
RUBY ON RAILS
ДЛЯ НАЧИНАЮЩИХ
ИЗУЧАЕМ РАЗРАБОТКУ
ВЕБ-ПРИЛОЖЕНИЙ
НА ОСНОВЕ RAILS
Майкл Хартл
ДМК
ИЗДАТЕЛЬСТВО

Майкл Хартл

Ruby on Rails для начинающих

Michael Hartl

Ruby on Rails Tutorial

Learn Web Development with Rails

Third Edition

Addison-Wesley
New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Майкл Хартл

Ruby on Rails для начинающих

Изучаем разработку веб-приложений на основе Rails

Москва, 2017

УДК 004.438Ruby on Rails
ББК 32.973.22
Х22

Хартл М.
Х22 Ruby on Rails для начинающих / пер. с анг. А. Разуваева. – М.: ДМК Пресс, 2017. – 572 с.: ил.

ISBN 978-5-97060-429-8

Ruby on Rails – один из наиболее популярных фреймворков для разработки веб-приложений, но его изучение и использование – не самая простая задача. Эта книга поможет вам решить ее независимо от того, имеете ли вы опыт веб-разработки вообще и Rails в частности. Известный автор и ведущий разработчик Rails Майкл Хартл познакомит вас с Rails на примере разработки трех приложений. Автор рассказывает не только о Rails, но также описывает основы Ruby, HTML, CSS и SQL, которые пригодятся вам при разработке своих веб-приложений. Начиная обсуждение каждой новой методики, Хартл доходчиво объясняет, как она помогает решать практические задачи, а затем демонстрирует ее применение в программном коде, достаточно простом и понятном.

Издание предназначено для всех программистов, желающих изучить Ruby on Rails.

УДК 004.438Ruby on Rails
ББК 32.973.22

ISBN 978-0-13-407770-3 (анг.)
ISBN 978-5-97060-429-8 (рус.)

Содержание

Предисловие

Моя бывшая компания (CD Baby) была одной из первых, громогласно заявивших о переходе на Ruby on Rails, а затем еще громче возвестившей о возврате на PHP (Google расскажет вам об этой драме). Эту книгу, написанную Майклом Хартлом, так настоятельно рекомендовали, что я должен был прочитать ее, и именно благодаря книге «Ruby on Rails для начинающих» я вернулся к Rails.

На своем пути я встречал много книг о Rails, и эта – одна из тех немногих, что, наконец, «зацепила» меня. Здесь все делается способом, типичным для Rails, который прежде казался мне крайне неестественным, но теперь, после изучения этой книги, пришло ощущение комфорта и естественности этого подхода. К тому же это единственная книга о Rails, которая соблюдает методику разработки через тестирование на всем своем протяжении; именно этот подход строго рекомендуется специалистами, но ранее я нигде не встречал такой его отчетливой демонстрации. Наконец, включив в примеры использование таких инструментов, как Git, GitHub и Heroku, автор дает почувствовать, что из себя представляет разработка проектов в реальном мире. Учебные примеры в этой книге не являются разрозненными фрагментами кода.

Линейное повествование – отличный формат. Лично я прочитал «Ruby on Rails для начинающих» за три полных дня[1], выполняя все примеры и задачи в конце каждой главы. Делайте все от начала и до конца, не перескакивая от одной темы к другой, и вы получите максимальную пользу.

Наслаждайтесь!

Derek Sivers (sivers.org),
основатель CD Baby

[1] Это нестандартная ситуация! Обычно прохождение всего учебника занимает *гораздо* больше, чем три дня.

Благодарности

«Ruby on Rails для начинающих» во многом обязан своим появлением моей предыдущей книге по Rails – «Rails Space» и, следовательно, моему соавтору Орилиусу Прочазка (Aurelius Prochazka). Я хотел бы поблагодарить Орилиуса и за работу над прошлой книгой, и за поддержку этой. Я также хочу поблагодарить Дебру Уильямс Коли (Debra Williams Cauley), редактора обеих этих книг; пока она будет брать меня с собой на бейсбол, я буду продолжать писать книги для нее.

Я хотел бы поблагодарить многих фанатов Ruby, учивших и вдохновлявших меня на протяжении многих лет. Это: Дэвид Хейнмейер Ханссон (David Heinemeier Hansson), Йехуда Кац (Yehuda Katz), Карл Лерхе (Carl Lerche), Джереми Кемпер (Jeremy Kemper), Ксавье Нория (Xavier Noria), Райан Бейтс (Ryan Bates), Джеффри Грозенбах (Geoffrey Grosenbach), Питер Купер (Peter Cooper), Мэтт Аимонетти (Matt Aimonetti), Марк Бейтс (Mark Bates), Грегг Поллак (Gregg Pollack), Вейн Е. Сегуин (Wayne E. Seguin), Ами Хой (Amy Hoy), Дэйв Челимски (Dave Chelimsky), Пэт Мэддокс (Pat Maddox), Том Престон-Вернер (Tom Preston-Werner), Крис Ванстрат (Chris Wanstrath), Чад Фаулер (Chad Fowler), Джош Сассер (Josh Susser), Оби Фернандес (Obie Fernandez), Ян Мак-Фарланд (Ian McFarland), Стивен Бристоль (Steven Bristol), Пратик Найк (Pratik Naik), Сара Мей (Sarah Mei), Сара Аллен (Sarah Allen), Вольфрам Арнольд (Wolfram Arnold), Алекс Чеффи (Alex Chaffee), Джилс Бокет (Giles Bowkett), Эван Дорн (Evan Dorn), Лонг Нгуен (Long Nguyen), Джеймс Линденбаум (James Lindenbaum), Адам Уиггинс (Adam Wiggins), Тихон Бернстам (Tikhon Bernstam), Рон Эванс (Ron Evans), Уайат Грин (Wyatt Greene), Майлз Форрест (Miles Forrest), Санди Метц (Sandi Metz), Райан Дэвис (Ryan Davis), Аарон Паттерсон (Aaron Patterson), сотрудники Pivotal Labs, команда Heroku, ребята из thoughtbot и команда GitHub. Наконец, многих, многих читателей – слишком многих, чтобы здесь перечислить их всех, – внесших большое количество предложений по улучшению и сообщивших об ошибках во время работы над этой книгой, и я с благодарностью признаю, что она получилась настолько хорошей во многом благодаря им.

Об авторе

Майкл Хартл – автор книги «Ruby on Rails для начинающих», одного из лучших введений в веб-разработку, сооснователь Softcover – платформы для упрощенной публикации электронных книг. Его предыдущий опыт включает написание учебника «RailsSpace», ныне серьезно устаревшего, и разработку Insoshi, некогда популярной платформы создания социальных сетей, написанной на Ruby on Rails. В 2011 году Майкл был награжден премией «Ruby Hero Award» за вклад в развитие сообщества пользователей Ruby. Закончил Гарвардский колледж, имеет степень кандидата физических наук, присвоенную в Калифорнийском технологическом институте, и является выпускником предпринимательских курсов Y Combinator.

Авторские права и лицензия

Глава 1

От нуля к развертыванию

Добро пожаловать в «Ruby on Rails для начинающих: Изучаем разработку веб-приложений на основе Rails». Цель данной книги – научить вас разрабатывать собственные веб-приложения с использованием популярного фреймворка Ruby on Rails. Если эта тема для вас в новинку, «Ruby on Rails для начинающих» даст вам полное представление о разработке веб-приложений, включая основополагающие знания о Ruby, Rails, HTML и CSS, базах данных, управлении версиями, тестировании и развертывании – достаточно для начала вашей карьеры веб-разработчика или предпринимателя в сфере компьютерных технологий. С другой стороны, если вы уже знакомы с веб-разработкой, эта книга поможет вам освоить основы фреймворка Rails, включая MVC и REST, генераторы, миграции, маршрутизацию и встроенный язык Ruby. В любом случае, когда вы закончите чтение этой книги, вы будете готовы извлечь максимум пользы из более продвинутых книг, блогов и скринкастов (видеоуроков), являющихся частью обширной образовательной экосистемы[1].

В данной книге используется комплексный подход к веб-разработке: в процессе обучения мы напишем три примера приложений с возрастающей сложностью; в разделе 1.3 мы начнем создание простенького приложения, затем, в главе 2, перейдем к более продвинутому и, наконец, в главах с 3 по 12 полностью сосредоточимся на большом учебном примере. Как можно догадаться, примеры приложений в этой книге не привязаны к какой-либо категории веб-сайтов; хотя заключительный пример будет иметь значительное сходство с весьма популярной социальной сетью Twitter (которая, по совпадению, изначально была написана на Rails). Основное внимание здесь будет уделяться общим принципам разработки, поэтому полученные знания станут для вас надежным фундаментом, вне зависимости от того, какие виды приложений вы будете создавать.

Я часто слышу вопрос: «Какими знаниями нужно обладать для изучения веб-разработки с этой книгой?» Как мы увидим в разделе 1.1.1, веб-разработка – сложная тема, особенно для полных новичков. Хотя изначально книга предназначалась

---

[1] Самую свежую версию книги (на англ. языке. – *Прим. перев.*) можно найти на сайте http://www.railstutorial.org/. Если вы читаете печатную версию книги, сверьте версию своего экземпляра с онлайн-версией на http://www.railstutorial.org/book.

для читателей с некоторым опытом программирования, она нашла множество читателей среди начинающих разработчиков. Учитывая это, в данном (третьем) издании значительно снижен порог входа в разработку с Ruby on Rails (см. блок 1.1).

Блок 1.1 ❖ Снижение порога входа

С целью снижения порога входа для начинающих разработчиков в третьем издании сделано следующее:

- использована стандартная среда разработки, размещенная в облаке (раздел 1.2), что позволяет обойти множество проблем, связанных с установкой и настройкой новой системы;
- использован «предустановленный стек» Rails, включающий встроенный фреймворк тестирования MiniTest;
- уменьшено количество внешних зависимостей (RSpec, Cucumber, Capybara, Factory Girl);
- использован облегченный и более гибкий подход к тестированию;
- исключены сложные в настройке варианты (Spork, RubyTest);
- меньше внимания уделяется особенностям, свойственным конкретной версии Rails, и больше – общим принципам веб-разработки.

Я надеюсь, что эти изменения сделают третье издание книги доступным для еще более широкой аудитории, чем предыдущие.

В этой первой главе мы начнем изучение фреймворка Ruby on Rails с установки необходимого программного обеспечения и настройки рабочего окружения (раздел 1.2). Затем создадим первое Rails-приложение с названием `hello_app`. В книге «Ruby on Rails для начинающих» особое значение придается хорошим привычкам в программировании, поэтому сразу после создания нового Rails-проекта мы поместим его в систему управления версиями Git (раздел 1.4). И, верите вы в это или нет, в этой главе мы даже развернем наше первое приложение в Сети (раздел 1.5).

В главе 2 мы создадим второй проект для демонстрации основ работы Rails-приложения. Для быстроты в этом *мини-приложении* (с названием `toy_app`) мы применим прием «скаффолдинга» (scaffolding, см. блок 1.2) для генерации кода. Поскольку код получится одновременно и сложным, и уродливым, мы оставим его в стороне и сосредоточимся на взаимодействии с приложением через его идентификатор *URI* (который часто называют адресом *URL*)[1], с использованием веб-браузера.

Остальная часть книги описывает разработку единственного большого учебного приложения (с названием `sample_app`), весь код которого будет написан с нуля. При этом мы будем использовать фиктивные объекты, приемы разработки через тестирование (Test-Driven Development, TDD) и интеграционные тесты. В главе 3 мы сначала создадим статические страницы, а затем добавим в них немного ди-

[1] Аббревиатура URI расшифровывается как Uniform Resource Identifier (универсальный идентификатор ресурсов), а аббревиатура URL – как Uniform Resource Locator (универсальный указатель ресурсов). На практике URL обычно соответствует тому, что «находится в адресной строке браузера».

намического контента. В главе 4 мы совершим небольшое турне по языку Ruby, лежащему в основе Rails. Затем, в главах с 5 по 10, завершим базовую часть приложения, создав макет сайта, модель данных пользователя, а также систему регистрации и аутентификации (включая активацию учетной записи и сброс пароля). Наконец, в главах 11 и 12 мы добавим поддержку микроблоггинга и немного социальных функций, чтобы закончить действующий пример сайта.

Блок 1.2 ❖ Скаффолдинг: быстрее, проще, заманчивее

С самого начала фреймворк Rails привлек к себе внимание знаменитым видеороликом от Дэвида Хейнмейера Ханссона (David Heinemeier Hansson) – создателя Rails, демонстрирующим создание микроблога за 15 минут. Этот и другие подобные видеоролики – отличный способ познакомиться с возможностями Rails, и я рекомендую посмотреть их. Но предупреждаю: они совершают свой удивительный пятнадцатиминутный подвиг, используя функцию под названием «скаффолдинг» (scaffolding), которая в значительной степени опирается на код, волшебным образом сгенерированный Rails-командой `generate scaffold`.

Работая над книгой по Ruby on Rails, мне приходилось бороться с искушением положиться на эту функцию, потому что это быстрее, проще, заманчивее. Но сложность и огромный объем кода, производимый генератором, могли стать непреодолимым препятствием для начинающего Rails-разработчика; вы, вероятно, сможете использовать его, но почти наверняка не сможете понять. Увлекшись скаффолдингом, вы рискуете превратиться в виртуозного генератора сценариев с весьма поверхностным знанием Rails.

В этой книге мы будем придерживаться (почти) полярно противоположного подхода: хотя в главе 2 создадим мини-приложение, сгенерировав код автоматически, основой книги все же является приложение, которое мы начнем писать в главе 3. На каждом этапе его разработки мы будем писать небольшие куски кода, достаточно простые для понимания, но все же требующие некоторых усилий для их усвоения. Результатом станет более глубокое понимание Rails, которое, в свою очередь, даст вам хорошую базу для написания практически любых типов веб-приложений.

1.1. Введение

«Ruby on Rails» (или просто «Rails») – это фреймворк для разработки веб-приложений на языке программирования Ruby. Со времен своего дебюта в 2004 году Ruby on Rails довольно быстро стал одним из самых мощных и популярных инструментов создания динамических веб-приложений. Rails используется множеством различных компаний: Airbnb, Basecamp, Disney, GitHub, Hulu, Kickstarter, Shopify, Twitter и Yellow Pages. Помимо этого, существует множество компаний, занимающихся разработкой веб-приложений и специализирующихся на Rails, таких как ENTP, thoughtbot, Pivotal Labs, Hashrocket и HappyFunCorp, плюс бесчисленное множество независимых консультантов, преподавателей и индивидуальных разработчиков.

Что же делает Rails таким замечательным? Во-первых, Ruby on Rails – это открытый исходный код, доступный на условиях лицензии MIT, и, как следствие, его можно загружать и использовать совершенно бесплатно. Rails также обязан своим успехом изящному и компактному дизайну. Используя податливость языка Ruby, лежащего в его основе, Rails фактически определяет предметно-ориентированный язык для разработки веб-приложений. В результате множество часто встречающихся задач веб-программирования, таких как динамическое создание разметки HTML, определение моделей данных и маршрутизация URL, легко решаются в Rails, а конечный код приложений получается кратким и читаемым.

Rails также быстро адаптируется к новым тенденциям в веб-технологиях. Например, Rails одним из первых полностью реализовал архитектурный стиль REST структурирования веб-приложений (мы будем изучать его на всем протяжении книги). И когда в других фреймворках появляются новые успешные приемы, создатель Rails, Дэвид Хейнмейер Ханссон (David Heinemeier Hansson), и рабочая группа Rails не стесняются использовать чужой опыт. Пожалуй, наиболее драматичным примером является слияние Rails с конкурирующим веб-фреймворком Merb, благодаря которому Rails получил модульный дизайн, стабильный API и улучшенную производительность.

Наконец, вокруг Rails сплотилось необычайно увлеченное и разнообразное сообщество: сотни разработчиков, представительные конференции, огромное количество гемов (пакетов, законченных решений конкретных задач, таких как постраничный вывод и выгрузка изображений), богатый набор информативных блогов и рог изобилия форумов и IRC-каналов. Большое количество Rails-программистов также облегчает работу с (неизбежными) ошибками, возникающими в процессе разработки: алгоритм «Ищи в Google сообщение об ошибке» практически всегда помогает найти подходящую статью в блоге или сообщение на форуме.

1.1.1. Необходимая квалификация

Формально эта книга не требует некоторого набора необходимых знаний. Она содержит учебные сведения не только о фреймворке Rails, но также о лежащем в его основе языке Ruby, встроенном фреймворке тестирования (MiniTest), командной строке Unix, HTML, CSS, немного JavaScript и даже чуть-чуть SQL. Это очень объемный материал, и потому для работы с книгой желательно иметь некоторый опыт использования HTML и программирования. Однако данную книгу использовало так много новичков для изучения веб-разработки с нуля, что даже если у вас мало опыта, я все же советую попробовать. Если содержимое учебника покажется сложным, вы всегда можете сделать шаг назад и начать с одного из ресурсов, перечисленных ниже. Другая возможная стратегия (рекомендованная многими читателями): прочитать книгу дважды; возможно, вы удивитесь, как много узнали в первый раз (и насколько проще читать по второму кругу).

Стоит ли вначале изучить Ruby? Ответ зависит от вашего личного стиля обучения и опыта в программировании. Если вы предпочитаете изучать все систематически, с самых основ, или прежде никогда не программировали, вам, возмож-

но, стоит начать с изучения Ruby, и в этом случае я рекомендую книгу «Учись программировать» Криса Пайна (Chris Pine)[1] и «Начало Ruby» Питера Купера (Peter Cooper)[2]. С другой стороны, множество начинающих Rails-разработчиков имеет своей целью создание веб-приложений и, скорее, предпочтет не корпеть над толстенной книгой, чтобы написать одну-единственную веб-страницу. В этом случае я рекомендую поработать с коротким интерактивным учебником «Try Ruby»[3], чтобы получить общее представление о Ruby. Если и после этого книга окажется для вас слишком сложной, попробуйте начать с «Learn Ruby on Rails» Дэниэла Кехо (Daniel Kehoe)[4] или «One Month Rails»[5] – обе книги ориентированы на начинающих, в отличие от данной книги.

Независимо от того, где вы начнете, к концу этой книги вы будете готовы к чтению других, более продвинутых ресурсов по Rails. Вот некоторые из тех, которые я могу рекомендовать:

- Code School[6]: хорошие интерактивные онлайн-курсы по программированию;
- Turing School of Software & Design[7]: очные, 27-недельные курсы в Денвере, Колорадо. Читатели этой книги могут получить скидку в $500, воспользовавшись промокодом RAILSTUTORIAL500;
- Tealef Academy[8]: хорошая образовательная онлайн-площадка, посвященная Rails-разработке (включая продвинутый материал);
- Thinkful[9]: онлайн-курс в паре с профессиональным инженером, работа по учебному плану, основанному на конкретном проекте;
- RailsCastsот[10] Райана Бейтса (Ryan Bates): отличные (в основном бесплатные) видеоуроки по Rails;
- RailsApps[11]: большой выбор подробных проектов и учебников, посвященных разным темам;
- Rails Guides[12]: актуальные руководства по Rails.

1.1.2. Типографские соглашения

Соглашения в этой книге главным образом очевидны. В этом разделе я упомяну лишь те, которые таковыми не являются.

---

1 http://www.shokhirev.com/mikhail/ruby/ltp/title.html. – *Прим. перев.*
2 http://www.amazon.com/gp/product/1430223634.
3 http://tryruby.org/.
4 http://learn-rails.com/learn-ruby-on-rails.html.
5 http://mbsy.co/7Zdc7.
6 https://my.getambassador.com/.
7 https://www.turing.io/friends/tutorial.
8 https://launchschool.com/.
9 https://www.thinkful.com/a/railstutorial.
10 http://railscasts.com/.
11 https://tutorials.railsapps.org/.
12 http://rusrails.ru/.

В этой книге присутствует множество примеров применения командной строки. Для простоты все они используют приглашение в стиле Unix (знак доллара):

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

Как отмечено в разделе 1.2, я рекомендую пользователям всех операционных систем (особенно Windows) использовать облачную среду разработки (раздел 1.2.1), в которой есть встроенная командная строка Unix (Linux). Это особенно полезно, потому что в Rails имеется множество команд, которые можно запустить из командной строки. Например, в разделе 1.3.2 мы запустим локальный веб-сервер командой **rails server**:

```
$ rails server
```

По аналогии с приглашением командной строки, здесь используется соглашение о разделителях каталогов в стиле Unix (то есть прямого слеша /). Например, конфигурационный файл **production.rb** учебного приложения будет представлен как:

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

Этот путь к файлу следует интерпретировать как относительный, начинающийся в корневом каталоге приложения, абсолютный путь к которому зависит от системы; в облачной интегрированной среде разработки (IDE) (раздел 1.2.1) этот путь выглядит так:

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

То есть полный путь к `production.rb` имеет вид:

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

Для простоты я часто буду опускать путь к корневому каталогу приложения и писать просто: **config/environments/production.rb**.

В книге нередко демонстрируется вывод разных программ (команд, системы управления версиями, программ на Ruby и т. д.). Из-за неисчислимого количества небольших различий между компьютерными системами вывод, который вы увидите, возможно, не всегда в точности совпадет с тем, что показан в тексте, но это не повод для беспокойства. Некоторые команды могут вызывать ошибки, связанные с особенностями вашей системы, поэтому, чтобы не решать сизифову задачу документирования всех таких погрешностей, я отсылаю вас к алгоритму «ищи сообщение об ошибке в Google», который, между прочим, является хорошей практикой для программирования. Если вы столкнетесь с проблемами в процессе обучения, я советую обратиться к ресурсам, список которых приводится в справочном разделе[1].

[1] https://www.railstutorial.org/#help.

Поскольку в книге (помимо всего прочего) рассматривается вопрос тестирования Rails-приложений, часто бывает полезно знать, что данный кусок кода приводит к неудаче (обозначается красным цветом) или, наоборот, к успеху тестирования (обозначается зеленым цветом). Для удобства программный код тестов будет отмечаться словами **КРАСНЫЙ** и **ЗЕЛЕНЫЙ**.

К каждой главе прилагаются упражнения, выполнение которых необязательно, но рекомендуется. Чтобы сделать основной текст независимым от упражнений, решения обычно не включаются в последующие листинги кода. В редких случаях, когда решение упражнения используется в дальнейшем, оно будет приведено в основном тексте.

Наконец, для удобства в книге применяются два соглашения, облегчающие понимание большого количества примеров кода. Во-первых, в некоторых листингах выделены одна или несколько строк, как показано ниже:

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

Такое выделение обычно указывает на наиболее важный новый код в данном примере и часто (хотя и не всегда) отражает разницу между этим и предыдущим листингом. Во-вторых, для краткости и простоты во многих листингах в книге используются вертикальные точки, например:

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

Эти точки обозначают пропущенный код, и их не нужно копировать буквально.

1.2. За работу

Установка Ruby, Rails и всего сопутствующего программного обеспечения поддержки может привести в отчаяние даже опытных Rails-разработчиков. Проблема осложняется большим разнообразием окружений: разные операционные системы, версии, предпочтения в выборе текстовых редакторов и интегрированных сред разработки (IDE) и т. д. Пользователи, уже установившие среду разработки, могут сразу переходить к настройкам, а новым пользователям (как указано в блоке 1.1) я предлагаю избежать проблем с установкой и настройкой за счет использования *облачной интегрированной среды разработки*. Облачная IDE запускается в обычном браузере и, следовательно, одинаково хорошо работает на любой платформе, что особенно полезно для операционных систем (например, Windows), в которых разработка с Rails исторически была сложным занятием. Если, несмотря на все

предполагаемые трудности, вы по-прежнему хотите создать локальную среду разработки, я рекомендую следовать инструкциям на InstallRails.com[1].

1.2.1. Среда разработки

Принимая во внимание множество своеобразных настроек и предпочтений, число вариантов окружений разработки может равняться числу Rails-программистов. Чтобы избежать этих сложностей, я адаптировал данную книгу под превосходную облачную среду разработки Cloud9. В частности, мне было очень приятно сотрудничать с Cloud9, чтобы предложить вам среду разработки, ориентированную на потребности именно этого издания учебника. Получившееся рабочее пространство Cloud9 предоставляется предварительно настроенным, со всем программным обеспечением (включая Ruby, RubyGems, Git), необходимым для профессиональной разработки с Rails. (Отдельно мы будем устанавливать только сам фреймворк Rails, но это сделано намеренно (раздел 1.2.2).) Облачная IDE также включает три основных компонента, необходимых для разработки веб-приложений: текстовый редактор, навигатор файловой системы и терминал командной строки (рис. 1.1). Кроме того, ее текстовый редактор поддерживает глобальный поиск «Найти в файлах», который я считаю необходимым для навигации по любым большим

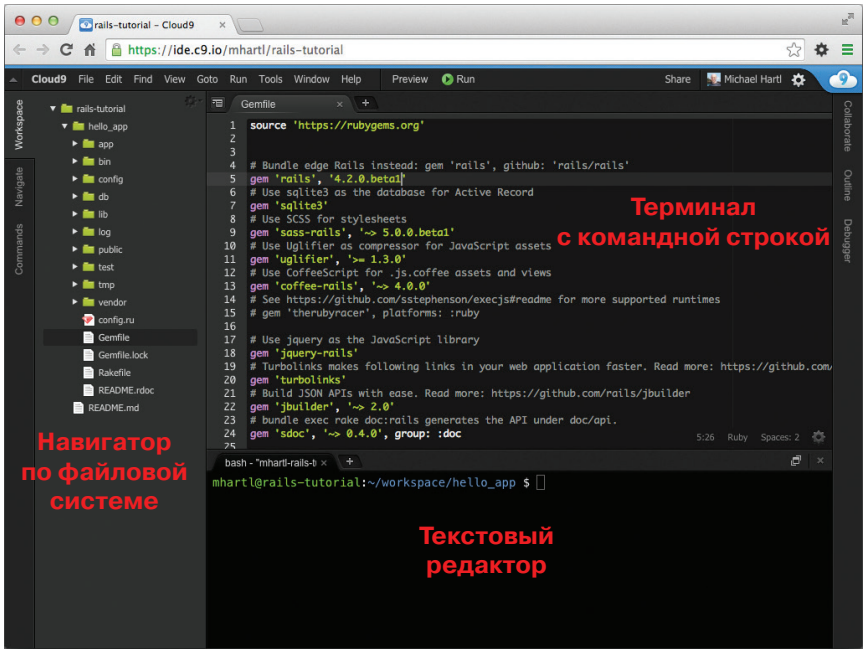


Рис. 1.1 ❖ Анатомия облачной IDE

[1] http://installrails.com/.

проектам Ruby или Rails[1]. И наконец, если вы решите использовать не только облачную IDE (я могу лишь приветствовать изучение других инструментов), на ее примере вы получите великолепное введение в основные возможности текстовых редакторов и других инструментов разработки.

Ниже приводится пошаговая инструкция, описывающая, как начать работу с облачной средой разработки:

- создайте бесплатную учетную запись в Cloud9[2];
- щелкните на ссылке **Goto your Dashboard** (Перейти к панели управления);
- щелкните на ссылке **Create New Workspace** (Создать новое рабочее пространство);
- как показано на рис. 1.2, создайте рабочее пространство `rails-tutorial` (не `rails_tutorial`), установите в настройках флажок **Private to the people I invite** (Только для приглашенных) и выберите значок **RailsTutorial** (не значок **Ruby on Rails**);

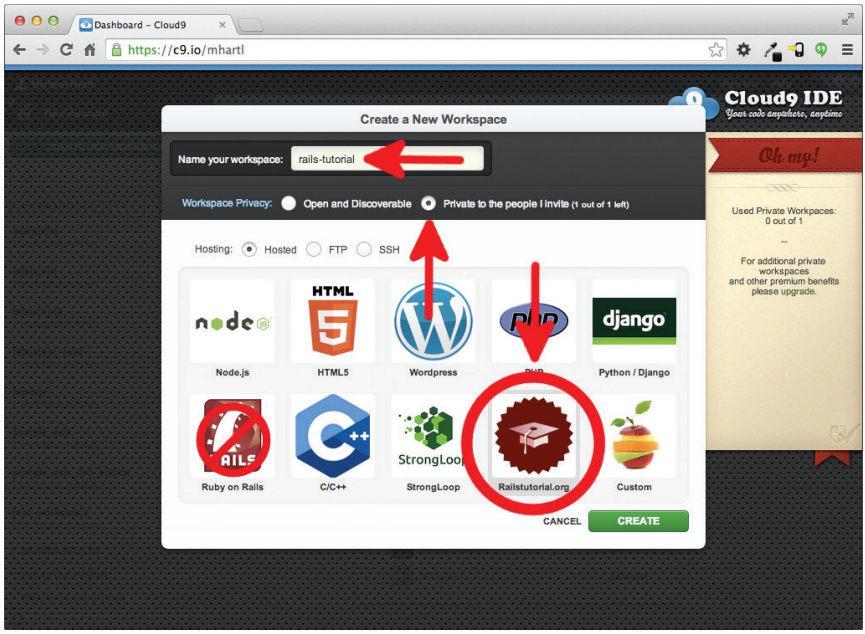


Рис. 1.2 ❖ Создание нового рабочего пространства в Cloud9

- щелкните на ссылке **Create** (Создать);
- после того как Cloud9 закончит подготовку рабочего пространства, выберите его и щелкните на ссылке **Start editing** (Начать редактирование).

---

[1] Например, чтобы найти определение функции `foo`, можно запустить глобальный поиск по фразе `def foo`.

[2] https://c9.io/web/sign-up/free.

Так как использование двух пробелов для отступов считается практически универсальным соглашением в Ruby, я также рекомендую изменить настройки редактора (по умолчанию используются четыре пробела). Как показано на рис. 1.3, щелкните на значке с изображением шестеренки в правом верхнем углу, выберите пункт **Code Editor (Ace)** (Редактор кода (Ace)) и измените параметр **Soft Tabs** (Мягкая табуляция). (Обратите внимание, что настройки применяются мгновенно и нет необходимости нажимать кнопку **Save** (Сохранить).)

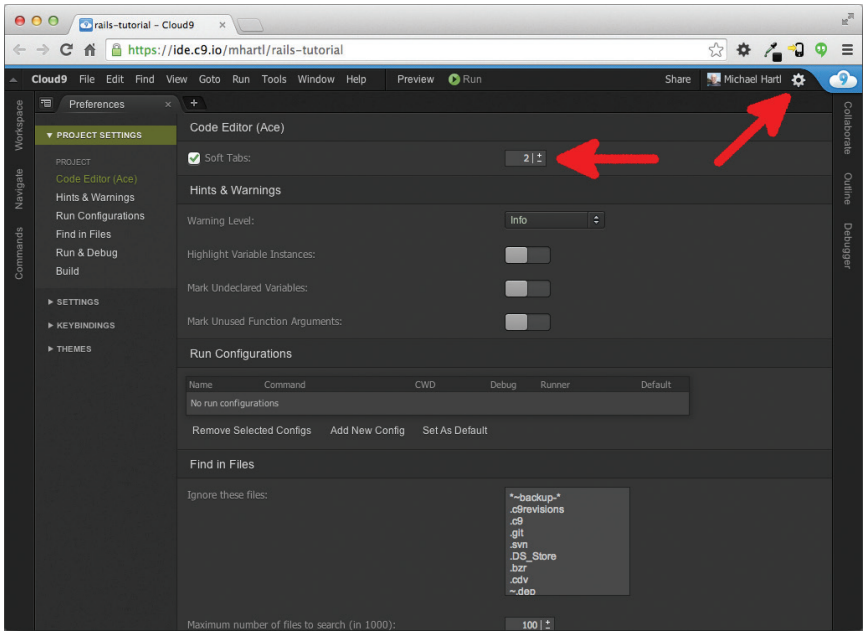


Рис. 1.3 ❖ Настройка использования двух пробелов для отступа в Cloud9

1.2.2. Установка Rails

Среда разработки, представленная в разделе 1.2.1, включает все ПО, необходимое для начала, кроме самого фреймворка Rails[1]. Для установки воспользуемся командой `gem` диспетчера пакетов *Ruby Gems*. Введите в командной строке терминала команду, как показано в листинге 1.1. (Если вы работаете в локальной системе, необходимо использовать обычное окно терминала; если в облачной IDE – область командной строки, как на рис. 1.1.)

Листинг 1.1 ❖ Установка Rails с определенным номером версии

```
$ gem install rails -v 4.2.0
```

[1] На данный момент в Cloud9 используется более старая версия Rails, и она несовместима с настоящей книгой, поэтому так важно установить его самостоятельно.

Параметр -v гарантирует установку определенной версии Rails. Это очень важно для получения результатов, совместимых с этой книгой.

1.3. Первое приложение

Следуя давним традициям в программировании, нашей первой целью будет создание программы, которая выводит фразу «hello, world» («Привет, мир!»). В частности, мы создадим простое приложение, которое отобразит строку «hello, world!» на веб-странице: в среде разработки на компьютере (раздел 1.3.4) и в Интернете (раздел 1.5).

Практически все приложения Rails начинаются одинаково – с команды `rails new`. Эта удобная команда создает скелет приложения Rails в выбранном каталоге. Тем, кто не стал использовать Cloud9 IDE, рекомендованную в разделе 1.2.1, сначала необходимо создать каталог `workspace` для проекта, если он еще не создан (листинг 1.2), и перейти в него. (В листинге 1.2 показаны команды `cd` и `mkdir`; если вы с ними пока не знакомы, прочитайте блок 1.3.)

Листинг 1.2 ❖ Создание каталога workspace для проекта (не нужно в облачной IDE)

```
$ cd                # Перейти в домашний каталог.
$ mkdir workspace   # Создать каталог workspace.
$ cd workspace/     # Перейти в каталог workspace.
```

Блок 1.3 ❖ Краткий курс командной строки Unix

Для читателей, пришедших из Windows или (в меньшей, но все еще значительной степени) из Macintosh OS X, командная строка Unix может выглядеть непривычной. К счастью, если вы используете рекомендованную облачную среду разработки, вы автоматически получаете доступ к командной строке Unix (Linux) в стандартной оболочке интерфейса, известной как Bash[1].

Основная идея командной строки проста: вводя короткие команды, пользователь может выполнить множество операций, таких как создание каталогов (`mkdir`), перемещение и копирование файлов (`mv` и `cp`), навигация в файловой системе за счет смены каталогов (`cd`). Командная строка может показаться примитивной тем, кто знаком в основном с графическим интерфейсом, внешность обманчива: это один из наиболее мощных инструментов в арсенале разработчика. Действительно, крайне редко удается увидеть рабочий стол опытного разработчика, на котором не было бы открыто несколько окон терминала с командной оболочкой.

Вообще говоря, это очень глубокая тема, но для следования за примерами в этой книге понадобится лишь несколько наиболее часто используемых команд, перечисленных в табл. 1.1. Для более глубокого изучения командной строки Unix читайте «Conquering the Command Line»[2] (Покоряя командную строку) Марка Бейтса (доступны бесплатная онлайн-версия и электронная книга с видеоуроками[3]).

---

1 https://ru.wikipedia.org/wiki/Bash.

2 http://conqueringthecommandline.com/book.

3 http://conqueringthecommandline.com/#pricing.

Следующий шаг в локальной и облачной системе – создание первого приложения, как показано в листинге 1.3. Обратите внимание, что в нем явно обозначен номер версии Rails (`_4.2.0_`) в виде части команды. Это гарантирует использование той же версии Rails для создания файловой структуры первого приложения, что была установлена в листинге 1.1. (Если команда из листинга 1.3 вернет ошибку, например `Could not find 'railties'` (Не могу найти 'railties'), значит, установлена неверная версия Rails, и необходимо перепроверить, была ли команда из листинга 1.1 выполнена в точности, как написано.)

Таблица 1.1 ❖ Некоторые часто используемые команды Unix

Описание	Команда	Пример
Список содержимого	`ls`	`$ ls -l`
Создать каталог	`mkdir <имя>`	`$ mkdir workspace`
Перейти в каталог	`cd <имя>`	`$ cd workspace/`
Перейти на один каталог вверх		`$ cd ..`
Перейти в домашний каталог		`$ cd ~ или просто $ cd`
Перейти в каталог внутри домашнего каталога		`$ cd ~/workspace/`
Переместить (переименовать) файл	`mv <источник><место назначения>`	`$ mv README.rdoc README.md`
Скопировать файл	`cp <источник><место назначения>`	`$ cp README.rdoc README.md`
Удалить файл	`rm <файл>`	`$ rm README.rdoc`
Удалить пустой каталог	`rmdir <каталог>`	`$ rmdir workspace/`
Удалить непустой каталог	`rm -rf <каталог>`	`$ rm -rf tmp/`
Объединить и показать содержимое файла	`cat <файл>`	`$ cat ~/.ssh/id_rsa.pub`

Листинг 1.3 ❖ Запуск railsnew (с указанием номера версии).

```
$ cd ~/workspace
$ rails _4.2.0_ 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
```

В конце листинга 1.3 видно, что реализация `rails new` автоматически запускает команду `bundle install` по завершении создания файлов. В разделе 1.3.1 мы детально обсудим, что это значит.

Таблица 1.2 ❖ Обзор начальной структуры Rails-каталогов

Файл/Каталог	Назначение
app/	Основной код приложения, включает модели, представления, контроллеры и вспомогательные функции
app/assets	Ресурсы приложения, такие как каскадные таблицы стилей (CSS), JavaScript-файлы и изображения
bin/	Двоичные выполняемые файлы
config/	Конфигурация приложения
db/	Файлы базы данных
doc/	Документация для приложения
lib/	Модули библиотек
lib/assets	Библиотека ресурсов приложения, таких как каскадные таблицы стилей (CSS), JavaScript-файлы и изображения
log/	Файлы журналов приложения
public/	Общедоступные данные (например, через браузер), такие как страницы с описанием ошибок
bin/rails	Программа для автоматического создания программного кода, открытия консольных сеансов или запуска локального веб-сервера
test/	Тесты приложения
tmp/	Временные файлы
vendor/	Код сторонних разработчиков, например расширения и гемы

[1] http://rack.github.io/.

***Таблица 1.2** (окончание)*

Файл/Каталог	Назначение
vendor/assets	Сторонние ресурсы, такие как каскадные таблицы стилей (CSS), JavaScript-файлы и изображения
README.rdoc	Краткое описание приложения
Rakefile	Служебные задачи для выполнения командой `rake`
Gemfile	Гемы, необходимые приложению
Gemfile.lock	Список гемов, обеспечивающий использование одинаковых версий гемов всеми копиями приложения
config.ru	Файл конфигурации для Rack[1]
.gitignore	Шаблоны файлов, которые должны игнорироваться Git

Отметьте, как много файлов и каталогов создает команда `rails`. Эта стандартная структура файлов и каталогов (рис. 1.4) является одним из многих преимуществ Rails – немедленный переход от нуля к (минимально) действующему приложению. Кроме того, так как эта структура является общей для всех приложений Rails, вы легко будете ориентироваться в чужом коде. Обзор типовых Rails-файлов представлен в табл. 1.2; с большинством из них мы познакомимся в остальной части книги. В частности, в разделе 5.2.1 мы обсудим каталог `app/assets`, часть конвейера ресурсов, который значительно упрощает организацию и развертывание активно используемых файлов (ресурсов), таких как каскадные таблицы стилей и JavaScript-файлы.

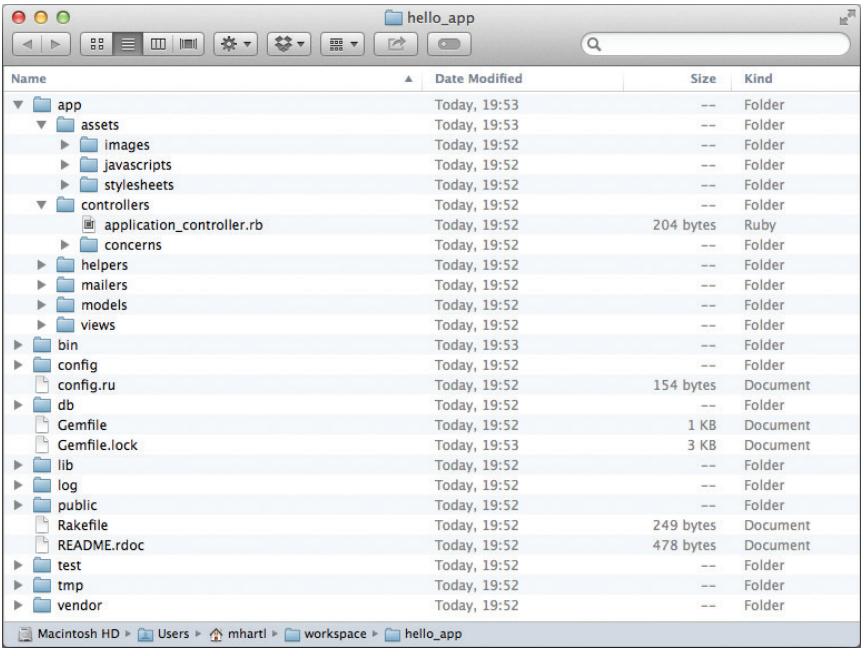


Рис. 1.4 ❖ Структура каталогов вновь созданного Rails-приложения

1.3.1. Компоновщик

Следующий этап после создания нового Rails-приложения – запуск *компоновщика* (bundler) для установки и включения необходимых гемов (пакетов). Как отмечалось в разделе 1.3, компоновщик автоматически запускается командой `rails` (как `bundle install`), но в этом разделе мы немного изменим комплект гемов приложения и вновь запустим компоновщик. Для этого нужно открыть файл `Gemfile` в текстовом редакторе. (В облачной IDE распахните дерево каталогов приложения в панели навигатора и дважды щелкните на файле `Gemfile`.) Некоторые детали и номера версий могут немного отличаться, но в целом результат должен выглядеть как на рис. 1.5 и в листинге 1.4. (Этот файл содержит программный код на языке Ruby, но не обращайте пока внимания на синтаксис; мы подробно поговорим о Ruby в главе 4.) Если файлы и каталоги выглядят иначе, чем на рис. 1.5, щелкните на значке шестеренки в навигаторе файлов и выберите пункт **Refresh File Tree** (Обновить дерево файлов). (Как правило, это необходимо делать всякий раз, когда дерево файлов выглядит не так, как ожидается.)

Листинг 1.4 ❖ Начальный Gemfile в каталоге hello_app

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

# Включить новую версию Rails: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.0'
# Использовать sqlite3 как базу данных для Active Record
gem 'sqlite3'
# Использовать SCSS для таблиц стилей
gem 'sass-rails', '~> 5.0.1'
# Использовать Uglifier для сжатия JavaScript
gem 'uglifier', '>= 1.3.0'
# Использовать CoffeeScript для ресурсов .js.coffee и представлений
gem 'coffee-rails', '~> 4.0.0'
# Дополнительную информацию о других средах выполнения ищите
# по адресу: https://github.com/sstephenson/execjs#readme

# Использовать jquery как библиотеку JavaScript
gem 'jquery-rails'
# Turbolinks ускоряет следование по ссылкам. Дополнительную информацию
# ищите по адресу: https://github.com/rails/turbolinks
gem 'turbolinks'
# Упрощает сборку JSONAPI. Дополнительную информацию
# ищите по адресу: 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

# Использовать Active Model has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Использовать Unicorn как сервер приложений
# gem 'unicorn'
```

```
# Использовать Capistrano для развертывания
# gem 'capistrano-rails', group: :development

group :development, :testdo
  # Позволяет вызвать 'debugger' в любом месте в коде, чтобы остановить
  # выполнение и открыть консоль отладчика
  gem 'byebug'

  # Доступ к консоли IRB на странице exceptions и /console в разработке
  gem 'web-console', '~> 2.0.0.beta2'

  # Увеличивает скорость разработки, сохраняя приложение запущенным в фоне.
  # Подробности по адресу: https://github.com/rails/spring
  gem 'spring'
end
```

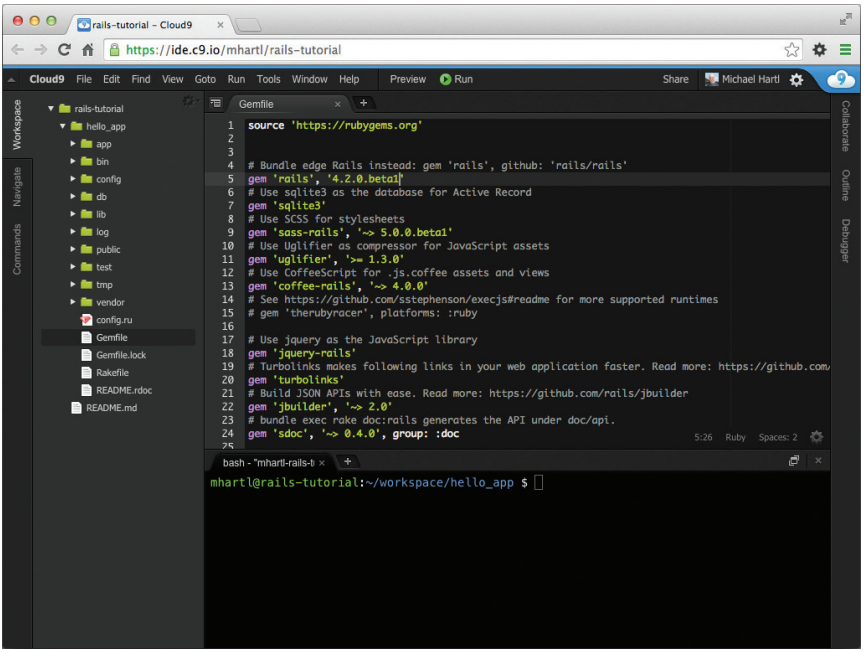


Рис. 1.5 ❖ Начальный Gemfile в текстовом редакторе

Многие строки в файле закомментированы символом #; их цель – показать некоторые часто используемые гемы, а также примеры синтаксиса компоновщика. Сейчас нам не понадобятся никакие новые гемы, кроме используемых по умолчанию.

Если не указать номер версии в команде `gem`, компоновщик установит самую последнюю, например:

```
gem 'sqlite3'
```

Есть два основных способа определения диапазона версий гема, позволяющих явно указать, что именно будет использовать Rails. Первый:

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

В этом случае будет установлена самая последняя версия гема `uglifier` (осуществляет сжатие файлов ресурсов) не ниже версии 1.3.0, даже если это будет версия 7.2. Второй метод:

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

Эта команда установит гем `coffee-rails` не ниже версии `4.0.0` и *не* выше `4.1`. Другими словами, запись `>=` всегда устанавливает самый последний гем, тогда как `~> 4.0.0` установит только незначительное обновление (например, с `4.0.0` до `4.0.1`), но не позволит установить значительное обновление (например, с `4.0` до `4.1`). К сожалению, практика показывает, что даже незначительные обновления гемов могут нарушить совместимость, поэтому в данной книге мы будем действовать осторожно, явно прописывая конкретные версии для всех гемов. Вы можете использовать самую последнюю версию любого гема, использовав конструкцию `~>` в файле `Gemfile` (однако я рекомендую этот шаг только опытным пользователям), но имейте в виду, что это может привести к совершенно непредсказуемому поведению примеров из этой книги.

Прописав в `Gemfile` из листинга 1.4 точные версии гемов, мы получим файл, показанный в листинге 1.5. Обратите внимание, что здесь гем `sqlite3` подключается только к окружению разработки и тестирования (раздел 7.1.1) – это предотвращает потенциальные конфликты с базой данных, которую использует фреймворк Heroku (раздел 1.5).

Листинг 1.5 ❖ Gemfile с явно указанными номерами версий гемов Ruby

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

gem 'rails',                '4.2.0'
gem 'sass-rails',           '5.0.1'
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
```

Скорректировав содержимое Gemfile-приложения, как показано в листинге 1.5, установите геммы командой `bundle install`[1]:

[1] Как отмечено в табл. 3.1, слово `install` можно опустить, так как команда `bundle` – это псевдоним для `bundle install`.

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

Команде `bundle install` может потребоваться некоторое время на выполнение, но по ее завершении приложение будет готово к работе.

1.3.2. Сервер Rails

Выполнив команды `rails new` в разделе 1.3 и `bundle install` в разделе 1.3.1, мы получили приложение, готовое к запуску, но как его запустить? Фреймворк Rails поставляется с программой командной строки, или сценарием, который запускает локальный веб-сервер, помогающий в разработке приложений. Точная команда зависит от используемого окружения: на локальной машине можно просто выполнить `rails server` (листинг 1.6), но в Cloud9 требуется дополнительно указать IP-адрес для приема соединений и номер порта, чтобы сервер Rails знал, какой адрес использовать, чтобы сделать приложение видимым для внешнего мира (листинг 1.7)[1]. (В Cloud9 для этого используются специальные переменные окружения `$IP` и `$PORT`. Узнать значения этих переменных можно командой `echo $IP` или `echo $PORT`.)

Листинг 1.6 ❖ Запуск Rails-сервера на локальной машине

```
$ 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
```

Листинг 1.7 ❖ Запуск Rails-сервера в cloudIDE

```
$ 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
```

Независимо от варианта я рекомендую запускать команду `rails server` во второй вкладке терминала, чтобы в первой по-прежнему можно было выполнять команды, как показано на рис. 1.6 и 1.7. (Если вы уже запустили сервер в первой вкладке, нажмите **Ctrl-C**, чтобы остановить его[2].) Если вы используете локальный

---

[1] Обычно веб-сайты запускаются на 80-м порте, но для этого часто требуются специальные привилегии, поэтому для сервера разработки нередко используют порты с более высокими номерами.

[2] «C» обозначает клавишу на клавиатуре, а не заглавную букву, поэтому не нужно нажимать еще и **Shift**.

