Convirtiendo estos requerimientos en afirmaciones y recordando que **feed** está en el modelo **User** (Listado 11.44) obtenemos la versión actualizada de la prueba del modelo **User** que se muestra en el Listado 12.41.

Listado 12.41: Una prueba para el status de avance. ROJO

test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  .
  .
  .

  test "feed should have the right posts" do
    michael = users(:michael)
    archer = users(:archer)
    lana = users(:lana)
    # Posts from followed user
    lana.microposts.each do |post_following|
      assert michael.feed.include?(post_following)
    end
  end
end
```

```

end
# Posts from self
michael.microposts.each do |post_self|
  assert michael.feed.include?(post_self)
end
# Posts from unfollowed user
archer.microposts.each do |post_unfollowed|
  assert_not michael.feed.include?(post_unfollowed)
end
end
end

```

Por supuesto, la implementación actual es sólo un avance prototípico, por lo que la nueva prueba inicialmente está en **ROJO**:

Listado 12.42: ROJO

```
$ bundle exec rake test
```

12.3.2 Una primera implementación del avance

Con los requerimientos de diseño del status de avance capturados en la prueba del [Listado 12.41](#), estamos listos para empezar a escribir el avance. Puesto que la implementación final del avance es más bien intrincada, la construiremos paso a paso. El primer paso es pensar en la clase de consulta que necesitamos. Necesitamos seleccionar todos los micromensajes de la tabla **microposts** cuyos ids correspondan con los usuarios que son seguidos por cierto usuario (o por él mismo). Podríamos escribir esto así:

```

SELECT * FROM microposts
WHERE user_id IN (<list of ids>) OR user_id = <user id>

```

Al escribir este código, intuimos que SQL soporta la palabra reservada **IN** que nos permite determinar la inclusión en un conjunto. (Afortunadamente, así es.)

Recuerde que en el avance prototípico de la [Sección 11.3.3](#), Active Record utiliza el método **where** para ejecutar el tipo de consulta anterior, como se muestra

en el [Listado 11.44](#). Entonces, nuestra consulta era muy simple; sólo queríamos obtener los micromensajes cuyo id de usuario correspondía con el usuario actual:

```
Micropost.where("user_id = ?", id)
```

Aquí, necesitamos algo un poco más complicado, del estilo:

```
Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
```

De estas condiciones vemos que necesitaremos un arreglo de ids que correspondan a los usuarios que son seguidos. Una forma de hacer esto es utilizar el método `map` de Ruby, disponible en cualquier objeto “enumerable”, es decir, cualquier objeto que consista de una colección de elementos (tal como un arreglo o un Hash).⁹ Vimos un ejemplo de este método en la [Sección 4.3.2](#); como ejemplo adicional, utilizaremos `map` para convertir un arreglo de enteros en un arreglo de cadenas:

```
$ rails console
>> [1, 2, 3, 4].map { |i| i.to_s }
=> ["1", "2", "3", "4"]
```

Situaciones como la recién ilustrada, donde el mismo método es invocado en cada elemento de la colección, son suficientemente comunes como para que haya una notación abreviada para ellas (la cual revisamos brevemente en la [Sección 4.3.2](#)) que utiliza un *ampersand* & y un símbolo correspondiente al método:

```
>> [1, 2, 3, 4].map(&:to_s)
=> ["1", "2", "3", "4"]
```

⁹El principal requerimiento es que los objetos enumerables deben implementar un método `each` para iterar a través de la colección.

Usando el método **join** (Sección 4.3.1), podemos crear una cadena compuesta de los ids separados por una coma y un espacio:

```
>> [1, 2, 3, 4].map(&:to_s).join(', ')
=> "1, 2, 3, 4"
```

Podemos utilizar el método anterior para construir el arreglo necesario de ids de usuarios seguidos, invocando **id** en cada elemento de **user.following**. Por ejemplo, para el primer usuario de la base de datos el arreglo sería como el siguiente:

```
>> User.first.following.map(&:id)
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

De hecho, como esta clase de construcción es tan útil, Active Record la proporciona por default:

```
>> User.first.following_ids
=> [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51]
```

Aquí el método **following_ids** es simplificado por Active Record basado en la asociación **has_many :following** (Listado 12.8); el resultado es que sólo necesitamos agregar **_ids** al nombre de la asociación para obtener los ids correspondientes a la colección **user.following**. Una cadena de ids de usuarios seguidos se muestra a continuación:

```
>> User.first.following_ids.join(', ')
=> "4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51"
```

Aunque no necesita hacer esto; cuando insertamos dentro de una cadena SQL, la interpolación `?` se encarga de hacerlo por usted (y de hecho elimina algunas incompatibilidades relativas al tipo de base de datos subyacente). Esto significa que podemos utilizar `following_ids` por sí solo.