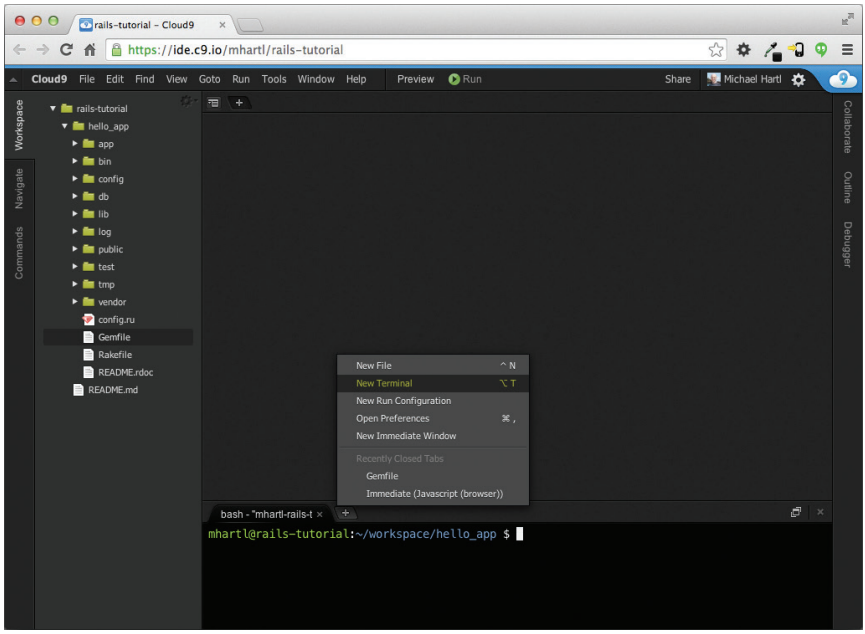


Рис. 1.6 ❖ Открытие новой вкладки терминала

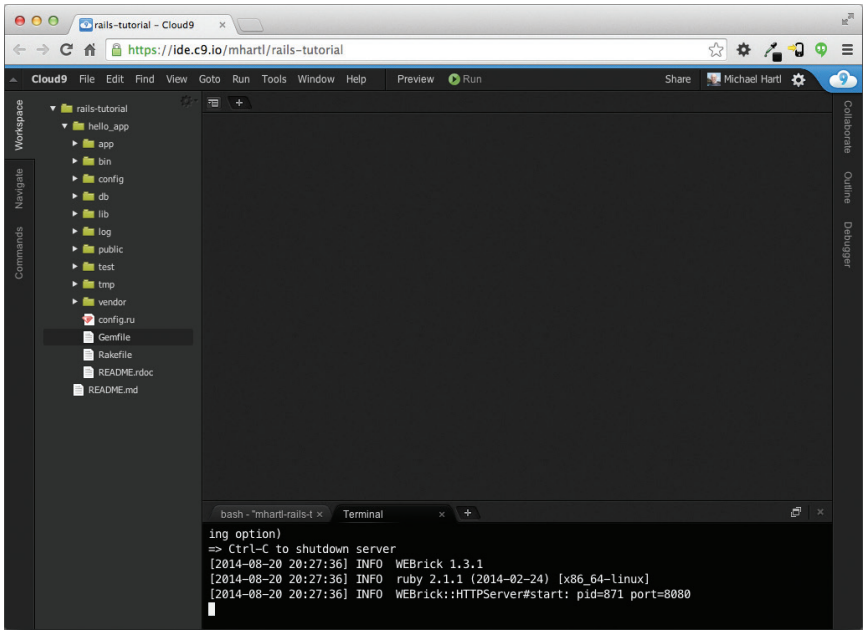


Рис. 1.7 ❖ Запуск Rails-сервера в отдельной вкладке

сервер, наберите в браузере адрес http://localhost:3000/; в облачной IDE щелкните на ссылке **Share** (Пригласить) и затем в открывшемся диалоге щелкните в поле **Application address** (Адрес приложения, см. рис. 1.8). В любом случае, результат должен быть похожим на рис. 1.9.

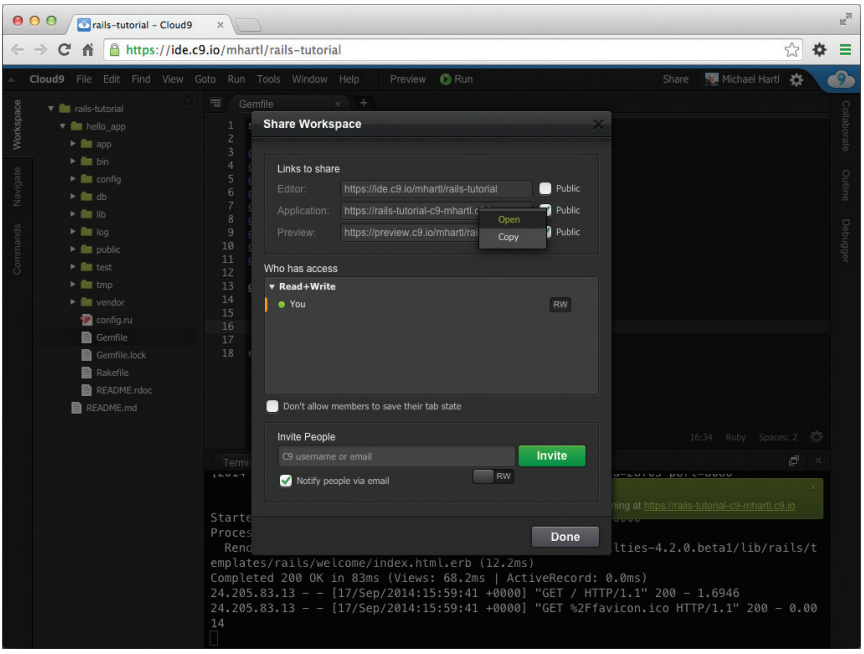


Рис. 1.8 ❖ Вызов страницы приложения в облачном рабочем пространстве

Чтобы увидеть информацию о нашем первом приложении, щелкните на ссылке **About your application's environment** (Об окружении вашего приложения). Хотя конкретные номера версий могут отличаться, в целом результат должен напоминать изображение на рис. 1.10. Очевидно, что в дальнейшем начальная страница Rails нам не понадобится, но сейчас приятно видеть ее работающей. Мы удалим ее (и заменим ее своей) в разделе 1.3.4.

1.3.3. Модель-представление-контроллер (MVC)

Даже на этой ранней стадии полезно получить общее представление о том, как работают Rails-приложения (рис. 1.11). Возможно, вы заметили в стандартной структуре Rails-приложения (рис. 1.4) каталог `app/` с тремя подкаталогами: `models`, `views` и `controllers`. Это намек, что Rails следует архитектурной схеме *модель-представление-контроллер* (Model-Miew-Controller, MVC), которая отделяет «логику предметной области» (или «бизнес-логику») от логики ввода и логики представления, связанной с графическим интерфейсом пользователя (GUI). В случае

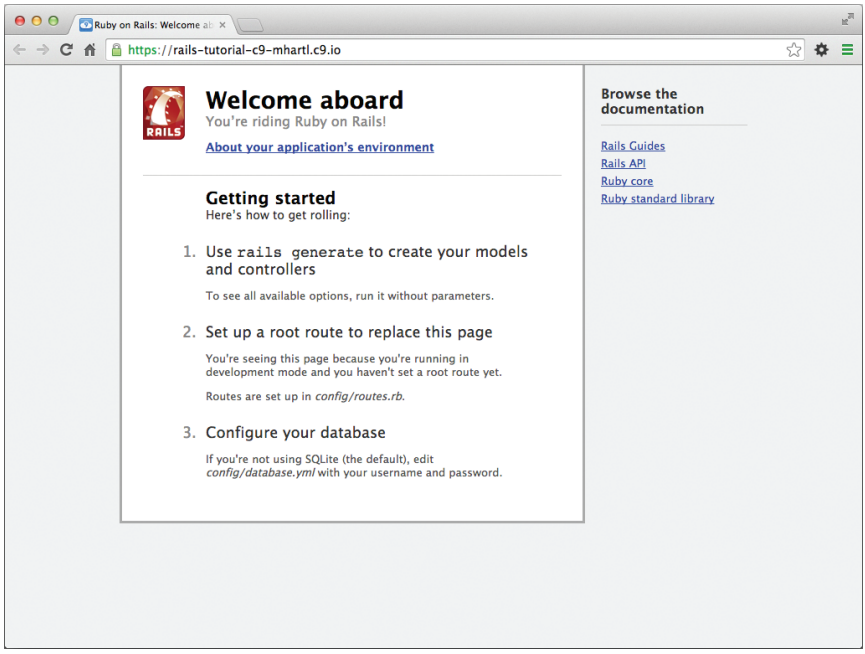


Рис. 1.9 ❖ Начальная Rails-страница по умолчанию

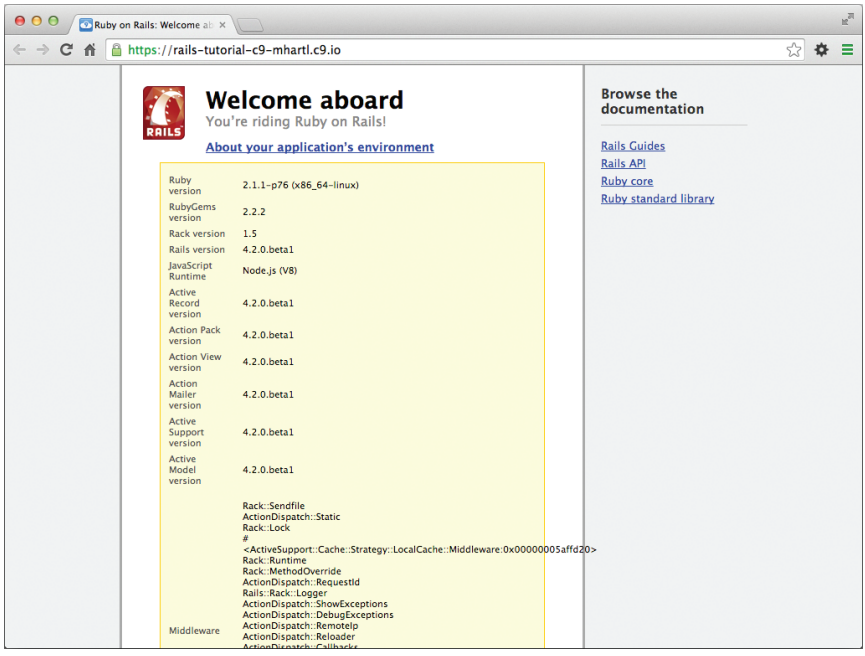


Рис. 1.10 ❖ Начальная страница с информацией об окружении приложения

веб-приложений «логика предметной области» обычно состоит из модели данных, представляющих пользователей, статьи, продукты, а GUI – это просто веб-страница в браузере.

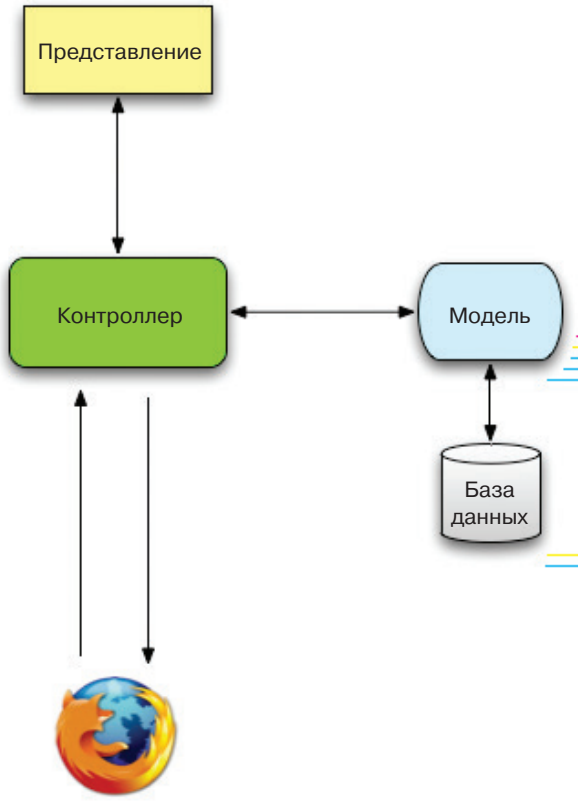


Рис. 1.11 ❖ Схематичное изображение архитектуры модель-представление-контроллер (MVC)

Взаимодействуя с приложением Rails, браузер посылает запрос, веб-сервер принимает его и передает контроллеру Rails, отвечающему за дальнейшую обработку. Иногда контроллер сразу отображает представление – шаблон, который преобразуется в разметку HTML и возвращается браузеру. В динамических сайтах гораздо чаще контроллер взаимодействует с моделью – объектом Ruby, который представляет элемент сайта (например, пользователя) и отвечает за связь с базой данных. После вызова модели контроллер отображает представление и возвращает браузеру готовую веб-страницу с разметкой HTML.

Если это обсуждение сейчас кажется вам немного абстрактным, не беспокойтесь; мы еще не раз будем возвращаться к этому разделу. В разделе 1.3.4 мы сделаем первую попытку применить MVC, но уже в разделе 2.2.2 обсудим MVC более детально в контексте мини-приложения. Наконец, в процессе создания учебного

приложения будут использоваться все аспекты MVC; мы охватим контроллеры и представления в разделе 3.2, модели – в разделе 6.1 и увидим совместную работу всей этой троицы в разделе 7.1.2.

1.3.4. Hello, world!

В качестве первого применения структуры MVC внесем тончайшие изменения в первое приложение, добавив метод контроллера, возвращающего строку «hello, world!». (Больше о методах действий контроллеров рассказывается в разделе 2.2.2.) В результате мы заменим начальную страницу Rails, изображенную на рис. 1.9, страницей «hello, world» – это и есть цель данного раздела.

Как можно догадаться, методы контроллеров определяются внутри контроллеров. Наш первый метод мы назовем `hello` и поместим в контроллер `ApplicationController` (контроллер приложения). В действительности сейчас это единственный контроллер, который есть в наличии, и в этом можно убедиться, выполнив команду:

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

(Мы начнем создавать собственные контроллеры в главе 2.) В листинге 1.8 показано определение метода hello, который использует функцию render, чтобы вернуть текст «hello, world!». (Не беспокойтесь сейчас о синтаксисе языка Ruby, мы разберемся с ним в главе 4.)

Листинг 1.8 ❖ Добавление метода действия hello в контроллер приложения (app/controllers/application_controller.rb)

```
class ApplicationController < ActionController::Base
  # Предотвратить атаки вида "подделка межсайтовых запросов" (CSRF),
  # возбудив исключение. Для библиотек, возможно, предпочтительнее будет
  # использовать :null_session instead.
  protect_from_forgery with: :exception

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

Определив метод, возвращающий нужную строку, необходимо сказать Rails, что именно он должен использоваться вместо вывода начальной страницы, изображенной на рис. 1.9. Для этого настроим *маршрутизатор* Rails, находящийся перед контроллером на рис. 1.11, и определим, куда посылать запросы, поступающие от браузера. (Для простоты я опустил маршрутизатор на рис. 1.11, но мы еще вернемся к вопросу маршрутизации в разделе 2.2.2.) В частности, нужно изменить начальную страницу, *корневой маршрут*, именно он определяет страницу, возвращаемую в ответ на обращение к *корневому URL*. Так как корневые URL совпадают с такими адресами, как http://www.example.com/ (когда за слешем ничего не следует), их часто для краткости называют / («слеш»).

Как видно в листинге 1.9, файл маршрутов (config/routes.rb) содержит закомментированную строку, в которой описывается, как построить корневой маршрут. Здесь «welcome» – это имя контроллера,«index» – метод внутри этого контроллера. Чтобы активировать корневой маршрут, раскомментируйте строку, удалив символ решетки, и замените кодом из листинга 1.10, который требует от Rails при обращении к корневому маршруту вызвать метод hello контроллера ApplicationController. (Как было отмечено в разделе 1.1.2, вертикальные точки обозначают пропущенный код, их не нужно копировать буквально.)

Листинг 1.9 ❖ Начальный (закомментированный) корневой маршрут (config/routes.rb)

```
Rails.application.routes.draw do
  .
  .
  .
  # Определить корневой маршрут к сайту можно как "root"
  # root 'welcome#index'
  .
  .
  .
end
```

Листинг 1.10 ❖ Установка корневого маршрута (config/routes.rb)

```
Rails.application.routes.draw do
  .
  .
  .
  # Определить корневой маршрут к сайту можно как "root"
  root 'application#hello'
  .
  .
  .
end
```

После изменений, показанных в листингах 1.8 и 1.10, корневой маршрут будет возвращать «hello, world!», как и требовалось (рис. 1.12).

1.4. Управление версиями с Git

Теперь, после создания нового и работающего приложения, воспользуемся моментом и сделаем то, что видится многим разработчикам Rails как практически необходимый шаг (хотя технически это необязательно), – поместим исходный код приложения в *систему управления версиями*. Системы управления версиями позволяют отслеживать изменения в коде проекта, облегчают совместную работу, а также могут откатывать любые непреднамеренные погрешности (такие как случайное удаление файлов). Грамотное использование системы управления вер-

сиями – необходимый навык для каждого профессионального разработчика программного обеспечения.

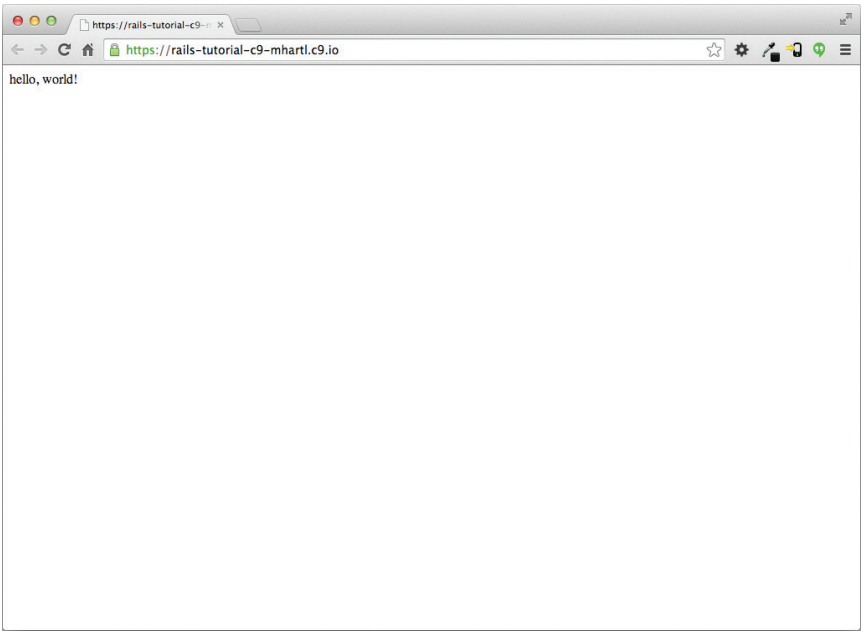


Рис. 1.12 ❖ Отображение «hello, world!» в браузере

Существует множество инструментов управления версиями, но сообщество Rails в значительной степени ориентировано на Git[1], распределенную систему управления версиями, первоначально разработанную Линусом Торвальдсом (Linus Torvalds) для хранения исходных текстов ядра Linux. Git – обширная тема, и мы лишь слегка коснемся ее в этой книге, но в Интернете можно найти много хороших бесплатных ресурсов; я особенно рекомендую краткий обзор «Bitbucket 101»[2] и книгу «Pro Git»[3] Скотта Чакона (Scott Chacon) для более подробного изучения. Помещение исходного кода в систему управления версиями Git *строго* рекомендуется не только потому, что это почти повсеместная практика в мире Rails, но также и потому, что это позволит легко создать резервную копию или открыть доступ к вашему коду (раздел 1.4.3), а еще развернуть приложение прямо здесь, в первой главе (раздел 1.5).

---

1 https://git-scm.com/.

2 https://confluence.atlassian.com/display/BITBUCKET/Clone+your+Git+repository+and+add+source+files.

3 http://git-scm.com/book/ru/v1.

1.4.1. Установка и настройка

Облачная IDE, рекомендованная в разделе 1.2.1, по умолчанию уже включает поддержку Git, то есть в этом случае нет необходимости устанавливать ее. В противном случае обращайтесь на сайт InstallRails.com (раздел 1.2), где приводятся инструкции по установке Git.

Первоначальная настройка системы

Прежде чем использовать Git, следует выполнить ряд первоначальных настроек. Это *системные* настройки, а значит, их необходимо сделать лишь единожды:

```
$ 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
```

Имейте в виду, что ваши имя и адрес электронной почты, указанные в настройках Git, будут доступны в любом вашем репозитории, который вы сделаете общедоступным. (Строго говоря, необходимы только первые две строки. Третья строка обеспечивает совместимость с будущими версиями Git. Четвертая – позволяет использовать команду `co` вместо более длинной `checkout`. Для максимальной совместимости с системами, где команда `co` не настроена, в книге будет использоваться полная команда `checkout`, но в жизни я почти всегда пишу `git co`.)

Первоначальная настройка репозитория

Сейчас мы разберем этапы создания каждого нового *репозитория* (иногда его называют *репо* для краткости). Сначала перейдите в корневой каталог первого приложения и инициализируйте новый репозиторий:

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

Затем добавьте все файлы проекта в репозиторий командой `git add -A`:

```
$ git add -A
```

Эта команда добавит все файлы из текущего каталога, кроме тех, что соответствуют шаблонам (правилам) в специальном файле `.gitignore`. Команда `rails new` автоматически создает файл `.gitignore`, соответствующий Rails-проекту, но в него также можно добавить дополнительные правила[1].

Добавленные файлы сначала помещаются в *промежуточную область*, где находятся незавершенные изменения в проекте. Посмотреть, какие файлы находятся там, можно с помощью команды `status`:

---

[1] В этой книге нам не понадобится редактировать его, тем не менее пример добавления правила в файл `.gitignore` будет показан в разделе 3.7.3 как часть дополнительных настроек тестирования.

```
$ 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
  .
  .
  .
```

(Результат очень длинный, поэтому здесь вместо большей части текста показаны вертикальные точки.)

Чтобы сообщить Git о необходимости сохранения изменений, используйте команду `commit`:

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

Параметр `-m` позволяет добавить сообщение для фиксации; если опустить его, Git откроет редактор, установленный в системе по умолчанию, и предложит ввести сообщение в нем. (Во всех примерах в книге будет использован флаг `-m`.)

Важно отметить, что фиксации Git *локальны*, они записываются только на машине, на которой сделаны. В разделе 1.4.4 мы разберем, как отправить изменения в удаленный репозиторий (командой `git push`).

Кстати, список фиксаций можно вызвать командой `log`:

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

    Initializerepository
```

В зависимости от длины истории репозитория может понадобиться ввести `q` для выхода.

1.4.2. Что дает использование репозитория Git?

Если раньше вы никогда не использовали систему управления версиями, вам может быть не очень понятно, зачем это нужно, поэтому позвольте мне привести всего один пример. Предположим, вы произвели некоторые случайные изменения, например (О, нет!) удалили крайне необходимый каталог `app/controllers/`.

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

Здесь для вывода содержимого каталога app/controllers/ использована Unix-команда ls, а для его удаления – команда rm (табл. 1.1). Параметр -rf означает «recursive force» («рекурсивно и принудительно») и обеспечивает рекурсивное удаление всех файлов, каталогов, подкаталогов и т. д., без дополнительного подтверждения.

Давайте проверим статус и посмотрим, что изменилось:

```
$ 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")
```

Как видите, файл был удален, но изменилось только «рабочее дерево»; изменения еще не зафиксированы. Это означает, что мы можем легко отменить изменения командой checkout с параметром -f для принудительной отмены текущих изменений:

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

Пропавшие каталоги и файлы вернулись. Какое облегчение!

1.4.3. Bitbucket

Теперь, когда проект помещен в систему управления версиями Git, пришло время отправить его на сайт Bitbucket[1], оптимизированный для хостинга и совместного использования Git-репозиториев. (В предыдущем издании вместо него использовался GitHub[2]; причины замены разъясняются в блоке 1.4.) Отправка копии Git-репозитория на сайт Bitbucket служит двум целям: полному резервному копированию программного кода (включая полную историю фиксаций) и простоте совместной разработки в будущем.

---

1 https://bitbucket.org/.

2 https://github.com/.

Блок 1.4 ❖ GitHub и Bitbucket

GitHub и Bitbucket – наиболее популярные сайты для хостинга Git-репозиториев. Они во многом похожи: оба позволяют размещать репозитории Git и использовать их для коллективной разработки, а также предоставляют удобные средства просмотра и поиска в репозиториях. Самое главное отличие (с точки зрения этой книги) в том, что GitHub бесплатно предлагает неограниченное количество репозиториев с открытым исходным кодом (с возможностью совместной работы), но берет плату за закрытые (частные) репозитории, в то время как Bitbucket позволяет бесплатно завести неограниченное количество закрытых репозиториев, но при этом ограничивает число совместно работающих людей (увеличить его можно за плату). Таким образом, выбор службы для конкретного репозитория зависит от ваших потребностей.
В предыдущем издании этой книги использовался GitHub из-за акцента на поддержку открытого исходного кода, но все возрастающая обеспокоенность по поводу безопасности привела меня к тому, что я рекомендую *все* репозитории веб-приложений делать закрытыми по умолчанию. Дело в том, что репозитории для веб-приложений могут содержать конфиденциальную информацию, такую как криптографические ключи или пароли, которая может быть использована для компрометации сайтов, выполняющих этот код. Конечно, можно организовать безопасную обработку информации (путем настроек игнорирования Git, например), но эти методы сложны и требуют значительного опыта.
На самом деле учебное приложение из этой книги вполне безопасно, чтобы выложить его в сеть, но не стоит всегда полагаться на этот факт. То есть для максимальной безопасности мы будем действовать с предельной осмотрительностью и по умолчанию использовать закрытые репозитории. Поскольку GitHub берет плату за закрытость, а Bitbucket предлагает неограниченное число закрытых репозиториев бесплатно, для наших целей Bitbucket подходит лучше, чем GitHub.

Начать работу с Bitbucket очень просто:

1. Создайте учетную запись на Bitbucket[1], если у вас ее еще нет.
2. Скопируйте свой *открытый ключ*[2] в буфер обмена. Пользователи облачной IDE могут увидеть его, выполнив команду `cat`, как показано в листинге 1.11, после чего его можно выделить и скопировать. Если вы используете другую систему и запуск команды из листинга 1.11 не дает никакого результата, тогда следуйте инструкции, описывающей установку открытого ключа в учетную запись Bitbucket[3].
3. Добавьте свой открытый ключ в Bitbucket, щелкнув на аватаре в правом верхнем углу и выбрав **Manage account** (Управление учетной записью), а затем **SSH keys** (SSH-ключи, см. рис. 1.13).

Листинг 1.11 ❖ Вывод открытого ключа командой cat

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

[1] https://bitbucket.org/account/signup/.

[2] https://ru.wikipedia.org/wiki/Криптосистема_с_открытым_ключом.

[3] Инструкция на русском языке: https://bitbucket.org/rtnm/gittertutorial/src.

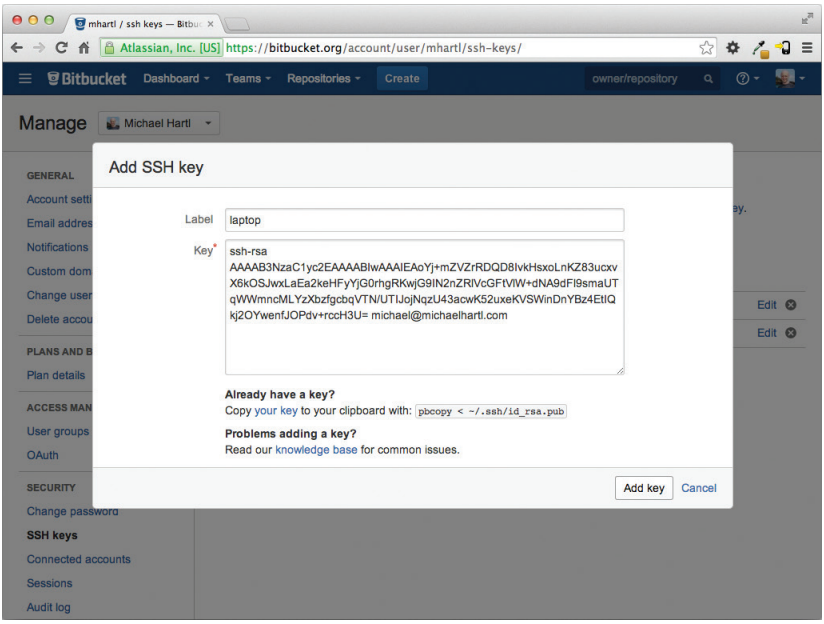


Рис. 1.13 ❖ Добавление открытого SSH-ключа

После добавления ключа щелкните на ссылке **Create** (Создать), чтобы создать новый репозиторий, как показано на рис. 1.14. Заполнив информацию о проекте, не забудьте установить флажок **This is a private repository** (Закрытый репозиторий). После щелчка на ссылке **Create repository** (Создать репозиторий) следуйте инструкциям в разделе **Command line ⇨ I have an existing project** (У меня уже есть проект), они должны выглядеть примерно как листинг 1.12. (Если они выглядят иначе, скорее всего, открытый ключ не был добавлен, и стоит попробовать добавить его еще раз.) При отправке репозитория ответьте **yes** на вопрос **Are you sure you want to continue connecting (yes/no)?** (Вы уверены, что хотите продолжить подключение?).

Листинг 1.12 ❖ Добавление Bitbucket и отправка репозитория

```
$ git remote add origin git@bitbucket.org:<username>/hello_app.git
$ git push -u origin --all        # отправляет репозиторий в первый раз
```

Команды в листинге 1.12 сообщают Git, что вы хотите добавить Bitbucket в качестве *источника* (origin) для вашего репозитория, а затем отправляют репозиторий в удаленный источник. (Не беспокойтесь о значении флага `-u`; если вам любопытно, поищите в сети по фразе: «git set upstream».) Конечно, следует заменить `<username>` фактическим именем пользователя. Например, команда, которую запустил я:

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

В результате действий, описанных выше, на сайте Bitbucket появится страница для репозитория `hello_app` с браузером файлов, полной историей фиксаций и большим количеством прочих приятных мелочей (рис. 1.15).

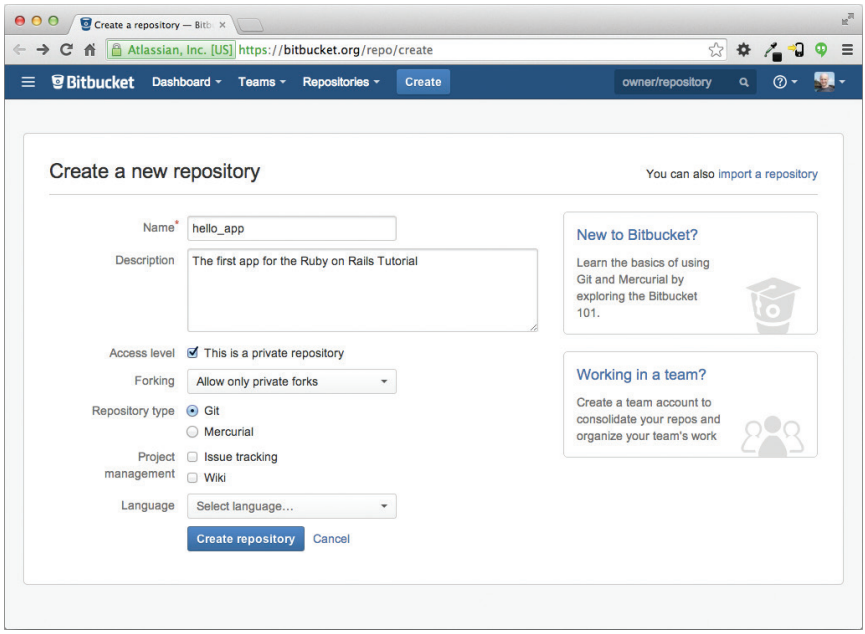


Рис. 1.14 ❖ Создание первого репозитория на сайте Bitbucket

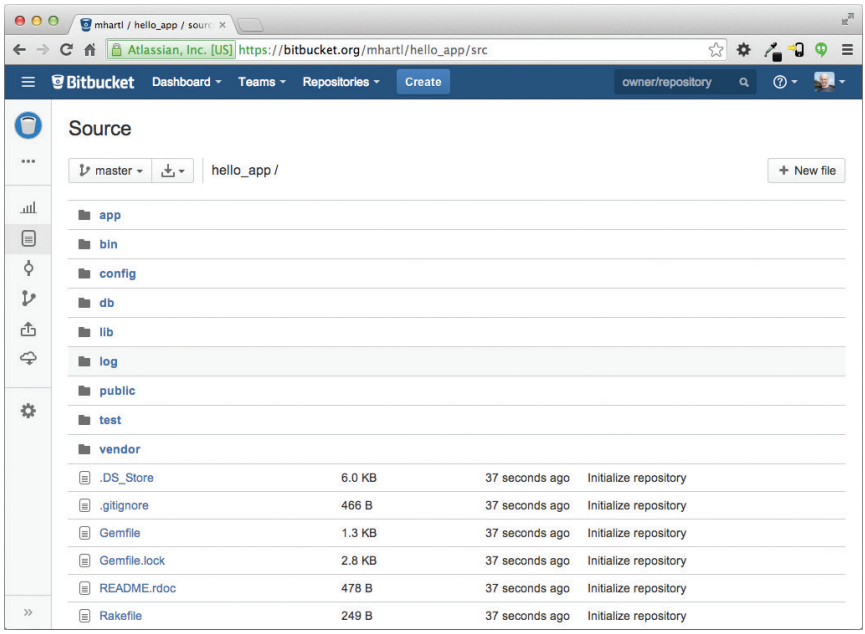


Рис. 1.15 ❖ Страница репозитория на сайте Bitbucket

1.4.4. Ветвление, редактирование, фиксация, слияние

Прошедшие все этапы в разделе 1.4.3 могли заметить, что Bitbucket не создает автоматически файл `README.rdoc` в репозитории, вместо этого он выводит на главной странице сообщение о его отсутствии (рис. 1.16). Это указывает на недостаточно широкую распространенность формата `rdoc`, чтобы Bitbucket поддерживал его автоматически, и на самом деле я и практически все известные мне разработчики предпочитают вместо него использовать формат *Markdown*. В этом разделе мы заменим файл `README.rdoc` на `README.md`, а также добавим в него описание своего проекта. В процессе мы увидим первый пример ветвления, редактирования, фиксации и слияния – именно такой рабочий процесс я рекомендую использовать в работе с Git[1].

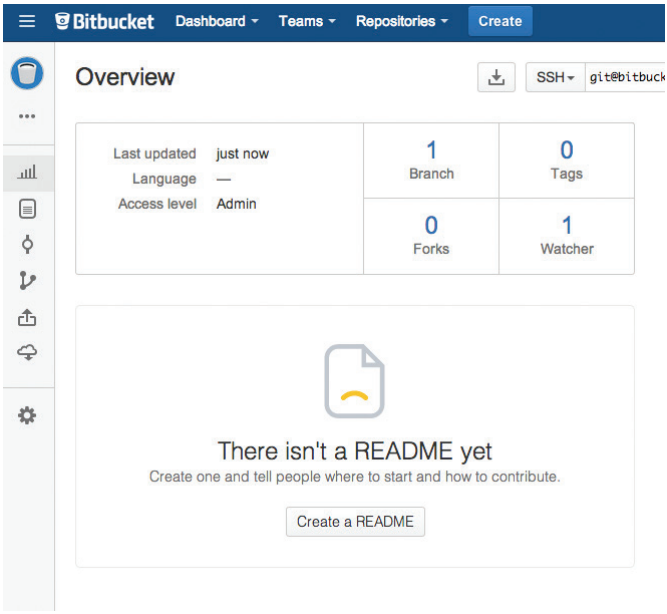


Рис. 1.16 ❖ Сообщение Bitbucket об отсутствии файла README

Ветвление

Git невероятно хорош в создании *ветвей*, которые фактически являются копиями репозитория, где можно производить изменения (возможно, экспериментальные), не модифицируя родительских файлов. В большинстве случаев родительский репозиторий – это *главная ветвь* (master), но можно создать новую рабочую, или тематическую, ветвь, выполнив команду `checkout` с флагом `-b`:

[1] Для удобства работы с Git-репозиториями я рекомендую использовать приложение Source Tree от Atlassian (описание на русском языке можно найти по адресу: http://www.teamlead.ru/pages/viewpage.action?pageId=112263556).

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

Вторая команда, `git branch`, просто перечисляет локальные ветви, а звездочкой * отмечена текущая ветвь. Обратите внимание, `git checkout -b modify-README` одновременно создает новую ветвь и делает ее текущей, на что указывает звездочка перед `modify-README`. (Если вы настроили команду `co` в разделе 1.4, вместо этого можете писать `git co -b modify-README`.)

Полностью оценить достоинства ветвления можно, только работая над проектом в составе коллектива[1], но ветви полезны даже для единственного разработчика, как в нашем случае. В частности, главная ветвь (master) изолируется от любых изменений в рабочих ветвях, поэтому, даже если мы *действительно* наворотим лишнего, всегда можно отказаться от изменений, вернувшись в главную ветвь и удалив рабочую. Мы увидим, как это делается, в конце раздела.

Между прочим, для столь малых изменений я бы не стал озадачиваться новой ветвью, но никогда не бывает слишком рано, чтобы начать практиковать хорошие привычки.

Редактирование

После создания локальной ветви отредактируем файл с описанием. Я предпочитаю использовать *язык разметки Markdown*[2] для этих целей, и если вы используете расширение файла `.md`, Bitbucket будет автоматически форматировать его. Итак, сначала запустим Git-версию Unix-команды `mv`, чтобы изменить имя файла:

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

Затем заполним `README.md` содержимым из листинга 1.13.

Листинг 1.13. Новый файл README, README.md

```
# Ruby on Rails для начинающих: "hello, world!"
Это первое приложение для
[*RubyonRailsTutorial*](http://www.railstutorial.org/)
[Майкл Хартл](http://www.michaelhartl.com/).
```

Фиксация

После внесения изменений можно взглянуть на статус ветки:

```
$ git status
On branch modify-README
Changes to be committed:
```

---

[1] Более подробно в главе «Ветвление в Git» – в книге «Pro Git» (https://git-scm.com/book/ru/v1/Ветвление_в_Git).

[2] https://ru.wikipedia.org/wiki/Markdown.

```
  (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
```

Сейчас мы могли бы использовать `git add -A`, как в разделе 1.4.1, но `git commit` предусматривает флаг `-a` для (очень частого) случая фиксации всех изменений существующих файлов (или файлов, созданных с использованием `git mv`, которые не считаются новыми для 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
```

Будьте осторожны, используя флаг `-a`; добавив новые файлы в проект после последней фиксации, вы должны сообщить Git о них, выполнив сначала `git add -A`.

Обратите внимание, что мы написали сообщение в настоящем времени (и, технически говоря, повелительном наклонении). Git моделирует фиксации как серии правок существующего кода, и в этом контексте имеет смысл описать, что каждая фиксация делает, а не что делала. Кроме того, такое использование соответствует сообщениям о фиксациях, генерируемым самой командой Git. Более подробно об этом можно почитать в статье «Блестящий новый стиль фиксаций»[1].

Слияние

Теперь, закончив с изменениями, надо слить рабочую ветвь с главной:

```
$ 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
```

Вывод Git часто включает такие строки, как `34f06b7`, которые связаны с внутренним представлением репозиториев Git. Ваши конкретные результаты будут отличаться в этих деталях, но в остальном они должны соответствовать выводу, показанному выше.

---

[1] https://github.com/blog/926-shiny-new-commit-styles.

После слияния изменений можно почистить ветви, удалив рабочую ветвь командой `git branch -d`, если она больше не нужна:

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

Этот шаг не является обязательным, и на практике рабочие ветви часто оставляют нетронутыми. Это позволяет переключаться между рабочей и главной ветвями, сливая их всякий раз, когда достигается естественный конечный пункт.

Как упоминалось выше, можно вообще отказаться от изменений в рабочей ветви командой `git branch -D`:

```
# Исключительно для иллюстрации; пользуйтесь,
# только если совсем все испортили в рабочей ветке
$ git checkout -btopic-branch
$ <действительно все испортили>
$ git add -A
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

В отличие от флага `-d`, флаг `-D` удалит ветку, даже если не было выполнено слияние.

Отправка

После обновления `README` можно отправить изменения в Bitbucket, чтобы увидеть результат. Так как одна отправка уже была сделана (раздел 1.4.3), в большинстве систем после этого можно опустить `origin master` и просто выполнить `git push`:

```
$ git push
```

Как было обещано в разделе 1.4.4, Bitbucket отформатировал новый файл с разметкой Markdown (рис. 1.17).

1.5. Развертывание

На этой ранней стадии мы уже готовы развернуть наше (все еще пустое) Rails-приложение. Этот шаг не является обязательным, но раннее и частое развертывание позволяет раньше обнаруживать проблемы в цикле разработки. Альтернативный вариант – развертывание только после напряженных усилий, в изолированном окружении разработки – часто приводит к неприятностям, когда наступает время запуска[1].

[1] Вообще, это не должно иметь значения для учебных приложений из этой книги, но если вы волнуетесь, что случайно и/или преждевременно опубликуете свое приложение, есть несколько способов, которые могут помочь избежать этого; один из них описан в разделе 1.5.4.

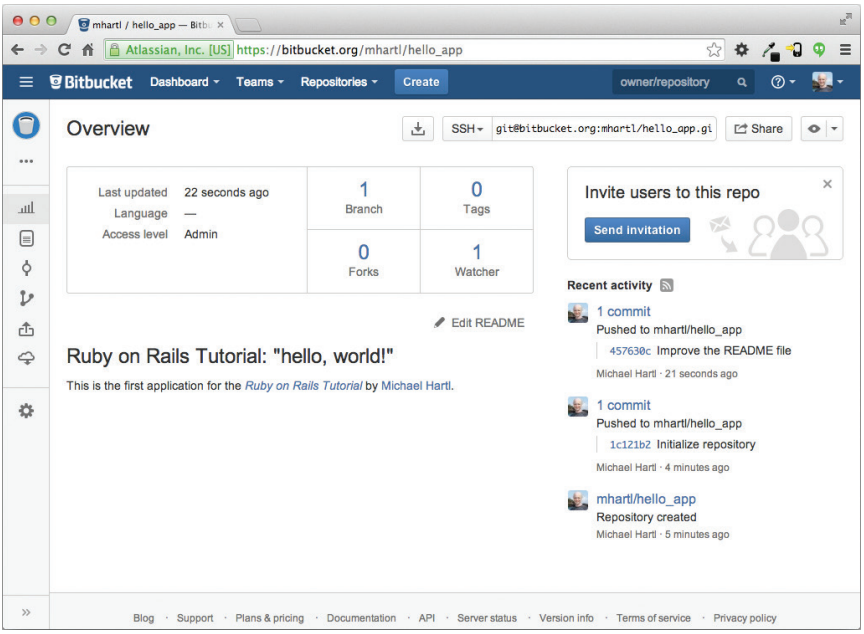


Рис. 1.17 ❖ Улучшенный файл README с разметкой Markdown

Раньше развертывание Rails-приложений было довольно сложной процедурой, но экосистема развертывания в Rails стремительно развивалась в течение нескольких последних лет, и теперь в ней есть несколько замечательных инструментов. Среди них общедоступные или виртуальные частные серверы на основе Phusion Passenger[1] (модуль для веб-серверов Apache и Nginx[2]), компании, предоставляющие полный комплекс услуг развертывания, такие как Engine Yard[3] и Rails Machine[4], и облачные службы развертывания, такие как Engine Yard Cloud[5], Ninefold[6] и Heroku[7].

Я отдаю предпочтение Heroku, хостинговой платформе, созданной специально для развертывания веб-приложений на Rails и других платформах. Heroku делает развертывание Rails-приложений смехотворно простым, если их исходный код хранится в системе управления версиями Git. (Это еще одна причина выполнить шаги установки Git из раздела 1.4, если вы до сих пор этого не сделали.) Кроме того, бесплатной службы Heroku более чем достаточно для выполнения очень

---

1 https://www.phusionpassenger.com/.
2 Произносится как «ин-джин-икс».
3 https://engineyard.com/.
4 https://railsmachine.com/.
5 https://cloud.engineyard.com/.
6 https://ninefold.com/.
7 https://www.heroku.com/.

многих задач, в том числе и для нужд этой книги. Я не заплатил ни цента за хостинг на платформе Heroku первых двух изданий, обработавшей для меня несколько миллионов запросов. Остальная часть раздела посвящена развертыванию нашего первого приложения на Heroku. Некоторые понятия довольно сложны, поэтому не переживайте, если не поймете всех деталей сразу; важно, чтобы в конце концов мы развернули приложение непосредственно в сети.

1.5.1. Установка Heroku

Платформа Heroku использует базу данных PostgreSQL[1] (произносится «пост-грескю-эль», для краткости ее часто называют «Postgres»), а это означает, что в окружение необходимо добавить гем `pg`, чтобы позволить Rails общаться с Postgres[2]:

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

Обратите внимание на дополнительный гем `rails_12factor`, который Heroku использует для работы со статическими ресурсами, такими как изображения и таблицы стилей.

Получившийся файл `Gemfile` показан в листинге 1.14.

Листинг 1.14 ❖ Gemfile с дополнительными гемами

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

gem 'rails',        '4.2.0'
gem 'sass-rails',   '5.0.1'
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'
```

---

[1] https://ru.wikipedia.org/wiki/PostgreSQL.

[2] Вообще говоря, желательно, чтобы окружение разработки максимально соответствовало эксплуатационному окружению. Это касается и баз данных, но для целей данного учебника мы всегда будем использовать SQLite локально и PostgreSQL – для эксплуатации. Дополнительная информация приводится в разделе 3.1.

```
  gem 'rails_12factor', '0.0.2'
end
```

Чтобы подготовить систему к развертыванию в эксплуатационном окружении, нужно выполнить `bundle install` со специальным флагом, предотвращающим локальную установку любых гемов из раздела `production` (в данном случае `pg` и `rails_12factor`):

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

Так как в листинге 1.14 были добавлены только гемы в раздел `production`, прямо сейчас эта команда не установит никаких дополнительных локальных гемов, но она добавит в `Gemfile.lock` гемы `pg` и `rails_12factor`. Зафиксируем полученные изменения:

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

Далее необходимо создать и настроить новую учетную запись Heroku. Для начала следует зарегистрироваться на сайте Heroku[1]. Затем – проверить, установлен ли клиент, командной строкой Heroku:

```
$ heroku version
```

Пользующиеся облачной IDE должны увидеть номер версии Heroku, это говорит о доступности клиента `heroku`, на других системах может потребоваться его установка через Heroku Toolbelt[2].

После установки интерфейса командной строки Heroku нужно выполнить команду `heroku login`, чтобы войти в систему и добавить SSH-ключ:

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

Наконец, командой `heroku create` нужно создать место на серверах Heroku для учебного приложения (листинг 1.15).

Листинг 1.15 ❖ Создание нового приложения на 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
```

Команда `heroku` создаст новый поддомен для приложения, который тут же станет доступным для просмотра. Однако там пока ничего нет, так что давайте займемся развертыванием.

1.5.2. Развертывание на Heroku, шаг первый

Первым шагом для развертывания является отправка главной ветви приложения в Heroku с помощью Git:

---

[1] http://api.heroku.com/signup.

[2] https://toolbelt.heroku.com/.

```
$ git push heroku master
```

(Может появиться несколько предупреждающих сообщений, которые сейчас можно игнорировать. Мы обсудим их ниже, в разделе 7.5.)

1.5.3. Развертывание на Heroku, шаг второй

Нет никакого шага под номером два! Все готово! Чтобы увидеть только что развернутое приложение, перейдите по адресу, который вы видели при выполнении `heroku create` (то есть в листинге 1.15). (Работающие на локальной машине вместо облачной IDE могут выполнить `heroku open`.) Результат показан на рис. 1.18. Страница идентична изображению на рис. 1.12, но теперь она запущена в эксплуатационном окружении – непосредственно в Интернете.

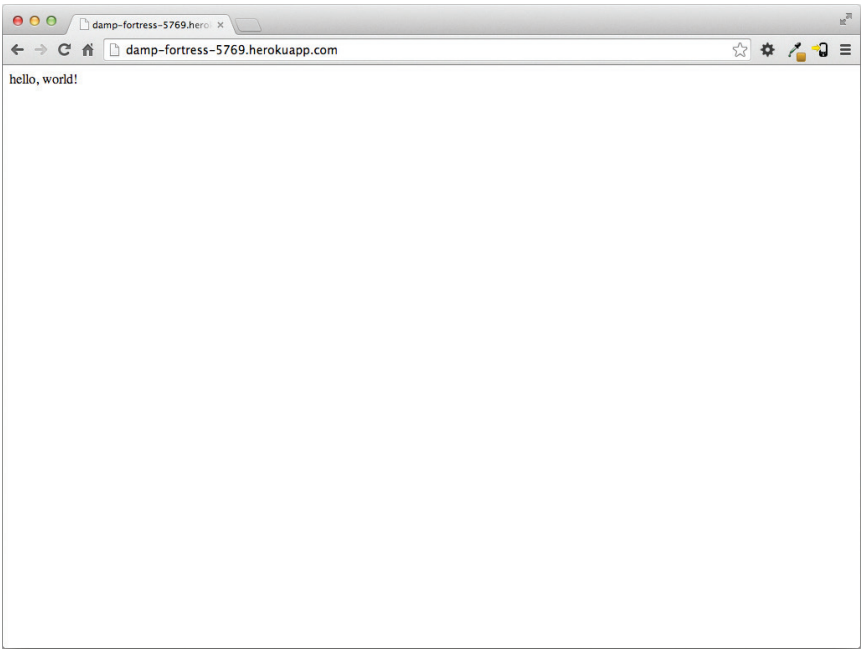


Рис. 1.18 ❖ Первое приложение из данной книги, запущенное в Heroku

1.5.4. Команды Heroku

Существует великое множество команд Heroku[1], и мы только слегка коснемся их в этой книге. Уделю минуту, только чтобы показать одну из них – команду переименования:

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

[1] https://devcenter.heroku.com/articles/heroku-command.

Не используйте сами это имя; я его уже занял! На самом деле вам вряд ли стоит делать это прямо сейчас; вполне достаточно адреса, автоматически предоставленного Heroku. Но если вы действительно хотите переименовать свое приложение, вы сможете сделать это, а заодно защититься от непрошеных визитеров, используя случайный или невнятный поддомен, например такой:

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

С таким случайным поддоменом никто не сможет посетить ваш сайт, если только вы сами не дадите адрес. (Между прочим, в качестве анонса удивительной компактности Ruby ниже приводится код, который я использовал для генерации случайных поддоменов:

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

Очень даже неплохо.)

В дополнение к поддоменам Heroku также поддерживает пользовательские домены. (Фактически сайт[1] этой книги живет на Heroku; если вы читаете электронную версию книги в сети, значит, прямо сейчас смотрите на сайт, размещенный на Heroku!) Обращайтесь к документации Heroku за дополнительной информацией о пользовательских доменах и других возможностях Heroku.

1.6. Заключение

Мы проделали длинный путь в этой главе: установка, настройка среды разработки, управление версиями и развертывание. В следующей главе, опираясь на полученные знания, мы построим *мини-приложение*, связанное с базой данных, чтобы просто оценить возможности Rails.

Если вы хотите поделиться с общественностью своим прогрессом, не стесняйтесь твитнуть или изменить свой статус в Facebook на что-то вроде этого:

Я изучаю Ruby on Rails с @railstutorial!
http://www.railstutorial.org/

Также я рекомендую зарегистрироваться в списке рассылки Rails Tutorial[2], который гарантирует вам получение очередных обновлений (и эксклюзивных купонов), касающихся книги.

1.6.1. Что мы узнали в этой главе

- Ruby on Rails – это фреймворк для разработки веб-приложений, написанный на языке программирования Ruby.

---

1 https://www.railstutorial.org/.

2 http://www.railstutorial.org/#email.

- Установка Rails, создание приложения, редактирование файлов – все это очень легко при использовании предварительно настроенной облачной среды разработки.
- В командную строку Rails встроена команда `rails`, которая может генерировать новые приложения (`rails new`) и запускать локальный сервер (`rails server`).
- Мы добавили метод контроллера и изменили корневой маршрут, чтобы создать приложение «hello, world».
- Поместив код приложения в систему управления версиями Git и отправив его в закрытый репозиторий Bitbucket, мы защитили себя от потери данных и получили возможность совместной работы над проектом.
- Мы развернули приложение в эксплуатационном окружении с помощью Heroku.

1.7. Упражнения

Примечание. *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

1. Измените метод `hello` в листинге 1.8 так, чтобы он выводил «hola, mundo!» вместо «hello, world!». *Дополнительно:* убедитесь, что Rails поддерживает символы, не принадлежащие набору ASCII, использовав перевернутый восклицательный знак «¡Hola, mundo!» (рис. 1.19)[1].

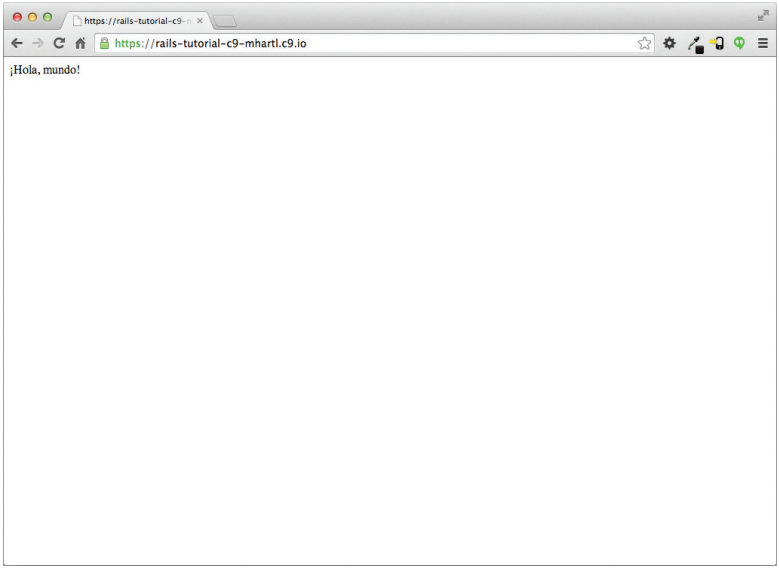


Рис. 1.19 ❖ Изменение действия контроллера для отображения «¡Hola, mundo!»

[1] Редактор может выдать сообщение, такое как: «invalid multibyte character» («неверный многобайтный символ»), но это не повод для беспокойства. Можете погуглить это сообщение, если вам интересно, как его устранить.

2. Следуя примеру метода действия `hello` из листинга 1.8, добавьте второй метод `goodbye`, который будет отображать текст «goodbye, world!». Отредактируйте файл маршрутов из листинга 1.10, чтобы корневой маршрут вел к методу `goodbye` вместо `hello` (рис. 1.20).

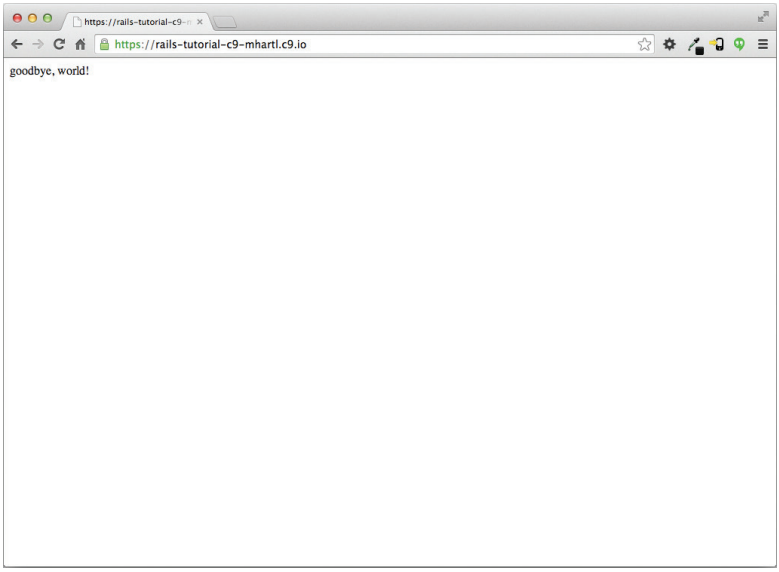


Рис. 1.20 ❖ Результат изменения корневого маршрута для вывода «goodbye, world!»

Глава 2

Мини-приложение

В этой главе мы разработаем миниатюрное демонстрационное приложение, чтобы похвастаться некоторыми возможностями Rails. Цель – получить общее представление о программировании Ruby on Rails (и веб-разработке в целом), быстро сгенерировав приложение с помощью *механизма скаффолдинга*, автоматически создающего большой объем функциональности. Как обсуждалось в блоке 1.2, в остальной части книги будет применяться противоположный подход, основанный на постепенной разработке приложения с объяснением всех новых понятий по мере их появления. Но для быстрого обзора (и получения чувства удовлетворения) нет ничего лучше скаффолдинга. Полученное в результате мини-приложение позволит взаимодействовать с ним через URL и понять структуру Rails-приложений, включая первый пример применения REST-архитектуры, предпочитаемой в Rails.

Как и предстоящее учебное приложение, это мини-приложение будет обслуживать *пользователей* и позволять им оставлять *микрособщения* (образуя минималистичное приложение в стиле Twitter). Оно будет обладать минимальной функциональностью, и многие из шагов будут походить на волшебство. Но не переживайте: начиная с главы 3 мы приступим к разработке полного учебного приложения, в ходе которой создадим подобный сайт с нуля, и здесь я буду давать многочисленные ссылки на соответствующие разделы в последующих главах. А пока запаситесь терпением – все-таки основная цель данной книги в том, чтобы удержать вас *от* этого поверхностного подхода для достижения более глубокого понимания Rails.

2.1. Проектирование приложения

В этом разделе мы обрисуем в общих чертах наши планы относительно мини-приложения. Как и в разделе 1.3, мы начнем с создания скелета приложения, вызвав команду `rails new` с конкретным номером версии Rails:

```
$ cd ~/workspace
$ rails _4.2.0_ new toy_app
$ cd toy_app/
```

Если эта команда вернет ошибку «Could not find ‘railties’», значит, у вас установлена не та версия Rails, и нужно перепроверить, была ли выполнена команда

из листинга 1.1 в точности, как написана. (Если вы используете облачную IDE, как рекомендовано в разделе 1.2.1, обратите внимание, что это второе приложение можно создать в том же рабочем пространстве, что и первое. Нет необходимости создавать новое рабочее пространство. Чтобы файлы появились в IDE, может понадобиться щелкнуть на значке шестеренки в навигаторе файлов и выбрать пункт **Refresh File Tree** (Обновить дерево файлов).

Далее в текстовом редакторе обновите файл `Gemfile`, как показано в листинге 2.1.

Листинг 2.1 ❖ Содержимое файла Gemfile для мини-приложения

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

gem 'rails',        '4.2.0'
gem 'sass-rails',   '5.0.1'
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
```

Отмечу, что листинг 2.1 идентичен листингу 1.14.

Как и в разделе 1.5.1, установим локальные гемы и предотвратим установку гемов, предназначенных для эксплуатационного окружения, добавив параметр `--without production`:

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

Наконец, поместим мини-приложение в систему управления версиями Git:

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

Необходимо также создать новый репозиторий, щелкнув на ссылке **Create** (Создать) на сайте Bitbucket (рис. 2.1), и отправить все в удаленный репозиторий:

```
$ git remote add origin git@bitbucket.org:<username>/toy_app.git
$ git push -u origin --all       # впервые отправляем репозиторий и ссылки
```

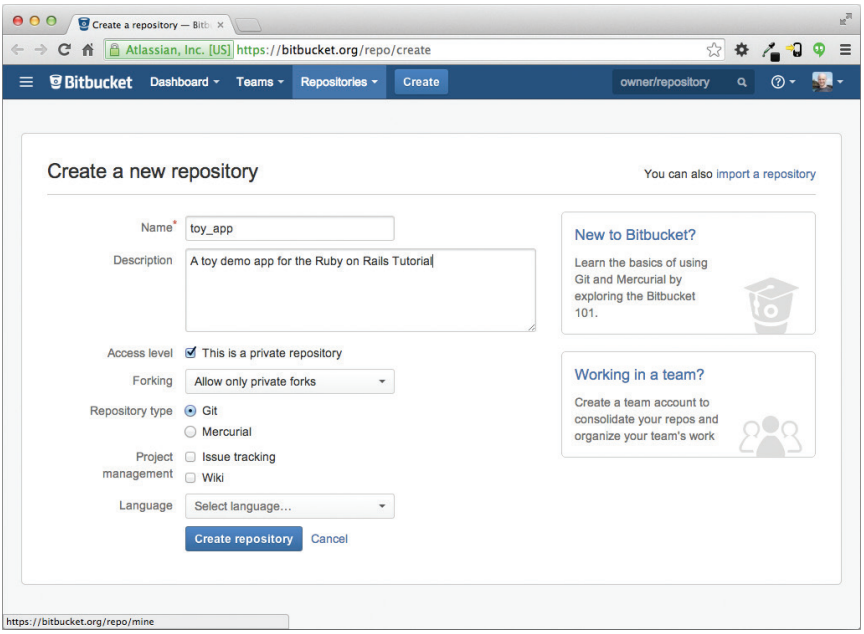


Рис. 2.1 ❖ Создание репозитория мини-приложения на Bitbucket

И наконец, никогда не бывает слишком рано, чтобы развернуть приложение, что я предлагаю сделать по той же схеме, что и в приложении «hello, world!», в листинге 1.8 и 1.9[1]. Затем зафиксируйте изменения и отправьте их в Heroku.

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

(Как и в разделе 1.5, может появиться несколько предупреждающих сообщений, которые на данном этапе можно игнорировать. Мы избавимся от них в разделе 7.5.) Кроме адреса приложения на сайте Heroku, результат должен совпадать с изображением на рис. 1.18.

Теперь мы готовы приступить к созданию самого приложения. Типичный первый шаг при разработке веб-приложений – создание *модели данных*, представляющей структуры, необходимые приложению. В нашем случае мини-приложение реализует упрощенный микроблог: только пользователи и короткие (микро)сообщения. Поэтому мы начнем с модели *пользователей* (раздел 2.1.1), затем добавим модель *микросообщений* (раздел 2.1.2).

[1] Созданные по умолчанию Rails-страницы, как правило, не работают в Heroku, поэтому сложно сказать, было ли успешным развертывание.

2.1.1. Модель пользователей

Вариантов модели данных для представления пользователей так же много, как разных форм регистрации в сети; мы пойдем самым простым путем. Пользователи мини-приложения будут иметь уникальный целочисленный идентификатор (`id`), открытое имя (`name`, типа `string`) и адрес электронной почты (`email`, тоже типа `string`). Итоговая модель данных для пользователей представлена на рис. 2.2.

users	
id	integer
name	string
email	string

Рис. 2.2 ❖ Модель данных для пользователей

Как будет показано в разделе 6.1.1, заголовок `users` на рис. 2.2 соответствует имени таблицы в базе данных, а атрибуты `id`, `name` и `email` – это имена столбцов в ней.

2.1.2. Модель микрособщений

Ядро модели микрособщений даже проще, чем модель пользователей: здесь есть только целочисленный идентификатор и текстовое содержимое (типа `text`)[1]. Но здесь есть дополнительная сложность: нам нужно связать каждое микрособщение с определенным пользователем. Мы реализуем это, добавив поле владельца сообщения `user_id`. Результат показан на рис. 2.3.

microposts	
id	integer
content	text
user_id	integer

Рис. 2.3 ❖ Модель данных микрособщений

Мы увидим в разделе 2.3.3 (и более полно в главе 11), как атрибут `user_id` позволит кратко выразить идею наличия у пользователя множества связанных с ним микрособщений.

2.2. Ресурс Users

В этом разделе мы реализуем модель данных пользователей из раздела 2.1.1 вместе с веб-интерфейсом к ней. Такая комбинация модели и интерфейса образует *ресурс Users*, который позволит думать о пользователях как об объектах, которые

[1] Так как микрособщения планируется сделать короткими, для их хранения вполне достаточно типа `string`, но использование типа `text` лучше выражает нашу цель, а также дает большую гибкость и позволяет не задумываться об ограничении длины.

можно создавать, читать, обновлять и удалять с использованием протокола HTTP. Как и было обещано во введении, ресурс Users будет создаваться механизмом скаффолдинга, который стандартно поставляется с каждым проектом Rails. Я настоятельно прошу не всматриваться в генерируемый код; на этом этапе он лишь запутает вас.

Скаффолдинг запускается передачей команды `scaffold` сценарию `rails generate`. Аргумент команды `scaffold` – это имя ресурса (в единственном числе, в данном случае `User`) вместе с дополнительными параметрами для атрибутов модели данных[1]:

```
$ 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
      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
```

[1] Имя генерируемой модели выбрано в соответствии с соглашением об именовании моделей, согласно которому они должны иметь единственное число, в отличие от ресурсов и контроллеров, имена которых имеют множественное число. Поэтому получаем `User` вместо `Users`.

Добавив поля `name:string` и `email:string`, мы придали модели User форму, соответствующую изображению на рис. 2.2. (Обратите внимание, что нет надобности включать параметр `id` – он создается автоматически и используется в качестве первичного ключа в базе данных.)

Чтобы продолжить создание мини-приложения, необходимо выполнить миграцию базы данных с помощью утилиты *Rake* (см. блок 2.1):

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

Она просто обновила базу данных, добавив модель `users`. (Мы узнаем больше о миграциях баз данных в разделе 6.1.1.) Обратите внимание: чтобы утилита Rake использовала версии команд, соответствующие указанным в файле `Gemfile`, запускать `rake` нужно с помощью `bundle exec`. Во многих системах, включая облачную IDE, можно опустить команду `bundle exec`, но в некоторых она необходима, поэтому я буду добавлять ее для полноты картины.

Теперь можно запустить локальный веб-сервер в отдельной вкладке (рис. 1.7)[1]:

```
$ rails server -b $IP -p $PORT    # Просто `rails server`, если запускаете локально
```

Сейчас мини-приложение должно быть доступно на локальном сервере, как описано в разделе 1.3.2. (Если используется облачная IDE, обязательно откройте страницу приложения в новой вкладке браузера, а не внутри самой IDE.)

Блок 2.1 ❖ Rake

В традициях `Unix` утилита `make` играет важную роль в сборке выполняемых программ из исходного кода; у многих хакеров на уровне мышечной памяти зафиксирована строка

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

обычно используемая для компиляции кода в Unix-системах (включая Linux и Mac OS X).

Rake – это *Ruby make*, make-подобный язык, написанный на Ruby. Rails очень широко использует Rake, особенно для решения бесчисленных мелких административных задач, возникающих при разработке веб-приложений, опирающихся на базы данных. Наиболее распространенной, пожалуй, является команда `rake db:migrate`, но есть и другие; полный список задач, связанных с базами данных, можно увидеть, передав параметр `-T db`:

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

Чтобы увидеть все доступные задачи Rake, запустите

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

[1] Сценарий `rails` построен так, что не нуждается в `bundle exec`.

Кого-то из вас список может ошеломить, но не волнуйтесь, вы не должны знать все (или даже большую часть) эти команды. К концу книги вы будете знать все самые важные из них.

2.2.1. Обзор пользователей

Если обратиться к корневому URL (как отмечено в разделе 1.3.4), мы увидим ту же страницу Rails, созданную по умолчанию, что изображена на рис. 1.9; но, сгенерировав каркас ресурса Users, мы дополнительно создали еще несколько страниц для управления пользователями. Например, страницу со списком всех пользователей по адресу /users и страницу для создания нового пользователя по адресу /users/new. Остальная часть этого раздела будет посвящена беглому обзору всех этих страниц. В процессе вам может быть полезно иногда заглядывать в табл. 2.1, где показаны соответствия между страницами и их адресами URL.

Начнем со страницы, отображающей список всех пользователей приложения. Она называется `index`. Как можно догадаться, изначально она отображает пустой список (рис. 2.4).

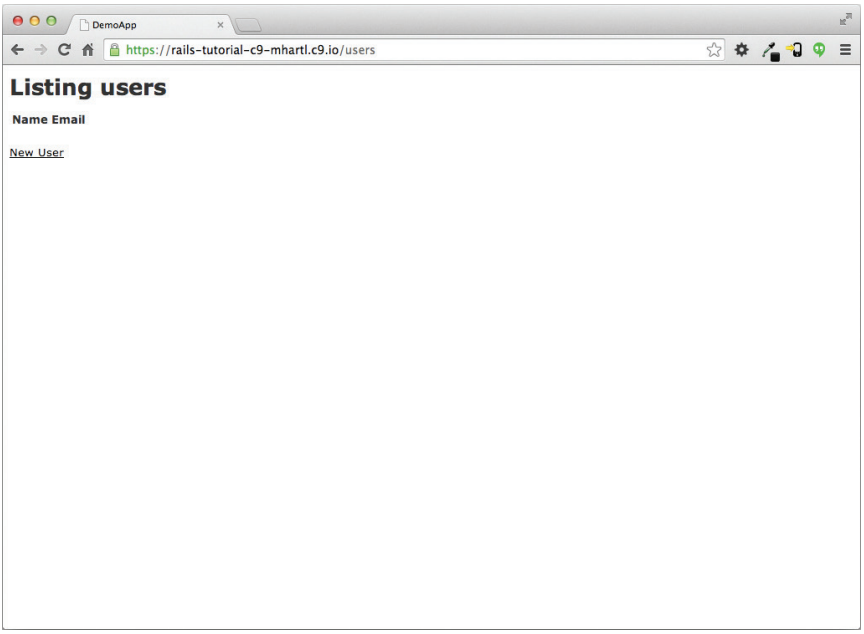


Рис. 2.4 ❖ Начальная страница ресурса Users (/users)

Чтобы создать нового пользователя, откроем страницу `new`, она показана на рис. 2.5. (Поскольку во всех адресах подразумевается наличие http://0.0.0.0:3000 или адреса облачной IDE, пока разработка будет вестись локально, я буду опускать ее.) В главе 7 она станет страницей регистрации пользователя.

Таблица 2.1 ❖ Соответствие между страницами и адресами URL для ресурса Users

URL	Метод	Назначение
/users	`index`	Страница со списком всех пользователей
/users/1	`show`	Страница с информацией о пользователе с `id 1`
/users/new	`new`	Страница создания нового пользователя
/users/1/edit	`edit`	Страница редактирования пользователя с `id 1`

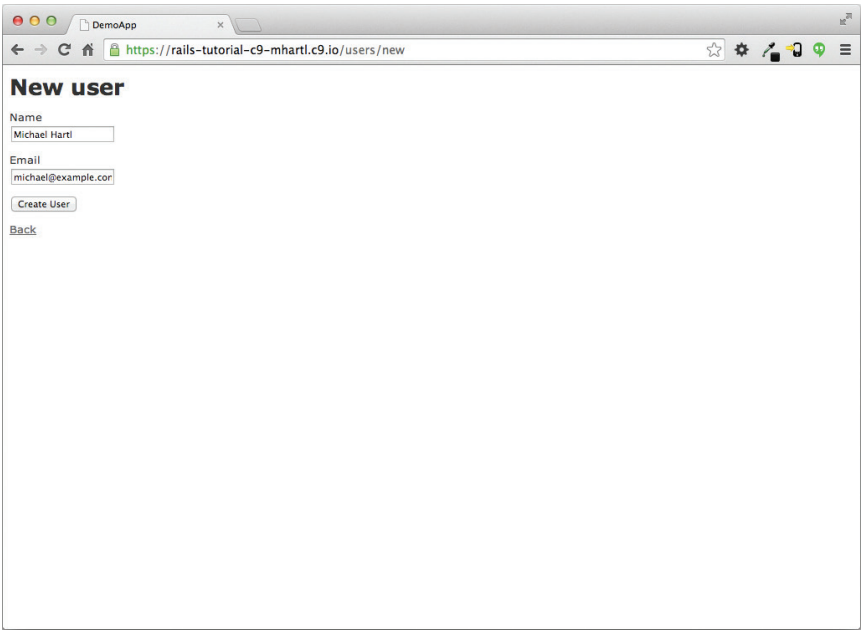


Рис. 2.5 ❖ Страница создания нового пользователя (/users/new)

Мы можем создать пользователя, введя его имя с адресом электронной почты в текстовые поля и щелкнув на кнопке **Create User** (Создать пользователя). В результате будет создана страница пользователя, изображенная на рис. 2.6. (Приветственное сообщение зеленого цвета выполнено с использованием метода *flash*, о котором рассказывается в разделе 7.4.2.) Обратите внимание на URL /users/1; как можно догадаться, число `1` – это просто атрибут `id` пользователя. В разделе 7.1 эта страница станет профилем пользователя.

Чтобы отредактировать данные о пользователе, откройте страницу `edit` (рис. 2.7). Изменив данные в полях и щелкнув на кнопке **Update User** (Изменить пользователя), мы изменим информацию о пользователе в мини-приложении (рис. 2.8). (Как вы узнаете в главе 6, эта информация хранится в базе данных.) Мы добавим функции редактирования/изменения пользователя в учебное приложение в разделе 9.1.

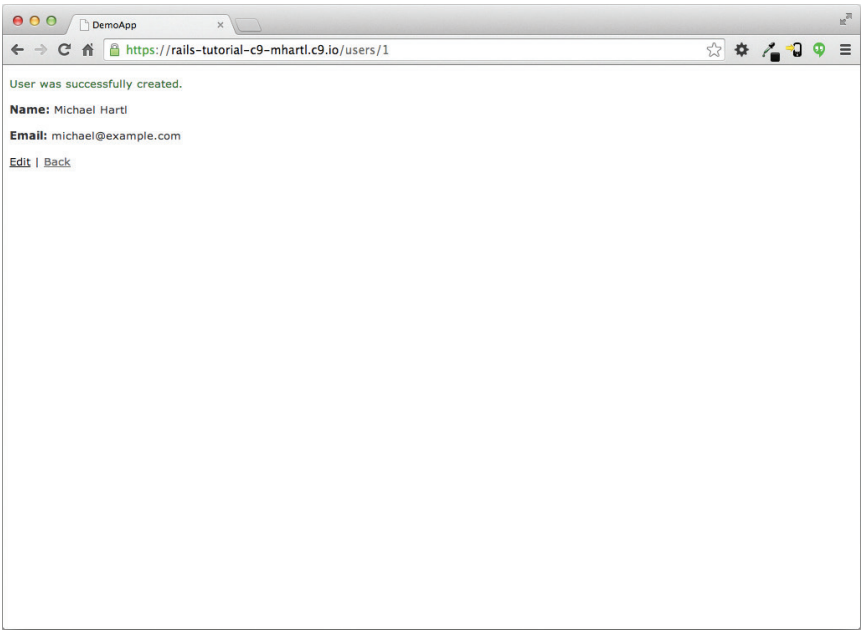


Рис. 2.6 ❖ Страница с информацией о пользователе (/users/1)

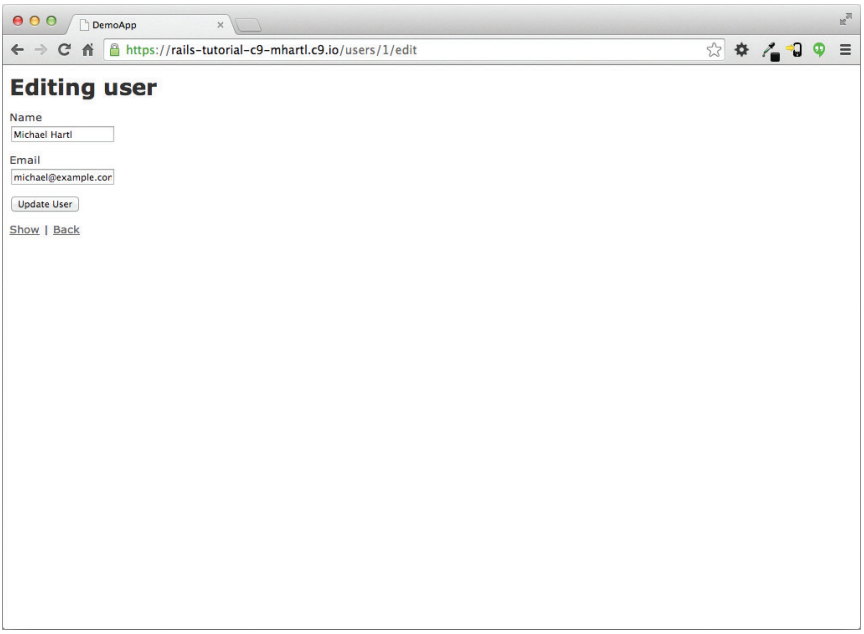


Рис. 2.7 ❖ Страница редактирования пользователя (/users/1/edit)

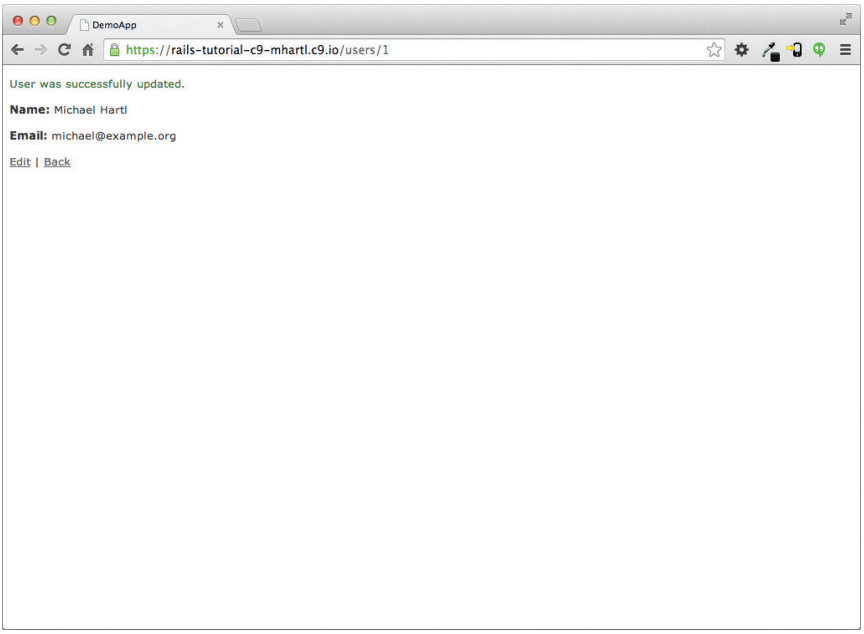


Рис. 2.8 ❖ Пользователь с обновленной информацией

Теперь создадим второго пользователя, повторно открыв страницу `new` и отправив второй набор данных; получившаяся страница `index` показана на рис. 2.9. В разделе 7.1 мы преобразуем этот список в более изысканную страницу, отображающую список всех пользователей.

Посмотрев, как создавать, отображать и редактировать пользователей, перейдем, наконец, к их удалению (рис. 2.10). Проверьте, что щелчок на ссылке, показанной на рис. 2.10, удаляет второго пользователя и на странице со списком остается один пользователь. (Если что-то не получится, убедитесь, что в браузере включена поддержка JavaScript; Rails использует JavaScript, чтобы запросить подтверждение на удаление пользователя.) Удаление пользователя в учебном приложении будет добавлено в разделе 9.4, где мы позаботимся об ограничении доступа к этой функции специальным классом административных пользователей.

2.2.2. MVC в действии

Теперь, завершив обзор ресурса Users, исследуем отдельную его часть в контексте схемы модель-представление-контроллер (MVC), представленной в разделе 1.3.3. Далее мы опишем результаты типичного запроса браузера – посещение страницы /users – в терминах MVC (рис. 2.11).

Ниже кратко описываются шаги, изображенные на рис. 2.11.

1. Браузер посылает запрос на адрес URL /users.
2. Rails передает запрос к адресу /users в метод `index` в контроллере Users.

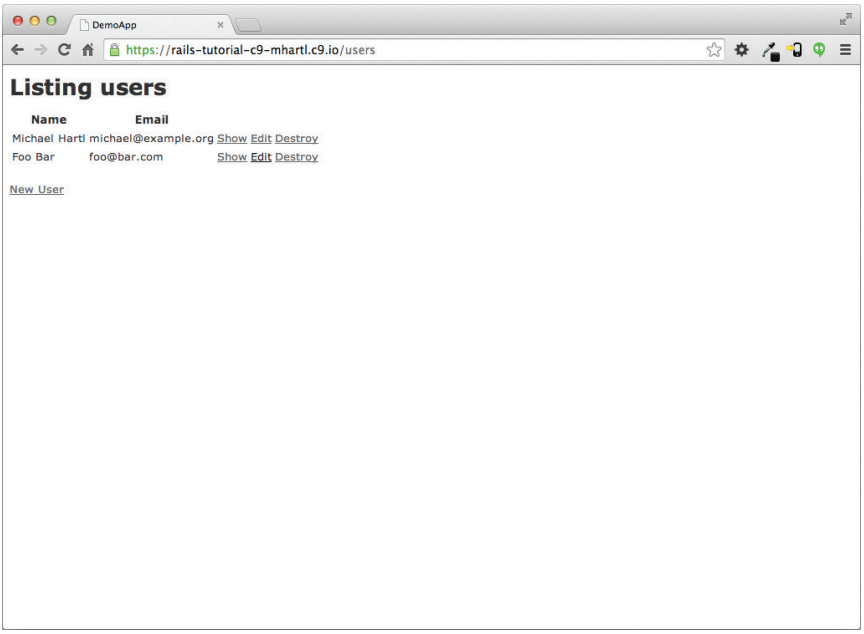


Рис. 2.9 ❖ Страница-список (/users) со вторым пользователем

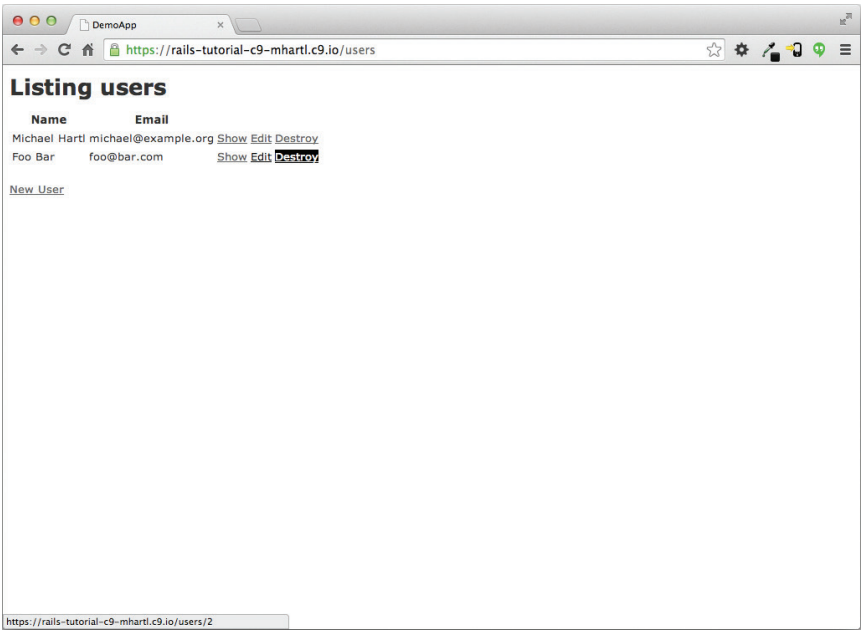


Рис. 2.10 ❖ Удаление пользователя

3. Метод `index` запрашивает у модели User список всех пользователей (`User.all`).
4. Модель User извлекает всех пользователей из базы данных.
5. Модель User возвращает список пользователей в контроллер.
6. Контроллер сохраняет список в переменной `@users` и передает ее представлению `index`.
7. Представление использует встроенный код на языке Ruby, чтобы отобразить страницу в виде разметки HTML.
8. Контроллер возвращает разметку HTML браузеру[1].

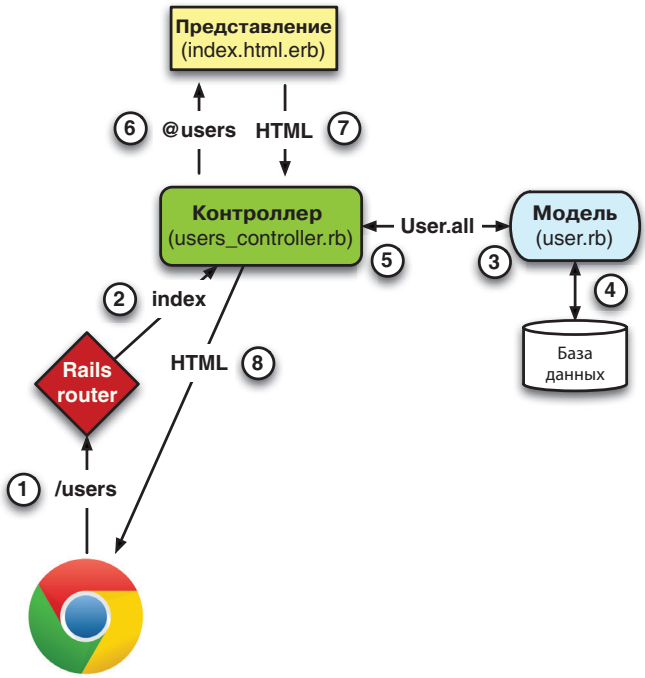


Рис. 2.11 ❖ Подробная схема MVC в Rails

Теперь подробно рассмотрим каждый из этих шагов. Начнем с запроса, отправленного браузером, то есть с результата ввода URL в адресной строке браузера или щелчка на ссылке (шаг 1 на рис. 2.11). Запрос достигает механизма маршрутизации Rail (шаг 2), который передает запрос соответствующему *методу контроллера*, опираясь на URL (и, как рассказывается в блоке 3.2, на тип запроса). Код, создающий отображение адресов URL в методы контроллера для ресурса Users,

[1] В некоторых источниках утверждается, что представление возвращает разметку HTML непосредственно браузеру (посредством веб-сервера, такого как Apache или Nginx). Независимо от деталей реализации я предпочитаю думать о контроллере как о центральном узле, через который проходят все информационные потоки приложения.

представлен в листинге 2.2; этот код фактически связывает пары URL/метод, которые мы видели в табл. 2.1. (Странная запись `:users` – это *символ*, о котором мы узнаем в разделе 4.3.3.)

Листинг 2.2 ❖ Маршруты Rails с правилами для ресурса Users (config/routes.rb)

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

Пока мы не ушли далеко от файла с маршрутами, воспользуемся моментом и свяжем корневой маршрут со страницей, отображающей список пользователей, чтобы корневой маршрут приводил к /users. Вспомните листинг 1.10

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

в который мы внесли следующие изменения

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

чтобы корневой маршрут приводил в метод `hello` контроллера Application. Теперь нам понадобится метод `index` в контроллере Users, и мы можем добиться этого, как показано в листинге 2.3. (Здесь я рекомендую удалить метод `hello` из контроллера Application, если вы добавили его в начале раздела.)

Листинг 2.3 ❖ Добавление корневого маршрута для ресурса users (config/routes.rb)

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

Страницы, описанные в разделе 2.2.1, соответствуют методам действий в *контроллере Users*, который, по сути, является набором связанных действий. Контроллер, сгенерированный механизмом скаффолдинга, схематично показан в листинге 2.4. Обратите внимание на запись `class UsersController < ApplicationController`, являющуюся примером объявления класса с наследованием. (Мы вкратце обсудим наследование в разделе 2.3.4 и раскроем обе темы более подробно в разделе 4.4.)

Листинг 2.4 ❖ Контроллер Users Controller в схематичной форме (app/controllers/users_controller.rb)

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

```
  .
  def index
    .
    .
    .
  end
  def show
    .
    .
    .
  end
  def new
    .
    .
    .
  end
  def edit
    .
    .
    .
  end
  def create
    .
    .
    .
  end
  def update
    .
    .
    .
  end
  def destroy
    .
    .
    .
  end
end
```

Таблица 2.2 ❖ Маршруты RESTful, поддерживаемые ресурсом Users из листинга 2.2

HTTP-запрос	URL	Метод	Назначение
GET	/users	**index**	Страница со списком всех пользователей
GET	/users/1	**show**	Страница с информацией о пользователе с id 1
GET	/users/new	**new**	Страница создания нового пользователя
POST	/users	**create**	Создает нового пользователя
GET	/users/1/edit	**edit**	Страница редактирования пользователя с id 1
PATCH	/users/1	**update**	Обновляет данные пользователя с id 1
DELETE	/users/1	**destroy**	Удаляет пользователя с id 1

Вы могли заметить, что методов больше, чем страниц. Методы `index`, `show`, `new` и `edit` соответствуют страницам из раздела 2.2.1, но есть еще дополнительные методы: `create`, `update` и `destroy`. Обычно они не отображают страниц (хотя могут); их основная цель в том, чтобы изменять информацию о пользователях в базе данных. Этот полный комплект методов действий контроллера, перечисленных в табл. 2.2, представляет реализацию архитектуры REST в Rails (блок 2.2), которая опирается на идею передачи репрезентативного состояния, сформулированную ученым в области информатики Роем Филдингом (Roy Fielding) в своей докторской диссертации[1]. Обратите внимание что некоторые адреса URL в табл. 2.2 перекрываются; например, обоим методам, `show` и `update`, соответствуют URL /users/1. Различие между ними заключено в методах HTTP запросов, для обработки которых они вызываются. Подробнее о методах HTTP запросов рассказывается в разделе 3.3.

Блок 2.2 ❖ REpresentational State Transfer (REST)

Если вы читали о разработке веб-приложений на основе Ruby on Rails, то могли видеть ссылки на «REST» (сокращенно от REpresentational State Transfer – передача репрезентативного состояния). REST – архитектурный стиль для разработки распределенных сетевых систем и приложений. Хотя теория REST довольно абстрактна, в контексте Rails-приложений поддержка REST означает, что большинство компонентов приложения (таких как пользователи и микрособщения) моделируется как ресурсы, которые могут быть созданы (Created), прочитаны (Read), изменены (Updated) и удалены (Deleted), – эти операции соответствуют CRUD-операциям в реляционных базах данных[2] и четырем основным методам HTTP-запросов: `POST`, `GET`, `PATCH` и `DELETE`. (Подробнее о запросах HTTP рассказывается в разделе 3.3 и особенно в блоке 3.2.)

Стиль RESTful разработки помогает разработчикам Rails-приложений решить, какие контроллеры и методы необходимы: вы просто структурируете приложение, используя ресурсы, которые могут создаваться, извлекаться, изменяться и удаляться. В случае с пользователями и микрособщениями все просто, так как это – естественные ресурсы по своей сути. В главе 12 мы увидим пример, где принципы REST позволят смоделировать более тонкую проблему, «Следование за пользователями», естественным и удобным способом.

В наших исследованиях отношений между контроллером Users и моделью User сосредоточимся на упрощенной версии метода `index`, показанной в листинге 2.5. (Код, созданный механизмом скаффолдинга, уродлив и запутан, так что я опустил большую его часть.)

---

[1] Рой Томас Филдинг (Fielding, Roy Thomas). «Архитектурные стили и дизайн сетевых архитектур программного обеспечения». Докторская диссертация, Калифорнийский университет, г. Ирвайн, 2000.

[2] https://ru.wikipedia.org/wiki/CRUD.

Листинг 2.5 ❖ Упрощенный метод index для мини-приложения (app/controllers/users_controller.rb)

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

В методе `index` находится строка `@users = User.all` (шаг 3 на рис. 2.11), которая запрашивает у модели User список всех пользователей из базы данных (шаг 4) и помещает его в переменную `@users` (шаг 5). Сама модель User показана в листинге 2.6. Она довольно проста, но к ней прилагается большое количество унаследованных функций (раздел 2.3.4 и раздел 4.4). В частности, за счет использования библиотеки *Active Record* код в листинге 2.6 заставляет `User.all` вернуть список всех пользователей из базы данных.

Листинг 2.6 ❖ Модель User для мини-приложения (app/models/user.rb)

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

Присвоив значение переменной `@users`, контроллер вызывает представление (шаг 6) из листинга 2.7. Переменные с именами, начинающимися со знака `@`, называются переменными экземпляра, они автоматически доступны в представлениях; в данном случае представление `index.html.erb` перебирает список `@users` и для каждого выводит строку HTML. (Помните, что не нужно пытаться понять этот код прямо сейчас. Он показан только для иллюстрации.)

Листинг 2.7 ❖ Представление для вывода списка пользователей (app/views/users/index.html.erb)

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

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

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
```

```
    <td><%= user.email %></td>
    <td><%= link_to 'Show', user %></td>
    <td><%= link_to 'Edit', edit_user_path(user) %></td>
    <td><%= link_to 'Destroy', user, method: :delete,
                                     data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>

</table>

<br>

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

Представление преобразует содержимое переменной в разметку HTML (шаг 7), которая затем возвращается контроллером в браузер для отображения (шаг 8).

2.2.3. Недостатки ресурса Users

Несмотря на полезность для общего знакомства с Rails, ресурс Users, сгенерированный механизмом скаффолдинга, имеет много серьезных недостатков.

- **Отсутствует проверка данных.** Модель User безропотно принимает такие данные, как пустые имена и недопустимые адреса электронной почты.
- **Отсутствует аутентификация.** У нас нет понятия регистрации и нет способа воспрепятствовать выполнению любых операций любыми пользователями.
- **Нет тестов.** Что технически не вполне верно, механизм скаффолдинга включает рудиментарные тесты, но в них нет тестов для проверки данных, аутентификации или каких-либо других нестандартных требований.
- **Нет стилей или макета.** Нет единого стиля оформления сайта или навигации.
- **Нет реального понимания.** Если бы вы понимали автоматически сгенерированный код, то, вероятно, не стали бы читать эту книгу.

2.3. Ресурс Microposts

Сгенерировав и исследовав ресурс Users, обратимся теперь к связанному с ним ресурсу Microposts. На протяжении этого раздела я рекомендую сравнивать между собой элементы этих ресурсов; вы должны увидеть, что они во многом повторяют друг друга. RESTful-структура Rails-приложений лучше всего усваивается посредством наблюдения за подобными повторяющимися формами – действительно, наблюдение сходства структур Users и Microposts, даже на этой ранней стадии, является одной из причин написания этой главы.

2.3.1. Микрообзор микрособщений

Так же как в случае с ресурсом Users, сгенерируем код для ресурса Microposts командой `rails generate scaffold`, в этом случае реализуя модель данных, изображенную на рис. 2.3[1]:

---

[1] Как и в случае с ресурсом User, генератор следует соглашению о выборе имен в единственном числе для моделей Rails; поэтому мы получим `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
```

(Если вы получите сообщение об ошибке, связанное со Spring, просто запустите команду снова.) Чтобы добавить в базу данных новую модель, нужно выполнить миграцию, как в разделе 2.2:

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

Теперь можно создавать микросообщения так же, как пользователей в разделе 2.2.1. Нетрудно догадаться, что генератор добавил в файл маршрутов правило для ресурса Microposts, как показано в листинге 2.8[1]. Так же, как для пользовате-

[1] Механизм скаффолдинга мог добавить дополнительные символы перевода строки. Это не повод для беспокойства, поскольку Ruby игнорирует их.

лей, маршрутное правило `resources :microposts` направляет запросы к URL микросообщений в методы контроллера Microposts, как показано в табл. 2.3.

Листинг 2.8 ❖ Маршруты с новым правилом для ресурса Microposts (config/routes.rb)

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

Таблица 2.3 ❖ Маршруты RESTful для ресурса Microposts

HTTP-запрос	URL	Метод	Назначение
GET	/microposts	**index**	Страница со списком всех микросообщений
GET	/microposts/1	**show**	Страница отображения микросообщения с id 1
GET	/microposts/new	**new**	Страница создания нового микросообщения
POST	/microposts	**create**	Создает новое микросообщение
GET	/microposts/1/edit	**edit**	Страница редактирования микросообщения с id 1
PATCH	/microposts/1	**update**	Обновляет данные микросообщения с id 1
DELETE	/microposts/1	**destroy**	Удаляет микросообщение с id 1

Сам Microposts в схематичной форме представлен в листинге 2.9. Обратите внимание, что он полностью *идентичен* листингу 2.4, отличаясь только именем `MicropostsController`, используемым вместо `UsersController`. Это отражение REST-архитектуры, характерной для обоих ресурсов.