Como resultado, la suposición inicial de

```
Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
```

de hecho, ¡funciona!. El resultado se muestra en el Listado 12.43.

Listado 12.43: El avance inicial funcionando. VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns true if a password reset has expired.
  def password_reset_expired?
    reset_sent_at < 2.hours.ago
  end

  # Returns a user's status feed.
  def feed
    Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
  end

  # Follows a user.
  def follow(other_user)
    active_relationships.create(followed_id: other_user.id)
  end
  .
  .
  .
end
```

El conjunto de pruebas debería estar en VERDE:

Listado 12.44: VERDE

```
$ bundle exec rake test
```

En algunas aplicaciones, esta implementación inicial podría ser suficiente para la mayoría de los casos prácticos, pero el [Listado 12.43](#) no contiene la implementación final; revise a ver si puede adivinar porqué no, antes de continuar a la siguiente sección. (*Sugerencia:* ¿Qué pasa si un usuario está siguiendo a otros 5000 usuarios?)

12.3.3 Subconsultas

Como sugerimos en la sección anterior, la implementación del status de la [Sección 12.3.2](#) no escala bien cuando el número de micromensajes en el avance es grande, como podría suceder si un usuario estuviera siguiendo, digamos, a otros 5000 usuarios. En esta sección, re-implementaremos el status de avance de forma que escale mejor con el número de usuarios seguidos.

El problema con el código de la [Sección 12.3.2](#) es que `following_ids` extrae *todos* los ids de los usuarios seguidos en memoria, y crea un arreglo con ellos. Puesto que la condición del [Listado 12.43](#) de hecho verifica la inclusión en un conjunto, debe haber una forma más eficiente de hacer esto, y de hecho SQL está optimizado para tales operaciones de conjuntos. La solución involucra llevar la búsqueda de ids de usuarios seguidos a la base de datos utilizando una *subconsulta*.

Empezaremos refactorizando el avance con el código ligeramente modificado que se muestra en el [Listado 12.45](#).

Listado 12.45: Usando parejas de llave-valor en el método `where` del avance.

VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns a user's status feed.
  def feed
    Micropost.where("user_id IN (:following_ids) OR user_id = :user_id",
      following_ids: following_ids, user_id: id)
  end
  .
  .
```

```
    .
end
```

Como preparación para el siguiente paso, hemos reemplazado

```
Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
```

con el equivalente

```
Micropost.where("user_id IN (:following_ids) OR user_id = :user_id",
                 following_ids: following_ids, user_id: id)
```

La sintaxis del signo de interrogación está bien, pero cuando queremos que la *misma* variable sea insertada en más de un lugar, la segunda sintaxis es más conveniente.

El razonamiento anterior implica que estaremos agregando una *segunda* ocurrencia de `user_id` en la consulta SQL. En particular, podemos reemplazar el código de Ruby

```
following_ids
```

con el código SQL

```
following_ids = "SELECT followed_id FROM relationships
                  WHERE follower_id = :user_id"
```

Este código contiene una subconsulta SQL, e internamente la consulta completa para el usuario 1 buscaría algo como esto:

```
SELECT * FROM microposts
WHERE user_id IN (SELECT followed_id FROM relationships
                   WHERE follower_id = 1)
      OR user_id = 1
```

Esta subconsulta se encarga de toda la lógica de conjuntos que será enviada a la base de datos, lo cual es más eficiente.

Con esta base, estamos listos para una implementación del status de avance más eficiente, como puede verse en el [Listado 12.46](#). Observe que, como ahora es SQL crudo, la cadena `following_ids` está *interpolada*, no escapada.