Листинг 2.9 ❖ Контроллер Microposts в схематичной форме (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
```

Чтобы сделать несколько настоящих микросообщений, нужно ввести информацию на странице /microposts/new (см. рис. 2.12).

Теперь создайте одно-два микросообщения так, чтобы хотя бы одно из них имело значение `1` в поле `user_id`, то есть принадлежало пользователю, созданному в разделе 2.2.1. Результат должен выглядеть, как показано на рис. 2.13.

2.3.2. Ограничение размеров микросообщений

Каждое *микросообщение*, как следует из названия, не должно превышать некоторой длины. Реализация этого ограничения в Rails легко достигается с помощью *валидаторов*, осуществляющих проверку допустимости; чтобы ограничить размер микросообщений 140 символами (как в Twitter), определим валидатор *длины*. Откройте файл `app/models/micropost.rb` в текстовом редакторе или IDE и добавьте в него код из листинга 2.10.

Листинг 2.10 ❖ Ограничение длины микросообщений 140 знаками (app/models/micropost.rb)

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

Код в листинге 2.10 может показаться таинственным – подробнее мы рассмотрим приемы проверки в разделе 6.2, – но его эффект станет очевиден, если перейти

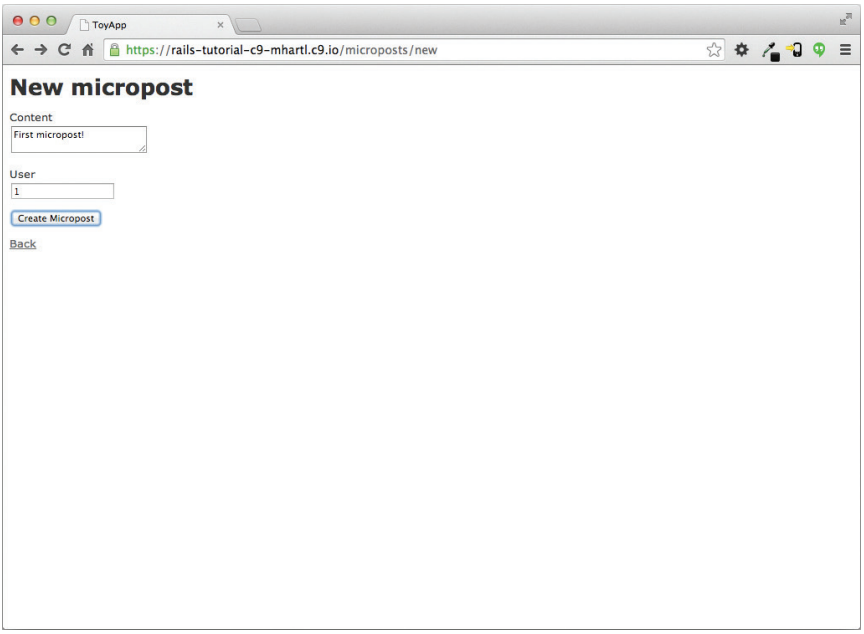


Рис. 2.12 ❖ Страница создания микросообщения (/microposts/new)

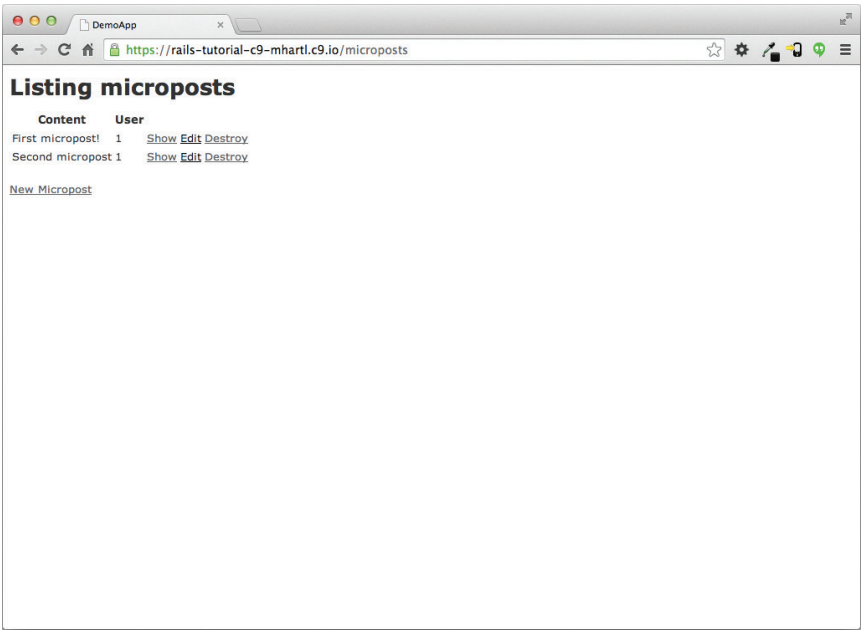


Рис. 2.13 ❖ Страница со списком микросообщений (/microposts)

на страницу создания микросообщений и ввести в поле больше 140 символов. Как видно на рис. 2.14, Rails выведет сообщение об ошибке, указывающее, что содержимое микросообщения является слишком длинным. (Мы узнаем больше о таких сообщениях в разделе 7.3.3.)

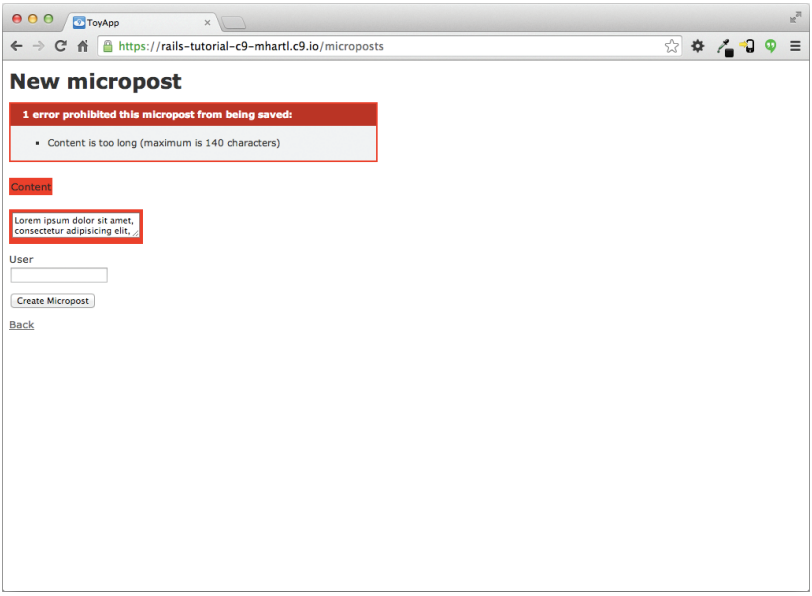


Рис. 2.14 ❖ Сообщение об ошибке для неудачно созданного микросообщения

2.3.3. Принадлежность множества микросообщений одному пользователю

Одна из наиболее мощных функций Rails – возможность формирования *связей* между различными моделями данных. В случае с моделью User каждый пользователь может иметь много микросообщений. Мы можем выразить это, расширив модели User и Micropost, как показано в листингах 2.11 и 2.12.

Листинг 2.11 ❖ Пользователь имеет много микросообщений (app/models/user.rb)

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

Листинг 2.12 ❖ Микросообщения принадлежат пользователю (app/models/micropost.rb)

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

Результат применения этой связи показан на рис. 2.15. За счет столбца user_id в таблице microposts Rails (с помощью Active Record) может вывести микросообщения, связанные с каждым пользователем.

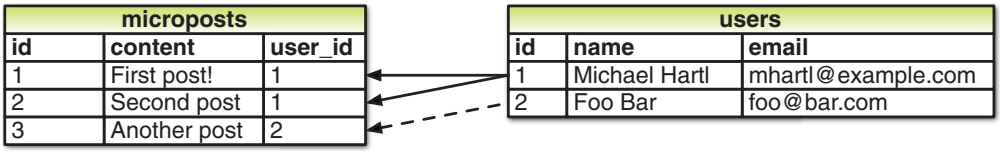


Рис. 2.15 ❖ Связь между микросообщениями и пользователями

В главах 11 и 12 мы будем использовать такую же связь, чтобы вывести на экран все микросообщения пользователя и создать Twitter-подобную ленту микросообщений. А пока выясним смысл связи «микросообщения–пользователь» с помощью консоли – удобного инструмента взаимодействий с Rails-приложениями. Сначала запустим консоль командой rails console в командной строке, а затем извлечем первого пользователя из базы данных, используя User.first (сохранив результаты в переменную first_user)[1]:

```
$ 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:
"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
```

(Я добавил последнюю строку с командой exit, только чтобы показать, как выйти из консоли. В большинстве систем с той же целью можно использовать **Ctrl-D**[2].) Здесь мы получили доступ к микросообщениям пользователя, используя метод first_user.microposts. Благодаря ему Active Record автоматически возвращает все микросообщения с идентификатором user_id, равным идентификатору first_user (в данном случае 1). Больше о возможностях связей в Active Record мы узнаем в главах 11 и 12.

---

[1] У вас приглашение командной строки, вероятно, будет выглядеть как-то так: 2.1.1 :001 >, но я буду использовать >>, поскольку версии Ruby могут отличаться.

[2] Как и в случае с комбинацией **Ctrl-C**, заглавная «D» означает клавишу «d» на клавиатуре, а не заглавную букву, поэтому не нужно зажимать кнопку **Shift** вместе с **Ctrl**.

2.3.4. Иерархия наследования

Закончим обсуждение мини-приложения кратким описанием иерархии классов контроллеров и моделей в Rails. Это обсуждение будет более интересно тем, кто имеет некоторый опыт объектно-ориентированного программирования (ООП); если вы не изучали ООП, можете просто пропустить этот раздел. В частности, если вы не знакомы с понятием классов (обсуждается в разделе 4.4), я предлагаю оставить этот раздел на потом.

Начнем со структуры наследования моделей. Сравнивая листинги 2.13 и 2.14, можно заметить, что модели User и Micropost наследуют (посредством левой угловой скобки <) свойства и методы класса `ActiveRecord::Base`, который является базовым для моделей, предоставляемых библиотекой Active Record; схема, резюмирующая это отношение, представлена на рис. 2.16. Благодаря наследованию `ActiveRecord::Base` наши объекты моделей получают возможность взаимодействовать с базой данных, обрабатывать столбцы базы данных как Ruby-атрибуты и т. д.

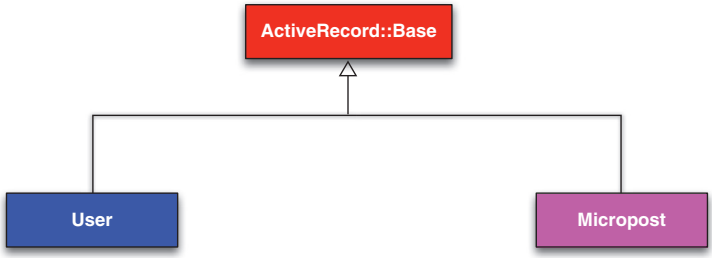


Рис. 2.16 ❖ Иерархия наследования для моделей User и Micropost

Листинг 2.13 ❖ Класс User с наследованием (app/models/user.rb)

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

Листинг 2.14 ❖ Класс Micropost с наследованием (app/models/micropost.rb)

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

Структура наследования контроллеров лишь немногим сложнее. Сравнивая листинг 2.15 с листингом 2.16, можно заметить, что контроллеры User и Micropost наследуют класс `ApplicationController`. В листинге 2.17 видно, что сам `ApplicationController` наследует `ActionController::Base`; это базовый класс контроллеров из библиотеки Action Pack. Отношения между этими классами показаны на рис. 2.17.

Листинг 2.15 ❖ Класс Users Controller с наследованием (app/controllers/users_controller.rb)

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

Листинг 2.16 ❖ Класс Microposts Controller с наследованием (app/controllers/microposts_controller.rb)

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

Листинг 2.17 ❖ Класс Application Controller с наследованием (app/controllers/application_controller.rb)

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

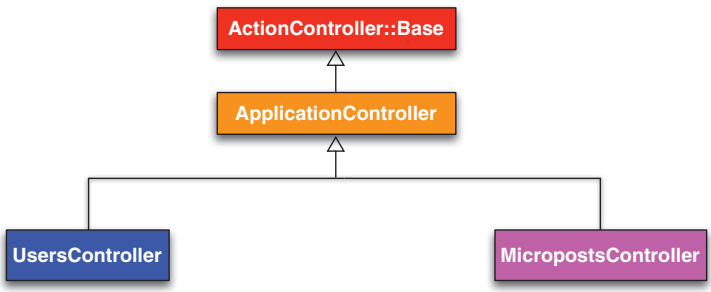


Рис. 2.17 ❖ Иерархия наследования контроллеров Users и Microposts

Как и при наследовании моделей, оба контроллера (Users и Microposts) получают большое количество функций за счет наследования базового класса (в данном случае `ActionController::Base`), включая возможность управления объектами моделей, фильтрации входящих запросов HTTP и отображения представлений в виде разметки HTML. Так как все контроллеры в Rails наследуют `ApplicationController`, правила, определенные в нем, автоматически применяются к каждому методу действия в приложении. Например, в разделе 8.4 мы увидим, как добавить вспомогательные методы для входа и выхода во все контроллеры учебного приложения.

2.3.5. Развертывание мини-приложения

Закончив создание ресурса Microposts, можно отправить его в репозиторий Bitbucket:

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

Обычно лучше фиксировать изменения часто и небольшими порциями, но для целей этой главы вполне допустимо выполнить единственную большую фиксацию в конце.

Теперь можно развернуть мини-приложение на сайте Heroku, как рассказывалось в разделе 1.5:

```
$ git push heroku
```

(Предполагается, что вы создали Heroku-приложение в разделе 2.1. Иначе нужно выполнить `heroku create`, а затем `git push heroku master`.)

Чтобы заставить работать базу данных приложения, нужно выполнить миграцию базы данных:

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

Эта команда добавит в базу данных на Heroku необходимые модели данных пользователей и микрособщений. После запуска миграции вы сможете запустить мини-приложение в эксплуатационном окружении, на основе базы данных PostgreSQL (рис. 2.18).

2.4. Заключение

Мы подошли к концу обзора Rails-приложения. У мини-приложения, разработанного в этой главе, есть несколько достоинств и масса недостатков.

Достоинства:

- общее представление oRails;
- введение в MVC;
- первое знакомство с архитектурой REST;
- начальное моделирование данных;
- действующее веб-приложение с базой данных.

Недостатки:

- нет собственного макета или стиля;
- нет статических страниц (таких как «Главная» или «О нас»);
- нет паролей пользователей;
- нет изображений пользователей;
- нет регистрации;
- нет безопасности;
- нет автоматической связи пользователь/микрособщение;

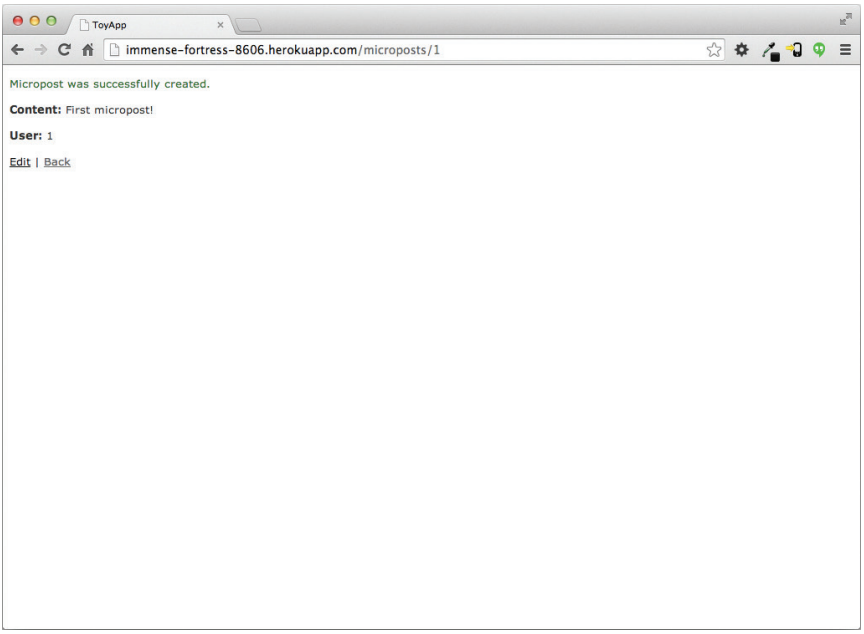


Рис. 2.18 ❖ Запуск мини-приложения в эксплуатационном окружении

- нет понятия «следования за пользователем»;
- нет ленты микрособщений;
- нет значимых тестов;
- **нет реального понимания**.

Остальная часть этой книги посвящена укреплению достоинств и устранению недостатков.

2.4.1. Что мы узнали в этой главе

- Механизм скаффолдинга позволяет автоматически создать код для моделей данных и взаимодействий с ними по сети.
- Механизм скаффолдинга хорошо подходит для быстрого старта, но плох для изучения.
- Для структурирования веб-приложений Rails использует схему модель-представление-контроллер (MVC).
- В интерпретации Rails REST-архитектура включает стандартный набор URL и методов действий контроллеров для взаимодействий с моделями данных.
- Rails поддерживает проверку данных и позволяет вводить ограничения на значения атрибутов модели данных.
- Rails содержит встроенные функции для определения связей между разными моделями данных.
- Мы можем взаимодействовать с Rails-приложением через командную строку, используя Rails-консоль.

2.5. Упражнения

Примечание. *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

1. Код в листинге 2.18 демонстрирует, как добавить проверку наличия содержимого в микрособщения, чтобы гарантировать отсутствие пустых сообщений. Проверьте, дает ли эта проверка поведение, изображенное на рис. 2.19.

Листинг 2.18 ❖ Код, проверяющий наличие содержимого микрособщений (app/models/micropost.rb)

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

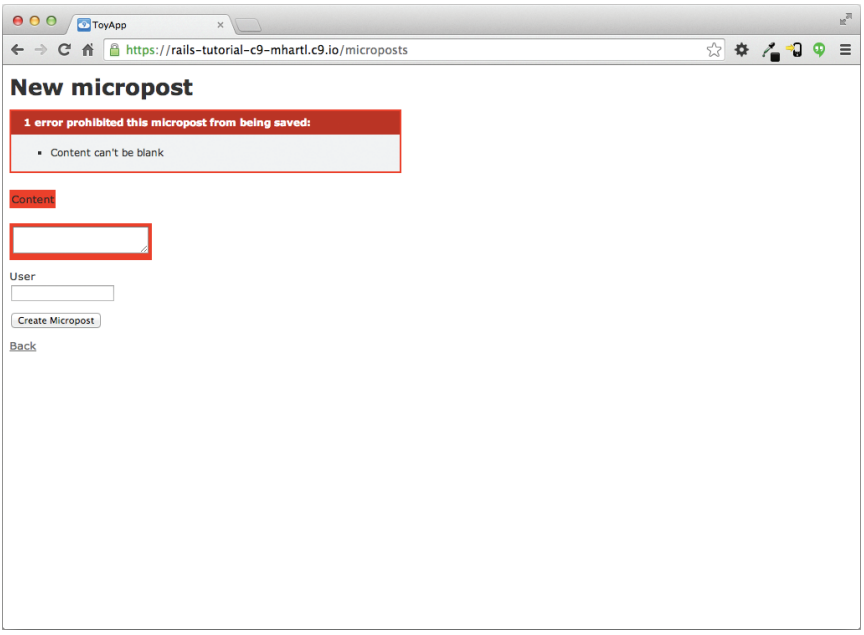


Рис. 2.19 ❖ Результат проверки наличия содержимого микрособщений

2. Измените листинг 2.19, заменив `FILL_IN` программным кодом, проверяющим наличие информации в атрибутах имени и адреса электронной почты в модели User (рис. 2.20).

Листинг 2.19 ❖ Добавление проверок наличия в модель User (app/models/user.rb)

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

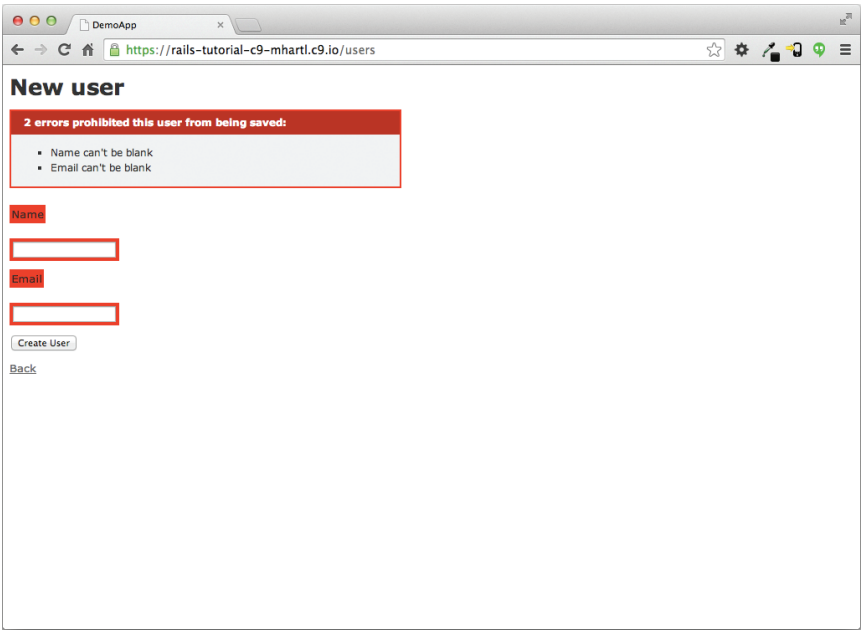


Рис. 2.20 ❖ Результат проверки заполнения полей в модели User

Глава 3

В основном статические страницы

В этой главе мы начнем разработку учебного приложения на профессиональном уровне, оно будет служить примером в остальной части книги. В конечном счете в нем будут реализованы поддержка пользователей и сообщений, полноценная система регистрации и аутентификации, но сейчас мы начнем с самого простого: создания статических страниц. Несмотря на кажущуюся простоту, это весьма поучительное упражнение и идеальное начало для нашего зарождающегося приложения.

Фреймворк Rails предназначен для создания динамических веб-сайтов, поддерживаемых базой данных, но он также хорош для создания статических страниц, которые можно написать на чистом HTML. В самом деле, использование Rails даже для обслуживания статических страниц дает неоспоримое преимущество: динамическое содержимое можно легко добавлять *небольшими* порциями. В этой главе мы узнаем, как это сделать. Попутно получим первый опыт *автоматизированного тестирования*, которое поможет нам быть увереннее в правильности своего кода. Кроме того, хороший набор тестов позволит нам уверенно *реорганизовывать* код, изменяя его форму, но не меняя функций.

3.1. Установка учебного приложения

Так же как в главе 2, перед началом работы необходимо создать новый Rails-проект, который мы в этот раз назовем `sample_app`, как показано в листинге 3.1[1].

---

[1] При работе в облачной IDE часто бывает полезна команда **Goto Anything** (Перейти к...), облегчающая навигацию по файловой системе, позволяя набирать только часть имени файла. В связи с этим существование в одном и том же проекте всех приложений сразу может быть неудобным из-за большого количества файлов с одинаковыми названиями. Например, при поиске файла `Gemfile` будут показаны шесть вариантов, так как в каждом приложении есть похожие файлы `Gemfile` и `Gemfile.lock`. Поэтому вы можете удалить первые два приложения, прежде чем продолжить, для этого перейдите в каталог `workspace` и выполните `rm -rf hello_app/toy_app/` (табл. 1.1). (Поскольку соответствующие репозитории были сохранены в Bitbucket, их всегда можно восстановить позднее.)

Если команды из листинга 3.1 вернут ошибку, например: «Could not find 'railties'» («Не могу найти 'railties'»), значит, установлена неверная версия Rails, и необходимо перепроверить, была ли команда из листинга 1.1 выполнена в точности, как написано.

Листинг 3.1 ❖ Создание нового учебного приложения (sample_app)

```
$ cd ~/workspace
$ rails _4.2.0_ new sample_app
$ cd sample_app/
```

(Так же как в разделе 2.1, пользователи облачной IDE могут создать проект в том же рабочем пространстве, что и приложения из предыдущих двух глав. Необязательно создавать новое.) Следующий шаг – добавление в `Gemfile` гемов, необходимых приложению. Листинг 3.2 идентичен листингам 1.5 и 2.1, за исключением гемов в группе `test`, которые нужны для дополнительных настроек тестирования (раздел 3.7).

> **Примечание.** *Если вы хотите установить сразу все гемы, необходимые учебному приложению, используйте код из листинга 11.66.*

Листинг 3.2 ❖ Gemfile для учебного приложения

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

gem 'rails',                '4.2.0'
gem 'sass-rails',           '5.0.1'
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
```

Как и в предыдущих двух главах, выполните `bundle install`, чтобы установить и подключить гемы, указанные в `Gemfile`, пропустив установку гемов для эксплуатационного окружения с помощью параметра `--without production`[1]:

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

Это позволит пропустить гем `pg` поддержки PostgreSQL в окружении разработки и использовать SQLite для разработки и тестирования. Heroku не рекомендует использовать разные базы данных для окружения разработки и эксплуатации, но для учебного приложения это не будет иметь никакого значения, а SQLite *гораздо* проще, чем PostgreSQL, в установке и настройке[2]. Если вы ранее установили версию гема (например, самого Rails), отличающуюся от указанной в `Gemfile`, лучше обновить гемы с помощью `bundle update`, чтобы убедиться в совпадении версий:

```
$ bundle update
```

Теперь осталось только инициализировать репозиторий Git:

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

Как и в первом приложении, я советую обновить файл `README` (в корневом каталоге приложения), чтобы сделать его более полезным и описательным. Начнем со смены формата RDoc на Markdown:

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

Затем заполним файл, как показано в листинге 3.3.

Листинг 3.3 ❖ Улучшенный файл README для учебного приложения

```
# 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/).
```

Теперь зафиксируем изменения:

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

В разделе 1.4.4 мы выполняли Git-команду `git commit -a -m "Сообщение"` с флагами `-a` (all changes – «все изменения») и `-m` с аргументом-сообщением. Как показано выше, Git позволяет свернуть два флага в один: `git commit -am "Сообщение"`.

---

[1] Стоит отметить, что `--without production` является «запоминаемым параметром», то есть он включится автоматически при следующем запуске `bundle install`.

[2] Я рекомендую научиться устанавливать и настраивать PostgreSQL в окружении разработки, но сейчас не самое подходящее для этого время. Когда время придет, наберите в Google «install configure postgresql <ваша система>» («установка и настройка postgresql <ваша система>», пользователи облачной IDE вместо «<ваша система>» должны указать Ubuntu).

Поскольку мы будем использовать это учебное приложение на протяжении оставшейся части книги, желательно создать новый репозиторий Bitbucket и отправить туда наш код:

```
$ git remote add origin git@bitbucket.org:<username>/sample_app.git
$ git push -u origin --all      # впервые отправляем репозиторий
```

Чтобы в дальнейшем избежать проблем при интеграции, было бы здорово развернуть приложение на Heroku, даже на этой ранней стадии. Так же как в главах 1 и 2, я предлагаю проделать это по той же схеме, как показано в листингах 1.8 и 1.9[1]. Затем зафиксировать изменения и отправить приложение в Heroku:

```
$ git commit -am "Addhello"
$ heroku create
$ git push heroku master
```

(Как и в разделе 1.5, может появиться несколько предупреждающих сообщений, которые на данном этапе можно игнорировать. Мы избавимся от них в разделе 7.5.) Кроме адреса приложения на сайте Heroku, результат должен совпадать с изображением на рис. 1.18.

Я рекомендую регулярно отправлять и разворачивать учебное приложение в процессе чтения, это автоматически обеспечит вас резервными копиями в удаленном хранилище и позволит обнаруживать ошибки настолько быстро, насколько это вообще возможно. Если вы столкнетесь с проблемами на Heroku, загляните в журналы, чтобы попытаться диагностировать проблему:

```
$ heroku logs
```

> **Примечание.** *Если в конечном итоге вы решите использовать Heroku для развертывания настоящего приложения, выполните настройки веб-сервера, как описывается в разделе 7.5.*

3.2. Статические страницы

Закончив со всеми приготовлениями, мы наконец-то готовы начать разработку учебного приложения. В этом разделе мы сделаем первые шаги в направлении динамических страниц, создав для начала набор методов-действий и представлений, содержащих только статическую разметку HTML[2]. Методы действий находятся

---

[1] Как рассказывалось в главе 2, созданные по умолчанию Rails-страницы, как правило, не работают на Heroku, поэтому сложно сказать, было ли развертывание успешным.

[2] Наш метод создания статических страниц является, пожалуй, самым простым, но это не единственный путь. Оптимальный метод на самом деле зависит от конкретных потребностей; если предполагается *большое* количество статических страниц, использование контроллера StaticPages может стать довольно громоздким, но в нашем учебном приложении нам понадобится лишь несколько страниц. Если вам нужно большое количество статических страниц, взгляните на гем `high_voltage`. Вы можете найти устаревшее, но все еще полезное обсуждение этого вопроса в блоге: http://blog.hasmanythrough.com/2008/4/2/simple-pages.

внутри контроллеров, которые, по сути, являются наборами методов действий, связанных общим назначением. Мы получили некоторое представление о них в главе 2 и придем к более глубокому пониманию, когда начнем более полно использовать REST-архитектуру (начиная с главы 6). Чтобы сориентироваться, полезно вспомнить структуру каталогов Rails из раздела 1.3 (рис. 1.4). В этом разделе мы в основном будем работать в каталогах `app/controllers` и `app/views`.

Как отмечалось в разделе 1.4.4, при использовании Git желательно делать что-то новое в отдельной, тематической, а не в основной (master) ветви. Поэтому, если вы пользуетесь системой управления версиями Git, выполните команду создания отдельной ветви для статических страниц:

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

(Первая команда просто гарантирует, что вы начнете из главной ветви и ветвь `static-pages` будет основана именно на ветви `master`. Можно пропустить эту команду, если вы уже в главной ветви.)

3.2.1. Рождение статических страниц

Чтобы начать работу со статическими страницами, сначала нужно сгенерировать контроллер, вызвав тот же сценарий `generate`, что использовался в главе 2. Так как создается контроллер для управления статическими страницами, назовем его `StaticPages`. Мы также планируем создать методы для страниц Home, Help и About, называться эти методы будут `home`, `help` и `about`. Сценарий `generate` принимает в качестве необязательного параметра список методов действий, поэтому включим действия для страниц Home и Help непосредственно в командную строку, намеренно опустив About, чтобы у нас была возможность узнать, как его добавить вручную (раздел 3.3). Получившаяся команда создания контроллера StaticPages показана в листинге 3.4.

Листинг 3.4 ❖ Создание контроллера StaticPages

```
$ 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
      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
```

Кстати, следует заметить, что `rails g` – это сокращение для `rails generate`, и это только одно из нескольких сокращений, поддерживаемых Rails (табл. 3.1). Для большей ясности в этой книге всегда используются полные команды, но в обычной жизни большинство Rails-разработчиков пользуется теми или иными сокращениями из табл. 3.1.

Прежде чем продолжить, сохраните файлы контроллера StaticPages в удаленном репозитории, если вы пользуетесь Git:

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

Таблица 3.1 ❖ Некоторые сокращения Rails

Полная команда	Сокращение
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ bundle install`	`$ bundle`
`$ rake test`	`$ rake`

Здесь последняя команда организует отправку в тематическую ветвь `static-pages`. В последующих операциях отправки можно опустить аргументы и писать просто

```
$ git push
```

Такая последовательность операций фиксации и отправки представляет шаблон, которому я обычно следую в своей работе, но для краткости отныне я в основном буду опускать подобные промежуточные операции фиксации.

Обратите внимание, что в листинге 3.4 мы передали имя контроллера в «верблюжьей» нотации, а получили название файла контроллера в форме записи с подчеркиваниями – для контроллера StaticPages был создан файл `static_pages_controller.rb`. Это просто соглашение, и в действительности с тем же успехом можно было передать имя контроллера в форме записи с подчеркиваниями:

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

Эта команда сгенерирует контроллер в файле с именем `static_pages_controller.rb`. Поскольку Ruby использует «верблюжью» нотацию для имен классов (раздел 4.4), при упоминании контроллеров я предпочитаю писать их в «верблюжьей» нотации, но это дело вкуса. (Поскольку в Ruby имена файлов обычно пишутся в форме записи с подчеркиваниями, Rails-генератор преобразует одну форму записи в другую с помощью метода `underscore`.)

Кстати, если вы когда-либо допустите ошибку, генерируя код, вам пригодится умение обратить процесс. В блоке 3.1 описывается несколько приемов аннулирования действий в Rails.

Блок 3.1 ❖ Аннулирование действий

Даже если вы очень осторожны, иногда при разработке Rails-приложений кое-что может пойти не так. К счастью, в Rails есть несколько возможностей, которые могут помочь вам откатить изменения. Одной из распространенных ситуаций является желание откатить сгенерированный код – например, когда вы передумали по поводу названия контроллера и хотите удалить созданные файлы. Поскольку Rails создает значительное количество вспомогательных файлов вместе с контроллером (см. листинг 3.4), это не так просто, как простое удаление файла контроллера; откат сгенерированного кода подразумевает удаление не только основного, но и всех остальных вспомогательных файлов. (Фактически, как мы видели в разделах 2.2 и 2.3, `rails generate` может автоматически внести изменения в файл `routes.rb`, которые хотелось бы также автоматически отменить.) В Rails это может быть осуществлено с помощью `rails destroy` с именем созданного элемента. В частности, следующие две команды отменяют друг друга:

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

Аналогично в главе 6 мы создадим *модель* командой:

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

Это действие может быть отменено с помощью:

```
$ rails destroy model User
```

(В данном случае, как оказывается, можно опустить остальные аргументы командной строки. Посмотрим, сможете ли вы понять причину этого после прочтения главы 6.)

Другая техника, связанная с моделями, позволяет откатывать *миграции*, которые мы слегка затронули в главе 2 и с которыми более подробно познакомимся в главе 6. Миграции изменяют состояние базы данных с помощью команды

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

Мы можем откатить один шаг миграции:

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

Чтобы откатить к самому началу (все миграции):

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

Как вы могли догадаться, подстановка любого другого числа вместо `0` откатит миграции до соответствующей версии, где номера версий образуются из последовательного списка миграций.

Познакомившись с этой техникой, мы неплохо подготовились к тому, чтобы справиться с путаницей, неизбежной при разработке.

Создание контроллера StaticPages автоматически обновляет файл маршрутов (config/routes.rb), упоминавшийся в разделе 1.3.4. Этот файл отвечает за реализацию маршрутов (см. рис. 2.11), определяющих соответствия между адресами URL и веб-страницами. Этот файл находится в каталоге config, в ней Rails хранит все необходимые файлы с настройками приложения (рис. 3.1).

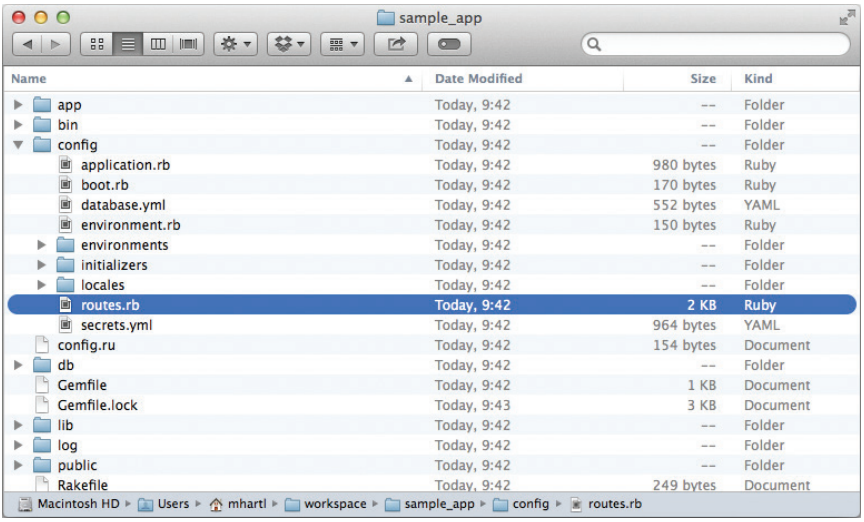


Рис. 3.1 ❖ Содержимое каталога config учебного приложения

Так как мы включили методы home и help в листинг 3.4, маршруты к ним уже были сохранены в файле, как видно в листинге 3.5.

Листинг 3.5 ❖ Маршруты к методам home и help в контроллере StaticPages (config/routes.rb)

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

Правило

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

направляет запросы к URL /static_pages/home в метод home контроллера StaticPages. Кроме того, ключевое слово get указывает, что маршрут предназначен для обработки запросов GET, одного из основных *HTTP-глаголов*, поддерживаемых протоколом передачи гипертекста (блок 3.2). В нашем случае это значит, что, обращаясь к методу home внутри контроллера StaticPages, мы автоматически полу-

чаем страницу по адресу /static_pages/home. Чтобы увидеть результат, запустите Rails-сервер, как описано в разделе 1.3.2:

```
$ rails server -b $IP -p $PORT  # Используйте просто `rails server`,
                                # если запускаете локально
```

Затем перейдите по адресу /static_pages/home (рис. 3.2).

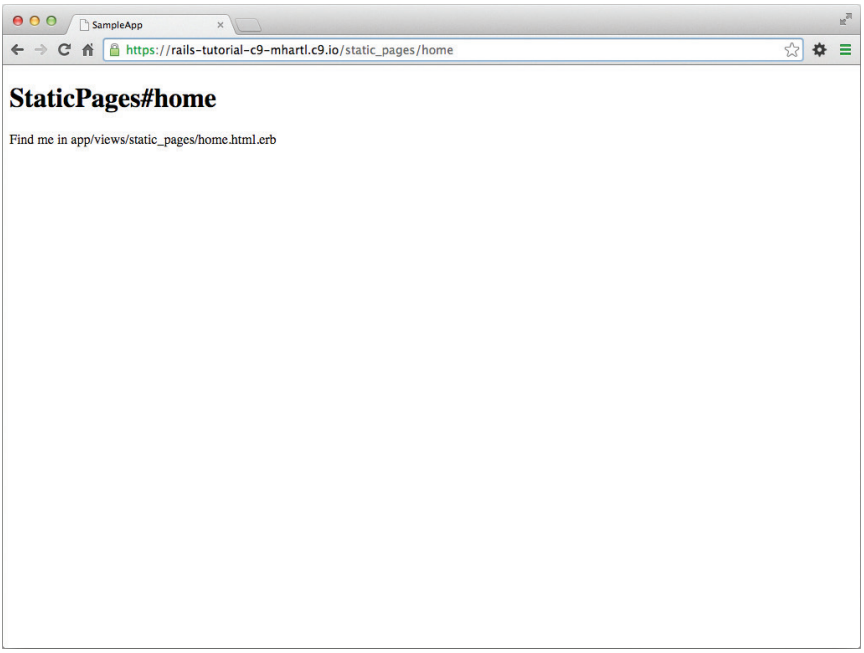


Рис. 3.2 ❖ Необработанное представление home (/static_pages/home)

Блок 3.2 ❖ GET и т. д.

Протокол передачи гипертекста (hypertext transfer protocol, HTTP) определяет основные операции GET, POST, PATCH и DELETE. Они относятся к операциям между *клиентом* (обычно веб-браузером, таким как Chrome, Firefox или Safari) и *сервером* (обычно веб-сервером, таким как Apache или Nginx). (Важно понимать, что при разработке Rails-приложений на локальном компьютере клиент и сервер физически находятся на одной машине, но в целом они разные.) Акцент на глаголы HTTP является типичным для веб-фреймворков (включая Rails), подверженных влиянию *REST-архитектуры*, основы которой мы рассмотрели в главе 2 и более подробно обсудим в главе 7.

GET является самой распространенной HTTP-операцией, используемой в Веб для *чтения* данных; она просто означает «получить страницу», и каждый раз, когда вы посещаете сайт, такой как http://www.google. com/ или http://www.wikipedia. org/, ваш браузер передает запрос GET. POST – это следующая по распространенности операция; запросы этого типа передаются браузером при отправке форм. В Rails-приложениях запросы POST обычно используются для *создания* чего-либо

(хотя HTTP позволяет использовать запросы POST для изменения существующих данных). Например, запрос POST формируется при отправке заполненной формы регистрации нового пользователя на удаленном сайте. Два других глагола, PATCH и DELETE, предназначены для *изменения* и *удаления* сущностей на удаленном сервере. Эти запросы встречаются реже, чем GET и POST, потому что браузеры не в состоянии отправлять их в естественном виде, но некоторые фреймворки (включая Ruby on Rails) могут делать *вид*, что браузеры отправляют такие запросы. В результате Rails поддерживает все четыре типа запросов: GET, POST, PATCH и DELETE.

Чтобы понять, как формируется эта страница, рассмотрим контроллер StaticPages в текстовом редакторе. Он должен напоминать код в листинге 3.6. Здесь можно заметить, что, в отличие от контроллеров Users и Microposts из главы 2, контроллер StaticPages не использует стандартных операций REST. И это нормально для статических страниц: архитектура REST – не панацея от всех проблем.

Листинг 3.6 ❖ Контроллер StaticPages, созданный командой в листинге 3.4 (app/controllers/static_pages_controller.rb)

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

Ключевое слово class сообщает, что в static_pages_controller.rb определяется *класс*, в данном случае с именем StaticPagesController. Классы – это лишь удобный способ организации *функций* (также называемых *методами*), таких как home и help, которые определяются с помощью ключевого слова def. Как обсуждалось в разделе 2.3.4, угловая скобка < указывает, что StaticPagesController *наследует* класс ApplicationController; как мы очень скоро увидим, это означает, что наши страницы по умолчанию получают большое количество функций. (Мы узнаем больше об этих двух классах и наследовании в разделе 4.4.)

В случае с контроллером StaticPages оба его метода изначально пусты:

```
def home
end

def help
end
```

В языке Ruby это означает, что методы просто ничего не делают. В Rails ситуация обстоит иначе: StaticPagesController – это класс Ruby, а так как он наследует ApplicationController, его методы получают поведение, характерное для Rails. Если открыть страницу по адресу URL /static_pages/home, Rails обратится к контроллеру StaticPages и выполнит код в методе home, а затем отобразит *представление* (view – V в MVC, как описывается в разделе 1.3.3), соответствующее данному методу. В данном случае метод home пуст, поэтому единственное, что происходит

при посещении /static_pages/home, – это отображение представления. Но как выглядит представление и где его найти?

Если еще раз посмотреть на вывод в листинге 3.4, можно найти соответствие между методами действий и представлениями: метод, такой как home, имеет соответствующее представление home.html.erb. В разделе 3.4 мы узнаем, что означает часть расширения .erb; судя по другой части расширения .html, вы вряд ли будете удивлены, что оно, в принципе, выглядит как HTML (листинг 3.7).

Листинг 3.7 ❖ Сгенерированное представление для страницы Home (app/views/static_pages/home.html.erb)

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

Аналогично выглядит представление для метода help (листинг 3.8).

Листинг 3.8 ❖ Сгенерированное представление для страницы Help (app/views/static_pages/help.html.erb)

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

Оба представления – лишь заглушки: у них есть заголовок (тег h1) и абзац (тег p), содержащий полный путь к соответствующему файлу.

3.2.2. Доработка статических страниц

В разделе 3.4 мы добавим немного (очень немного) динамического содержимого, а пока, будучи статичными, эти представления из листингов 3.7 и 3.8 подчеркивают важный момент: Rails-представления могут просто содержать статическую разметку HTML. Это значит, что можно изменять страницы Home и Help даже без знания Rails, как показано в листингах 3.9 и 3.10.

Листинг 3.9 ❖ Измененная разметка HTML для страницы 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>
```

Листинг 3.10 ❖ Измененная разметка HTML для страницы 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>.
  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>
```

Результат изменений, представленных в листингах 3.9 и 3.10, показан на рис. 3.3 и рис. 3.4.

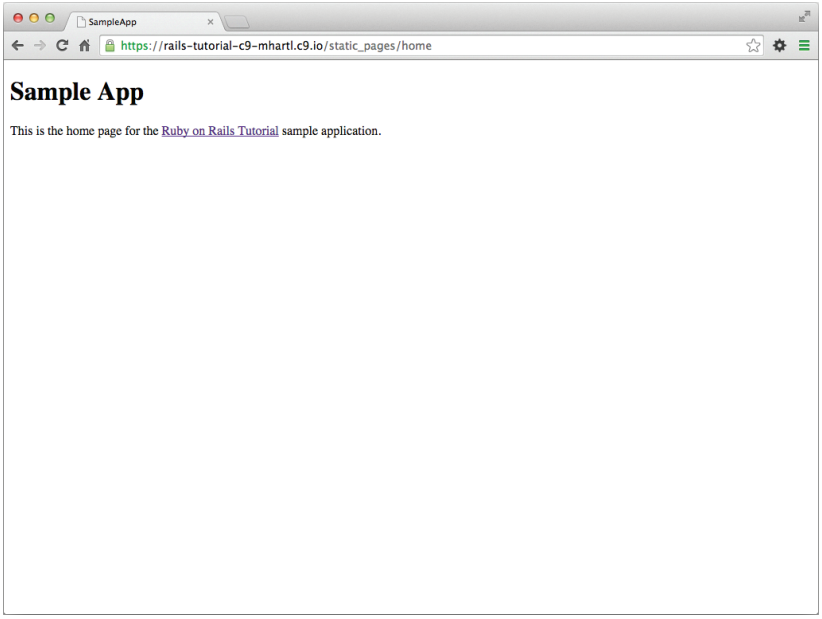


Рис. 3.3 ❖ Измененная страница Home

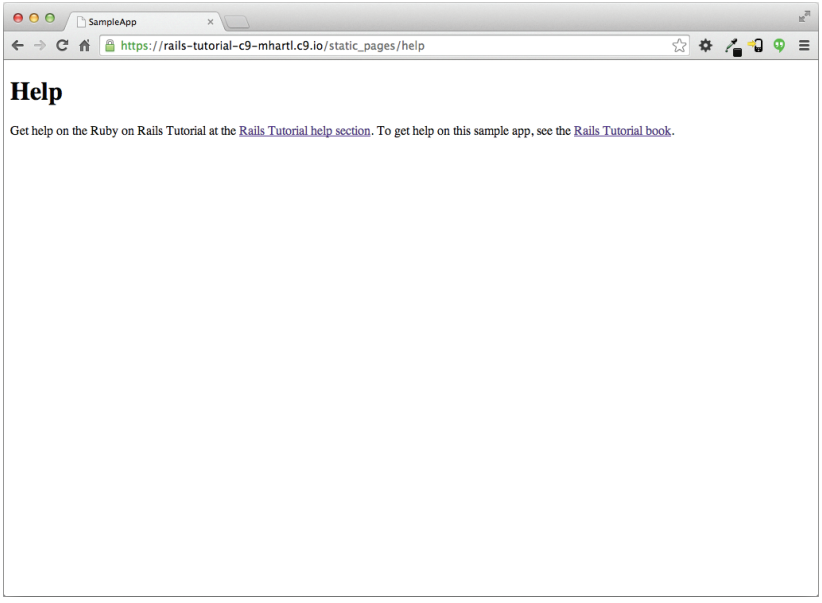


Рис. 3.4 ❖ Измененная страница Help

3.3. Начало работы с тестированием

Создав и заполнив страницы Home и Help учебного приложения, мы собираемся добавить страницу About. При совершении изменений такого характера хорошей привычкой является добавление *автоматизированных тестов*, помогающих убедиться в правильной реализации какой-либо функции. Итоговый *набор тестов*, разрабатываемый в процессе построения приложения, будет служить страховкой и выполняемой документацией для исходного кода приложения. При правильном подходе написание тестов позволяет даже *ускорить* разработку, несмотря на необходимость писать дополнительный код, так как в конечном счете мы будем тратить меньше времени на поиски ошибок. Это верно, только если вы мастер в написании тестов, и это еще одна важная причина начать заниматься этим как можно раньше.

Практически все разработчики Rails согласны, что тестирование является хорошей идеей, но существуют разные мнения о деталях этого процесса. Особенно оживленная дискуссия идет по поводу методологии разработки через тестирование (Test-Driven Development, TDD)[1], согласно который программист сначала пишет тесты, завершающиеся неудачей, а затем – код приложения, обеспечивающий прохождение тестов. В этой книге применяется простой и интуитивно понятный подход к тестированию с использованием TDD, когда это удобно (блок 3.3).

Блок 3.3 ❖ Когда тестировать

Решая, когда и как тестировать, полезно понимать – *зачем* тестировать. На мой взгляд, автоматизированные тесты дают три основных преимущества:

1) тесты защищают от *регрессий*, когда действовавшая ранее функция прекращает работать по какой-то причине;
2) тесты позволяют с большей уверенностью *реорганизовывать* код (то есть изменять его форму без изменения функциональности);
3) тесты играют роль *клиента* для кода приложения, помогая определить его дизайн и интерфейс с другими частями системы.

Ни одно из указанных преимуществ не *требует* писать тесты заранее, однако во многих обстоятельствах разработка через тестирование (TDD) оказывается весьма ценным инструментом. Решение, когда и как выполнять тестирование, зависит и от того, насколько комфортно для вас написание тестов; многие разработчики обнаруживают, что, улучшив свои навыки в создании тестов, они становятся склонными начинать именно с них. Решение также зависит от сложности тестов относительно кода приложения, насколько точно известна желаемая функциональность и с какой вероятностью она в будущем может нарушиться.

В этом отношении мог бы пригодиться набор рекомендаций, когда писать тесты первыми (или писать тесты вообще). Вот несколько таких советов, основанных на моем собственном опыте:

- когда тест короткий или простой относительно кода тестируемого приложения, желательно написать тест первым;

[1] См., например, статью «TDDisdead. Longlivetesting» от создателя Rails Дэвида Хейнмейера Ханссона (David Heinemeier Hansson).

- когда желаемое поведение не полностью понятно, первым желательно написать код приложения, а затем тест для систематизации результата;
- поскольку безопасность является главным приоритетом, лучше, если ошибка сначала проявится при написании тестов модели безопасности;
- при обнаружении каждой ошибки пишите тест для ее воспроизведения и защиты от регрессии, а затем пишите код приложения, исправляющий ее;
- не особенно много смысла писать тесты для кода (такого как подробная HTML-структура), который почти наверняка изменится в будущем;
- пишите тесты перед реорганизацией кода, сосредоточившись на тестировании кода, подверженного ошибкам, который с большой вероятностью может быть нарушен.

На практике эти рекомендации означают, что обычно в первую очередь пишутся тесты для моделей и контроллеров, а во вторую – интеграционные тесты (тестируют функциональность через модели, представления и контроллеры). А когда пишется не слишком хрупкий и не подверженный ошибкам код, или код, который с большой вероятностью изменится (как это часто бывает с представлениями), мы обычно пропускаем тестирование вообще.

Нашими главными инструментами тестирования будут *тесты контроллеров* (начиная с этого раздела), *тесты моделей* (начиная с главы 6) и *интеграционные тесты* (начиная с главы 7). Интеграционные тесты имеют особую ценность, поскольку позволяют имитировать действия пользователя, взаимодействующего с приложением через браузер. Интеграционные тесты, в конце концов, станут нашей основной технологией тестирования, но тесты контроллеров позволят нам начать с чего попроще.

3.3.1. Наши первые тесты

А сейчас пришло время добавить страницу About в наше приложение. Как мы увидим, тест страницы короток и прост, поэтому последуем рекомендациям из блока 3.3 и сначала напишем тест. Затем мы используем этот тест в качестве проводника для написания кода приложения.

На первых порах тестирование может показаться сложной задачей, требующей обширных знаний Rails и Ruby. Поэтому на ранней стадии написание тестов может казаться безнадежно пугающим. К счастью, Rails уже проделал за нас самую трудную часть – команда `rails generate controller` (листинг 3.4) автоматически создала файл тестов, чтобы мы могли начать не с пустого места:

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

Давайте заглянем в него (листинг 3.11).

Листинг 3.11 ❖ Созданные по умолчанию тесты для контроллера StaticPages **ЗЕЛЕНЫЙ** (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
```

На данном этапе не требуется детально понимать синтаксис листинга 3.11, и все же в нем можно определить два теста, по одному для каждого метода действия контроллера, которые мы включили в листинге 3.4. Каждый тест просто запрашивает действие и проверяет (через утверждение) успешность выполнения. Здесь ключевое слово `get` подсказывает, что тесты интерпретируют страницы Home и Help как самые обычные веб-страницы, возвращаемые в ответ на запрос `GET` (блок 3.2). Ответ `:success` – это абстрактное представление возвращаемого кода состояния HTTP (в данном случае **200 OK**). Другими словами, следующий тест:

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

говорит: «Проверим страницу Home, послав запрос `GET` методу `home`, и убедимся, что в ответ возвращается код состояния “success” (успех)».

Чтобы начать цикл тестирования, нужно запустить набор тестов и убедиться, что тесты выполняются благополучно. Сделать это можно с помощью утилиты `rake` (блок 2.1)[1]:

Листинг 3.12 ❖ ЗЕЛЕНЫЙ

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

Как и требовалось, наш первоначальный набор тестов выполняется без ошибок (**ЗЕЛЕНЫЙ**). (Вы не увидите зеленого цвета, пока не добавите отчетов о минитестах в разделе 3.7.1.) Кстати, тестам требуется некоторое время для выполнения, это связано с двумя факторами: (1) запуск *сервера Spring* для предварительной загрузки компонентов окружения Rails, это происходит только в первый раз; и (2) издержки, связанные со временем запуска Ruby. (Второй фактор улучшается при использовании Guard, как предлагается в разделе 3.7.3.)

---

[1] Как отмечалось в разделе 2.2, в некоторых системах, включая облачную IDE, необязательно использовать полную команду `bundle exec`, рекомендованную в разделе 1.2.1. Я употребляю ее здесь лишь для полноты картины. На практике обычно я пропускаю `bundle exec`, пока не получу ошибку, и только тогда повторяю попытку с `bundle exec`, и смотрю, не заработало ли.

3.3.2. Красный

Как отмечалось в блоке 3.3, разработка через тестирование предполагает, что сначала создаются тесты, терпящие неудачу, затем код приложения, обеспечивающий их прохождение, и последующий рефакторинг кода при необходимости. Поскольку в большинстве инструментов тестирования неудачные попытки прохождения тестов представлены красным цветом, а успешные – зеленым, эта последовательность известна как цикл «Красный, зеленый, рефакторинг». В этом разделе мы закончим первый шаг в этом цикле, написав **КРАСНЫЙ** тест, заканчивающийся неудачей. Затем, в разделе 3.3.3, перейдем к **ЗЕЛЕНОМУ** и в разделе 3.4.3 – к рефакторингу[1].

Итак, на первом шаге напишем для страницы About тест, терпящий неудачу. Следуя модели из листинга 3.11, нетрудно предугадать правильный тест, он показан в листинге 3.13.

Листинг 3.13 ❖ Тест для страницы About **КРАСНЫЙ**
(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
```

Как видите, тест для страницы About (выделенный фрагмент в листинге 3.13) в точности повторяет тесты страниц Home и Help, отличаясь только со словом «about» вместо «home» или «help».

Как и требовалось, сразу после создания тест терпит неудачу:

Листинг 3.14 ❖ **КРАСНЫЙ**

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

[1] По умолчанию `rake test` окрашивает красным тесты, завершившиеся неудачей, но успешные не окрашивает зеленым. Организация настоящего цикла красный–зеленый описывается в разделе 3.7.1.

3.3.3. Зеленый

Теперь, когда у нас есть тест, терпящий неудачу (**КРАСНЫЙ**), будем использовать его сообщения об ошибках, чтобы прийти к успешному тесту (**ЗЕЛЕНЫЙ**), реализовав рабочую страницу About.

Начнем с исследования сообщения об ошибке, которое выдает неуспешный тест[1]:

Листинг 3.15 ❖ КРАСНЫЙ

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

Сообщение говорит об отсутствии маршрута, совпадающего с желаемой комбинацией метод/контроллер, и как бы намекает, что необходимо добавить строку в файл маршрутов. Мы можем сделать это по шаблону из листинга 3.5, как показано в листинге 3.16.

Листинг 3.16 ❖ Добавление маршрута about **КРАСНЫЙ** (config/routes.rb)

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

Выделенная строка в листинге 3.16 требует от Rails передать `GET`-запрос к URL `/static_pages/about` в метод `about` контроллера StaticPages.

Запустив набор тестов еще раз, мы увидим, что он остался **КРАСНЫМ**, но сообщение об ошибке изменилось:

Листинг 3.17 ❖ КРАСНЫЙ

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

Теперь сообщение указывает на отсутствие метода `about` в контроллере StaticPages, который мы можем добавить, по аналогии с методами `home` и `help`, как показано в листинге 3.18.

Листинг 3.18 ❖ Добавление метода about в контроллер StaticPages **КРАСНЫЙ** (app/controllers/static_pages_controller.rb)

```
class StaticPagesController < ApplicationController

  def home
```

[1] В некоторых системах может потребоваться прокрутить трассировку стека, помогающую отследить источник ошибки в исходном коде. В разделе 3.7.2 рассказывается, как настроить фильтрацию трассировки, чтобы удалить ненужные строки.

```
  end

  def help
  end

  def about
  end
end
```

Но, как и раньше, набор тестов остался **КРАСНЫЙ**, однако сообщение об ошибке опять изменилось:

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

Оно указывает на отсутствие шаблона, который в Rails, по сути, то же самое, что и представление. Как рассказывалось в разделе 3.2.1, метод с именем `home` связан с представлением `home.html.erb`, находящимся в каталоге `app/views/static_pages`, а значит, в том же каталоге нужно создать новый файл с именем `about.html.erb`.

Способ создания файла зависит от настроек системы, но большинство текстовых редакторов позволяет щелкнуть правой кнопкой мыши внутри каталога, где нужно создать файл и выбрать пункт контекстного меню **New File** (Создать файл). Как вариант можно использовать меню **File** (Файл), чтобы создать новый файл, и при его сохранении выбрать требуемый каталог. Наконец, можно использовать мой любимый фокус с Unix-командой `touch`:

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

Несмотря на то что команда `touch` предназначена для установки времени последнего изменения файлов или каталогов, как побочный эффект она создает новый (пустой) файл, если он отсутствует. (В облачной IDE может понадобиться обновить дерево каталогов, как уже говорилось в разделе 1.3.1.)

После создания файла `about.html.erb` его нужно заполнить содержимым из листинга 3.19.

Листинг 3.19 ❖ Код для страницы About **ЗЕЛЕНЫЙ**
(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>
```

Теперь при попытке выполнить `rake test` набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 3.20 ❖ ЗЕЛЕНЫЙ

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

Разумеется, никогда нелишне взглянуть на страницу в браузере, чтобы убедиться, что наши тесты совсем не сошли с ума (рис. 3.5).

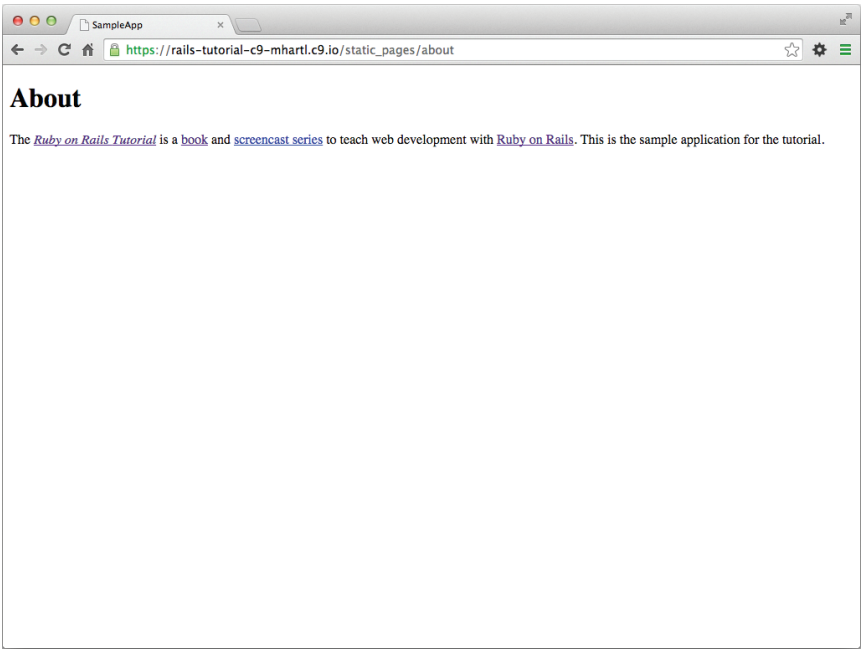


Рис. 3.5 ❖ Новая страница About (/static_pages/about)

3.3.4. Рефакторинг

Теперь, получив **ЗЕЛЕНЫЙ** набор тестов, можно переходить к реорганизации кода. В процессе разработки код часто начинает «дурно пахнуть», то есть становится уродливым, раздутым или наполненным повторяющимися фрагментами. Компьютеру, конечно, все равно, но людям – нет, поэтому важно хранить код в чистоте, периодически выполняя рефакторинг. Наше учебное приложение слишком маленькое, чтобы реорганизовывать его прямо сейчас, но когда дурной запах начнет просачиваться через все щели, мы займемся рефакторингом (в разделе 3.4.3).

3.4. Слегка динамические страницы

После создания методов действий и представлений для нескольких статических страниц мы сделаем их *чуть-чуть* динамичнее, добавив содержимое, которое будет меняться в зависимости от страницы: мы добавим в каждую страницу свой за-

головок, он будет отражать ее содержимое. Является ли это *действительно* динамическим контентом – спорно, но в любом случае это закладывает необходимую основу для реализации динамического содержимого в главе 7.

Наш план заключается в добавлении в страницы Home, Help и About заголовков, разных для разных страниц. Для этого нам потребуется добавить тег `<title>` в соответствующие представления. Большинство браузеров отображает содержимое тега `<title>` в заголовке своего окна, к тому же он важен для поисковой оптимизации. Мы снова будем использовать полный цикл «Красный, зеленый, рефакторинг»: сначала добавим простые тесты для заголовков страниц (**КРАСНЫЙ**), затем добавим заголовки во все три страницы (**ЗЕЛЕНЫЙ**) и в самом конце задействуем файл *макета*, чтобы устранить повторения (рефакторинг). В конце этого раздела все три страницы получат заголовки в формате: «<имя страницы> | Ruby on Rails Tutorial Sample App», и первая часть заголовка будет изменяться в зависимости от страницы (табл. 3.2).

Таблица 3.2 ❖ Статические (в основном) страницы для учебного приложения

Страница	URL	Основной заголовок	Изменяющийся заголовок
Home	/static_pages/home	`"Ruby on Rails Tutorial SampleApp"`	`"Home"`
Help	/static_pages/help	`"Ruby on Rails Tutorial SampleApp"`	`"Help"`
About	/static_pages/about	`"Ruby on Rails Tutorial SampleApp"`	`"About"`

Команда `rails new` (листинг 3.1) создала файл макета по умолчанию, но для нас будет весьма поучительно изначально не обращать на него внимания, а чтобы было проще, изменим его имя:

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

Обычно этого не нужно делать в реальном приложении, но нам проще будет понять его назначение, если предварительно выведем файл из строя.

3.4.1. Тестирование заголовков (КРАСНЫЙ)

Чтобы добавить заголовки страниц, нужно познакомиться со структурой типичной веб-страницы, она приводится в листинге 3.21.

Листинг 3.21 ❖ HTML-структура типичной веб-страницы

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

Структура в листинге 3.21 включает объявление *типа документа* – в самом верху страницы, которое сообщает браузерам, какая версия HTML используется (в данном случае HTML5)[1]; раздел `head` с текстом «Greeting» в теге `title`; и раздел `body` с текстом «Hello, world!» в теге `p` (абзац). (Отступы необязательны – HTML нечувствителен к ним и игнорирует и табуляцию, и пробелы – но они делают структуру документа более удобной для просмотра.)

Напишем простые тесты для каждого из заголовков в табл. 3.2, объединив тесты из листинга 3.13 с методом `assert_select`, проверяющим наличие определенного тега HTML (его иногда называют «селектором», отсюда и имя `assert_select`)[2]:

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

Эта инструкция проверяет наличие тега<title> со строкой «Home | Ruby on Rails Tutorial Sample App». Применив эту идею ко все трем статическим страницам, мы получим тесты, показанные в листинге 3.22.

Листинг 3.22 ❖ Тесты контроллера StaticPages с тестами заголовков **КРАСНЫЙ** (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
```

(Если вас беспокоит повторение базового заголовка «Ruby on Rails Tutorial Sample App», загляните в упражнения в разделе 3.6.)

Теперь убедимся, что набор тестов стал **КРАСНЫМ**:

---

[1] HTML продолжает развиваться; явно указав тип документа, мы увеличили вероятность правильного отображения наших страниц браузерами в будущем. Простой тег<!DOCTYPE html> характерен для самого последнего HTML-стандарта – HTML5.

[2] Список утверждений (`assert_*`), часто используемых в тестах, можно найти в статье о тестировании в Rails: http://guides.rubyonrails.org/testing.html#available-assertions.

Листинг 3.23 ❖ КРАСНЫЙ

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

3.4.2. Добавление заголовков страниц (ЗЕЛЕНЫЙ)

А сейчас добавим заголовок в каждую страницу, чтобы обеспечить успешное выполнение тестов из раздела 3.4.1. Применив базовую структуру HTML из листинга 3.21 к странице Home в листинге 3.9, получаем листинг 3.24.

Листинг 3.24 ❖ Представление для страницы Home с полной HTML-структурой **КРАСНЫЙ** (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>
```

Соответствующая веб-страница показана на рис. 3.6[1].

По образу и подобию изменим страницы Help (листинг 3.25) и About (листинг 3.26).

Листинг 3.25 ❖ Представление для страницы Help с полной HTML-структурой **КРАСНЫЙ** (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
```

[1] Большинство скриншотов в этой книге сделано в Google Chrome, но рис. 3.6 получен в Safari, так как Chrome не отображает полного заголовка страницы.

```
        Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

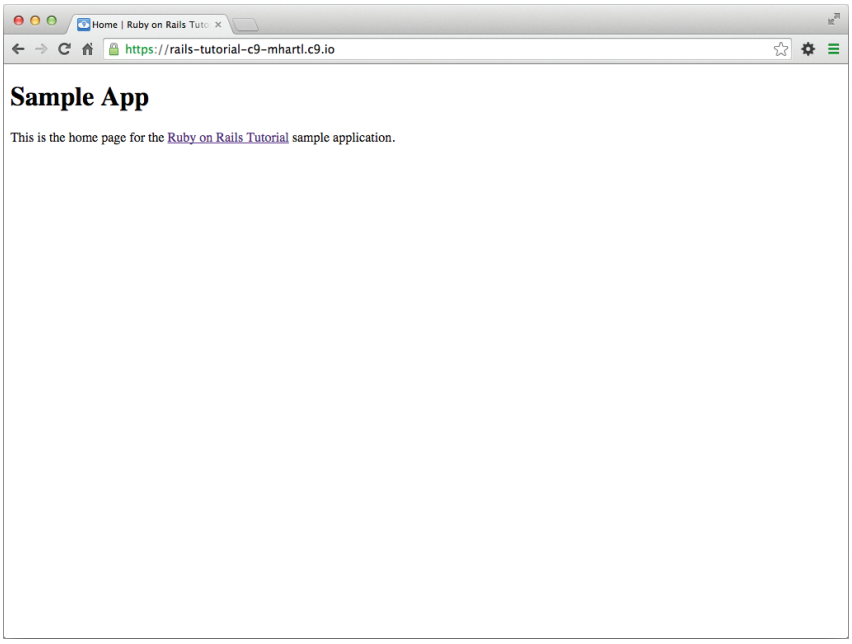


Рис. 3.6 ❖ Страница Home с заголовком

Листинг 3.26 ❖ Представление для страницы About с полной HTML-структурой **ЗЕЛЕНЫЙ** (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>
```

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 3.27 ❖ ЗЕЛЕНЫЙ

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

3.4.3. Макеты и встроенный код на Ruby (рефакторинг)

Мы многого добились в этом разделе, создав три действующие страницы с использованием Rails-контроллеров и методов действий, но они содержат исключительно статическую разметку HTML и, следовательно, не раскрывают всех возможностей Rails. Кроме того, они имеют повторяющиеся фрагменты:

- заголовки страниц почти (но не совсем) одинаковые;
- во всех трех страницах повторяется текст «Ruby on Rails Tutorial Sample App»;
- в каждой странице повторяется вся HTML-структура.

Повторяющийся код нарушает важный принцип Не Повторяй Себя (Don't Repeat Yourself, DRY); в этом разделе мы осушим код, удалив повторения. В конце снова запустим тесты из раздела 3.4.2, чтобы убедиться, что заголовки в ходе рефакторинга ничего не сломали.

Как это ни парадоксально, но первым шагом на пути устранения дублирования будет его усиление: мы сделаем названия страниц, которые в настоящее время очень похожи, полностью идентичными. Это позволит легко удалить все повторения одним махом.

Данный прием опирается на *встраивание* в представления программного кода на Ruby. Поскольку в заголовках страниц Home, Help и About есть переменная составляющая, мы будем использовать специальную Rails-функцию `provide` для установки разных заголовков в разных страницах. Увидеть, как это работает, можно, заменив буквальный заголовок «Home» в `home.html.erb` кодом из листинга 3.28.

Листинг 3.28 ❖ Представление для страницы Home с заголовком, включающим код на Ruby **ЗЕЛЕНЫЙ** (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>
```

Листинг 3.28 – это первый пример со встроенным кодом на Ruby (*embedded Ruby*, или *ERb*: теперь вы знаете, почему HTML-представления имеют расширение `.html.erb`.) ERb – это основная система шаблонов для включения динамического содержимого в веб-страницы[1]. Код

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

указывает (с помощью `<% ... %>`), что Rails должен вызвать функцию `provide` и связать строку `"Home"` с меткой `:title`[2]. Затем в заголовке используется тесно связанная запись `<%= ... %>` для вставки заголовка в шаблон с помощью Ruby-функции `yield`[3]:

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

(Разница между этими двумя типами встроенного кода на Ruby заключается в том, что конструкция `<% ... %>` просто *выполняет* код, находящийся внутри ее, а конструкция `<%= ... %>` выполняет код *и вставляет* результат в шаблон.) Получившаяся страница совершенно не изменилась, только переменная часть заголовка теперь динамически генерируется встроенным кодом на Ruby.

Проверим, что все это работает, запустив набор тестов и убедившись, что он все такой же **ЗЕЛЕНЫЙ**:

Листинг 3.29 ❖ ЗЕЛЕНЫЙ

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

Теперь можно сделать соответствующие замены в страницах Help и About (листинги 3.30 и 3.31).

Листинг 3.30 ❖ Представление для страницы Help с заголовком на основе встроенного Ruby **ЗЕЛЕНЫЙ** (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
```

---

[1] Существует еще одна популярная система, Haml (http://www.haml-lang.com/). Лично мне она очень нравится, но пока она недостаточно стандартизована, чтобы использовать ее во вводной книге.

[2] Опытные Rails-программисты, возможно, ожидали увидеть здесь `content_for`, но она плохо работает с ресурсами. Функция `provide` является ее заменой.

[3] Изучавшие Ruby прежде могли бы предположить, что Rails передает содержимое в блок, и эта догадка будет верна. Но для разработки веб-приложений с Rails это не так уж важно знать.

```
      <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>
```

Листинг 3.31 ❖ Представление для страницы About с заголовком на основе встроенного Ruby **ЗЕЛЕНЫЙ** (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>
```

Теперь, после встраивания Ruby в заголовки, все наши страницы получили следующую структуру:

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

Другими словами, все страницы получили одинаковую структуру, в том числе содержимое тега `title`, исключая данные внутри тега `body`.

Чтобы вынести за скобки повторяющуюся структуру, Rails предоставляет специальный файл макета `application.html.erb`, который мы переименовали в начале раздела (раздел 3.4) и который теперь восстановим:

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

Чтобы получить работающий макет, нужно заменить указанный в нем заголовок по умолчанию на тот, что был создан встраиванием кода на Ruby:

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

Получившийся макет представлен в листинге 3.32.

Листинг 3.32 ❖ Макет сайта учебного приложения **ЗЕЛЕНЫЙ** (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>
```

Обратите внимание на специальную строку

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

Этот код отвечает за вставку содержимого страницы в макет. Нет необходимости понимать, как это на самом деле работает; куда важнее, что использование этого макета обеспечивает при посещении страницы, например `/static_pages/home`, преобразование содержимого `home.html.erb` в разметку HTML и вставку ее на место `<%= yield %>`.

Также стоит отметить, что по умолчанию Rails-макет включает несколько дополнительных строк:

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

Этот код организует подключение стилей и сценариев JavaScript-приложения, являющихся частью ресурсов (раздел 5.2.1), совместно с Rails-методом `csrf_meta_tags`, предотвращающим подделку межсайтовых запросов (CSRF), одного из видов вредоносных веб-атак.

Сейчас представления в листингах 3.28, 3.30 и 3.31 все еще содержат полную структуру HTML, включенную в макет, поэтому удалим ее, оставив только внутреннее содержимое. Получившиеся после очистки представления показаны в листингах 3.33, 3.34 и 3.35.

Листинг 3.33 ❖ Страница Home после удаления HTML-структуры **ЗЕЛЕНЫЙ** (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>
```

Листинг 3.34 ❖ Страница Help после удаления HTML-структуры **ЗЕЛЕНЫЙ** (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>
```

Листинг 3.35 ❖ Страница About после удаления HTML-структуры **ЗЕЛЕНЫЙ** (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>
```

Страницы Home, Help и About с этими представлениями выглядят точно так же, как прежде, но в них теперь гораздо меньше повторений.

Опыт показывает, что даже простой рефакторинг чреват ошибками, и из-за него все может пойти наперекосяк. Это одна из причин создания хорошего набора тестов. Вместо перепроверки каждой страницы на правильность – эта процедура не слишком сложна вначале, но быстро становится громоздкой по мере разрастания приложения – мы можем просто убедиться, что набор тестов все еще **ЗЕЛЕНЫЙ**:

Листинг 3.36 ❖ **ЗЕЛЕНЫЙ**

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

Это не доказательство верности нашего кода, но значительно увеличивает такую вероятность, тем самым обеспечивая страховку для защиты от будущих ошибок.

3.4.4. Установка корневого маршрута

Теперь, когда мы подготовили страницы нашего сайта и начали создавать достойный набор тестов, установим корневой маршрут приложения. Как уже говорилось в разделах 1.3.4 и 2.2.2, для этого нужно отредактировать файл `routes.rb` и связать адрес / со страницей по нашему выбору, в данном случае со страницей Home. (Я также рекомендую заодно удалить метод `hello` из контроллера Application, если вы добавили его в разделе 3.1.) Как показано в листинге 3.37, это означает замену сгенерированного правила `get` в листинге 3.5 следующим кодом:

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

Это меняет URL `static_pages/home` на пару контроллер/действие `static_pages#home`, которая обеспечивает передачу запроса `GET` к адресу / в метод `home` контроллера StaticPages. Получившийся файл маршрутов показан на рис. 3.7. (Обратите внимание, что теперь предыдущий маршрут `static_pages/home` больше не будет работать.)

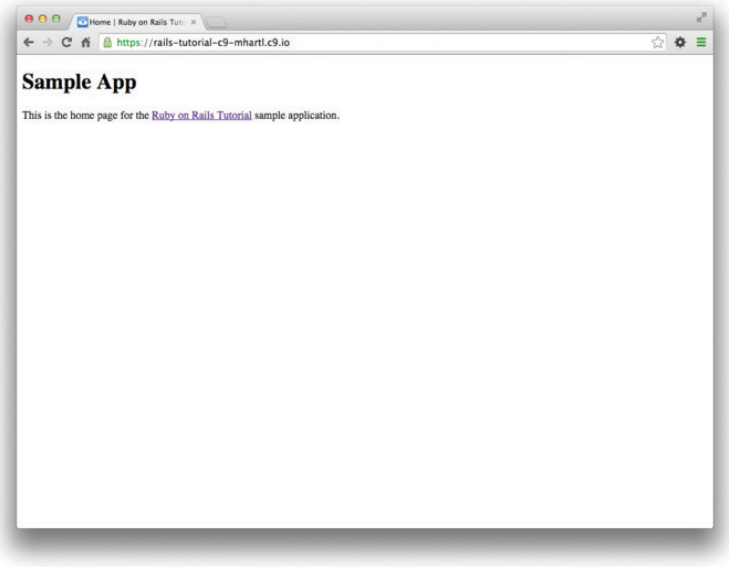


Рис. 3.7 ❖ Страница Home связана с корневым маршрутом

Листинг 3.37 ❖ Определение корневого маршрута к странице Home (config/routes.rb)

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

3.5. Заключение

На первый взгляд, в этой главе мы мало чего добились: мы начали со статических страниц, а закончили... *в основном* статическими страницами. Но внешность обманчива: задействовав в разработке Rails-контроллеры, методы и представления, мы теперь в состоянии добавить произвольный объем динамического содержимого в наш сайт. Задача всей оставшейся части книги – показать, как именно это работает.

Прежде чем двигаться дальше, потратим немного времени, чтобы зафиксировать изменения и объединить их с основной ветвью. Еще в разделе 3.2 мы создали в Git ветвь для разработки статических страниц. Если вы не производили фиксации во время чтения, сначала выполните фиксацию, указывающую, что мы достигли точки остановки:

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

Затем объедините изменения главной ветвью, используя тот же прием, что и в разделе 1.4.4[1]:

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

Достигнув точки остановки, такой как эта, обычно хорошо бы отправить код в удаленный репозиторий (Bitbucket, если вы следовали шагам в разделе 1.4.3):

```
$ git push
```

Я также рекомендую развернуть приложение на Heroku:

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

Здесь мы позаботились о том, чтобы запустить все тесты до развертывания, это хорошая привычка в разработке.

3.5.1. Что мы изучили в этой главе

- В третий раз мы прошли через полную процедуру создания нового Rails-приложения с нуля, установив необходимые гемы, отправив его в удаленный репозиторий и развернув в эксплуатационной среде.
- Сценарий `rails` создает новые контроллеры, получив команду `rails generate controller ControllerName <необязательные названия действий>`.
- Новые маршруты определяются в файле `config/routes.rb`.
- Rails-представления могут содержать статическую разметку HTML или встроенный код Ruby (ERb).

---

[1] Если появится сообщение об ошибке, говорящее, что файл с идентификатором процесса Spring (pid) будет перезаписан при слиянии, просто удалите файл, выполнив команду `rm -f *.pid` в командной строке.

- Механизм автоматизированного тестирования позволяет писать наборы тестов, которые придают процессу разработки новые возможности, позволяя уверенно реорганизовывать код и обнаруживать регрессии.
- В разработке через тестирование используется цикл «Красный, зеленый, рефакторинг».
- Макеты Rails позволяют определить общий шаблон для страниц в приложении и устранить повторяющийся код.

3.6. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Начиная с этого момента и до конца книги я рекомендую решать упражнения в отдельной ветке:

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

Это предотвратит конфликты с основной книгой.

Когда вы будете удовлетворены решениями, можете отправить ветку с упражнениями в удаленный репозиторий (если вы его установили):

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

(Последний шаг здесь выполняет переключение на основную ветку как этап подготовки к продолжению разработки учебного приложения, но при этом *не* происходит слияния изменений с основной ветвью, что предотвращает появление конфликтов с основной частью книги.) В будущих главах ветви и сообщения фиксаций, само собой, будут отличаться, но основная идея та же.

1. Вы могли заметить некоторые повторения в тестах контроллера StaticPages (листинг 3.22). В частности, базовый заголовок «Ruby on Rails Tutorial Sample App» одинаков во всех тестах заголовков. Используя специальную функцию `setup`, которая автоматически вызывается перед каждым тестом, убедитесь, что тест из листинга 3.38 все такой же **ЗЕЛЕНЫЙ**. (В нем используется *переменная экземпляра*, о которых вкратце рассказывалось в разделе 2.2.2 и более подробно будет рассказываться далее, в разделе 4.4.5, в сочетании с *интерполяцией строк*, о которой я расскажу в разделе 4.2.2.)
2. Создайте страницу Contact для учебного приложения[1]. Следуя образцу из листинга 3.13, сначала напишите тесты для проверки существования стра-

[1] Это упражнение решено в разделе 5.3.1.

ницы по адресу URL /static_pages/contact и проверки заголовка «Contact | Ruby on Rails Tutorial Sample App». Добейтесь успешного выполнения тестов, как при создании страницы About в разделе 3.3.3, включая заполнение страницы Contact содержимым из листинга 3.39. (Обратите внимание, что для сохранения независимости упражнений изменения в листинге 3.39 не включены в листинг 3.38.)

Листинг 3.38 ❖ Тесты контроллера StaticPages с базовым заголовком. **ЗЕЛЕНЫЙ** (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
```

Листинг 3.39 ❖ Код для предложенной страницы Contact (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. Продвинутые настройки тестирования

Этот дополнительный раздел описывает настройку тестов, используемую в серии видеоуроков Ruby on Rails Tutorial[1]. Он делится на три основные части: улучшен-

[1] http://screencasts.railstutorial.org/.

ные отчеты об успешном/неуспешном прохождении тестов (раздел 3.7.1), утилита для фильтрации трассировки стека (раздел 3.7.2) и автоматизированный запуск тестов, определяющий изменение файлов (раздел 3.7.3). В этом разделе демонстрируется не самый простой код, и он представлен только для удобства; не стоит ожидать, что вы с ходу поймете его.

Изменения в этом разделе должны осуществляться в основной ветви:

```
$ git checkout master
```

3.7.1. Средства составления отчетов minitest

Чтобы заставить Rails-тесты по умолчанию показывать **КРАСНЫЙ** и **ЗЕЛЕНЫЙ** цвета, я рекомендую добавить в файл вспомогательного инструмента тестирования[1] код из листинга 3.40, задействовав тем самым пакет `minitest-reporters` из листинга 3.2.

Листинг 3.40 ❖ Настройка тестов – **КРАСНЫЙ** и **ЗЕЛЕНЫЙ** цвета (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
  # Настроить все точки входа в test/fixtures/*.yml для всех тестов в алфавитном порядке.
  fixtures :all

  # Вспомогательные методы, используемые всеми тестами, следует добавлять сюда...
end
```

На рис. 3.8 видно, как теперь **КРАСНЫЙ** переходит в **ЗЕЛЕНЫЙ** в облачной IDE.

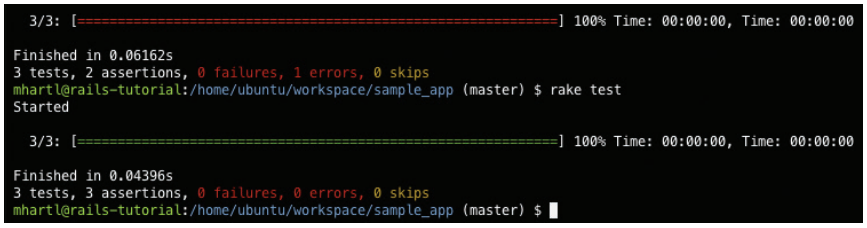


Рис. 3.8 ❖ **КРАСНЫЙ** переходит в **ЗЕЛЕНЫЙ** в облачной IDE

[1] В листинге 3.40 встречаются строки в одинарных и двойных кавычках. Это связано с тем, что `rails new` создает строки в одинарных кавычках, в то время как в документации к средствам составления отчетов minitest (https://github.com/kern/minitest-reporters) используются двойные кавычки. Такое смешение типов кавычек распространено в Ruby; больше информации об этом приводится в разделе 4.2.2.

3.7.2. Настраиваем backtrace

При появлении ошибки или неудачи во время тестирования выводится «трассировка стека» (список функций, вызванных приложением до момента сбоя), которая помогает проследить, как двигался тест через приложение. Обычно трассировка содержит информацию, очень полезную для диагностики проблем, но в некоторых системах (в том числе и в облачной IDE) она пересекает весь код приложения и гемов, в том числе самого фреймворка Rails. Полученная трассировка получается неудобно длинной, тем более что зачастую источник проблемы находится в приложении, а не в гемах.

Решение – отфильтровать трассировку и удалить ненужные строки. Для этого нужен гем `mini_backtrace`, включенный в листинг 3.2 вместе с *глушителем трассировки*. В облачной IDE большинство ненужных строк содержат элемент `rvm` (Ruby Version Manager – диспетчер версий Ruby), поэтому я рекомендую использовать глушитель, как показано в листинге 3.41, чтобы отфильтровать их.

Листинг 3.41 ❖ Добавление глушителя для RVM (config/initializers/backtrace_silencers.rb)

```
# Перезапустите сервер после изменения этого файла

# Вы можете добавить глушитель для библиотек, которыми пользуетесь, но не хотите видеть
# в трассировке
Rails.backtrace_cleaner.add_silencer { |line| line =~ /rvm/ }

# Также можно удалить все глушители, если проблема происходит из кода фреймворка
# Rails.backtrace_cleaner.remove_silencers!
```

Как отмечается в комментариях в листинге 3.41, после добавления глушителя нужно перезагрузить локальный веб-сервер.

3.7.3. Автоматизация тестирования с помощью Guard

Одно из неудобств команды `rake test` связано с необходимостью переключаться в командную строку и запускать тесты вручную. Избавиться от этого неудобства можно с помощью Guard[1], инструмента автоматизации запуска тестов. Guard следит за изменениями в файловой системе. Например, если изменить файл `static_pages_controller_test.rb`, он запустит только этот тест. Более того, можно настроить Guard так, что при изменении файла `home.html.erb` (например) автоматически запустится тест `static_pages_controller_test.rb`.

Файл `Gemfile` в листинге 3.2 уже включает гем `guard` в приложение, поэтому нам остается только инициализировать его:

```
$ 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
```

[1] https://github.com/guard/guard.

Теперь отредактируем полученный файл `Guardfile`, чтобы Guard запускал тесты после обновления интеграционных тестов и представлений (листинг 3.42). (Учитывая его длину и сложность, я рекомендую просто скопировать его содержимое.)

Листинг 3.42 ❖ Измененный файл Guardfile

```
# Определяет правила сопоставления для 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

# Возвращает интеграционные тесты, соответствующие данному ресурсу.
def integration_tests(resource = :all)
  if resource == :all
    Dir["test/integration/*"]
  else
    Dir["test/integration/#{resource}_*.rb"]
```

```
  end
end

# Возвращает тесты контроллера, соответствующие данному ресурсу.
def controller_test(resource)
  "test/controllers/#{resource}_controller_test.rb"
end

# Возвращает все тесты, соответствующие данному ресурсу.
def resource_tests(resource)
  integration_tests(resource) << controller_test(resource)
end
```

Строка

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

заставляет Guard использовать сервер Spring, входящий в состав Rails, для ускорения загрузки, а также не дает Guard выполнить весь набор тестов на запуске.

Чтобы предотвратить конфликт между Spring и Git при использовании Guard, добавьте каталог `spring/` в файл `.gitignore`, именно его использует Git для определения списка игнорируемых файлов или каталогов. В облачной IDE это можно сделать так:

1) щелкните на ярлыке с изображением шестеренки в панели инструментов навигатора по файловой системе (рис. 3.9);

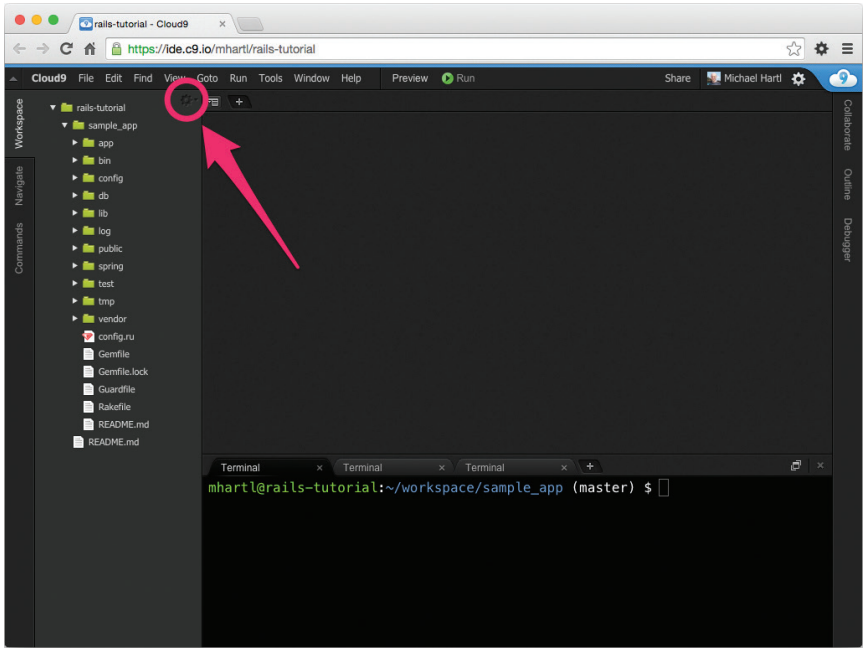


Рис. 3.9 ❖ Малозаметный ярлык настройки на панели навигатора

2) выберите пункт **Show hidden files** (Показать скрытые файлы), чтобы в корневом каталоге приложения появился файл `.gitignore` (рис. 3.10);

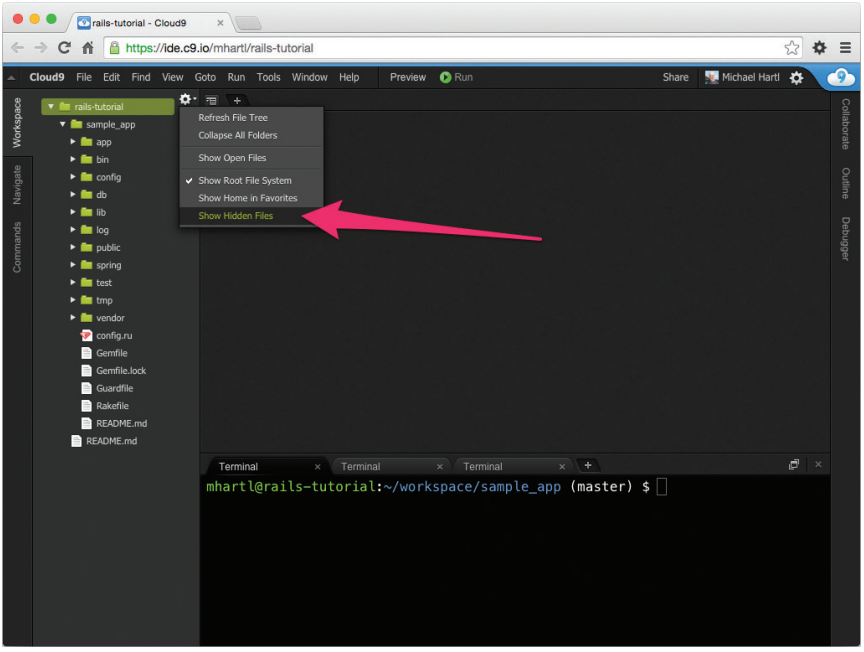


Рис. 3.10 ❖ Показать скрытые файлы в навигаторе

3) откройте файл `.gitignore` двойным щелчком на нем (рис. 3.11) и заполните его содержимым из листинга 3.43.

Листинг 3.43 ❖ Добавление Spring в файл .gitignore

```
# Полную информацию об игнорировании файлов ищите по адресу:
# https://help.github.com/articles/ignoring-files
# Если понадобится игнорировать файлы, созданные текстовым редактором или операционной
# системой, возможно, лучше добавить их в глобальный список игнорирования
#   git config --global core.excludesfile '~/.gitignore_global'

# Игнорировать конфигурацию компоновщика
/.bundle

# Игнорировать базу данных SQLite по умолчанию
/db/*.sqlite3
/db/*.sqlite3-journal

# Игнорировать все журналы и временные файлы
/log/*.log
/tmp

# Игнорировать файлы Spring
/spring/*.pid
```

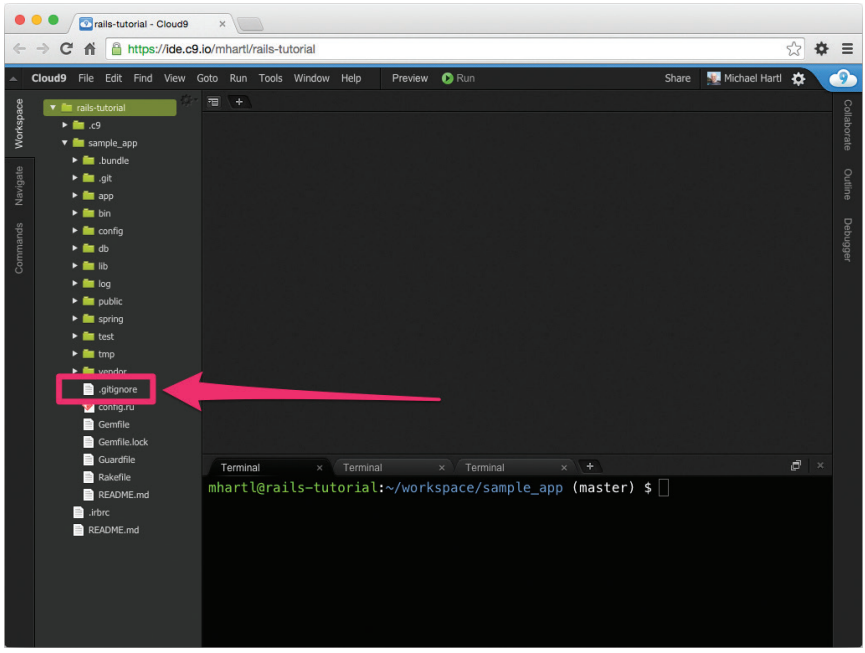


Рис. 3.11 ❖ Обычно скрытый файл .gitignore стал видимым

Сервер Spring временами ведет себя довольно странно (по крайней мере, на момент написания этой книги), и иногда Spring-процессы будут накапливаться и снижать производительность тестов. Если тесты кажутся необычно медленными – это повод проверить системные процессы и удалить ненужные (блок 3.4).

Блок 3.4 ❖ Unix-процессы

В Unix-подобных системах, таких как Linux и OS X, каждая пользовательская и системная задача находится в четко определенном контейнере, который называется *процессом*. Чтобы увидеть все процессы в системе, можно воспользоваться командой `ps` с параметром `aux`:

```
$ ps aux
```

Чтобы отфильтровать процессы по типам, можно пропустить результаты работы команды `ps` через инструмент поиска по шаблону `grep`, использовав оператор конвейера `|`:

```
$ ps aux | grep spring
ubuntu 12241 0.3 0.5 589960 178416 ? Ssl Sep20 1:46
spring app | sample_app | started 7 hours ago
```

В этом выводе можно заметить множество деталей, самой важной из которых является *идентификатор (id) процесса*, или `pid`. Остановить ненужный процесс мож-

но командой `kill`, передав с ее помощью сигнал завершения (с кодом 9[1]) процессу по его идентификатору `pid`:

```
$ kill -9 12241
```

Я рекомендую этот прием для остановки отдельных процессов, таких как Rails-сервер (идентификатор `pid`, которого можно получить командой `ps aux | grep server`), но иногда бывает удобнее останавливать сразу все процессы, соответствующие определенному имени программы, например все процессы `spring`, захламляющие систему. В этом случае стоит сначала попробовать остановить их с помощью самой команды `spring`:

```
$ spring stop
```

Иногда этот прием не срабатывает, тогда можно остановить все процессы с именем `spring`, вызвав команду `pkill`:

```
$ pkill -9 -f spring
```

Каждый раз, когда процесс ведет себя не так, как ожидается, или кажется зависшим, запустите сначала `ps aux`, чтобы увидеть, что происходит, и только потом `kill -9<pid>` или `pkill -9 -f <имя>`.

После настройки Guard нужно открыть новый терминал и (так же как с Rails-сервером в разделе 1.3.2) запустить его в командной строке:

```
$ bundle exec guard
```

Правила в листинге 3.42 оптимизированы для этой книги, они автоматически запускают (например) интеграционные тесты при изменении контроллера. Чтобы запустить *все тесты*, нажмите **ENTER** в строке приглашения `guard>`. (Иногда может появляться ошибка соединения с сервером Spring. Чтобы решить проблему, просто снова нажмите **ENTER**.)

Чтобы выйти из Guard, нажмите **Ctrl-D**. Чтобы добавить дополнительные правила в Guard, обратитесь к примерам в листинге 3.42, прочитайте файл Guard README[2] и вики-страницу Guard[3].

[1] https://ru.wikipedia.org/wiki/Сигналы_(UNIX).

[2] https://github.com/guard/guard.

[3] https://github.com/guard/guard/wiki.

Глава 4

Ruby со вкусом Rails

Основанная на примерах из главы 3, эта глава рассматривает некоторые элементы Ruby, важные для Rails. Ruby многогранен, но, к счастью, для разработки веб-приложений на Rails нам достаточно относительно небольшой его части. Кроме того, она немного отличается от того, что обычно дают во введении в Ruby. Цель этой главы – заложить основательный фундамент знаний о Ruby со вкусом Rails, независимо от того, был ли у вас предыдущий опыт программирования на этом языке. Здесь рассматривается очень много нового, и я не жду, что вы поймете все с первого раза. Я буду часто ссылаться на эту главу в последующем.

4.1. Мотивация

Как мы видели в предыдущей главе, можно создать скелет приложения Rails и даже начать его тестирование, практически не зная основ Ruby. Мы сделали это, опираясь на включенный в книгу код тестов, разбирая каждое сообщение об ошибке, пока не добились успешного выполнения всего набора тестов. Однако такая ситуация не может длиться вечно, поэтому мы откроем эту главу парой дополнений к сайту, которые поставят нас лицом к лицу с нашим ограниченным знанием Ruby.

Последнее, что мы сделали в нашем новом приложении, – обновили практически статические страницы, использовав Rails-макет, чтобы избавиться от повторяющегося кода в представлениях, как показано в листинге 4.1 (он повторяет листинг 3.32).

Листинг 4.1 ❖ Макет сайта учебного приложения (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>
```

Давайте сосредоточимся на одной строке в этом листинге:

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track' => true %>
```

Здесь используется встроенная Rails-функция `stylesheet_link_tag` (описание которой можно найти в справочнике поRails API)[1], подключающая `application.css` для всех типов носителей информации[2] (включая мониторы компьютеров и принтеры). Для опытного Rails-разработчика эта строка выглядит просто, но в ней заключено не менее четырех идей, которые могут сбить с толку: встроенные Rails-методы, вызов метода без скобок, символы и хэши. Мы раскроем все эти идеи в этой главе.

Помимо того что Rails поставляется с огромным количеством встроенных функций для использования в представлениях, есть возможность создавать новые. Такие функции называются *вспомогательными*; чтобы узнать, как создать свою вспомогательную функцию, рассмотрим для начала строку с заголовком из листинга 4.1:

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

Она опирается на определение заголовка страницы (через `provide`) в каждом представлении:

```
<% 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>
```

Но что случится, если мы не предоставим его? Отличное решение – определить *базовый заголовок*, используемый в каждой странице, с дополнительным переменным заголовком для конкретной страницы. Мы уже *почти* достигли этого в текущей схеме, с одним маленьким недостатком: если удалить вызов `provide` в одном из представлений, полный заголовок будет выглядеть так:

```
| Ruby on Rails Tutorial Sample App
```

Другими словами, у нас есть подходящий базовый заголовок, но в его начале стоит вертикальная черта `|`.

---

[1] http://v32.rusrails.ru/layouts-and-rendering-in-rails/structuring-layouts.

[2] https://webref.ru/css/media.

Для решения проблемы отсутствующего заголовка страницы определим собственную вспомогательную функцию с именем full_title. Она будет возвращать базовый заголовок «Ruby on Rails Tutorial Sample App», если заголовок страницы не определен, и добавлять вертикальную черту перед заголовком страницы в противном случае (листинг 4.2)[1].

Листинг 4.2 ❖ Определение вспомогательного метода full_title (app/helpers/application_helper.rb)

```
module ApplicationHelper

  # Возвращает полный заголовок на основе заголовка страницы.
  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
```

Теперь, определив вспомогательную функцию, можно упростить шаблон, заменив

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

на

```
<title><%= full_title(yield(:title)) %></title>
```

как показано в листинге 4.3.

Листинг 4.3 ❖ Шаблон сайта со вспомогательной функцией full_title **ЗЕЛЕНЫЙ** (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>
```

[1] Если вспомогательная функция относится к определенному контроллеру, ее следует поместить в соответствующий файл; например, вспомогательные функции для контроллера StaticPages обычно должны находиться в app/helpers/static_pages_helper.rb. В нашем случае функция full_title будет использоваться всеми страницами сайта, а для таких функций в Rails имеется специальный файл: app/helpers/application_helper.rb.

```
    <%= yield %>
  </body>
</html>
```

Чтобы задействовать новую вспомогательную, можно удалить ненужное слово «Home» из страницы Home, позволив ей вернуться к базовому заголовку. Но сначала обновим тест заголовка – проверим отсутствие строки "Home" (листинг 4.4).

Листинг 4.4 ❖ Обновленные тесты для заголовка страницы Home **КРАСНЫЙ** (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
```

Запустим набор тестов, чтобы убедиться, что один тест завершается неудачей.

Листинг 4.5 ❖ **КРАСНЫЙ**

```
$ bundle exec rake test
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
```

Чтобы обеспечить успешное выполнение теста, удалим строку provide из представления страницы Home, как показано в листинге 4.6.

Листинг 4.6 ❖ Страница Home без собственного заголовка страницы **ЗЕЛЕНЫЙ** (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>
```

Теперь все тесты должны выполняться успешно:

Листинг 4.7 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

> **Примечание.** *Предыдущие примеры включали часть вывода команды* `rake test`*, в том числе количество успешных и неудачных тестов, но для краткости, начиная с этого момента, я буду опускать эту информацию.*

Так же как строка подключения стилей к приложению, код в листинге 4.2 может выглядеть просто для опытного Rails-разработчика, но он *полон* важных Ruby-идей: модули, определения методов, необязательные аргументы методов, комментарии, присваивание локальным переменным, логические значения, управление потоком выполнения, интерполяция строк и возвращение значений. Все эти идеи тоже будут раскрыты в этой главе.

4.2. Строки и методы

Нашим основным инструментом изучения Ruby будет *консоль Rails*, утилита командной строки для работы с Rails-приложениями, впервые представленная в разделе 2.3.3. Сама консоль опирается на интерактивную оболочку Ruby (`irb`), и потому ей доступна вся мощь языка Ruby. (Как будет показано в разделе 4.4.4, она также имеет доступ и к Rails-окружению.)

Тем, кто пользуется облачной IDE, я рекомендую включить пару параметров конфигурации `irb`. С помощью простого текстового редактора `nano` создайте в корневом каталоге файл `.irbrc` и заполните его содержимым из листинга 4.8:

```
$ nano ~/.irbrc
```

Настройки в листинге 4.8 устанавливают упрощенный режим приглашения к вводу и отключают раздражающее автоматическое оформление отступов.