Listado 12.46: La implementación final del status de avance. VERDE

app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  # Returns a user's status feed.
  def feed
    following_ids = "SELECT followed_id FROM relationships
                     WHERE follower_id = :user_id"
    Micropost.where("user_id IN (#{following_ids})
                     OR user_id = :user_id", user_id: id)
  end
  .
  .
  .
end
```

Este código tiene una formidable combinación de Ruby, Rails y SQL, realiza el trabajo y lo hace bien:

Listado 12.47: VERDE

```
$ bundle exec rake test
```

Por supuesto, aún la subconsulta no va a escalar para siempre. Para sitios muy grandes, probablemente debería generar el status de avance de forma asíncrona usando una tarea de fondo, pero tales sutilezas de escalamiento están más allá del alcance de este tutorial.

Con el código del [Listado 12.46](#), nuestro status de avance está completo. Recuerde de la [Sección 11.3.3](#) que la página principal ya incluye este status: como recordatorio, la acción `home` aparece nuevamente en el [Listado 12.48](#). En

el Capítulo 11, el resultado fue sólo un avance prototípico (Figura 11.14), pero con la implementación del Listado 12.46 que se observa en la Figura 12.23, la página principal muestra ahora el avance completo.

Listado 12.48: La acción `home` con un avance paginado.

`app/controllers/static_pages_controller.rb`

```
class StaticPagesController < ApplicationController

  def home
    if logged_in?
      @micropost = current_user.microposts.build
      @feed_items = current_user.feed.paginate(page: params[:page])
    end
  end
  .
  .
  .
end
```

En este momento, estamos listos para mezclar nuestros cambios con la rama principal:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user following"
$ git checkout master
$ git merge following-users
```