Листинг 4.8 ❖ Добавление конфигурации irb (~/.irbrc)

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

Независимо от того, последуете вы моему совету или нет, вы сможете запустить консоль в командной строке, как показано ниже:

```
$ rails console
Loading development environment
>>
```

По умолчанию консоль запускается в *окружении разработки*, одном из трех отдельных окружений, имеющихся в Rails (также есть *тестовое* и *эксплуатационное*). Это различие не будет иметь большого значения в данной главе, но мы познакомимся поближе с окружениями в разделе 7.1.1.

Консоль – это замечательный инструмент обучения. Не бойтесь экспериментировать в ней – вы (почти наверняка) ничего не сломаете. Используя консоль, нажимайте клавиши **Ctrl-C**, если застряли, или **Ctrl-D**, чтобы вообще выйти из нее. В оставшейся части главы порой будет полезно консультироваться со справочником по Ruby API[1]. Он содержит много (пожалуй, даже *слишком много*) информации; например, чтобы узнать больше о строках Ruby, загляните в раздел с описанием класса `String`.

4.2.1. Комментарии

Комментарии в Ruby начинаются со знака решетки # (знак хэша, или (более поэтично) «октоторп») и простираются до конца строки. Ruby игнорирует комментарии, но они полезны для читателей (а зачастую и для самого автора!). В следующем фрагменте

```
# Возвращает полный заголовок на основе заголовка страницы.
def full_title(page_title = '')
  .
  .
  .
end
```

первая строка является комментарием, описывающим цель функции, что определяется ниже.

Обычно в консольных сеансах нет необходимости в комментариях, но в учебных целях я в дальнейшем буду иногда их использовать, например:

```
$ rails console
>> 17 + 42   # Сложение целых чисел
=> 59
```

Если в процессе чтения вы будете опробовать примеры у себя, вы можете опускать комментарии, если хотите; консоль в любом случае их игнорирует.

4.2.2. Строки

Строки – это, пожалуй, самая важная структура данных для веб-приложений, так как веб-страницы, посылаемые браузерам, в конечном счете состоят из строк символов. Начнем исследование строк в консоли:

```
$ rails console
>> ""         # Пустая строка
=> ""
>> "foo"      # Непустая строка
=> "foo"
```

[1] http://ruby-doc.org/.

Эти *строковые литералы* созданы с использованием символа двойной кавычки ". Консоль выводит результат интерпретации каждой строки, которым для строкового литерала является сама строка.

Строки можно объединять оператором +:

```
>> "foo" + "bar" # Конкатенация строк
=> "foobar"
```

Здесь результатом выражения "foo" плюс "bar" является строка "foobar"[1].

Другой способ создания строк – посредством *интерполяции* с применением специального синтаксиса #{}[2]:

```
>> first_name = "Michael" # Присваивание переменной
=> "Michael"
>> "#{first_name} Hartl"  # Интерполяция
=> "Michael Hartl"
```

Здесь мы *присвоили* значение "Michael" переменной first_name, а затем интерполировали ее в строку"#{first_name} Hartl". Точно так же можно присвоить переменным значения обеих строк:

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name # Конкатенация с пробелом между строками
=> "MichaelHartl"
>> "#{first_name} #{last_name}" # Эквивалентная интерполяция
=> "Michael Hartl"
```

Отметим, что последние два выражения являются равнозначными, но я предпочитаю версию с интерполяцией; добавление одного пробела " " кажется мне немного неуклюжим.

Вывод

Чтобы *вывести* строку, обычно используется Ruby-функция puts (произносится как «пут-эс»):

```
>> puts "foo" # вывести строку
foo
=> nil
```

Метод puts имеет *побочный эффект*: выражение puts "foo"выведет строку на экран и затем вернет буквально «ничего»: nil– это специальное значение в Ruby, обозначающее «вообще ничего». (В дальнейшем, для простоты, я буду иногда опускать часть => nil.)

---

[1] О происхождении строк «foo» и «bar» можно узнать из статьи: http://lurklurk.com/Foobar.

[2] Программисты, знакомые с Perl или PHP, могут сравнить эту конструкцию с автоматической интерполяцией переменных со знаком доллара в выражениях вроде "foo $bar".

Как видно из примеров выше, puts автоматически добавляет в вывод символ перевода строки \n. А родственный ему метод print – нет:

```
>> print "foo"   # выведет строку (как puts, но без символа перевода строки)
foo=> nil
>> print "foo\n" # Даст тот же результат, что и puts "foo"
foo
=> nil
```

Строки в одиночных кавычках

Во всех примерах, представленных до сих пор, использовались *строки в двойных кавычках*, но Ruby также поддерживает строки *в одиночных кавычках*. Во многих отношениях оба типа строк практически идентичны:

```
>> 'foo'    # Строка в одинарных кавычках
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

Но есть важное отличие – Ruby не интерполирует строки в одиночных кавычках:

```
>> '#{foo} bar' # Строки в одиночных кавычках не интерполируются
=> "\#{foo} bar"
```

Обратите внимание, что консоль возвращает значения в виде строк в двойных кавычках, что требует использовать обратный слеш для *экранирования* специальных символов, таких как #.

Строки в двойных кавычках можно использовать везде, где можно применять строки в одиночных кавычках, плюс они поддерживают интерполяцию. Но тогда зачем нужны строки в одиночных кавычках? Они часто бывают полезными, потому что являются истинными литералами и хранят в точности те символы, что вы вводите. Например, символ обратного слеша – это специальный символ в большинстве систем, как в литерале символа перевода строки \n. Если нужно, чтобы переменная содержала обратный слеш как таковой, в одиночных кавычках это сделать проще:

```
>> '\n' # Комбинация двух символов: обратного слеша и n
=> "\\n"
```

Так же как в случае с символом # в предыдущем примере, обратный слеш в языке Ruby необходимо экранировать еще одним, дополнительным обратным слешем; внутри строк в двойных кавычках сам обратный слеш представлен *двумя* обратными слешами. Для такого небольшого примера в этом немного экономии, но при большом количестве элементов, нуждающихся в экранировании, это может серьезно помочь:

```
>> 'Новые строки (\n) и табуляция (\t) используют обратный слеш \.'
=> "Новые строки (\\n) и табуляция (\\t) используют обратный слеш \\."
```

Наконец, следует отметить, что одинарные и двойные кавычки, в сущности, взаимозаменяемы, и вы часто будете видеть, как исходный код переключается между ними без какой-либо видимой системы. Ничего не остается, как сказать: «Добро пожаловать в Ruby!»

4.2.3. Объекты и передача сообщений

Все в Ruby, включая строки и даже `nil`, – это *объекты*. Мы увидим технический смысл этого выражения в разделе 4.4.2, но я не думаю, что кто-нибудь когда-нибудь понял суть объектов, прочитав определение в книге; вы должны создать свое интуитивное понимание объектов, увидев множество примеров.

Проще описать, что объекты *делают*, на какие сообщения реагируют. Объект-строка, например, может реагировать на сообщение `length`, возвращая число символов в строке:

```
>> "foobar".length # Передача сообщения "length" строке
=> 6
```

Как правило, сообщения, посылаемые объектам, – это *методы*, которые являются функциями, определенными для этих объектов[1]. Строки также откликаются на метод `empty?`:

```
>> "foobar".empty?
=> false
>> "".empty?
=> true
```

Обратите внимание на знак вопроса в конце метода `empty?`. Это стандарт Ruby, обозначающий тип *boolean* возвращаемого значения: `true` (истина) или `false` (ложь). Логические значения особенно полезны для управления потоком:

```
>> s = "foobar"
>> if s.empty?
>>   "The string is empty"
>> else
>>   "The string is nonempty"
>> end
=> "The string is nonempty"
```

Чтобы указать более одного условия, можно использовать `elsif`(`else`+ `if`):

```
>> if s.nil?
>>   "The variable is nil"
>> elsif s.empty?
>>   "The string is empty"
>> elsif s.include?("foo")
```

[1] Заранее извиняюсь за случайные переключения между терминами *функция* и *метод* на протяжении этой главы; в Ruby это одно и то же: все методы являются функциями, а все функции – методами, поскольку все программные элементы являются объектами.

```
>>   "The string includes 'foo'"
>> end
=> "The string includes 'foo'"
```

Логические значения можно объединять с помощью операторов `&&` («И»), `||` («ИЛИ») и ! («НЕ»):

```
>> 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
```

Поскольку все в Ruby является объектом, значение `nil` тоже является объектом, поэтому имеет свой комплект методов. Одним из них является метод `to_s`, преобразующий практически любой объект в строку:

```
>> nil.to_s
=> ""
```

Это значение выглядит как пустая строка, в чем можно убедиться, составив *цепочку* сообщений, передаваемых объекту `nil`:

```
>> nil.empty?
NoMethodError: undefined method `empty?' for nil:NilClass
>> nil.to_s.empty?# Цепочка сообщений
=> true
```

Здесь мы видим, что объект `nil` сам по себе не реагирует на метод `empty?`, но `nil.to_s` реагирует.

Существует специальный метод для проверки значения на равенство `nil`:

```
>> "foo".nil?
=> false
>> "".nil?
=> false
>> nil.nil?
=> true
```

Код

```
puts "x is not empty" if !x.empty?
```

демонстрирует альтернативное использование ключевого слова `if`; Ruby позволяет писать инструкции, которые выполняются, только когда инструкция, следую-

щая за if, возвращает истинное значение. Имеется ключевое слово unless, имеющее противоположный смысл, которое работает по той же схеме:

```
>> string = "foobar"
>> puts "The string '#{string}' is nonempty." unless string.empty?
The string 'foobar' is nonempty.
=> nil
```

Уникальность объекта nil в том, что это *единственный* Ruby-объект, который возвращает ложное значение в логическом контексте, кроме самого значения false. В этом можно убедиться с помощью оператора !! (читается как «бэнг-бэнг»), который дважды отрицает объект, приводя его к логическому значению:

```
>> !!nil
=> false
```

Для всех других объектов возвращается *true*, даже для числа 0:

```
>> !!0
=> true
```

4.2.4. Определение методов

Консоль позволяет определять методы, подобные методу действия home в листинге 3.6 или вспомогательной функции full_title в листинге 4.2. (Определение методов в консоли – не самое простое мероприятие, и для этой цели обычно используются файлы, но это удобно для демонстрационных целей.) Например, определим функцию string_message, которая принимает один *аргумент* и возвращает разные сообщения для пустого и непустого аргументов:

```
>> 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!
```

Как видно в последнем примере, аргумент можно вообще опустить (а вместе с ним и скобки). Это возможно благодаря объявлению

```
def string_message(str = '')
```

содержащему аргумент со значением по умолчанию, в данном случае с пустой строкой. Это делает аргумент `str` необязательным, и если опустить его, он автоматически получит указанное значение по умолчанию.

Обратите внимание, что Ruby-функции могут *возвращать результат неявно* – значение последней выполненной инструкции, в данном случае одну из двух строк, в зависимости от значения аргумента `str`. В Ruby также есть явный оператор возврата значения из функции; следующая функция эквивалентна приведенной выше:

```
>> def string_message(str = '')
>>   return "It's an empty string!" if str.empty?
>>   return "The string is nonempty."
>> end
```

(Внимательный читатель может заметить, что второй оператор `return` фактически не нужен, – как последнее выражение в функции, строка `"The string is nonempty."` будет возвращена независимо от его наличия, но использование `return` в обоих местах придает коду приятную симметрию.)

Также важно понимать, что имя аргумента функции не имеет значения. Другими словами, в примере выше имя `str` можно заменить любым другим допустимым именем переменной, таким как `the_function_argument`, и он будет работать совершенно так же:

```
>> 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. Еще раз о вспомогательной функции заголовка

Теперь мы в состоянии понять вспомогательную функцию `full_title` из листинга 4.2[1], определение которой повторно приводится в листинге 4.9, но на этот раз с комментариями и примечаниями.

---

[1] И все же есть еще кое-что, пока непонятное для нас, – как Rails связывает все это вместе: как отображает URL в методы, как обеспечивает доступность вспомогательной функции `full_title` в представлениях и т. д. Это интересная тема, и я рекомендую исследовать ее далее, но точное знание, как работает Rails, не является необходимым условием для работы с Rails. Для более глубокого понимания я рекомендую прочитать книгу «The Rails 4 Way» Оби Фернандеса (Obie Fernandez).

Листинг 4.9 ❖ title_helper с примечаниями (app/helpers/application_helper.rb)

```
module ApplicationHelper

  # Возвращает полный заголовок для страницы.          # Документирующий комментарий
  deffull_title(page_title = '')                       # Опpеределение метода,
                                                       # необязательный аргумент
    base_title = "Ruby on Rails Tutorial Sample App" # Присваивание переменной
    if page_title.empty?                               # Логическая проверка
      base_title                                       # Неявно возвращаемое значение
    else
      page_title + " | " + base_title                  # Соединение строк
    end
  end
end
```

Все эти элементы – определение функции (с необязательным аргументом), присваивание значения переменной, логическая проверка, управление потоком выполнения и конкатенация строк[1] – собраны вместе, чтобы сделать компактный вспомогательный метод для использования в шаблоне нашего сайта. Последний элемент – это `module ApplicationHelper`: модули позволяют собрать вместе родственные методы и включить их в Ruby-классы с помощью `include`. Если вы пишете на обычном Ruby, значит, вы пишете модули и сами явно включаете их, но в Rails вспомогательные модули подключаются автоматически. В результате метод `full_title` автоматически, как по волшебству, оказывается доступен во всех представлениях.

4.3. Другие структуры данных

Любое веб-приложение в конечном счете – это строки. Для *изготовления* этих строк необходимо также использовать другие структуры данных. В этом разделе мы познакомимся с некоторыми структурами данных Ruby, играющими важную роль в создании Rails-приложений.

4.3.1. Массивы и диапазоны

Массив – это лишь список элементов, следующих в определенном порядке. Мы еще не обсуждали массивов, но знание их особенностей дает хорошую основу для понимания хэшей (раздел 4.3.3) и аспектов моделирования данных в Rails (таких как ассоциации `has_many`, представленные в разделе 2.3.3 и более широко раскрытые в разделе 11.1.3).

---

[1] Очень заманчиво было здесь использовать интерполяцию – в действительности этот способ был применен во всех предыдущих версиях книги, – но дело в том, что вызов `provide` превращает строку в так называемый объект `SafeBuffer`. Интерполяция и вставка в шаблон представления в таком случае приводят к чрезмерному экранированию (over-escape) любого текста, вставляемого в разметку HTML, поэтому такой заголовок, как «Help's on the way», будет преобразован в «Help's on the way». (Спасибо читателю Джереми Флейшману (Jeremy Fleischman) за указание на эту тонкость.)

Мы потратили много времени на изучение строк, которые поддерживают естественный способ перехода к массивам – метод split:

```
>> "foo bar      baz".split       # Разобьет строку на трехэлементный массив.
=> ["foo", "bar", "baz"]
```

Результатом этой операции является массив из трех строк. По умолчанию split делит строку по пробелам, но есть возможность использовать любые другие разделители:

```
>> "fooxbarxbazx".split('x')
=> ["foo", "bar", "baz"]
```

Как это принято в большинстве языков программирования, нумерация элементов в Ruby-массивах *начинается с нуля*, то есть первый элемент массива имеет индекс 0, второй – индекс 1 и т. д.:

```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]            # Для доступа к элементам массива в Ruby используются квадратные скобки.
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]           # Индексы могут быть отрицательными!
=> 17
```

Мы видим здесь, что для доступа к элементам массива в Ruby используются квадратные скобки. В дополнение к этой записи Ruby предлагает синонимы для некоторых часто используемых элементов[1]:

```
>> a               # Просто чтобы напомнить, что содержится в 'a'
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
=> 17
>> a.last == a[-1] # Сравнение с помощью==
=> true
```

В последней строке используется оператор равенства ==, который поддерживается в Ruby, так же как во многих других языках, наряду с родственным оператором != («не равно»), и т. д.:

---

[1] Метод second, используемый здесь, пока не является частью языка Ruby непосредственно, он добавляется фреймворком Rails. В этом примере он доступен, потому что консоль автоматически включает Rails-расширения в Ruby.

```
>> x = a.length      # Массивы, как и строки, имеют метод 'length'.
=> 3
>> x == 3
=> true
>> x == 1
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
```

В дополнение к length (первая строка в примере выше) массивы имеют множество других методов:

```
>> 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]
```

Обратите внимание, что ни один из перечисленных методов не меняет самого массива a. Чтобы *изменить* массив, следует использовать соответствующие «бэнг»-методы (такое название обусловлено наличием восклицательного знака в их именах, который в английском языке произносится как «бэнг»):

```
>> a
=> [42, 8, 17]
>> a.sort!
=> [8, 17, 42]
>> a
=> [8, 17, 42]
```

Добавлять новые данные в массивы можно с помощью метода push или эквивалентного ему оператора <<:

```
>> a.push(6)                    # Добавит 6 в массив
=> [42, 8, 17, 6]
>> a << 7                       # Добавит 7 в массив
=> [42, 8, 17, 6, 7]
```

```
>> a << "foo" << "bar"            # Добавит сразу два новых элемента
=> [42, 8, 17, 6, 7, "foo", "bar"]
```

Последний пример показывает возможность объединения нескольких операций добавления в цепочки. Кроме того, в отличие от массивов во многих других языках, Ruby-массивы могут содержать данные разных типов (в данном случае целые числа и строки).

Выше мы видели, что split преобразует строку в массив. Обратная операция выполняется с помощью метода join:

```
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join                         # Объединить без разделителя.
=> "428177foobar"
>> a.join(', ')                   # Объединить через запятую и пробел.
=> "42, 8, 17, 7, foo, bar"
```

С массивами тесно связаны *диапазоны*, суть которых становится более понятной, если преобразовать такой диапазон в массив с помощью метода to_a:

```
>> 0..9
=> 0..9
>> 0..9.to_a        # Ошибка, метод to_a вызывается для 9.
NoMethodError: undefined method `to_a' for 9:Fixnum
>> (0..9).to_a      # Чтобы вызвать метод to_a для диапазона, его нужно заключить в скобки.
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Даже при том, что конструкция 0..9 является допустимым диапазоном, второе выражение показывает, что для вызова методов диапазонов нужно добавлять скобки.

Диапазоны полезны для извлечения элементов массива:

```
>> a = %w[foobarbazquux]              # %w создает массив строк.
=> ["foo", "bar", "baz", "quux"]
>> a[0..2]
=> ["foo", "bar", "baz"]
```

Особенно полезным трюком является использование индекса –1 в конце диапазона для выборки каждого элемента от заданной точки до конца массива *без* необходимости явно использовать длину массива:

```
>> a = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..(a.length-1)]                 # Явное использование длины массива.
=> [2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..-1]                           # Трюк с индексом -1.
=> [2, 3, 4, 5, 6, 7, 8, 9]
```

Поддерживаются также диапазоны символов:

```
>> ('a'..'e').to_a
=> ["a", "b", "c", "d", "e"]
```

4.3.2. Блоки

И массивы, и диапазоны имеют множество методов, принимающих *блоки*, которые являются одними из самых мощных и в то же время самых непонятных элементов Ruby:

```
>> (1..5).each { |i| puts 2 * i }
2
4
6
8
10
=> 1..5
```

Здесь вызывается метод `each` диапазона `(1..5)`, которому передается блок `{ |i| puts 2 * i }`. Вертикальными линиями вокруг имени переменной `|i|` в Ruby обозначаются блоковые переменные, посредством которых метод оперирует блоком. В данном случае метод `each` диапазона обрабатывает блок с одной локальной переменной, с именем `i`, и просто выполняет блок для каждого значения в диапазоне.

Фигурные скобки – это один из способов обозначить блок, но есть и другой способ:

```
>> (1..5).each do |i|
?>   puts 2 * i
>> end
2
4
6
8
10
=> 1..5
```

Блоки могут состоять более чем из одной строки, и часто именно так и бывает. В этой книге мы будем следовать общепринятому соглашению об использовании фигурных скобок только для коротких, однострочных блоков, и использовать синтаксис `do..end` для длинных однострочных и многострочных блоков:

```
>> (1..5).each do |number|
?>   puts 2 * number
>>   puts '--'
>> end
2
--
4
--
6
--
8
--
10
```

```
--
=> 1..5
```

Здесь я использовал `number` вместо `i`, просто чтобы подчеркнуть, что имя переменной может быть любым.

Без владения основами программирования нет короткого пути к пониманию блоков; с ними нужно много работать, и в конечном итоге вы привыкнете к ним[1]. К счастью, люди способны к обобщениям на основе конкретных примеров; вот еще несколько блоков, в том числе пара с использованием метода `map`:

```
>> 3.times { puts "Betelgeuse!" } # 3.times принимает блок без переменной
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
=> 3
>> (1..5).map { |i| i**2 }        # Оператор ** вычисляет "степень"
=> [1, 4, 9, 16, 25]
>> %w[abc]                        # %w создает массивы строк
=> ["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"]
```

Как видите, метод `map` возвращает результат применения указанного блока к каждому элементу в массиве или диапазоне. В последних двух примерах блок внутри `map` включает вызов конкретного метода переменной блока. В таких случаях обычно используется сокращенный вариант:

```
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
>> %w[A B C].map(&:downcase)
=> ["a", "b", "c"]
```

(В этом странном, но компактном коде используется *символ* (symbol), о котором рассказывается в разделе 4.3.3.) Самое интересное в этой конструкции, что изначально она была добавлена в Ruby on Rails, но так всем понравилась, что теперь включена в ядро Ruby.

И в качестве последнего примера блоков рассмотрим тест из файла, представленный в листинге 4.4:

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

---

[1] С другой стороны, искушенные программисты могут извлечь пользу, зная, что блоки – это *замыкания*, объединяющие анонимные функции с внешними данными.

Сейчас не важно понимание деталей (на самом деле я не смогу их объяснить без подготовки), но из наличия ключевого слова do можно сделать вывод, что тело теста является блоком. Метод test принимает строковый аргумент (описание) и блок, затем выполняет тело блока как часть набора тестов.

Кстати, теперь вы в состоянии понять строку Ruby, которую я бросил в разделе 1.5.4 для определения случайных субдоменов:

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

Давайте разберем ее шаг за шагом:

```
>> ('a'..'z').to_a                    # Массив алфавитных символов
=> ["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            # Перемешать.
=> ["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]      # Выбрать первые восемь элементов.
=> ["f", "w", "i", "a", "h", "p", "c", "x"]
>> ('a'..'z').to_a.shuffle[0..7].join  # Объединить в одну строку.
=> "mznpybuj"
```

4.3.3. Хэши и символы

Хэши – это массивы, не ограниченные целочисленными индексами. (По этой причине в некоторых языках, особенно в Perl, хэши иногда называют *ассоциативными массивами*.) Вместо целочисленных индексов в хэшах используются хэш-индексы, или *ключи*, которые могут быть практически любыми объектами. Например, в качестве ключей можно использовать строки:

```
>> user = {}                          # {} - пустой хэш.
=> {}
>> user["first_name"] = "Michael"     # Ключ "first_name", значение "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"        # Ключ "last_name", значение "Hartl"
=> "Hartl"
>> user["first_name"]                 # Доступ к элементам как в массиве.
=> "Michael"
>> user                               # Литеральное представление хэша
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Хэши обозначаются фигурными скобками с парами ключ/значение; фигурные скобки без пар ключ/значение, то есть {}, это пустой хэш. Важно отметить, что фигурные скобки хэшей не имеют ничего общего с фигурными скобками блоков. (Да, это может привести к путанице.) Хотя хэши напоминают массивы, важно помнить, что они не гарантируют следования элементов в каком-то определенном порядке[1]. Если порядок важен, используйте массив.

[1] Версии Ruby 1.9 и выше в действительности гарантируют хранение элементов хэшей в том же порядке, в каком они вводились, но не стоит на это рассчитывать.

Для определения хэшей часто используется их литеральное представление – с ключами и значениями, разделенными символом => (который называют «hashrocket» (хэш-ракета)):

```
>> user = { "first_name" => "Michael", "last_name" => "Hartl" }
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

Здесь я использовал обычное для Ruby соглашение, поместив дополнительные пробелы с двух сторон хэша, – они игнорируются при выводе в консоль. (Не спрашивайте меня, почему так принято; вероятно, раньше какому-то влиятельному программисту Ruby понравился внешний вид лишних пробелов, и они застряли в стандартах.)

До сих пор мы использовали строки в качестве ключей хэшей, но в Rails гораздо чаще вместо них используются *символы*. Символы выглядят как строки, но с двоеточием в начале и без кавычек. Например, :name – это символ. Символы можно считать строками, но без всего дополнительного багажа[1]:

```
>> "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
```

Символы – это специальный тип данных в Ruby, почти не используемый в других языках, поэтому они могут показаться странными на первый взгляд, но в Rails они встречаются довольно часто, так что вы быстро к ним привыкнете. В отличие от строк, в символах можно использовать не все символы:

```
>> :foo-bar
NameError: undefined local variable or method `bar' for main:Object
>> :2foo
SyntaxError
```

Символы должны начинаться только с буквы и могут содержать лишь знаки, обычные для слов.

С точки зрения символов как ключей хэшей хэш user можно определить так:

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> user[:name]       # Доступ к значению ключа :name.
=> "MichaelHartl"
>> user[:password]   # Доступ к значению отсутствующего ключа.
=> nil
```

---

[1] Благодаря меньшему количеству багажа символы легче сравнивать друг с другом; строки должны сравниваться посимвольно, а символы можно сравнивать за один шаг. Это делает их идеальными для использования в качестве ключей хэшей.

Из последнего примера видно, что при попытке получить значение для отсутствующего ключа возвращается nil.

Поскольку использование символов в качестве ключей является общепринятой практикой, Ruby (начиная с версии 1.9) поддерживает новый синтаксис специально для этого случая:

```
>> 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
```

Второй вариант синтаксиса предусматривает замену комбинации символ/хэш-ракета именем ключа с двоеточием и значением:

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

Эта конструкция ближе к обозначению хэшей в других языках (таких как JavaScript) и пользуется все возрастающей популярностью в Rails-сообществе. Поскольку оба варианта синтаксиса продолжают широко использоваться, необходимо уметь распознавать их. К сожалению, это может вызывать путаницу, потому что :name является допустимым символом (как таковым), а name: сам по себе не имеет значения. Дело в том, что :name => и name: фактически являются одним и тем же только *внутри литерала хэша*, поэтому

```
{ :name => "Michael Hartl" }
```

и

```
{ name: "Michael Hartl" }
```

эквивалентны, но в других случаях для обозначения символа следует использовать форму записи :name (с двоеточием впереди).

Значениями хэшей может быть практически все, даже другие хэши, как показано в листинге 4.10.

Листинг 4.10 ❖ Вложенные хэши

```
>> params = {}                  # Определение хэша с именем 'params'.
=> {}
>> 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"
```

Такие хэши в хэшах, или *вложенные хэши*, широко используются в Rails, как будет показано в разделе 7.3.

Так же как массивы и диапазоны, хэши имеют метод each. Рассмотрим, например, хэш с именем flash и с ключами для двух условий, :success и :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."
```

Обратите внимание, что, в отличие от метода each массивов, который принимает блок только с одной переменной, метод each для хэшей принимает две, *ключ* и *значение*. То есть метод each выбирает в каждой итерации сразу *пару* ключ/значение.

В последнем примере использован полезный метод inspect, возвращающий строку с литеральным представлением своего объекта:

```
>> puts (1..5).to_a             # Вывести массив как строку.
1
2
3
4
5
>> puts (1..5).to_a.inspect     # Вывести литерал массива.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

Кстати, использование inspect для вывода объекта – достаточно обычное явление, для этого даже есть специальное сокращение – функция p[1]:

```
>> p :name           # Выведет то же, что и 'puts :name.inspect'
:name
```

4.3.4. Еще раз о CSS

Пришло время вернуться к строке из листинга 4.1, используемой в шаблоне для подключения каскадных таблиц стилей:

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track' => true %>
```

Теперь мы почти готовы ее понять. Как упоминалось в разделе 4.1, Rails определяет специальную функцию для подключения таблиц стилей, и

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

[1] На самом деле есть тонкое различие: функция p возвращает объект, а puts возвращает nil. Спасибо читательнице Катаржине Шивек (Katarzyna Siwek) за замечание.

– это вызов такой функции. Но кое-что остается пока неясным. Во-первых, где скобки? В Ruby они не являются обязательными; следующие два выражения эквивалентны:

```
# Скобки в вызовах функций можно опустить.
stylesheet_link_tag('application', media: 'all',
                                   'data-turbolinks-track' => true)
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

Во-вторых, аргумент `media` определенно выглядит как хэш, но где фигурные скобки? Когда хэш является *последним* аргументом в вызове функции, фигурные скобки можно опустить, то есть следующие два выражения эквивалентны:

```
# Если хэш - последний аргумент в вызове функции, фигурные скобки можно опустить.
stylesheet_link_tag 'application', { media: 'all',
                                     'data-turbolinks-track' => true }
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

Далее, почему в паре ключ/значение используется старый синтаксис с хэш-ракетой? Это связано с тем, что новый синтаксис

```
data-turbolinks-track: true
```

нельзя использовать из-за дефисов. (Вспомните, как в разделе 4.3.3 говорилось, что дефисы не могут использоваться в символах.) Это вынуждает использовать старый синтаксис:

```
'data-turbolinks-track' => true
```

Наконец, почему Ruby правильно интерпретирует строки

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

даже с разрывом между элементами? Потому что в данном контексте Ruby не проводит различий между знаками перевода строки и другими пробельными знаками[1]. *Причиной*, по которой я разбил код на части, является моя привычка удерживать длину строк исходного кода в рамках 80 знаков для разборчивости[2].

---

[1] Знак перевода строки – это знак, завершающий одну строку и начинающий новую. В коде он представлен знаком `\n`.

[2] Такой *подсчет* вручную может свести с ума, поэтому во многих текстовых редакторах есть визуальные средства, помогающие в этом. Например, если внимательно посмотреть на рис. 1.5, можно заметить тонкую вертикальную черту справа, которая помогает удерживать длину строк исходного кода в пределах 80 знаков. В облачной IDE (раздел 1.2.1) такая линия включена по умолчанию. Пользователи TextMate смогут найти ее в настройках **View ⇨ Wrap Column ⇨ 78** (Вид ⇨ Переносить строки в поз. ⇨ 78). В Sublime Text можно использовать **View ⇨ Ruler ⇨ 78** (Вид ⇨ Линейка ⇨ 78) или **View ⇨ Ruler ⇨ 80** (Вид ⇨ Линейка ⇨ 80).

Теперь мы видим, что строка

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

вызывает функцию `stylesheet_link_tag` с двумя аргументами: строкой, указывающей путь к таблице стилей, и хэшем с двумя элементами, определяющими тип носителя и требующими от Rails использовать turbolinks[1], новую особенность, появившуюся в Rails 4.0. Благодаря операторным скобкам `<%= %>` результаты вставляются в шаблон встроенным Ruby (ERb), и если в браузере заглянуть в исходный код страницы, можно увидеть разметку HTML, необходимую для подключения таблиц стилей (листинг 4.11). (Там же можно увидеть некоторые дополнительные параметры вроде `?body=1`, после названий файлов CSS. Они вставляются фреймворком Rails, чтобы браузеры перезагружали CSS после их изменения на сервере.)

Листинг 4.11 ❖ Разметка HTML, подключающая таблицы CSS

```
<link data-turbolinks-track="true" href="/assets/application.css"
  media="all" rel="stylesheet" />
```

Если заглянуть в сам CSS-файл, перейдя по адресу `http://localhost:3000/assets/application.css`, можно увидеть, что он (за исключением нескольких комментариев) пуст. Мы приступим к его наполнению в главе 5.

4.4. Ruby-классы

Выше говорилось, что в Ruby все и вся являются объектами, и в этом разделе мы наконец определим несколько собственных объектов. Ruby, как и многие другие объектно-ориентированные языки, для организации методов использует *классы*; эти классы затем применяются для создания объектов. Если вы новичок в объектно-ориентированном программировании, это может звучать как бред, так что давайте рассмотрим несколько конкретных примеров.

4.4.1. Конструкторы

Мы видели много примеров использования классов для создания экземпляров объектов, но нам еще предстоит сделать то же самое явно. Например, мы создавали экземпляры строк с помощью двойных кавычек, которые являются *литеральным конструктором* строк:

```
>> s = "foobar"    # Литеральный конструктор строки с использованием двойных кавычек
=> "foobar"
>> s.class
=> String
```

Как видите, строки имеют метод `class`, просто возвращающий класс, которому они принадлежат.

[1] https://habrahabr.ru/post/167161/.

Вместо литерального конструктора можно воспользоваться аналогичным именованным конструктором, что подразумевает вызов метода new класса[1]:

```
>> s = String.new("foobar")    # Именованный конструктор строк
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

Он действует подобно литеральному конструктору, но более явно говорит о наших намерениях.

Массивы в этом контексте работают так же, как строки:

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

Хэши, напротив, действуют иначе. В отличие от конструктора массива Array.new, принимающего начальное значение массива, Hash.new принимает значение *по умолчанию*, которое будет возвращаться в ответ на попытку обратиться к несуществующему ключу:

```
>> h = Hash.new
=> {}
>> h[:foo]          # Попытка обращения к несуществующему ключу :foo.
=> nil
>> h = Hash.new(0)  # В ответ на попытку обратиться к несуществующему ключу возвращать 0
                    # вместо nil.
=> {}
>> h[:foo]
=> 0
```

Когда метод вызывается относительно самого класса, как в случае с new, он называется *методом класса*. Результатом вызова new является объект соответствующего класса, также называемый *экземпляром* класса. Метод, вызываемый относительно экземпляра, например length, называется *методом экземпляра*.

4.4.2. Наследование классов

Знакомясь с классами, полезно выяснить *иерархию класса* с помощью метода superclass:

```
>> s = String.new("foobar")
=> "foobar"
>> s.class# Определить класс s.
=>String
>> s.class.superclass              # Найти базовый класс для String.
```

[1] Результаты зависят от версии Ruby. В этом примере предполагается, что используется версия Ruby 1.9.3 или выше.

```
=>Object
>> s.class.superclass.superclass   # В Ruby 1.9 используется новый
                                   # базовый класс BasicObject
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

Диаграмма этой иерархии наследования представлена на рис.4.1. Как видите, суперклассом для `String` является класс `Object`, а суперклассом для `Object` – класс `BasicObject`, но `BasicObject` не имеет суперкласса. Этой модели следуют все объекты в Ruby: иерархия наследования классов может быть очень глубокой, но каждый класс в конечном счете наследует `BasicObject`, у которого нет суперкласса. Это техническое значение выражения «все и вся в Ruby являются объектами».

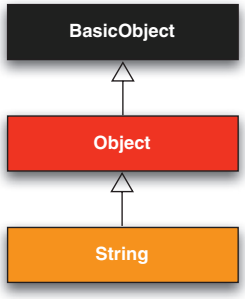


Рис. 4.1 ❖ Иерархия наследования класса String

Проникнуть в суть классов невозможно, не попытавшись создать хотя бы один свой класс. Давайте создадим класс `Word` с методом `palindrome?`, возвращающим `true`, если слово одинаково читается справа налево и слева направо:

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

Ниже показано, как его можно использовать:

```
>> w = Word.new                 # Создать новый объект Word.
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

Если этот пример покажется вам немного надуманным, хорошо – так и было задумано. Было бы странно создавать новый класс, только чтобы определить метод, принимающий строку в аргументе. Поскольку слово *является* строкой, бо-

лее естественным решением будет *унаследовать* класс `String` в `Word`, как показано в листинге 4.12. (Выйдите из консоли и войдите в нее снова, чтобы удалить старое определение `Word`.)

Листинг 4.12 ❖ Определение класса Word в консоли

```
>> class Word < String       # Word наследует String.
>>   # Возвращает true, если строка является палиндромом.
>>   def palindrome?
>>     self == self.reverse  # self - сама строка.
>>   end
>> end
=> nil
```

`Word < String` – так в Ruby определяется наследование (кратко обсуждалось в разделе 3.2), которое гарантирует, что в дополнение к новому методу `palindrome?` слова (экземпляры `Word`) будут иметь все те же методы, что и строки (экземпляры `String`):

```
>> s = Word.new("level")   # Создать экземпляр Word, инициализировать строкой "level".
=> "level"
>> s.palindrome?           # Экземпляры Word имеют метод palindrome?.
=> true
>> s.length                # Экземпляры Word также наследуют все строковые методы.
=> 5
```

Класс `Word` наследует `String`, и мы можем явно увидеть его иерархию в консоли:

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

Она также изображена на рис. 4.2.

Обратите внимание в листинге 4.12, что проверка применяется к строке внутри класса `Word`. Ruby позволяет сделать это с помощью ключевого слова `self`: внутри класса `Word` ключевое слово `self` представляет сам объект, а значит, его можно использовать

```
self == self.reverse
```

чтобы проверить, является ли слово палиндромом[1]. Фактически в методах и атрибутах класса `Word` ключевое слово `self` можно не использовать (если не выполняется присваивание), поэтому тот же результат можно получить так:

```
self == reverse
```

---

[1] Больше о классах Ruby и ключевом слове `self` можете узнать в статье по адресу http://www.railstips.org/blog/archives/2006/11/18/class-and-instance-variables-in-ruby/.

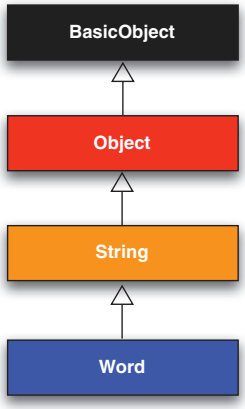


Рис. 4.2 ❖ Иерархия наследования (невстроенного) класса Word из листинга 4.12

4.4.3. Изменение встроенных классов

Наследование – мощный механизм, но в случае с палиндромами более естественно было бы добавить метод `palindrome?` в сам класс `String`, чтобы его (среди прочих) можно было вызвать для строковых литералов, что мы сейчас и сделаем:

```
>> "level".palindrome?
NoMethodError: undefined method `palindrome?' for "level":String
```

Немного удивительно, что Ruby разрешает это; Ruby-классы можно открывать и изменять, что позволяет простым смертным, как мы, самостоятельно добавлять в них методы:

```
>> class String
>>   # Вернуть true, если строка является палиндромом.
>>   def palindrome?
>>     self == self.reverse
>>   end
>> end
=> nil
>> "deified".palindrome?
=> true
```

Я и не знаю, что круче: то, что Ruby позволяет добавлять методы во встроенные классы, или то, что слово «deified» («боготворить») является палиндромом.

Изменение встроенных классов – мощный прием, но с большой властью приходит большая ответственность, и считается дурным тоном добавлять методы во встроенные классы, не имея *по-настоящему* серьезной причины для этого. В Rails есть несколько серьезных причин; например, в веб-приложениях часто бывает желательно, чтобы переменные не оставались *чистыми* – имя пользователя не должно быть пустым или состоять из одних пробелов, – поэтому Rails добавляет метод

blank? в Ruby. Так как Rails-консоль автоматически включает Rails-расширения, мы можем увидеть это здесь (не сработает в простом irb):

```
>> "".blank?
=> true
>> " ".empty?
=> false
>> " ".blank?
=> true
>> nil.blank?
=> true
```

Мы видим, что строка пробелов *непустая*, но она *чистая*. Обратите также внимание, что значение nil является чистым; так как значение nil не является строкой – это намек, что Rails фактически добавляет метод blank? в базовый класс, который наследует String, то есть (как мы видели в начале этого раздела) в класс Object. Еще несколько примеров Rails-дополнений в Ruby-классы мы увидим в разделе 8.4.

4.4.4. Класс контроллера

Все эти разговоры о классах и наследовании, возможно, у кого-то вызвали смутные воспоминания, потому что мы видели их раньше, в контроллере StaticPages (листинг 3.18):

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

Теперь вы в состоянии оценить, по крайней мере приблизительно, что этот код означает: StaticPagesController – это класс, наследующий ApplicationController и определяющий методы home, help и about. Так как каждый сеанс Rails-консоли загружает локальное окружение Rails, мы можем даже создать контроллер в явном виде и изучить его иерархию классов[1]:

```
>> controller = StaticPagesController.new
=> #<StaticPagesController:0x22855d0>
```

[1] От вас не требуется досконально знать, что делает каждый класс в этой иерархии. Даже я не знаю, что они все делают, хотя программирую на Ruby on Rails с 2005 года. Это означает, что или (а) я чрезвычайно некомпетентен, или (б) можно быть квалифицированным Rails-разработчиком, не зная всех его внутренностей. Для нашей с вами пользы я уповаю на последнее.

```
>> 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
```

Диаграмма этой иерархии представлена на рис. 4.3.

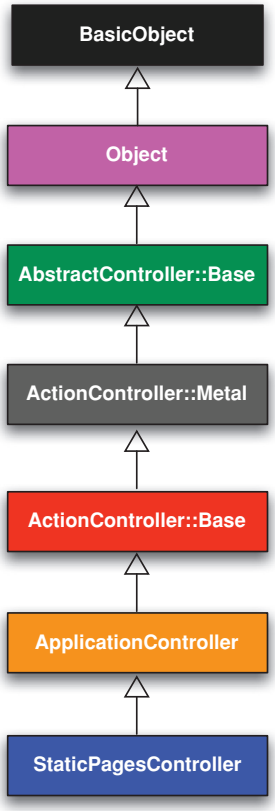


Рис. 4.3 ❖ Иерархия наследования контроллера StaticPages

Мы можем даже вызвать методы действий контроллера в консоли, которые являются самыми обычными методами:

```
>> controller.home
=> nil
```

Здесь метод `home` вернул `nil`, потому что он ничего не делает.

Но постойте: методы действий не возвращают значений, по крайней мере, не такие. Целью метода `home`, как мы видели в главе 3, является отображение вебстраницы, а не возврат значения. И я бы точно не забыл, если бы когда-либо где-либо вызывал `StaticPagesController.new`. Что происходит?

А происходит вот что: Rails *написан на* Ruby, но Rails – это не Ruby. Некоторые Rails-классы используются как обычные объекты Ruby, но некоторые льют воду на «волшебную мельницу» Rails. Фреймворк Rails – это нечто особенное, и его следует изучать и исследовать отдельно от Ruby.

4.4.5. Класс User

Наше путешествие по Ruby мы закончим определением собственного класса `User`, предвосхищающего модель User, которая появится в главе 6.

До сих пор мы вводили определения классов в консоли, но это быстро надоедает. Поэтому теперь создайте файл `example_user.rb` в корневом каталоге приложения и заполните его содержимым листинга 4.13.

Листинг 4.13 ❖ Пример определения класса User (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
```

Здесь довольно много чего, поэтому будем разбираться с этим шаг за шагом. Первая строка

```
attr_accessor :name, :email
```

создает *методы доступа к атрибутам* с именем пользователя и адресом электронной почты. В результате появляются два метода – «getter» (метод чтения) и «setter» (метод записи), – которые позволяют читать (get) и записывать (set) значения в *переменные экземпляра* `@name` и `@email`, коротко упоминавшиеся в разделах 2.2.2 и 3.6. Принципиальная важность переменных экземпляров в Rails заключается в том, что они автоматически становятся доступны в представлениях. Но вообще они используются как переменные, доступные в любой точке Ruby-класса. (Более подробно об этом мы поговорим чуть ниже.) Переменные экземпляра всегда начинаются со знака `@` и имеют значение `nil`, если не определены.

Первый метод –initialize–, специальный в Ruby: он автоматически вызывается, когда выполняется User.new. Конкретно этот метод initialize принимает один аргумент, attributes:

```
def initialize(attributes = {})
  @name = attributes[:name]
  @email = attributes[:email]
end
```

Здесь переменная attributes имеет *значение по умолчанию* – пустой хэш, что дает возможность определить пользователя без имени и адреса электронной почты. (Как рассказывалось в разделе 4.3.3, хэши возвращают nil при обращении к несуществующим ключам, поэтому attributes[:name] вернет nil в отсутствие ключа :name, так же как attributes[:email].)

Наконец, класс User определяет метод formatted_email, использующий значения переменных @name и @email для создания отформатированной версии адреса электронной почты пользователя путем интерполяции строк (раздел 4.2.2):

```
def formatted_email
  "#{@name} <#{@email}>"
end
```

Переменные @name и @email автоматически доступны в методе formatted_email, так как являются переменными экземпляра (на это указывает знак @).

Давайте запустим консоль, загрузим класс User и поиграем с ним:

```
>> require './example_user'                # Загрузить пример example_user.
=> true
>> example = User.new
=> #<User:0x224ceec @email=nil, @name=nil>
>> example.name                           # nil, потому что attributes[:name] == nil
=> nil
>> example.name = "Example User"          # Присвоить имя
=> "Example User"
>> example.email = "user@example.com"     # и адрес электронной почты
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

Символ точки в первой команде обозначает «текущий каталог», соответственно, путь './example_user' требует от Ruby искать файл примера в этом каталоге. Последующий код создает пустой экземпляр класса User и заполняет его поля имени и адреса электронной почты (это возможно благодаря строке attr_accessor в листинге 4.13). Инструкция

```
example.name = "Example User"
```

присвоит переменной @name строку "Example User" (аналогичная операция выполняется с атрибутом email), которая затем используется методом formatted_email.

Передав хэш в вызов метода `initialize`, можно создать другого пользователя, с заранее определенными атрибутами (напомню, что фигурные скобки вокруг хэша можно опустить, если он является последним аргументом, как рассказывалось в разделе 4.3.4):

```
>> 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>"
```

Начиная с главы 7 вы увидите, что инициализация объектов передачей хэшей в аргументах широко используется в Rails-приложениях (прием *массового присваивания*).

4.5. Заключение

На этом мы завершаем обзор языка Ruby. В главе 5 мы начнем применять полученные знания, приступив к разработке учебного приложения.

Файл `example_user.rb`, созданный в разделе 4.4.5, нам больше не пригодится, поэтому я рекомендую удалить его:

```
$ rm example_user.rb
```

Затем зафиксируйте остальные изменения в основном репозитории проекта, отправьте код в Bitbucket и разверните на Heroku:

```
$ git status
$ git commit -am "Add a full_title helper"
$ git push
$ bundle exec rake test
$ git push heroku
```

4.5.1. Что мы узнали в этой главе

- В Ruby существует большое количество методов для работы со строками и символами.
- Все в Ruby является объектом.
- Ruby поддерживает определение методов через ключевое слово `def`.
- Ruby поддерживает определение классов через ключевое слово `class`.
- Rails-представления могут содержать статическую разметку HTML или код на встроенном Ruby (ERb).
- Ruby поддерживает встроенные структуры данных, включая массивы, диапазоны и хэши.
- Ruby-блоки – это гибкая конструкция, которая позволяет (среди прочего) выполнять перебор перечисляемых структур данных.
- Символы (symbols) – это метки, как строки без какой-либо дополнительной структуры.
- Ruby поддерживает наследование объектов.

- Поддерживается возможность открывать и изменять встроенные Ruby-классы.
- Слово «deified» – палиндром.

4.6. Упражнения

Примечание. *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

1. Заменяя знаки вопроса в листинге 4.14 соответствующими методами, напишите функцию перемешивания букв в строке, используя методы `split`, `shuffle` и `join`.
2. Используя листинг 4.15 как руководство к действию, добавьте метод `shuffle` в класс `String`.
3. Создайте три хэша с именами `person1`, `person2` и `person3` для хранения имен и фамилий под ключами `:first` и `:last`. Затем создайте хэш `params`, чтобы элемент `params[:father]` ссылался на `person1`, элемент `params[:mother]` – на `person2` и элемент `params[:child]` – на `person3`. Проверьте что, например, `params[:father][:first]` возвращает правильное значение.
4. Найдите онлайн-версию документации с описанием Ruby API и почитайте о методе `merge` класса `Hash`. Что значит следующее выражение?

```
{ "a" => 100, "b" => 200 }.merge({ "b" => 300 })
```

Листинг 4.14 ❖ Заготовка для функции перемешивания символов в строке

```
>> def string_shuffle(s)
>>   s.?('').?.?
>> end
>> string_shuffle("foobar")
=> "oobfra"
```

Листинг 4.15 ❖ Заготовка метода shuffle в классе String

```
>> class String
>>   def shuffle
>>     self.?('').?.?
>>   end
>> end
>> "foobar".shuffle
=> "borafo"
```

Глава 5

Заполнение макета

В процессе краткого обзора Ruby в главе 4 мы узнали, как подключить таблицы стилей к учебному приложению (раздел 4.1), но (как было отмечено в разделе 4.3.4) таблицы стилей пока пусты. В этой главе мы начнем их заполнять, подключив фреймворк CSS к приложению, а затем добавим немного собственных стилей[1]. Также мы начнем заполнять макет ссылками на страницы (например, Home и About), созданные ранее (раздел 5.1). Попутно мы познакомимся с частичными шаблонами, маршрутами и ресурсами, а также уделим немного внимания Sass (раздел 5.2). Мы закончим эту главу, сделав первый важный шаг – реализовав возможность регистрации пользователей на нашем сайте (раздел 5.4).

Большинство изменений в этой главе заключается в добавлении и редактировании разметки в макете сайта учебного приложения, а это (основываясь на рекомендациях из блока 3.3) именно та работа, которую мы обычно выполняем, полностью или частично, без тестирования. То есть большую часть времени мы проведем в текстовом редакторе и браузере, используя приемы TDD только при добавлении страницы контактов (раздел 5.3.1). Тем не менее мы добавим важные новые тесты и напишем первые *интеграционные тесты*, проверяющие корректность ссылок в окончательном макете (раздел 5.3.4).

5.1. Добавление некоторых структур

Эта книга посвящена разработке веб-приложений, а не веб-дизайну, но работа над приложением с *дрянным* внешним видом удручает, поэтому в этом разделе мы добавим некоторые структуры в макет и немного облагородим внешний вид с помощью CSS. В дополнение к нашим собственным CSS-правилам мы будем использовать Bootstrap[2], фреймворк для веб-дизайна с открытым исходным кодом от компании Twitter. Мы также придадим *коду* немного стиля, так сказать, используя частичные шаблоны, чтобы очистить основной макет от ненужного хлама.

При создании веб-приложений обычно бывает полезно как можно раньше определить общий вид пользовательского интерфейса. Поэтому на протяжении осталь-

---

[1] Спасибо читателю Колму Тьюту (Colm Tuite) за его замечательную работу по переводу учебного приложения на CSS-фреймворк Bootstrap.

[2] http://getbootstrap.com/.

ной части книги я часто буду использовать *макеты* (в контексте веб-разработки их часто называют *каркасами*) – черновые наброски внешнего вида приложения[1]. В этой главе мы будем разрабатывать главным образом статические страницы, начатые в разделе 3.2, включая логотип сайта, заголовок с навигацией по сайту и подвал сайта. Макет наиболее важной из страниц, Home, показан на рис. 5.1. Конечный результат можно увидеть на рис. 5.7. Он отличается в некоторых деталях – например, в конечном варианте на страницу добавлен логотип Rails, – но это нормально, так как макет и не должен быть абсолютно точным.

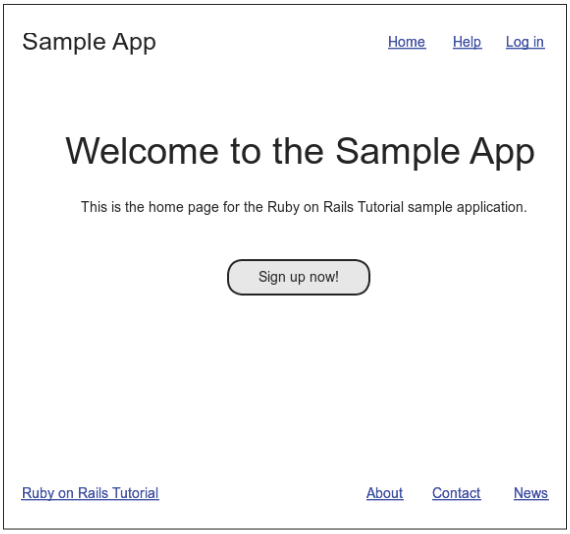


Рис. 5.1 ❖ Макет страницы Home учебного приложения

Как обычно, если вы используете Git для управления версиями, сейчас самое время создать новую ветку:

```
$ git checkout master
$ git checkout -b filling-in-layout
```

5.1.1. Навигация по сайту

В качестве первого шага перед добавлением ссылок и стилей добавим в файл макета сайта `application.html.erb` (последний раз он был показан в листинге 4.3) дополнительную разметку HTML, включая несколько дополнительных разделов, несколько CSS-классов и заготовку навигации по сайту. Полный файл представлен в листинге 5.1; объяснения различных частей последуют сразу за ним. Если вы не хотите откладывать удовольствие, можете посмотреть на результат, представленный на рис. 5.2. (Примечание: внешний вид (пока) ничем не впечатляет.)

[1] Макеты в этой книге созданы с помощью замечательного онлайн-приложения Mockingbird (https://gomockingbird.com/).

Листинг 5.1 ❖ Макет сайта с дополнительной разметкой (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>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

Рассмотрим новые элементы в листинге 5.1 сверху вниз. Как отмечалось в разделе 3.4.1, Rails по умолчанию использует HTML5 (об этом говорит тег `<!DOCTYPE html>`); так как стандарт HTML5 относительно новый и некоторые браузеры (особенно старые версии Internet Explorer) не полностью его поддерживают, мы включили расширение («HTML5 shim (или shiv)»)[1] на языке JavaScript (https://github.com/aFarkas/html5shiv)[2], помогающее обойти это затруднение:

---

[1] Слова *shim* и *shiv* взаимозаменяемы в данном контексте; первое – действительный термин, основанный на английском слове, означающем «шайба, прокладка или тонкая полоска материала, которая используется для сопоставления частей или уменьшения износа», второе («нож или бритва, используемая в качестве оружия»), по-видимому, игра слов от имени первоначального автора этого расширения Сьерда Висшера (Sjoerd Visscher).

[2] https://ru.wikipedia.org/wiki/HTML5_Shiv.

```
<!--[if lt IE 9]>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
  </script>
<![end if]-->
```

Следующая строка выглядит немного странно:

```
<!--[if lt IE 9]>
```

Она выполняет закомментированную строку, если версия Microsoft Internet Explorer (IE) ниже 9 (`if lt IE 9`). Этот странный синтаксис `[if lt IE 9]` *не* является частью Rails; это условный комментарий[1], поддерживаемый браузерами Internet Explorer специально для подобных ситуаций. Это удобно хотя бы потому, что можно подключить расширение поддержки HTML5 только в версиях браузера IE ниже 9, не затрагивая браузеров, таких как Firefox, Chrome и Safari.

Следующий раздел – это `header` (заголовок) для логотипа (текстового) сайта с парой разделов (в тегах `div`) и списком элементов с навигационными ссылками:

```
<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>
```

Здесь тег `header` включает элементы, которые должны находиться вверху страницы. Мы присвоили ему три CSS-класса[2] – `navbar`, `navbar-fixed-top` и `navbar-inverse`, перечислив их через пробел:

```
<header class="navbar navbar-fixed-top navbar-inverse">
```

Всем HTML-элементам можно присвоить классы и *идентификаторы* `id` - ярлыки, используемые для применения стилей CSS (раздел 5.1.2). Основное различие между классами и идентификаторами в том, что классы можно использовать много раз на странице, а идентификаторы – лишь один. В данном случае все `navbar` классы имеют особое значение для фреймворка Bootstrap, который мы установим и начнем использовать в разделе 5.1.2.

Внутри тега `header` находится тег `div`:

```
<div class="container">
```

---

[1] https://ru.wikipedia.org/wiki/Условный_комментарий.

[2] Классы CSS не имеют никакого отношения к классам Ruby.

Тег `div` – это обобщенный раздел; он не делает ничего, кроме деления документа на отдельные части. В прежних версиях HTML теги `div` использовались для оформления почти всех разделов сайта, но в HTML5 были добавлены элементы `header`, `nav` и `section` для выделения разделов, общих для многих приложений. В данном случае тегу `div` присвоен CSS-класс `container`. Как и в случае с тегом `header`, он имеет особое значение для Bootstrap.

За тегом `div` следует код на встроенном Ruby:

```
<%= 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>
```

Здесь использована вспомогательная функция `link_to` из Rails, создающая ссылки (которые в разделе 3.2.2 мы создавали непосредственно, с помощью тега `a`); первый аргумент в вызове `link_to` – это текст ссылки, второй – адрес URL. В разделе 5.3.3 мы заменим адреса URL *именованными маршрутами*, а пока будем использовать заглушки `'#'`, обычно применяемые в веб-дизайне. Третий аргумент – это хэш дополнительных параметров, в данном случае идентификатор `"logo"` класса CSS для ссылки на учебное приложение. (При создании остальных трех ссылок хэш дополнительных параметров не используется, и в этом нет ничего необычного, так как это необязательный аргумент.) Вспомогательные функции Rails часто принимают хэш дополнительных параметров подобным образом, что дает возможность добавлять произвольные HTML-параметры, оставаясь в рамках Rails.

Второй элемент внутри `div` – это список навигационных ссылок, созданный с помощью тега *ненумерованного списка* `ul` и тега *элемента списка* `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>
```

Формально тег `<nav>` здесь не нужен, он служит лишь признаком, что это именно навигационные ссылки. Но классы `nav`, `navbar-nav` и `navbar-right` в теге `ul` имеют особое значение для Bootstrap и служат цели автоматического применения стилей после подключения Bootstrap CSS, что мы проделаем в разделе 5.1.2. После того как Rails обработает этот макет и встроенный Ruby, список будет выглядеть, как показано ниже (вы можете убедиться в этом, открыв исходный код страницы в браузере)[1]:

[1] Интервалы могут немного отличаться, но это не страшно, потому что (как было сказано в разделе 3.4.1) HTML нечувствителен к пробелам.

```
<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>
```

Именно этот текст будет отправлен браузеру.

Заключительная часть макета – это элемент `div` для основного содержимого:

```
<div class="container">
  <%= yield %>
</div>
```

Как и прежде, класс `container` имеет специальное значение для Bootstrap. Как мы узнали в разделе 3.4.3, метод `yield` вставляет содержимое каждой страницы в макет сайта.

Если не брать в расчет подвал сайта, который мы добавим в разделе 5.1.3, макет практически завершен, и мы можем посмотреть на результаты, открыв страницу Home. Чтобы извлечь пользу из ожидающих своего часа элементов стиля, добавим несколько дополнительных элементов в представление `home.html.erb` (листинг 5.2).

Листинг 5.2 ❖ Страница Home со ссылкой на страницу регистрации (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/' %>
```

В рамках подготовки к реализации функции регистрации пользователей (глава 7) первый вызов `link_to` создает заглушку для ссылки:

```
<a href="#" class="btn btn-lg btn-primary">Sign up now!</a>
```

Класс CSS `jumbotron` в теге `div` имеет особое значение для Bootstrap, так же как классы `btn`, `btn-lg` и `btn-primary` в кнопке запуска процедуры регистрации.

Второй вызов `link_to` демонстрирует применение вспомогательной функции `image_tag`, которая принимает в качестве аргумента путь к изображению и хэш с дополнительными параметрами, в данном случае атрибут `alt` для тега `image`

в виде символа. Чтобы это сработало, нужен файл изображения с именем `rails.png`, который можно загрузить с главной страницы сайта книги (http://rubyonrails.org/images/rails.png) и поместить его в каталог `app/assets/images/`. Если вы используете облачную IDE или другую Unix-подобную систему, можете сделать это с помощью утилиты `curl`[1]:

```
$ curl -O http://rubyonrails.org/images/rails.png
$ mv rails.png app/assets/images/
```

Если вторая команда не сработала, что иногда случается в облачной IDE по не понятным мне причинам, попробуйте перезапускать первую команду `curl`, пока файл не загрузится. (За дополнительной информацией о команде `curl` обращайтесь к главе 3 в книге «Conquering the Command Line»[2].) Так как в листинге 5.2 мы использовали вспомогательную функцию `image_tag`, Rails будет автоматически искать все картинки в каталоге ресурсов `app/assets/images/` (раздел 5.2).

Чтобы прояснить действие `image_tag`, посмотрим на произведенную ею разметку HTML[3]:

```
<img alt="Rails logo"
src="/assets/rails-9308b8f92fea4c19a3a0d8385b494526.png" />
```

Rails добавляет строку `9308b8f92fea4c19a3a0d8385b494526` (в вашей системе она будет отличаться), чтобы гарантировать уникальность имени файла, это помогает браузеру загрузить правильную картинку в случае ее обновления (а не получать ее из кэша браузера). Обратите внимание, что атрибут `src` *не* включает каталог `images`, вместо этого он использует каталог `assets`, общий для всех ресурсов (изображения, сценарии на JavaScript, таблицы стилей CSS и т. д.). На сервере Rails связывает изображения в каталоге `assets` с правильным каталогом `app/assets/images`, но, поскольку браузер считает, что все ресурсы находятся в одном каталоге, это позволяет ему ускорить работу с ними. Атрибут `alt` – это текст для программ, которые не загружают картинки (например, программа чтения с экрана для слабовидящих).

Теперь мы наконец готовы увидеть плоды наших трудов (рис. 5.2). Говорите, «не впечатляет»? Может быть. К счастью, мы проделали хорошую работу, присвоив нашим HTML-элементам классы с говорящими именами, которые позволят придать сайту стиль с помощью CSS.

5.1.2. Bootstrap и собственные стили CSS

В разделе 5.1.1 мы связали многие HTML-элементы с CSS-классами, и это дало нам большую гибкость при построении макета. Как отмечалось в разделе 5.1.1, многие из этих классов специфичны для Bootstrap, фреймворка, распростра-

---

[1] В OS X `curl` можно установить с помощью диспетчера пакетов Homebrew: `brew install curl`.

[2] http://conqueringthecommandline.com/book/curl.

[3] Вы могли заметить, что вместо пары тегов `<img>...</img>` использован единственный тег `<img ... />`. Теги такого вида известны как *самозакрывающиеся*.

няемого компанией Twitter, упрощающего создание привлекательного веб-дизайна и элементов пользовательского интерфейса для приложений HTML5. В этом разделе соединим Bootstrap с небольшим количеством собственных CSS-правил, чтобы придать некоторый стиль учебному приложению. Стоит отметить, что при использовании Bootstrap дизайн приложения автоматически становится *адаптивным*, гарантируя корректное отображение на широком спектре устройств.

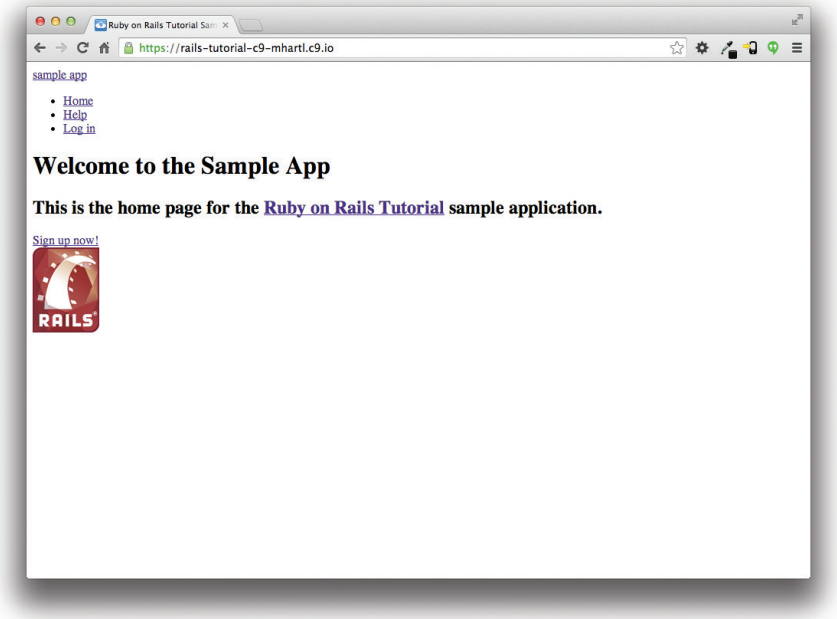


Рис. 5.2 ❖ Страница Home без добавления CSS

Для начала добавим Bootstrap. В Rails-приложениях это можно сделать с помощью гема `bootstrap-sass`, как показано в листинге 5.3. Фреймворк Bootstrap изначально использует язык Less CSS10[1] для создания динамических таблиц стилей, но конвейер ресурсов в Rails по умолчанию поддерживает (очень похожий) язык Sass (раздел 5.2); `bootstrap-sass` преобразует Less в Sass и делает все необходимые файлы Bootstrap доступными для текущего приложения[2].

Листинг 5.3 ❖ Добавление гема bootstrap-sassgem в файл Gemfile

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

gem 'rails',                '4.2.0'
```

[1] http://lesscss.org/.

[2] Механизм обслуживания ресурсов позволяет использовать Less; читайте об этом в описании гемма `less-rails-bootstrap` (https://rubygems.org/gems/less-rails-bootstrap).

```
gem 'bootstrap-sass',       '3.2.0.0'
.
.
.
```

Как обычно, запустим `bundle install`, чтобы установить Bootstrap:

```
$ bundle install
```

Хотя `rails generate` автоматически создает отдельный CSS-файл для каждого контроллера, подключить их в нужном порядке довольно сложно, поэтому для простоты поместим все определения стилей CSS в один файл. Итак, создадим этот файл:

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

(Здесь использован трюк с командой `touch` из раздела 3.3.3, но вы можете создать файл любым удобным вам способом.) Здесь важно и название каталога, и расширение. Каталог

```
app/assets/stylesheets/
```

является частью комплекта ресурсов (раздел 5.2), и все таблицы стилей из него автоматически будут добавлены в файл `application.css`, подключаемый к макету сайта. Более того, имя файла `custom.css.scss` содержит расширение `.css`, обозначающее файл CSS, и расширение `.scss`, обозначающее файл «Sassy CSS», что указывает конвейеру ресурсов на необходимость обрабатывать его с помощью Sass. (Мы не будем пользоваться Sass до раздела 5.2.2, но сейчас это нужно гему `bootstrap-sass`, чтобы сотворить свое волшебство.)

Внутри файла для нестандартных стилей CSS можно использовать функцию `@import`, чтобы подключить Bootstrap (вместе со связанной утилитой Sprockets), как показано в листинге 5.4[1].

Листинг 5.4 ❖ Добавление стилей из Bootstrap (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";
```

Эти две строки в листинге 5.4 подключают весь фреймворк Bootstrap CSS. После перезапуска веб-сервера для принятия изменений (комбинацией **Ctrl-C** с последующим выполнением команды `rails server`, как описывалось в разделе 1.3.2) вы получите результат, как на рис. 5.3. Текст расположен неудачно, и логотип оказался неоформленным, но цвета и кнопка регистрации выглядят многообещающе.

Теперь мы добавим несколько стилей CSS, которые будут использоваться повсюду на сайте для оформления макета и каждой отдельной страницы (листинг 5.5). Результат изображен на рис. 5.4. (В листинге 5.5 довольно много правил;

---

[1] Если все это выглядит несколько загадочным, мужайтесь: я просто следую инструкции из файла `README` гема `bootstrap-sass`.

чтобы понять, что делает конкретно то или иное правило, часто бывает полезным закомментировать его, окружив парами символов /* и */, и посмотреть на изменения.)

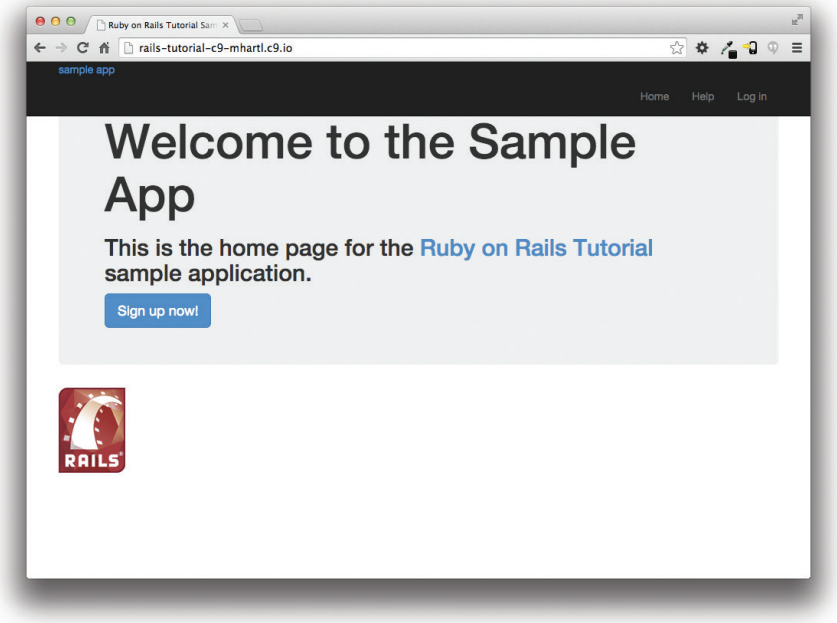


Рис. 5.3 ❖ Учебное приложение с Bootstrap CSS

Листинг 5.5 ❖ Добавление стилей, применяемых для оформления всех страниц (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* общие */

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}
```

```
.center h1 {
  margin-bottom: 10px;
}
```

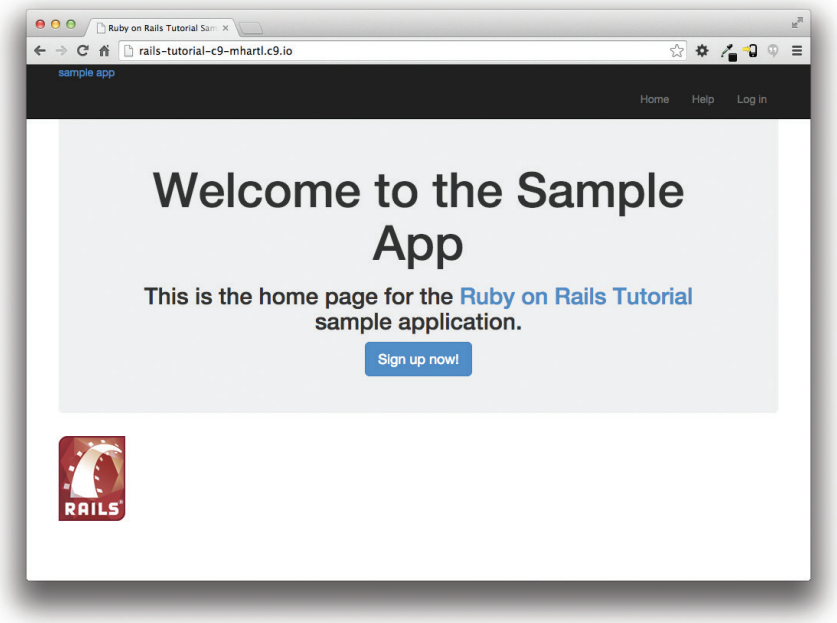


Рис. 5.4 ❖ Результат добавления нескольких отступов и прочих общих стилей

Обратите внимание, что код CSS в листинге 5.5 представлен в виде последовательных правил. Обычно правила CSS начинаются с названия класса, идентификатора `id`, тега HTML или любой их комбинации, за которыми следует список команд оформления. Например, команда

```
body {
  padding-top: 60px;
}
```

задает отступ в 60 пикселей от верхнего края страницы. Благодаря классу `navbar-fixed-top` в теге `header` Bootstrap фиксирует панель навигации вверху страницы, то есть отступ служит для отделения основного текста от панели навигации. (Поскольку цвет навигационной панели по умолчанию отличается от использовавшегося в Bootstrap 2.0, нам пришлось добавить класс `navbar-inverse`, чтобы сделать ее темной, а не белой.) Далее, правило

```
.center {
  text-align: center;
}
```

связывает класс `center` со свойством `text-align: center`. Другими словами, точка `.` в `.center` указывает, что правило относится к классу. (В листинге 5.7 мы увидим, что знак решетки `#` связывает правило с идентификатором `id`.) Это означает, что элементы внутри любого тега (такого как `div`) с классом `center` будут выравниваться по центру страницы. (Пример такого класса мы видели в листинге 5.2.)

Несмотря на то что в Bootstrap изначально включены CSS-правила для привлекательного оформления, мы добавим немного собственных правил оформления текста на нашем сайте, представленных в листинге 5.6. (Не все они используются в странице Home, но каждое правило будет использовано в какой-то части учебного приложения.) Результат добавления листинга 5.6 показан на рис. 5.5.

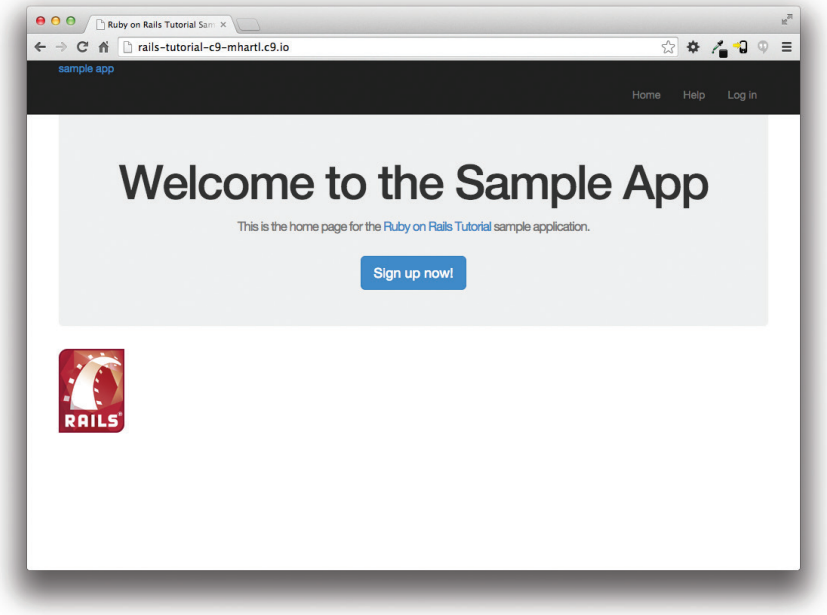


Рис. 5.5 ❖ Результат добавления нескольких стилей для оформления текста

Листинг 5.6 ❖ Дополнительные правила CSS для улучшения оформления текста (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";
.
.
.
/* правила для текста */

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: #777;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}
```

Наконец, добавим несколько правил оформления логотипа сайта, который представляет собой просто текст «sample app». Правила в листинге 5.7 преобразуют буквы текста в верхний регистр, а также изменяют их размер, цвет и положение. (Мы связали правила с идентификатором `id`, так как предполагается, что логотип сайта будет на странице в единственном экземпляре, но вы можете связать правила с классом.)

Листинг 5.7 ❖ Правила CSS для оформления логотипа сайта (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";
.
.
.
/* заголовок */

#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;
}
```

Объявление `color: #fff` изменяет цвет текста логотипа на белый. Цвета в языке HTML можно кодировать тремя парами шестнадцатеричных чисел, по одной для каждого из основных цветов – красного, зеленого и синего (именно в этом порядке). Код `#ffffff` соответствует максимальной интенсивности всех трех цветов, что дает в результате чистый белый цвет, а `#fff` – это сокращенная форма записи `#ffffff`. Стандарт CSS дополнительно определяет большое количество синонимов для часто употребляемых цветов, в том числе `white` (белый) для `#fff`. Результат применения правил CSS из листинга 5.7 показан на рис. 5.6.

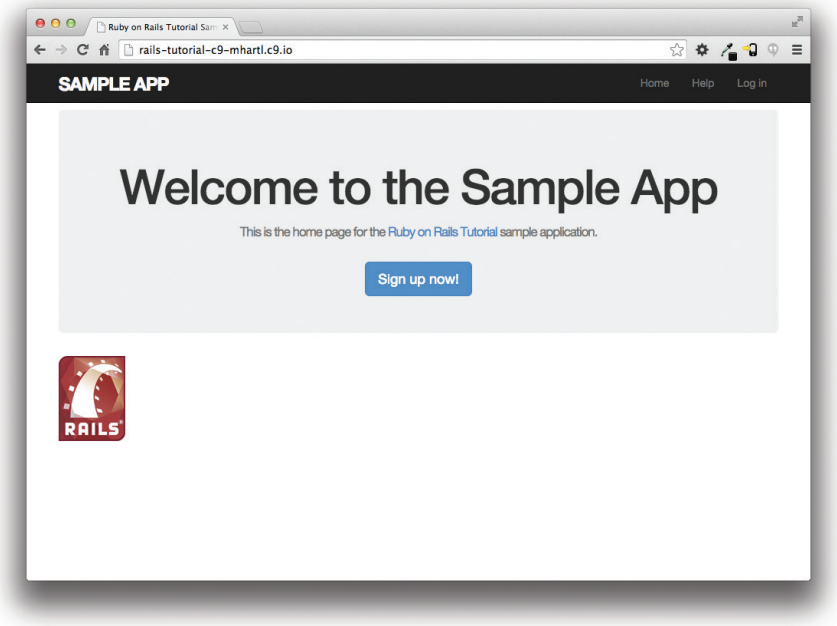


Рис. 5.6 ❖ Учебное приложение с оформленным логотипом

5.1.3. Частичные шаблоны

Макет в листинге 5.1 выполняет свою работу, но он начинает понемногу захламляться. Подключение библиотеки JavaScript для поддержки HTML5 занимает три строки и использует странный IE-специфичный синтаксис. Было бы хорошо вынести его в какое-нибудь отдельное место. Кроме того, заголовок HTML формирует обособленную логическую единицу, и его тоже было бы неплохо оформить отдельно. Мы можем сделать это, используя удобную особенность Rails – частичные шаблоны. Давайте сначала посмотрим, как будет выглядеть макет после определения частичных шаблонов (листинг 5.8).

Листинг 5.8 ❖ Макет сайта с частичными шаблонами для таблицы стилей и заголовка (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>
```

В листинге 5.8 мы заменили строки подключения библиотеки JavaScript вызовом вспомогательной функции `render`:

```
<%= render 'layouts/shim' %>
```

В данном случае вспомогательная функция отыщет файл `app/views/layouts/_shim.html.erb`, обработает его содержимое и вставит результат в представление[1]. (Напоминаю, что конструкция `<%= ... %>` служит для выполнения Ruby-выражений и вставки результата в макет.) Обратите внимание на символ подчеркивания в начале имени файла `_shim.html.erb`; это универсальное соглашение об именовании частичных шаблонов, которое, среди прочего, позволяет заметить все шаблоны в каталоге с первого взгляда.

Конечно, чтобы использовать частичные шаблоны, нужно создать соответствующие файлы и заполнить их чем-то; в случае с шаблоном загрузки JavaScript-библиотеки это просто строки из листинга 5.1, как показано в листинге 5.9.

Листинг 5.9 ❖ Частичный шаблон для загрузки JavaScript-библиотеки (app/views/layouts/_shim.html.erb)

```
<!--[if lt IE 9]>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
  </script>
<![endif]-->
```

[1] Многие Rails-разработчики используют каталог `shared` для частичных шаблонов, совместно используемых разными представлениями. Я предпочитаю хранить в папке `shared` «служебные» шаблоны, которые используются в нескольких страницах, а шаблоны, необходимые буквально в каждой странице (такие как части макета сайта), – хранить в каталоге `layouts`. (Мы создадим каталог `shared` в главе 7.) Это кажется мне более логичным, впрочем, хранение всех шаблонов в общей папке `shared` ничуть не хуже.

Аналогично выполняются перенос заголовка в частичный шаблон (см. листинг 5.10) и его вставка в макет вызовом `render`. (Как это часто бывает с частичными шаблонами, файл придется создать вручную, в текстовом редакторе.)

Листинг 5.10 ❖ Частичный шаблон с заголовком сайта (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>
```

Теперь, узнав, как создавать частичные шаблоны, добавим в каждую страницу сайта подвал, чтобы он шел вместе с заголовком. Вы уже наверняка догадались, что мы назовем его `_footer.html.erb` и поместим в каталог `layouts` (листинг 5.11)[1].

Листинг 5.11 ❖ Частичный шаблон с подвалом сайта (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>
```

Как и в заголовке, внутренние ссылки на страницы About и Contact в подвале также создаются с помощью `link_to`, а адреса URL заглушены символом `'#'` до лучших времен. (Тег `footer`, как и `header`, впервые появился в HTML5.)

Включается частичный шаблон подвала в макет так же, как и все остальные (листинг 5.12).

---

[1] Вы, возможно, удивлены, что мы используем и тег `footer`, и класс `.footer`. Дело в том, что тег имеет понятное значение для читателя, а класс необходим для Bootstrap. Вместо `footer` вполне можно было бы использовать тег `div`.

Листинг 5.12 ❖ Макет сайта с подключенным шаблоном подвала (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 %>
      <%= render 'layouts/footer' %>
    </div>
  </body>
</html>
```

Конечно же, без дополнительного оформления (листинг 5.13) подвал будет выглядеть не очень привлекательно. Результаты добавления стилей показаны на рис. 5.7.

Листинг 5.13 ❖ Дополнительные правила CSS для оформления подвала сайта (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* подвал */

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;
}

footer ul li {
  float: left;
  margin-left: 15px;
}
```

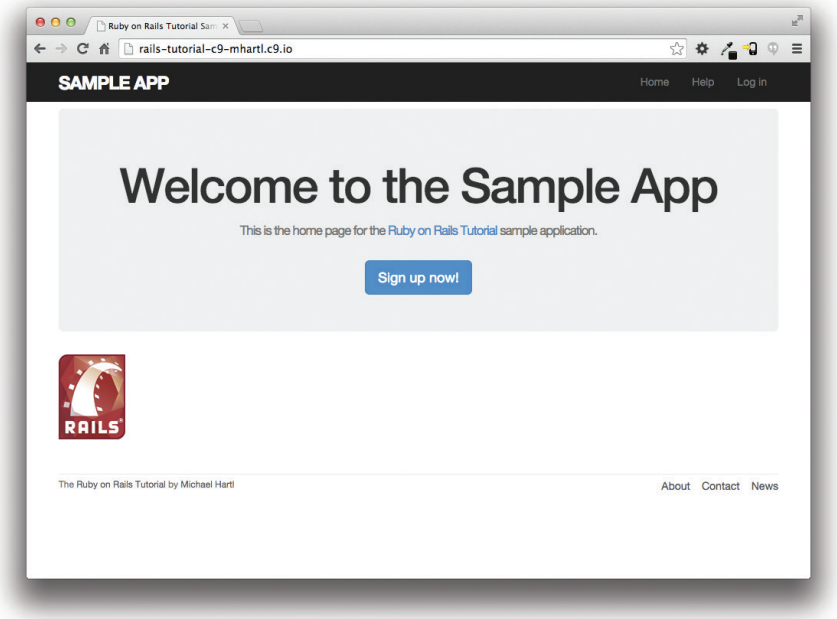


Рис. 5.7 ❖ Страница Home с подвалом

5.2. Sass и конвейер ресурсов

Одним из наиболее значительных нововведений в последних версиях Rails является *конвейер ресурсов* (asset pipeline), который значительно упрощает создание статических ресурсов и управление ими. К числу ресурсов относятся, например, CSS, JavaScript и изображения. В этом разделе сначала дается общий обзор конвейера ресурсов, а затем демонстрируются приемы использования Sass – удивительного инструмента создания CSS.

5.2.1. Конвейер ресурсов

Конвейер ресурсов включает множество внутренних изменений в Rails, но с точки зрения типичного Rails-разработчика важно знать и понимать три основных

аспекта: каталоги ресурсов, файлы манифестов и препроцессоры[1]. Давайте рассмотрим каждый из них по очереди.

Каталоги ресурсов

В версиях Rails 3.0 и ниже статические ресурсы хранились в каталоге `public/`:

- `public/stylesheets`;
- `public/javascripts`;
- `public/images`.

Файлы в этих каталогах (даже в версиях выше 3.0) автоматически возвращаются в ответ на запросы по таким адресам, как `http://www.example.com/stylesheets`, и т. д.

В более поздних версиях определены *три* канонических каталога для статических ресурсов, каждый со своим собственным назначением:

- `app/assets`: ресурсы для данного приложения;
- `lib/assets`: ресурсы для библиотек, написанных вашей командой разработчиков;
- `vendor/assets`: ресурсы сторонних поставщиков.

Как можно догадаться, каждый из этих каталогов имеет подкаталоги для каждого класса ресурсов, например:

```
$ ls app/assets/
images/  javascripts/  stylesheets/
```

Теперь нетрудно понять причины выбора места хранения файла CSS, созданного в разделе 5.1.2: `custom.css.scss` принадлежит учебному приложению, поэтому он помещен в `app/assets/stylesheets`.

Файлы манифестов

После сохранения ресурсов в предназначенных им каталогах можно создать *файлы манифестов* (декларации), чтобы сообщить Rails (через гем `Sprockets`), как объединить их и сформировать в один файл. (Это относится к CSS и JavaScript, но не к изображениям.) Для примера заглянем в начальный файл манифеста для таблиц стилей приложения (листинг 5.14).

Листинг 5.14 ❖ Файл манифеста для таблиц стилей приложения (app/assets/stylesheets/application.css)

```
/*
 * Это файл манифеста, он будет скомпилирован в файл
 * application.css, включающий все нижеперечисленные файлы.
 *
 * Все файлы CSS и SCSS в этом каталоге, а также в lib/assets/stylesheets,
```

[1] Структура этого раздела опирается на замечательную статью «The Rails 3 Asset Pipeline in (about) 5 Minutes» Майкла Ерасмуса (Michael Erasmus). Дополнительные подробности см. в статье «Asset Pipeline» в руководстве к Rails (http://guides.rubyonrails.org/asset_pipeline.html).

```
 * vendor/assets/stylesheets или vendor/assets/stylesheets, если они есть,
 * могут быть указаны здесь с относительными путями.
 *
 * Вы можете добавлять сюда свои стили для приложения. Они появятся в конце
 * скомпилированного файла и потому будут иметь высший приоритет перед любыми другими
 * стилями в других файлах CSS/SCSS в этом каталоге. Как правило, для каждого нового
 * раздела стилей  лучше создать новый файл.
 *
 *= require_tree .
 *= require_self
 */
```

Ключевыми строками на самом деле здесь являются CSS-комментарии, именно их использует Sprockets для включения нужных файлов:

```
/*
 .
 .
 .
 *= require_tree .
 *= require_self
*/
```

Строка

```
*= require_tree .
```

гарантирует включение всех файлов CSS из каталога `app/assets/stylesheets` (и вложенных подкаталогов) в таблицу стилей CSS приложения. Строка

```
*= require_self
```

указывает, где в последовательности загрузки будет включен код самого файла `application.css`.

Rails включает вполне адекватный файл манифеста по умолчанию, поэтому в данной книге нам не придется вносить в них изменения, но при необходимости вы сможете получить более подробную информацию об этом в статье «Asset Pipeline» в руководстве к Rails (http://guides.rubyonrails.org/asset_pipeline.html).

Препроцессоры

После сборки ресурсов Rails подготавливает их для добавления в макет сайта, прогоняя через несколько препроцессоров и комбинируя их с помощью файлов манифестов. Выбор препроцессора определяется расширением файла; чаще всего на практике используются: `.scss` – для Sass, `.coffee` – для CoffeeScript и `.erb` – для встроенного Ruby (ERb). Впервые мы говорили о ERb в разделе 3.4.3, о Sass – в разделе 5.2.2. В этой книге нам не понадобится CoffeeScript, но если вам интересно узнать об этом элегантном маленьком языке, который компилируется в JavaScript, советую начать с видеоурока об основах CoffeeScript[1].

[1] http://railscasts.com/episodes/267-coffeescript-basics.

Препроцессоры можно объединять в цепочки, например файл

```
foobar.js.coffee
```

будет обработан препроцессором CoffeeScript, а файл

```
foobar.js.erb.coffee
```

будет обработан препроцессорами CoffeeScript и ERb (справа налево, то есть CoffeeScript сначала).

Эффективность на этапе эксплуатации

Одно из самых замечательных свойств конвейера ресурсов заключается в автоматической оптимизации ресурсов для максимальной эффективности в эксплуатационном окружении. Традиционный подход к организации CSS и JavaScript подразумевает распределение функциональности по нескольким файлам и использование приятного глазу форматирования (с большим количеством отступов). Это удобно для программистов, но неэффективно в режиме эксплуатации. В частности, включение множества крупноразмерных файлов может серьезно увеличить время загрузки страниц, а это один из наиболее значимых факторов, влияющих на впечатления, получаемые пользователями от сайта. Конвейер ресурсов избавляет от необходимости выбирать между скоростью и комфортом: можно работать со множеством отформатированных файлов в среде разработки, а затем использовать конвейер ресурсов с целью создания эффективных файлов для эксплуатационного окружения. В частности, конвейер объединяет все таблицы стилей приложения в один CSS-файл (`application.css`), весь код на JavaScript – в один JavaScript-файл (`application.js`), а затем минимизирует их, убирая ненужные пробелы и отступы, увеличивающие размер файла. В результате мы получаем лучшее для обоих миров: комфорт и удобство при разработке, скорость и эффективность в период эксплуатации.

5.2.2. Синтаксически безупречные таблицы стилей

Sass – это язык определения таблиц стилей, который во многом улучшает CSS. В этом разделе мы познакомимся с двумя наиболее важными улучшениями: *вложениями* и *переменными*. (Третье, *примеси*, представлено в разделе 7.1.1.)

Как отмечалось в разделе 5.1.2, Sass поддерживает формат, называемый SCSS (обозначается расширением файлов `.scss`), который является надстройкой над CSS; то есть SCSS только *добавляет* возможности в CSS, а не определяет новый синтаксис[1]. Это означает, что любой допустимый CSS-файл будет допустимым SCSS-файлом, что удобно для проектов с уже имеющимися правилами оформления. В нашем случае мы использовали SCSS с самого начала для совместимости

[1] Более старый формат `.sass`, также поддерживаемый Sass, определяет новый язык, более компактный (и с меньшим числом фигурных скобок), менее удобный для существующих проектов и более сложный в изучении для тех, кто уже знаком с CSS.

с Bootstrap. Поскольку конвейер ресурсов в Rails автоматически использует Sass для обработки файлов с расширением .scss, файл custom.css.scss будет обработан препроцессором Sass перед упаковкой для доставки браузеру.

Вложения

Часто в таблицах стилей предусматриваются правила, применяемые к вложенным элементам. Например, в листинге 5.5 есть правило, применяемое к .center и к .center h1:

```
.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

В Sass эти два правила можно заменить одним:

```
.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}
```

Вложенное правило h1 автоматически наследует контекст .center.

Здесь есть второй кандидат на вложение, который требует несколько иного синтаксиса. В листинге 5.7 присутствует код:

```
#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;
}
```

Идентификатор логотипа #logo используется дважды, в первый раз сам по себе, второй раз – с атрибутом hover (он отвечает за внешний вид при наведении указателя мыши). Чтобы вложить второе правило, необходимо сослаться на родительский элемент #logo; в SCSS это делается с помощью символа амперсанда &:

```
#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 заменяет `&:hover` на `#logo:hover` при преобразовании SCSS в CSS.

Оба этих приема вложения элементов применяются в правилах CSS оформления подвала в листинге 5.13, который может быть трансформирован вот так:

```
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;
    }
  }
}
```

Преобразование листинга 5.13 вручную – хорошее упражнение, и вам стоит убедиться, что стили по-прежнему работают правильно после этого.

Переменные

Sass позволяет определять *переменные* для устранения дублирования и придания коду дополнительной выразительности. Например, взгляните на листинги 5.6 и 5.13, в них есть повторные упоминания одного и того же цвета:

```
h2 {
  .
  .
  .
  color: #777;
}
.
.
.
footer {
  .
  .
  .
  color: #777;
}
```

В данном случае #777 – это светло-серый, и ему можно дать имя, определив переменную:

```
$light-gray: #777;
```

Это позволит переписать SCSS вот так:

```
$light-gray: #777;
.
.
.
h2 {
  .
  .
  .
  color: $light-gray;
}
.
.
.
footer {
  .
  .
  .
  color: $light-gray;
}
```

Поскольку имена переменных, такие как $light-gray, являются более понятными, чем #777, часто бывает полезным определить переменные даже для неповторяющихся значений. Действительно, фреймворк Bootstrap определяет большое количество переменных со значениями цвета, с которыми можно ознакомиться в документации к Bootstrap[1]. На этой странице переменные определяются с по-

[1] http://getbootstrap.com/customize/#less-variables.

мощью языка Less, а не Sass, но гем bootstrap-sass преобразует их в Sass-эквиваленты. Угадать соответствие несложно; там, где в языке Less ставится значок @, в Sass используется знак доллара $. Глядя на страницу с переменными, можно заметить, что переменная с определением светло-серого цвета уже существует:

```
@gray-light: #777;
```

То есть благодаря гему bootstrap-sass должна существовать соответствующая SCSS-переменная $gray-light. Ее можно использовать вместо нашей переменной $light-gray:

```
h2 {
  .
  .
  .
  color: $gray-light;
}
.
.
.
footer {
  .
  .
  .
  color: $gray-light;
}
```

В результате применения вложений и переменных ко всему файлу SCSS получается файл, представленный в листинге 5.15. В нем использованы Sass-переменные (из Bootstrap) и встроенные названия цветов (например, white для #fff). Обратите внимание, насколько лучше стали выглядеть правила для тега footer.

Листинг 5.15 ❖ Файл SCSS, преобразованный с помощью вложений и переменных (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* примеси, переменные и пр.*/

$gray-medium-light: #eaeaea;

/* общие */

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
```

```
  resize: vertical;
}

.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}

/* правила для текста */

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;
}

/* заголовок */

#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;
  }
}
```

```
/* подвал */

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 имеет гораздо более широкие возможности, позволяющие упрощать таблицы стилей, но в листинге 5.15 используются лишь наиболее важные. Дополнительную информацию об этом ищите на сайте Sass[1].

5.3. Ссылки в макете

Теперь, после добавления оформления сайта, пришло время приступить к заполнению ссылок, которые мы заглушили символом `'#'`. Конечно, можно просто жестко прописать их:

```
<a href="/static_pages/about">About</a>
```

но это не Rails Way™. При этом было бы здорово, если бы URL страницы About имел вид `/about`, а не `/static_pages/about`. Более того, в Rails традиционно используются *именованные маршруты*, которые предполагают примерно такой код:

```
<%= link_to "About", about_path %>
```

Этот код имеет более прозрачный смысл и становится более гибким, поскольку позволяет изменить определение `about_path`, и URL изменяется везде, где используется `about_path`.

---

[1] http://sass-scss.ru/.

Полный список запланированных ссылок представлен в табл. 5.1 вместе с соответствующими адресами URL и маршрутами. О первом маршруте мы уже позаботились в разделе 3.4.4, а остальные, кроме последнего, реализуем до конца этой главы. (Определение последнего будет дано в главе 8.)

Таблица 5.1 ❖ Маршруты и соответствующие им адреса URL для ссылок на сайте

Страница	URL	Именованный маршрут
Home	/	`root_path`
About	/about	`about_path`
Help	/help	`help_path`
Contact	/contact	`contact_path`
Signup	/signup	`signup_path`
Login	/login	`login_path`

5.3.1. Страница Contact

Прежде чем двигаться дальше, добавим страницу Contact, которая была оставлена в качестве упражнения в главе 3. Тесты представлены в листинге 5.16, они просто следуют той же модели, что и тесты в листинге 3.22.

Листинг 5.16 ❖ Тесты для страницы Contact **КРАСНЫЙ**
(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
```

Сейчас набор тестов в листинге 5.16 должен давать **КРАСНЫЙ** цвет:

Листинг 5.17 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Процедура добавления страницы Contact в точности повторяет процедуру добавления страницы About, как описывалось в разделе 3.3: сначала обновим маршруты (листинг 5.18), затем добавим в контроллер Static Pages метод `contact` (листинг 5.19) и, наконец, создадим представление Contact (листинг 5.20).

Листинг 5.18 ❖ Добавление маршрута для страницы Contact **КРАСНЫЙ** (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
```

Листинг 5.19 ❖ Добавление метода действия для страницы Contact **КРАСНЫЙ** (app/controllers/static_pages_controller.rb)

```
class StaticPagesController < ApplicationController
  .
  .
  .
  def contact
  end
end
```

Листинг 5.20 ❖ Представление для страницы Contact **ЗЕЛЕНЫЙ** (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">contactpage</a>.
</p>
```

Убедимся, что набор тестов возвращает **ЗЕЛЕНЫЙ** цвет:

Листинг 5.21 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

5.3.2. Маршруты в Rails

Чтобы добавить именованные маршруты к статическим страницам учебного приложения, необходимо отредактировать файл маршрутов `config/routes.rb`, с помощью которого Rails определяет соответствия между именами маршрутов и адресами URL. Начнем с обзора маршрута для страницы Home (он был опре-

делен в разделе 3.4.4), который представляет особый случай, а затем определим набор маршрутов для всех остальных статических страниц.

Мы уже видели три примера определения корневого маршрута, начиная с

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

в первом приложении (листинг 1.10), затем

```
root 'users#index'
```

в мини-приложении (листинг 2.3), и наконец

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

в учебном приложении (листинг 3.37). В каждом случае метод `root` передает запрос по адресу / в контроллер и метод по нашему выбору. Определение корневого маршрута таким способом имеет еще один важный эффект – создается именованный маршрут, к которому можно обращаться по имени, а не через необработанный адрес URL. В данном случае это маршруты `root_path` и `root_url`, они отличаются наличием во втором случае полного адреса URL:

```
root_path -> '/'
root_url -> 'http://www.example.com/'
```

В этой книге мы последуем общему соглашению об использовании формы `_path` во всех случаях, кроме переадресации, когда будет использоваться форма `_url`. (Это связано с тем, что стандарт HTTP технически требует полный адрес URL для переадресации, хотя в большинстве браузеров поддерживаются оба способа.)

Чтобы определить именованные маршруты для страниц Help, About и Contact, нужно изменить правила `get` из листинга 5.18, переписав строки, имеющие вид:

```
get 'static_pages/help'
```

вот так:

```
get 'help' => 'static_pages#help'
```

Второй вариант передает запрос `GET` к адресу URL `/help` в метод `help` контроллера StaticPages, то есть вместо громоздкого `/static_pages/help` можно использовать более краткий адрес URL `/help`. Как и в случае с корневым маршрутом, в результате создаются два именованных маршрута, `help_path` и `help_url`:

```
help_path -> '/help'
help_url -> 'http://www.example.com/help'
```

Изменив подобным образом маршруты для остальных статических страниц из листинга 5.18, получим определения маршрутов, как показано в листинге 5.22.

Листинг 5.22 ❖ Маршруты к статическим страницам (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. Использование именованных маршрутов

После определения маршрутов, как показано в листинге 5.22, их можно использовать в макете сайта. Это просто означает заполнение второго аргумента в функции `link_to` правильным именованным маршрутом. Например, превратим

```
<%= link_to "About", '#' %>
```

в

```
<%= link_to "About", about_path %>
```

и т. д.

Начнем с шаблона заголовка, `_header.html.erb` (листинг 5.23), содержащего ссылки на страницы Home и Help. Попутно свяжем логотип со страницей Home, последовав общепринятому соглашению.

Листинг 5.23 ❖ Шаблон заголовка со ссылками (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>
```

Именованный маршрут для ссылки **Login** мы создадим в главе 8, поэтому оставим пока `'#'`.

В шаблоне подвала, `_footer.html.erb`, тоже имеются ссылки – на страницы About и Contact (листинг 5.24).

Листинг 5.24 ❖ Шаблон подвала со ссылками (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>
```

Теперь в нашем макете есть ссылки на все статические страницы, созданные в главе 3; например, маршрут /about ведет к странице About (рис. 5.8).