Luego podemos subir el código al repositorio remoto y desplegar la aplicación en producción:

```
$ git push
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

El resultado es un status de avance funcional en una web viva (Figura 12.24).

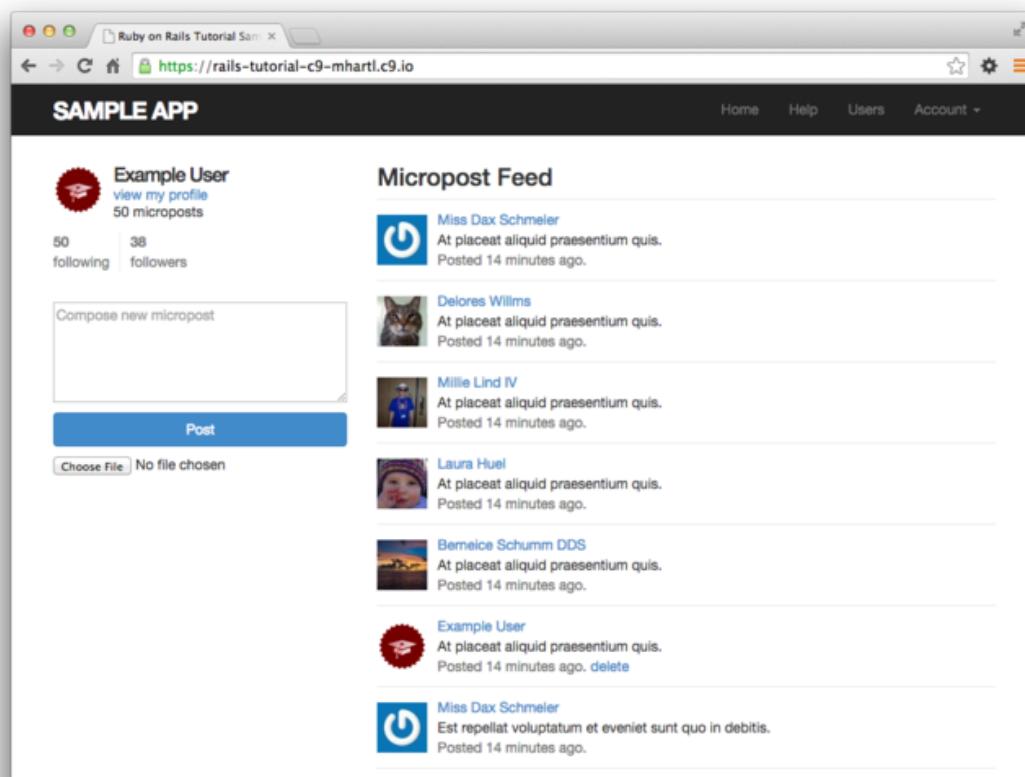


Figura 12.23: La página Home con un status de avance funcional.

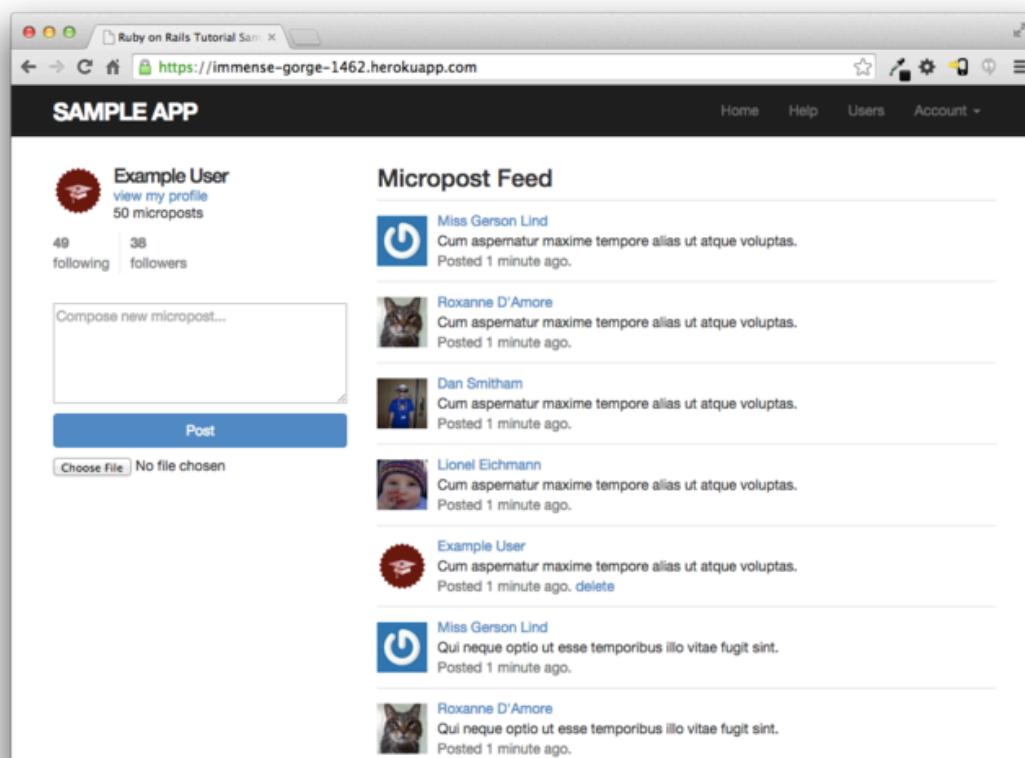


Figura 12.24: Un status de avance funcional en una web viva.

12.4 Conclusión

Con la adición del status de alimentación, hemos terminado la aplicación de ejemplo del *Tutorial de Ruby on Rails*. Esta aplicación incluye ejemplos de todas las características principales de Rails, incluyendo modelos, vistas, controladores, plantillas, parciales, filtros, validaciones, llamadas de retorno, asociaciones `has_many / belongs_to` y `has_many :through`, seguridad, pruebas y despliegue.

A pesar de esta impresionante lista, aún hay mucho qué aprender acerca del desarrollo web. Como un primer paso en este proceso, esta sección contiene algunas sugerencias de futuro aprendizaje.

12.4.1 Guía de recursos adicionales

Hay una gran variedad de recursos de Rails tanto en tiendas como en internet—de hecho, la provisión es tan vasta que puede ser abrumadora. Las buenas noticias son que, habiendo llegado hasta aquí, usted está listo para casi cualquier cosa que se encuentre allá afuera. Aquí están algunas sugerencias para seguir aprendiendo:

- [Los videos del Tutorial de Ruby on Rails](#): Yo ofrezco un curso completo en video basado en este libro. Además de cubrir todo el material del libro, los videos están llenos de sugerencias, trucos y demos del tipo *vea cómo se hace* que son difíciles de reflejar en un texto escrito. Están disponibles a la venta a través del [sitio web del Tutorial de Ruby on Rails](#).
- [RailsCasts](#): Le sugiero que empiece a visitar el [archivo de episodios de RailsCasts](#) y revise los temas que le causen curiosidad.
- [La Academia Tealeaf](#): Montones de campamentos de entrenamiento de desarrolladores en persona han surgido en los últimos años, por lo que le recomiendo buscar uno en su área, pero la [Academia Tealeaf](#) está disponible en línea por lo que puede cursarse desde cualquier parte. Tealeaf es especialmente una buena elección si usted desea retroalimentación de un instructor dentro del contexto de un plan de estudios estructurado.

- La [Escuela de Turing de Software y Diseño](#): un programa de entrenamiento en Ruby/Rails/JavaScript, de tiempo completo, con 27 semanas de duración que se imparte en Denver, Colorado . La mayoría de sus estudiantes inician con una experiencia en programación limitada pero tienen la determinación de ampliarla rápidamente. Turing garantiza a sus estudiantes que encontrarán empleo luego de graduarse o les devuelven el costo de la matrícula o inscripción. Los lectores del Tutorial de Rails pueden obtener un descuento de [\\$500 dólares](#) utilizando el código RAILSTUTORIAL500.
- [Bloc](#): un campamento de entrenamiento en línea con un plan de estudios estructurado, tutoría personalizada, y un enfoque en el aprendizaje a través de proyectos concretos. Utilice el cupón BLOCLOVESHARTL para obtener un descuento de \$500 dólares en la cuota de inscripción.
- [Thinkful](#): Una clase en línea que lo contacta con un ingeniero profesional conforme trabaja en un proyecto. Los temas incluyen Ruby on Rails, desarrollo front-end, diseño web, y ciencia de datos.
- [Pragmatic Studio](#): Cursos de Ruby and Rails en línea impartidos por Mike y Nicole Clark.
- [RailsApps](#): Aplicaciones educativas de ejemplo en Rails.
- [Code School](#): Una gran variedad de cursos de programación interactivos.
- [Desarrollo en Ruby Orientado a Pruebas, de Bala Paraná](#): Un curso en línea avanzado que se enfoca en el Desarrollo orientado a Pruebas en Ruby puro.
- Libros de Ruby and Rails: Para aprender más de Ruby, le recomiendo [*BEGINNING RUBY*](#) de Peter Cooper, [*The Well-Grounded Rubyist*](#) de David A. Black, [*Eloquent Ruby*](#) de Russ Olsen y [*The Ruby Way*](#) de Hal Fulton. Para aprender más de Rails, le recomiendo [*Agile Web Development with Rails*](#) de Sam Ruby, Dave Thomas, y David Heinemeier Hansson, [*The Rails 4 Way*](#) de Obie Fernandez y Kevin Faustino y [*Rails 4 in Action*](#) de Ryan Bigg y Yehuda Katz.

12.4.2 Qué aprendimos en este capítulo

- La expresión de Rails `has_many :through` nos permite modelar relaciones de datos complejos.
- El método `has_many` recibe varios argumentos opcionales, incluyendo el nombre de la clase del objeto y la llave foránea.
- Usando `has_many` y `has_many :through` con los nombres de clase y las llaves foráneas apropiadas, podemos modelar tanto relaciones activas (seguir) como pasivas (ser seguido).
- El ruteo de Rails soporta rutas anidadas.
- El método `where` es una herramienta flexible y poderosa para crear consultas a la base de datos.
- Rails acepta la creación de consultas de bajo nivel en SQL, si es necesario.
- Juntando todo lo que hemos aprendido en este libro, hemos implementado exitosamente el seguimiento de usuarios con un status de avance de los micromensajes provenientes de los usuarios seguidos.

12.5 Ejercicios

Nota: El *Manual de Soluciones para los Ejercicios*, con soluciones para cada ejercicio del libro *Tutorial de Ruby on Rails*, se incluye de forma gratuita en cada compra realizada en www.railstutorial.org.

Si desea una sugerencia acerca de cómo evitar conflictos entre los ejercicios y el tutorial principal, revise la nota del ejercicio sobre ramas temáticas en la Sección 3.6.

1. Escriba pruebas para las estadísticas de las páginas principal y la de perfil. *Sugerencia:* Agréguelas a la prueba del Listado 11.27. (¿Porqué no necesitamos probar las estadísticas de la página principal por separado?)

2. Escriba una prueba para verificar que la primera página de avance aparece en la página principal como es requerido. Una plantilla se muestra en el [Listado 12.49](#). Observe la forma de escapar el código HTML utilizando `CGI.escapeHTML`; vea si puede descubrir porqué esto es necesario. (Intente remover el escapado e inspeccione con cuidado el código de la página para el contenido del micromensaje que no coincide.)

Listado 12.49: Probando el HTML del avance. [VERDE](#)

test/integration/following_test.rb

```
require 'test_helper'

class FollowingTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
    log_in_as(@user)
  end
  .
  .
  .
  test "feed on Home page" do
    get root_path
    @user.feed.paginate(page: 1).each do |micropost|
      assert_match CGI.escapeHTML(FILL_IN), FILL_IN
    end
  end
end
```