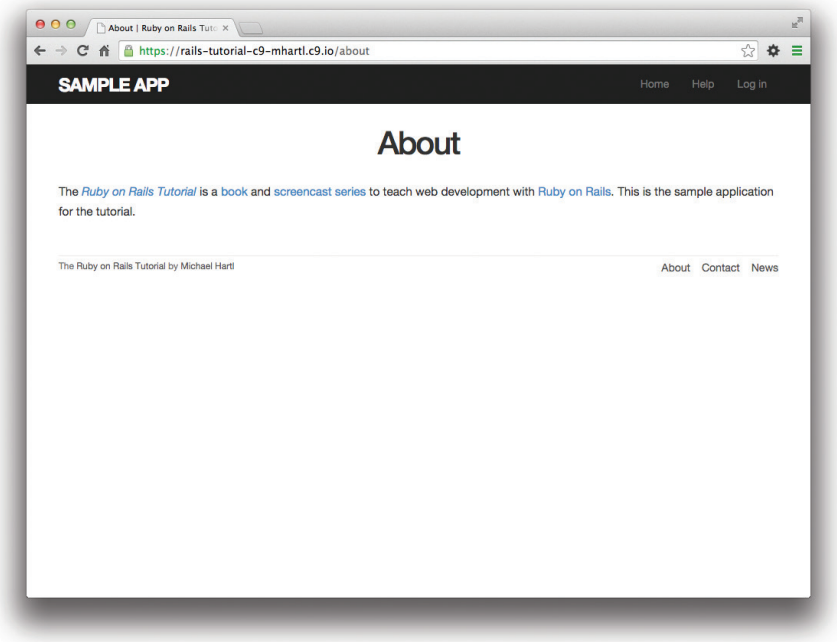


Рис. 5.8 ❖ Маршрут /about ведет к странице About

5.3.4. Тесты для проверки ссылок в макете

Теперь, после заполнения нескольких ссылок в макете, было бы неплохо протестировать их. Можно, конечно, сделать это вручную через браузер, сначала посетив домашнюю страницу, а затем «прощелкав» все ссылки, но этот способ быстро становится утомительным. Вместо этого мы сымитируем перечисленные действия с помощью *интеграционных тестов*, которые позволят проверить поведение приложения в комплексе. Начнем с теста, который назовем site_layout:

```
$ rails generate integration_test site_layout
      invoke test_unit
      create   test/integration/site_layout_test.rb
```

Обратите внимание, что Rails-генератор автоматически добавляет _test в конец имени файла с тестами. Наш план тестирования ссылок макета подразумевает проверку HTML-структуры сайта:

1) выполнить переход по корневому маршруту (страница Home);
2) убедиться, что возвращается правильная страница;
3) проверить корректность ссылок на страницы Home, Help, About и Contact.

В листинге 5.25 показано, как с помощью интеграционных тестов Rails преобразовать эти шаги в код, начиная с метода `assert_template`, проверяющего шаблон, на основе которого создается страница Home[1].

Листинг 5.25 ❖ Тесты для проверки ссылок **ЗЕЛЕНЫЙ**
(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
```

В листинге 5.25 использованы более продвинутые возможности метода `assert_select` в сравнении с показанными в листингах 3.22 и 5.16. В данном случае используется синтаксис, позволяющий проверить наличие определенной ссылки, указав имя тега `a` и атрибут `href`, как в строке:

```
assert_select "a[href=?]", about_path
```

Rails автоматически вставляет значение `about_path` вместо знака вопроса (экранируя при необходимости любые специальные символы), обеспечивая проверку наличия HTML-тега вида

```
<a href="/about">...</a>
```

Обратите внимание: утверждение для проверки корневого маршрута проверяет наличие *двух* ссылок (одна – для логотипа, и одна – для элемента в меню навигации):

```
assert_select "a[href=?]", root_path, count: 2
```

Таким способом проверяется присутствие обеих ссылок на страницу Home, определенных в листинге 5.23.

---

[1] Некоторые разработчики настаивают, что каждый отдельный тест не должен содержать несколько утверждений. Я считаю это неоправданно сложным и вызывающим дополнительные издержки, если перед каждым тестом требуется выполнить общие настройки. Кроме того, хорошо написанные тесты рассказывают связную историю, и разбиение на куски нарушает повествование. Поэтому я предпочитаю использовать множественные утверждения, полагаясь на то, что Ruby (с помощью MiniTest) укажет мне точную строку с утверждением, потерпевшим неудачу.

Таблица 5.2 ❖ Примеры использования assert_select

Код	Разметка HTML
`assert_select "div"`	`<div>foobar</div>`
`assert_select "div", "foobar"`	`<div>foobar</div>`
`assert_select "div.nav"`	`<div class="nav">foobar</div>`
`assert_select "div#profile"`	`<div id="profile">foobar</div>`
`assert_select "div[name=yo]"`	`<div name="yo">hey</div>`
`assert_select "a[href=?]", '/', count: 1`	`<a href="/">foo</a>`
`assert_select "a[href=?]", '/', text: "foo"`	`<a href="/">foo</a>`

В табл. 5.2 приводится еще несколько примеров использования `assert_select`. Несмотря на то что `assert_select` является очень мощным и гибким инструментом (он поддерживает гораздо больше вариантов использования, чем показано здесь), опыт подсказывает, что разумнее использовать более простой подход и проверять только элементы HTML (например, ссылки в макете сайта), которые вряд ли изменятся со временем.

Чтобы выполнить новые тесты из листинга 5.25, нужно запустить только интеграционное тестирование, выполнив Rake-задачу:

Листинг 5.26 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test:integration
```

Если все пошло хорошо, запустите полный набор тестов, чтобы убедиться, что он **ЗЕЛЕНЫЙ**:

Листинг 5.27 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

После добавления интеграционных тестов для ссылок в макете мы сразу же заметим любые регрессии с помощью набора тестов.

5.4. Регистрация пользователей: первый шаг

Кульминацией нашей работы над макетом и маршрутами в этом разделе станет маршрут к странице регистрации, что означает создание второго контроллера. Это первый важный шаг в реализации механизма регистрации новых пользователей на нашем сайте; следующий шаг, определение модели пользователя, мы сделаем в главе 6 и закончим работу в главе 7.

5.4.1. Контроллер Users

Первый свой контроллер, Static Pages, мы создали еще в разделе 3.2. Теперь пришло время создать второй – контроллер Users. Как и прежде, используем `generate`, чтобы создать простейший контроллер, отвечающий нашим текущим потребностям, а именно с одной страницей-заглушкой для регистрации новых пользователей. Следуя традиционной архитектуре REST, предпочитаемой Rails, назовем

метод создания новых пользователей new, его можно создать автоматически, если передать в аргументе команде generate. Результат показан в листинге 5.28.

Листинг 5.28 ❖ Создание контроллера Users (с методом действия 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
```

Как и требовалось, мы получили контроллер Users с методом действия new (листинг 5.30) и заглушку представления (листинг 5.31). Дополнительно был создан минимальный набор тестов для страницы создания нового пользователя (листинг 5.32), который должен выполняться успешно:

Листинг 5.29 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Листинг 5.30 ❖ Начальный контроллер Users с методом действия new (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController

  def new
  end
end
```

Листинг 5.31 ❖ Начальное представление new для контроллера Users (app/views/users/new.html.erb)

```
<h1>Users#new</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

Листинг 5.32 ❖ Тесты для страницы создания нового пользователя **ЗЕЛЕНЫЙ** (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. Адрес URL страницы регистрации

У нас появилась действующая страничка создания новых пользователей с адресом /users/new, но из табл. 5.1 мы помним, что вместо этого адреса URL должен использоваться адрес /signup. Последуем примерам из листинга 5.22 и добавим правило get '/signup' для адреса к странице регистрации нового пользователя, как показано в листинге 5.33.

Листинг 5.33 ❖ Маршрут для страницы регистрации (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
```

Теперь используем новый именованный маршрут, чтобы добавить ссылку в определение кнопки на странице Home. Как и в случае с другими маршрутами, get 'signup' автоматически возвращает именованный маршрут signup_path, который мы вставим в листинг 5.34. Добавление тестов для страницы регистрации остается вам в качестве упражнения (раздел 5.6.)

Листинг 5.34 ❖ Связывание кнопки со страницей регистрации (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/' %>
```

Наконец, изменим представление-заглушку для страницы регистрации (листинг 5.35).

Листинг 5.35 ❖ Начальная страница регистрации (заглушка) (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>
```

На этом мы закончили со ссылками и именованными маршрутами, по крайней мере пока не добавим маршрут для входа (глава 8). Получившаяся страница регистрации нового пользователя (с адресом URL /signup) показана на рис. 5.9.

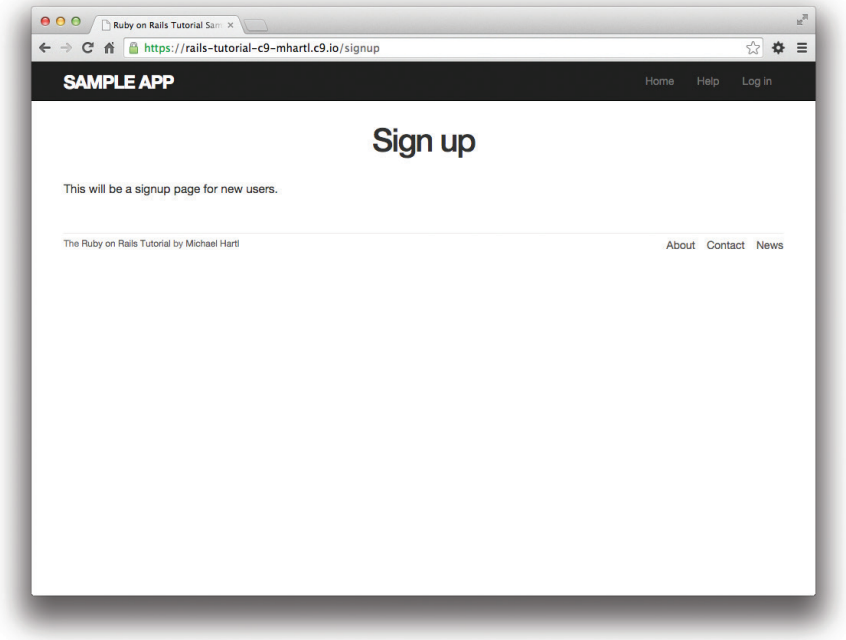


Рис. 5.9 ❖ Новая страница регистрации с адресом /signup

5.5. Заключение

В этой главе мы выковали макет нашего приложения и отполировали маршруты. Остальная часть книги посвящена реализации деталей учебного приложения: сначала мы реализуем поддержку пользователей, которые смогут регистрироваться, входить в свою учетную запись и выходить из нее; затем добавим поддержку микросообщений, и, наконец, возможность следовать за другими пользователями.

Если вы используете Git, слейте произведенные изменения с основной ветвью:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Finish layout and routes"
```

```
$ git checkout master
$ git merge filling-in-layout
```

Затем отправьте в Bitbucket:

```
$ git push
```

И разверните на сайте Heroku:

```
$ git push heroku
```

В результате должно получиться действующее учебное приложение (рис. 5.10).

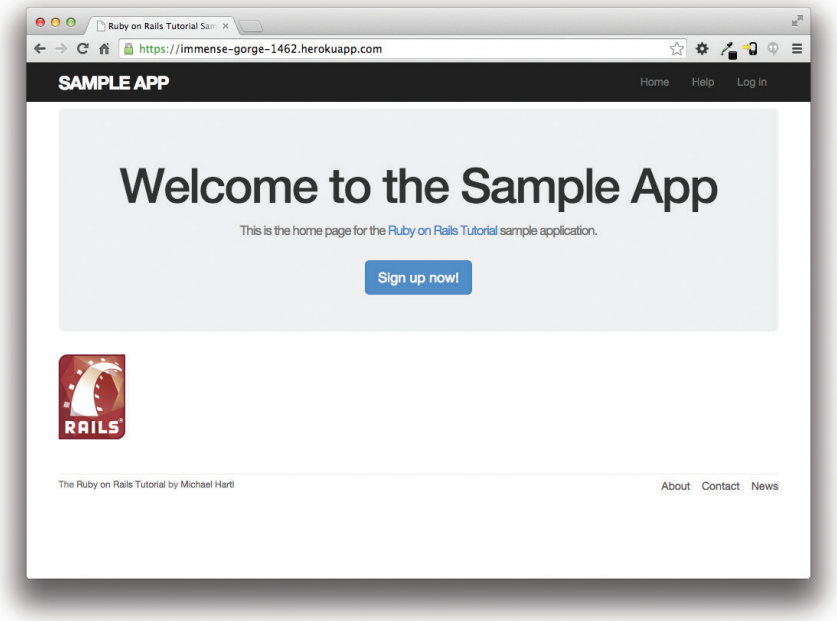


Рис. 5.10 ❖ Учебное приложение на сервере в Интернете

5.5.1. Что мы узнали в этой главе

- Используя HTML5, можно определить макет сайта с логотипом, заголовком, подвалом и основным содержимым.
- Частичные шаблоны в Rails помогают удобно разместить разметку в отдельных файлах.
- CSS позволяет применить стили к макету сайта на основе CSS-классов и идентификаторов.
- Фреймворк Bootstrap ускоряет создание сайта с привлекательным дизайном.
- Sass и механизм ресурсов позволяют избежать дублирования в CSS и произвести эффективную упаковку результатов для эксплуатационного окружения.

- Rails дает возможность определить собственные правила маршрутизации добавлением именованных маршрутов.
- Интеграционные тесты эффективно имитируют переход от страницы к странице в браузере.

5.6. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Как было предложено в разделе 5.2.2, преобразуйте правила CSS для подвала, представленные в листинге 5.13, в правила SCSS, представленные в листинге 5.15.
2. Добавьте в интеграционные тесты (листинг 5.25) код, проверяющий ссылку на страницу регистрации с использованием метода `get`, и убедитесь, что заголовок полученной страницы является верным.
3. В тестах удобно использовать вспомогательную функцию `full_title`, подключив `ApplicationHelper`, как показано в листинге 5.36. После этого можно проверить правильность заголовка с помощью кода из листинга 5.37 (расширенное решение предыдущего упражнения). Это решение весьма хрупкое, хотя бы потому, что теперь любая опечатка в базовом заголовке (например, «Ruby on Rails Tutoial») не будет обнаружена тестами. Решите эту проблему, создав тест со вспомогательной функцией `full_title`, для чего понадобится создать файл для тестирования приложения, а затем заменить метки `FILL_IN` в листинге 5.38. (В нем использован метод `assert_equal`, проверяющий равенство своих аргументов с помощью оператора `==`.)

Листинг 5.36 ❖ Включение ApplicationHelper в тесты (test/test_helper.rb)

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all
  include ApplicationHelper
  .
  .
  .
end
```

Листинг 5.37 ❖ Использование вспомогательной функции full_title в тесте **ЗЕЛЕНЫЙ** (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
```

Листинг 5.38 ❖ Непосредственный тест вспомогательной функции 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
```

Глава 6

Моделирование пользователей

Главу 5 мы закончили созданием страницы-заглушки для регистрации пользователей (раздел 5.4). В следующих пяти главах мы выполним обещание, скрытое в этой только зарождающейся странице регистрации. В этой главе мы сделаем первый важный шаг – создадим *модель данных*, описывающую пользователей нашего сайта, и реализуем механизм хранения этих данных. В главе 7 мы реализуем саму процедуру регистрации и создадим страницу профиля пользователя. После этого разрешим им осуществлять вход и выход (глава 8), а в главе 9 (раздел 9.2.1) узнаем, как защитить страницы от несанкционированного доступа. Наконец, в главе 10 добавим активацию аккаунта (подтверждающую действительность адреса электронной почты) и сброс пароля. В главах с 6 по 10 показана разработка полноценной системы входа и аутентификации. Как вы, возможно, знаете, для Rails существует множество готовых решений в этой области; блок 6.1 поясняет, почему, по крайней мере в первый раз, развертывание собственной системы является лучшей идеей.

Блок 6.1 ❖ Прикручивание собственной системы аутентификации

Фактически всем веб-приложениям в настоящее время требуется какая-либо система входа и аутентификации. Неудивительно, что большинство веб-фреймворков реализует множество вариантов подобных систем, и Rails не исключение. В качестве примеров систем аутентификации и авторизации можно привести: Clearance[1], Authlogic[2], Devise[3] и CanCan[4] (так же как решения, не связанные напрямую с Rails, построенные на основе OpenID[5] или OAuth[6]). Возникает резонный вопрос: зачем

[1] https://github.com/thoughtbot/clearance.

[2] https://github.com/binarylogic/authlogic.

[3] https://github.com/plataformatec/devise.

[4] http://railscasts.com/episodes/192-authorization-with-cancan.

[5] https://ru.wikipedia.org/wiki/OpenID.

[6] https://ru.wikipedia.org/wiki/OAuth.

изобретать велосипед? Почему бы просто не использовать готовое решение, вместо того чтобы прикручивать свое?
С одной стороны, практика показывает, что на большинстве сайтов аутентификация требует серьезной настройки, а модификация стороннего продукта – зачастую даже более сложная задача, чем создание собственной системы с нуля. К тому же готовые решения – это «черные ящики» с неясным внутренним устройством; когда вы пишете свою систему, у вас гораздо больше шансов разобраться в ней. Кроме того, последние дополнения к Rails (раздел 6.3) существенно упростили создание собственных систем аутентификации. Наконец, если вы *все же решите* использовать стороннюю систему, вам будет гораздо проще в ней разобраться и настроить, если у вас уже будет опыт создания собственной такой системы.

6.1. Модель User

Конечная цель следующих трех глав – создание страницы регистрации для нашего сайта (ее макет показан на рис. 6.1), но на данном этапе мы не готовы получать информацию о новых пользователях, ее попросту негде хранить. Поэтому первый шаг к регистрации пользователей – создание структуры данных для получения и хранения информации о них.

Рис. 6.1 ❖ Макет страницы регистрации пользователей

Структуру, описывающую модель данных, в Rails называют, что достаточно естественно, *моделью* (буква M в аббревиатуре MVC, как описывается в разделе 1.3.3). По умолчанию для долгосрочного хранения информации в Rails ис-

пользуется *база данных*, а для взаимодействий с ней применяется библиотека *Active Record*[1]. Библиотека Active Record включает массу методов для создания, хранения и поиска объектов данных, без использования языка структурированных запросов (SQL)[2], применяемого в реляционных базах данных[3]. Кроме того, в Rails есть функции, называемые *миграциями*, которые позволяют писать определения данных на чистом Ruby, без необходимости изучать SQL-язык определения данных (Data Definition Language, DDL). Как результат Rails почти полностью изолирует разработчиков от деталей хранения данных. В этой книге благодаря использованию SQLite (в окружении для разработки) и PostgreSQL (в эксплуатационном окружении Heroku, см. раздел 1.5) мы разовьем эту тему до точки, где нам едва ли когда-нибудь придется задумываться о том, как Rails хранит данные.

Как обычно, если для управления версиями вы используете Git, сейчас самое время создать новую ветвь, в которой будет разрабатываться модель пользователей:

```
$ git checkout master
$ git checkout -b modeling-users
```

6.1.1. Миграции базы данных

В классе `User`, созданном нами в разделе 4.4.5, уже имеются атрибуты `name` и `email`. Он послужил нам полезным примером, но ему не хватало крайне необходимого механизма *сохранения*: когда мы создавали объект `User` в консоли Rails, он исчезал сразу после выхода из нее. В этом разделе мы создадим модель пользователей, объекты которой не будут исчезать так легко.

Так же как в разделе 4.4.5, мы начнем с модели пользователя, содержащей два атрибута: `name` и `email`, последний из которых будет служить уникальным именем пользователя[4]. (Атрибут для хранения пароля мы добавим в разделе 6.3.) Листинг 4.13 демонстрирует определение модели с помощью Ruby-метода `attr_accessor`:

```
class User
  attr_accessor :name, :email
  .
  .
  .
end
```

---

[1] Название происходит от шаблона «ActiveRecord» (https://ru.wikipedia.org/wiki/ActiveRecord), определение которого приводится в книге «Шаблоны корпоративных приложений» Мартина Фаулера (Фаулер Мартин. Шаблоны корпоративных приложений. М.: Вильямс, 2009. ISBN: 978-5-8459-1611-2).

[2] Произносится как «эс-кю-эль».

[3] https://ru.wikipedia.org/wiki/Реляционная_база_данных.

[4] Используя адреса электронной почты в качестве имени пользователя, мы открываем теоретическую возможность связи с пользователями в будущем (глава 10).

Напротив, при моделировании в Rails нет необходимости идентифицировать атрибуты явно. Как отмечалось выше, для хранения данных Rails по умолчанию использует реляционные базы данных с *таблицами*, состоящими из *строк* и *столбцов*. Например, для хранения пользователей с именами и адресами электронной почты создадим таблицу `users` со столбцами `name` и `email` (где каждая строка соответствует одному пользователю). Пример такой таблицы показан на рис. 6.2. Она соответствует модели данных на рис. 6.3 (это лишь первый вариант, полная модель показана на рис. 6.4). Дав столбцам имена `name` и `email`, мы дали Active Record возможность автоматически определить атрибуты объекта `User`.

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

Рис. 6.2 ❖ Пример данных в таблице users

users	
id	integer
name	string
email	string

Рис. 6.3 ❖ Первый вариант модели данных User

По аналогии с листингом 5.28 создадим контроллер Users (с методом действия `new`) командой

```
$ rails generate controller Users new
```

Существует похожая команда для создания моделей: `generate model`. В листинге 6.1 показано, как с ее помощью создать модель User с атрибутами `name` и `email`.

Листинг 6.1 ❖ Создание модели 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
```

(Обратите внимание, что, в отличие от соглашения о выборе множественного числа для имен контроллеров, для именования моделей используется единственное число: контроллер *Users*, но модель *User*.) Дополнительными параметрами `name:string` и `email:string` мы требуем от Rails создать два атрибута с указанными

именами и типами (в данном случае string). Сравните это с включением имен методов действий в листингах 3.4 и 5.28.

В результате выполнения команды generate создается новый файл *миграции*. Миграции описывают порядок постепенного изменения структуры базы данных и позволяют адаптировать модель данных к изменяющимся требованиям. В отношении модели User миграция создается автоматически, сценарием создания модели; она создает таблицу users с двумя столбцами, name и email, как показано в листинге 6.2. (В разделе 6.2.5 мы увидим, как создать миграцию с нуля.)

Листинг 6.2 ❖ Миграция для модели User (создает таблицу 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
```

Обратите внимание на присутствие *отметки времени* (timestamp) в начале имени файла миграции, соответствующей моменту создания миграции. Когда-то давно файлам давались имена с префиксами в виде увеличивающихся целых чисел, что приводило к конфликтам в группах разработчиков, когда несколько программистов создавали миграции с совпадающими номерами. Использование отметок времени помогает легко избежать подобных коллизий, кроме маловероятных случаев одновременного (с точностью до секунды) создания миграций.

Сама миграция состоит из метода change, определяющего изменения в базе данных. В листинге 6.2 метод change создает таблицу в базе данных вызовом Rails-метода create_table. Метод create_table принимает блок (раздел 4.3.2) с одной переменной, в данном случае t (от «table» (таблица)). Внутри блока с помощью объекта t метод create_table создает столбцы name и email, оба с типом string[1]. Для таблицы выбрано имя во множественном числе (users), в отличие от единственного числа имени модели (User), что соответствует лингвистическому соглашению, принятому в Rails: модель представляет единственного пользователя, тогда как таблица в базе данных хранит сведения о множестве пользователей. Заключительная строка в блоке, t.timestamps null: false, – это специальная команда, создающая два *волшебных столбца* – created_at и updated_at – с отметками времени, которые автоматически записываются в момент создания и обновления записи с информацией о пользователе. (Знакомство с этими столбцами мы начнем в разделе 6.1.3.)

[1] Совершенно не важно, как объект t с этим справляется; в этом и заключается красота уровней абстракции – нам не нужно об этом знать. Можно просто доверить объекту t сделать свою работу.

Полная модель данных, представленная миграцией из листинга 6.2, показана на рис. 6.4. (Обратите внимание на появившиеся волшебные столбцы, отсутствовавшие в первом варианте на рис. 6.3.)

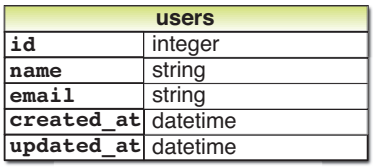


Рис. 6.4 ❖ Модель данных User, созданная миграцией в листинге **6.2**

Запустить миграцию можно командой `rake` (блок 2.1):

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

(Мы уже запускали эту команду в похожем контексте, в разделе 2.2.) При первом запуске `db:migrate` создаст файл `db/development.sqlite3` – базу данных SQLite[1]. Увидеть структуру базы данных можно, открыв файл `development.sqlite3` в обозревателе базы данных для SQLite[2]. (Использующие облачную IDE должны сначала загрузить файл базы данных на локальный диск, как показано на рис. 6.5.) Результат приведен на рис. 6.6; сравните со схемой на рис. 6.4. Возможно, вы заметили отсутствие в миграции столбца `id`. Как рассказывалось в разделе 2.2, он создается автоматически и играет роль уникального идентификатора каждой строки.

Многие миграции (включая все миграции в этой книге) являются *обратимыми*, то есть можно провести «обратную миграцию» и отменить изменения с помощью Rake-задачи `db:rollback`:

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

(В блоке 3.1 демонстрируется еще один прием обращения миграций.) На самом деле эта Rake-задача выполняет команду `drop_table`, удаляющую таблицу `users`. Все это работает потому, что метод `change` знает – `drop_table` является обратной командой по отношению к `create_table`, то есть способ отката миграции легко определяется. Для необратимых миграций, таких как удаление столбца из таблицы, вместо единственного метода `change` нужно определять отдельные методы `up` и `down`. Дополнительную информацию о миграциях ищите в руководстве к Rails (http://rusrails.ru/rails-database-migrations).

Если вы откатили базу данных, выполнив команду выше, проведите миграцию еще раз, прежде чем продолжить:

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

---

[1] http://sqlite.org/ (см. также https://ru.wikipedia.org/wiki/SQLite).

[2] http://sqlitebrowser.org/ (существует также удобное расширение для Mozilla Firefox, обладающее похожими возможностями: https://addons.mozilla.org/ru/firefox/addon/sqlite-manager/).

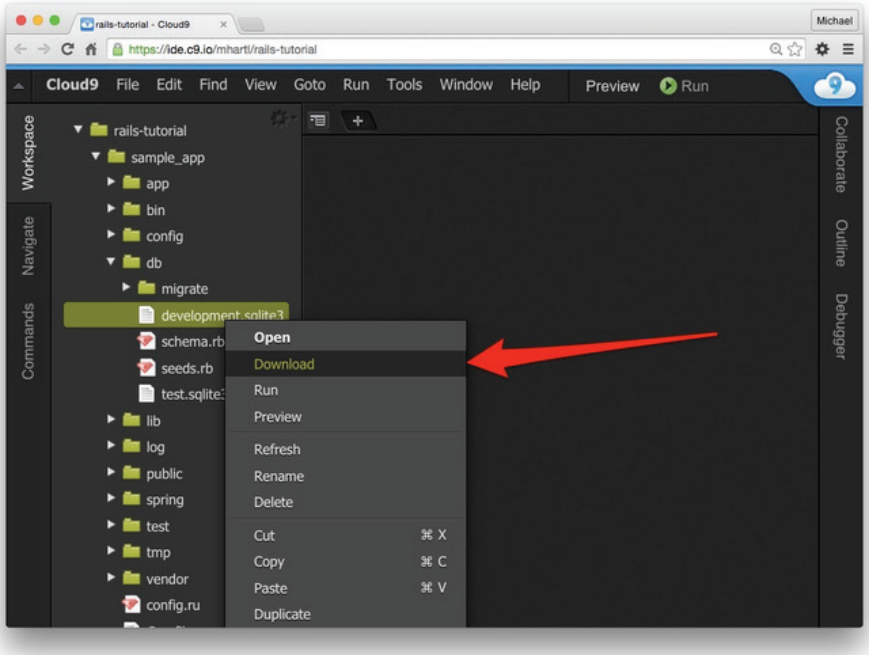


Рис. 6.5 ❖ Загрузка файла из облачной IDE

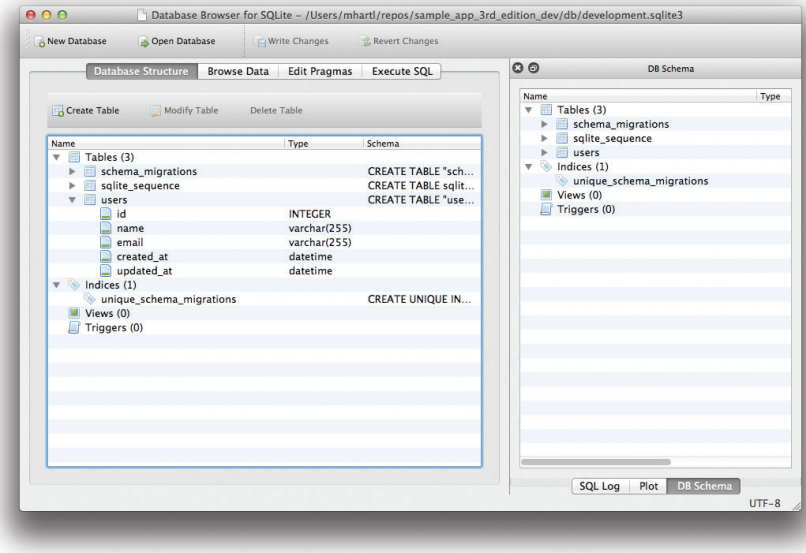


Рис. 6.6 ❖ Обозреватель базы данных для SQLite с новой таблицей users

6.1.2. Файл модели

Создание модели User привело к появлению файла миграции (листинг 6.2), и на рис. 6.6 мы увидели результат применения этой миграции: файл development.sqlite3 обновился, и в нем появилась таблица users со столбцами id, name, email, created_at и updated_at. Команда в листинге 6.1 также создала саму модель. Остальную часть этого раздела мы посвятим ее изучению.

Сначала рассмотрим код модели User в файле user.rb, хранящийся в каталоге app/models/. Он почти пуст (листинг 6.3).

Листинг 6.3 ❖ Совершенно новая модель User(app/models/user.rb)

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

В разделе 4.4.2 отмечалось, что означает синтаксис class User < ActiveRecord::Base: класс User наследует ActiveRecord::Base, поэтому модель User автоматически получает всю функциональность класса ActiveRecord::Base. Это знание не принесет никакой пользы, если не знать, что содержит ActiveRecord::Base, поэтому начнем с конкретных примеров.

6.1.3. Создание объектов User

Роль инструмента изучения модели данных, как и прежде, будет играть Rails-консоль. Так как пока нам не требуется производить какие-либо изменения в базе данных, запустим консоль в изолированном окружении (песочнице):

```
$ rails console --sandbox
Loading development environment in sandbox
Any modifications you make will be rolled back on exit
>>
```

Как указывает сообщение «Все изменения, которые вы сделаете, будут отменены при выходе», консоль, запущенная в песочнице, отменит все изменения в базе данных, внесенные во время сеанса.

В разделе 4.4.5 мы создавали новые объекты вызовом User.new, доступ к которому мы смогли получить только после подключения файла example_user.rb из листинга 4.13. С моделями ситуация иная; как рассказывалось в разделе 4.4.4, Rails-консоль автоматически загружает окружение Rails, включая модели. Благодаря этому мы можем создавать новые объекты, ничего не подключая дополнительно:

```
>> User.new
=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>
```

Так консоль представляет объект класса User.

При вызове без параметров метод User.new возвращает объект с пустыми атрибутами. В разделе 4.4.5 класса User принимал хэш значений для установки атрибутов объекта; такое решение было обусловлено библиотекой Active Record, которая позволяет инициализировать объекты тем же способом:

```
>> 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>
```

Атрибуты имени и адреса электронной почты были установлены, как и ожидалось.

Понятие о *допустимости* очень важно для понимания объектов моделей Active Record. Более полно эту тему мы рассмотрим в разделе 6.2, а пока отметим, что данный объект `User` допустим, и в этом можно убедиться, вызвав логический метод `valid?`.

```
>> user.valid?
true
```

До сих пор мы никак не касались базы данных: `User.new` только создает объект в памяти, а `user.valid?` просто проверяет допустимость этого объекта. Чтобы сохранить объект `User` в базе данных, необходимо вызвать его метод `save`:

```
>> 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
```

Метод `save` возвращает `true`, если сохранение прошло успешно, и `false` в противном случае. (Сейчас все операции сохранения должны выполняться успешно, так как пока нет никаких проверок корректности; в разделе 6.2 мы столкнемся с неудачными попытками сохранения.) Для справки: Rails-консоль также выводит SQL-команду, соответствующую вызову `user.save` (а именно `INSERT INTO "users"...`). В этой книге нам не понадобится использовать SQL[1], поэтому с этого момента я буду опускать обсуждение команд SQL, но вы можете многое узнать, изучая код SQL, соответствующий командам Active Record.

Возможно, вы заметили, что столбцы `created_at`,`updated_at` и `id` в новом объекте получили значения `nil`. Давайте посмотрим, изменилось ли что-то после команды `save`:

```
>> user
=> #<Userid: 1, name: "MichaelHartl", email: "mhartl@example.com",
created_at: "2014-07-24 00:57:46", updated_at: "2014-07-24 00:57:46">
```

[1] Кроме раздела 12.3.3.

Как видите, столбец id получил значение 1, а столбцы created_at иupdated_at – текущее время и дату[1]. Сейчас столбцы created_at иupdated_at хранят идентичные значения; но они могут различаться, как будет показано в разделе 6.1.5.

Экземпляры модели User, так же как экземпляры класса User в разделе 4.4.5, предоставляют доступ к своим атрибутам с помощью точечной нотации:

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "mhartl@example.com"
>> user.updated_at
=> Thu, 24 Jul 2014 00:57:46 UTC +00:00
```

Как мы увидим в главе 7, часто бывает удобно создать и сохранить модель в два приема, это делалось выше, но Active Record позволяет также объединить эти действия в один шаг:

```
>> 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">
```

Обратите внимание: на этот раз User.create вернул не true или false, а сам объект User, который при желании можно присвоить переменной (как показано во второй команде).

Команда destroy выполняет обратное действие:

```
>> 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">
```

Команда destroy также возвращает рассматриваемый объект, хотя я не могу вспомнить, чтобы когда-либо использовал эту ее особенность. Кроме того, уничтоженный объект все еще существует в памяти:

---

[1] Если время "2014-07-24 00:57:46" показалось вам странным, имейте в виду – я не пишу эти строки после полуночи; в базу данных записывается координированное универсальное время (Coordinated Universal Time), которое во многих отношениях соответствует времени по Гринвичу (UTC). Вот выдержка из «NIST Time and Frequency FAQ» (http://www.nist.gov/pml/div688/faq.cfm):
Вопрос: Почему для обозначения координированного универсального времени используется аббревиатура UTC вместо CUT (Coordinated Universal Time)?
Ответ: В 1970 г. система координированного универсального времени разрабатывалась международной консультативной группой технических экспертов Международного союза электросвязи (International Telecommunication Union, ITU). Эксперты решили, что лучше определить единую аббревиатуру для использования на всех языках и тем самым минимизировать беспорядок. Так как единогласное соглашение недостижимо при использовании порядка слов в английском (CUT) или французском (TUC) языке, в качестве компромисса была выбрана аббревиатура UTC.

```
>> foo
=> #<Userid: 3, name: "Foo", email: "foo@bar.com", created_at: "2014-07-24 01:05:42",
updated_at: "2014-07-24 01:05:42">
```

Как же тогда узнать, был ли уничтожен объект в действительности? И как получить сохраненные и неуничтоженные объекты из базы данных? Чтобы ответить на все эти вопросы, нужно знать, как использовать Active Record для поиска объектов.

6.1.4. Поиск объектов User

Библиотека Active Record поддерживает несколько способов поиска объектов. Давайте используем их, чтобы найти первого созданного пользователя и убедиться, что третий пользователь (foo) был уничтожен. Начнем с существующего пользователя:

```
>> 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">
```

Мы передали значение id пользователя в вызов User.find, и библиотека Active Record вернула пользователя с этим значением в атрибуте id.

Теперь посмотрим, существует ли в базе данных пользователь с id = 3:

```
>> User.find(3)
ActiveRecord::RecordNotFound: Couldn't find User with ID=3
```

Так как в разделе 6.1.3 мы уничтожили третьего пользователя, Active Record не нашла его в базе данных. Вместо этого метод find вызвал *исключение* – именно таким способом библиотека сообщает об исключительных событиях во время выполнения программы. В данном случае несуществующий идентификатор id заставил find вызвать исключение ActiveRecord::RecordNotFound[1].

В дополнение к обобщенному методу find в арсенале Active Record имеется еще один похожий метод, позволяющий искать пользователей по определенным атрибутам:

```
>> 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">
```

Поскольку мы будем использовать адреса электронной почты в качестве имен пользователей, этот вид поиска пригодится нам, когда мы приступим к реализации механизма входа на сайт (глава 7). Если вас волнует эффективность find_by при большом количестве пользователей – ваше беспокойство вполне обосновано, но вы немного забегаете вперед; мы обсудим эту проблему и ее решение с помощью индексов в разделе 6.2.5.

---

[1] Исключения и обработка исключений – сложная тема, связанная с программированием на языке Ruby, и мы не будем касаться ее в этой книге. И все же исключения играют важную роль, и я предлагаю познакомиться с ними, используя одну из книг, рекомендованных в разделе 12.4.1.

А теперь закончим этот раздел знакомством с парой более обобщенных способов поиска пользователей. Первый способ реализует метод first:

```
>> 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">
```

Метод first просто возвращает первого пользователя в базе данных. Второй способ реализует метод 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">]>
```

Как видите, User.all возвращает всех пользователей в базе данных в виде объекта класса ActiveRecord::Relation, который фактически является массивом (раздел 4.3.1).

6.1.5. Обновление объектов User

После создания объектов их часто приходится изменять. Сделать это можно двумя способами. Во-первых, можно присваивать значения отдельным атрибутам объекта, как это делалось в разделе 4.4.5:

```
>> user               # Просто чтобы вспомнить имеющиеся атрибуты
=> #<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
```

Обратите внимание: заключительный шаг необходим, чтобы записать изменения в базу данных. Мы можем увидеть, что произойдет без этого шага, если вызвать метод reload, который загрузит объекты из базы данных:

```
>> user.email
=> "mhartl@example.net"
>> user.email = "foo@bar.com"
=> "foo@bar.com"
>> user.reload.email
=> "mhartl@example.net"
```

Теперь, после изменения атрибута пользователя и вызова user.save, столбцы created_at и updated_at получили разные значения, как и было обещано в разделе 6.1.3:

```
>> user.created_at
=> "2014-07-24 00:57:46"
```

```
>> user.updated_at
=> "2014-07-24 01:37:32"
```

Второй основной способ заключается в использовании метода update_attributes, изменяющего сразу несколько атрибутов[1]:

```
>> user.update_attributes(name: "The Dude",
>> email: "dude@abides.org")
=> true
>> user.name
=> "The Dude"
>> user.email
=> "dude@abides.org"
```

Метод update_attributes принимает хэш атрибутов и, в случае успеха, выполняет не только обновление, но и сохранение (возвращая true как признак успешного сохранения). Обратите внимание: если какая-нибудь из проверок завершится неудачей, что может произойти, например, когда в хэше отсутствует атрибут, обязательный для сохранения (как будет реализовано в разделе 6.3), вызов update_attributes потерпит неудачу. Если необходимо обновить только один атрибут, используйте update_attribute, чтобы обойти это ограничение:

```
>> user.update_attribute(:name, "The Dude")
=> true
>> user.name
=> "The Dude"
```

6.2. Проверка объектов User

Созданная нами модель User теперь имеет действующие атрибуты name и email, но они абсолютно универсальны: сейчас в них можно записать любые строки (в том числе и пустые). Однако имя и адрес электронной почты – это нечто более определенное. Например, атрибут name не должен содержать пустую строку, а строка в атрибуте email должна соответствовать определенному формату, характерному для адресов электронной почты. Кроме того, так как адреса электронной почты предполагается использовать в качестве уникальных имен пользователей при регистрации, они не должны повторяться в базе данных.

Проще говоря, мы не должны позволять записывать в атрибуты name и email любые строки; требуется реализовать определенные ограничения для их значений. Active Record позволяет накладывать такие ограничения с помощью *проверок* (они упоминались в разделе 2.3.2). В этом разделе мы рассмотрим несколько наиболее распространенных случаев, реализовав проверку *наличия*, *длины*, *формата* и *уникальности*. В разделе 6.3.2 мы добавим заключительную проверку: *под-*

[1] Для метода update_attributes имеется аналог с более коротким именем, update, но я предпочитаю более длинную версию из-за сходства с версией метода в единственном числе, update_attribute.

тверждение). А в разделе 7.3 узнаем, как выводить сообщения об ошибках, когда пользователи предоставляют недопустимые данные.

6.2.1. Проверка допустимости

Как отмечалось в блоке 3.3, разработка через тестирование не всегда оправдана, но проверка моделей – это именно тот случай, когда TDD подходит идеально. Никогда нельзя быть уверенным, что данная проверка делает именно то, что ожидается, если прежде не написать тест, терпящий неудачу, а затем обеспечить его успешное выполнение.

Далее мы будем действовать по следующему алгоритму: возьмем заведомо *допустимый* объект модели, запишем в один из его атрибутов недопустимое значение, а затем с помощью тестов убедимся, что они оценивают его как недопустимый. Но прежде напишем тесты, чтобы убедиться в допустимости исходного объекта. При таком подходе, когда в дальнейшем тесты проверки действительности потерпят неудачу, мы будем уверены, что это произошло не потому, что исходная модель сразу была недопустимой.

Для простоты возьмем за основу тесты, подготовленные командой из листинга 6.1, хотя они практически пустые (листинг 6.4).

Листинг 6.4 ❖ Начальные тесты для модели User (test/models/user_test.rb)

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

Чтобы написать тест для допустимого объекта, создадим заведомо допустимый объект `@user` модели User с помощью специального метода `setup` (кратко обсуждался в упражнениях к главе 3), который автоматически будет запускаться перед каждым тестом. Так как `@user` – это переменная экземпляра, она автоматически будет доступна всем тестам, и мы сможем тестировать ее допустимость методом `valid?` (раздел 6.1.3). Результат показан в листинге 6.5.

Листинг 6.5 ❖ Тест заведомо допустимого объекта user **ЗЕЛЕНЫЙ** (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
```

Здесь использован простой метод assert, который будет завершаться успехом, получив true от @user.valid?, и терпеть неудачу, получив false.

Так как пока модель User не реализует никаких проверок, тест должен завершаться успехом:

Листинг 6.6 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test:models
```

Команда rake test:models запускает только тесты моделей (сравните с rake test:integration из раздела 5.3.4).

6.2.2. Проверка наличия

Проверка *наличия* является самой простой – она всего лишь проверяет присутствие значения в указанном атрибуте. Например, в данном разделе мы реализуем проверку заполнения полей name и email, которая будет выполняться перед сохранением объекта в базе данных. В разделе 7.3.3 мы увидим, как использовать ее в форме регистрации новых пользователей.

Начнем с тестирования проверки наличия значения в атрибуте name, взяв за основу тест из листинга 6.5. Как показано в листинге 6.7, достаточно просто записать строку из пробелов в атрибут name переменной @user, а затем убедиться (вызовом метода assert_not), что полученный объект User недопустим.

Листинг 6.7 ❖ Тест проверки атрибута name КРАСНЫЙ (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
```

Сейчас набор тестов модели должен быть **КРАСНЫМ**:

Листинг 6.8 ❖ КРАСНЫЙ

```
$ bundle exec rake test:models
```

Как мы уже видели в упражнениях к главе 2, для проверки наличия значения в атрибуте name можно использовать метод validates с аргументом presence: true,

как показано в листинге 6.9. Аргумент `presence: true` – это *хэш параметров* с одним элементом – как рассказывалось в разделе 4.3.4, фигурные скобки можно опустить, если хэш передается методу в последнем аргументе. (В разделе 5.1.1 уже отмечалось, что хэши параметров широко используются в Rails.)

Листинг 6.9 ❖ Проверка наличия значения в атрибуте name **ЗЕЛЕНЫЙ** (app/models/user.rb)

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

Листинг 6.9 может показаться таинственным, но это не так, `validates` – обычный метод. Ту же проверку можно записать с круглыми скобками:

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

Давайте вернемся в консоль, чтобы посмотреть, как действует только что добавленная проверка[1]:

```
$ rails console --sandbox
>> user = User.new(name: "", email: "mhartl@example.com")
>> user.valid?
=> false
```

Здесь мы проверили допустимость переменной `user` вызовом метода `valid?`, который возвращает `false`, если объект не проходит одну или несколько проверок, и `true`, когда все проверки завершаются успехом. В настоящий момент реализована только одна проверка, поэтому мы точно знаем, какая именно завершилась неудачей, и все же полезно заглянуть в объект `errors`, созданный проверкой:

```
>> user.errors.full_messages
=> ["Name can't be blank"]
```

(Сообщение об ошибке подсказывает, что Rails выполняет проверку наличия значения в атрибуте методом `blank?`, который встречался нам в конце раздела 4.4.3.)

Так как такой объект недопустим, попытка сохранить его в базе данных автоматически потерпит неудачу:

```
>> user.save
=> false
```

Как результат тест из листинга 6.7 теперь должен быть **ЗЕЛЕНЫМ**:

Листинг 6.10 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test:models
```

[1] Я буду опускать вывод команд, таких как `User.new`, если в нем нет ничего полезного.

Следуя модели из листинга 6.7, легко написать тест для реализации проверки наличия значения в атрибуте email (листинг 6.11), а также прикладной код, гарантирующий успешное его выполнение (листинг 6.12).

Листинг 6.11 ❖ Тест проверки атрибута email **КРАСНЫЙ** (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
```

Листинг 6.12 ❖ Проверка наличия значения в атрибуте email **ЗЕЛЕНЫЙ** (app/models/user.rb)

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

Теперь проверки наличия значений готовы, и набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 6.13 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

6.2.3. Проверка длины

Мы потребовали от модели User наличия непустого имени у каждого пользователя, но это еще не все: имена пользователей будут отображаться на сайте, поэтому мы должны реализовать некоторое ограничение их длины. С опытом, полученным в разделе 6.2.2, это несложно.

В выборе максимальной длины нет ничего хитрого; просто примем 50 символов как разумную верхнюю границу. Это будет означать, что имена длиной в 51 символ

будут считаться слишком длинными. Кроме того, есть вероятность, хотя и небольшая, что адрес электронной почты пользователя превысит максимальную длину строки, 255 символов, допустимую для многих баз данных. Так как проверка формата в разделе 6.2.4 не накладывает такого ограничения, мы добавим его здесь для полноты. Получившиеся тесты показаны в листинге 6.14.

Листинг 6.14 ❖ Тест для проверки длины атрибута name **КРАСНЫЙ** (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
```

Для удобства мы использовали операцию «умножения строки», чтобы создать строку длиной в 51 символ. В консоли можно посмотреть, как действует эта операция:

```
>> "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> ("a" * 51).length
=> 51
```

Тест проверки длины строки в атрибуте `email` создает адрес длиннее допустимого на один символ:

```
>> "a" * 244 + "@example.com"
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
@example.com"
>> ("a" * 244 + "@example.com").length
=> 256
```

Сейчас тест из листинга 6.14 должен быть **КРАСНЫМ**:

Листинг 6.15 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Чтобы он увенчался успехом, нужно указать проверяемый параметр, то есть length, вместе с параметром maximum, устанавливающим верхнюю границу (листинг 6.16).

Листинг 6.16 ❖ Проверка длины атрибута name ЗЕЛЕНЫЙ (app/models/user.rb)

```
class User < ActiveRecord::Base
  validates :name, presence: true, length: { maximum: 50 }
  validates :email, presence: true, length: { maximum: 255 }
end
```

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 6.17 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Набор тестов снова завершается успехом, и можно переходить к более сложной проверке формата электронной почты.

6.2.4. Проверка формата

Проверки атрибута name накладывают минимальные ограничения – допустимым считается любое непустое имя длиной не более 50 символов, – но атрибут email должен соответствовать более строгим требованиям. На данный момент отклоняются только пустые значения, но далее в этом разделе мы потребуем, чтобы адреса электронной почты соответствовали известному образцу user@example.com.

Важно понимать, что невозможно создать исчерпывающие тесты и проверки, – они могут быть лишь достаточно хорошими, чтобы пропустить большинство допустимых и отклонить большинство недопустимых адресов электронной почты. Начнем с пары тестов, включающих наборы допустимых и недопустимых адресов. Для этого познакомимся с интересным приемом создания массивов строк %w[], как показано ниже:

```
>> %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
```

В этом примере метод each перебирает все элементы массива addresses (раздел 4.3.2). Вооружившись этим приемом, мы готовы написать несколько простых тестов для проверки формата электронной почты.

Такая проверка весьма сложна и чревата ошибками, поэтому начнем с передачи тестам нескольких *допустимых* адресов, чтобы выявить любые ошибки в реализации проверки. Другими словами, убедимся не только в том, что проверка отклонит недопустимые адреса, такие как *user@example,com*, но и примет допустимые адреса, такие как *user@example.com*. (Прямо сейчас они будут приняты, так как на данный момент допустимыми считаются любые непустые адреса.) Тест для проверки типичных допустимых адресов электронной почты показан в листинге 6.18.

Листинг 6.18 ❖ Тесты для допустимых адресов электронной почты **ЗЕЛЕНЫЙ** (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
```

Обратите внимание, что здесь мы включили в утверждение дополнительный второй аргумент с собственным сообщением, которое выводится, если проверка потерпит неудачу:

```
assert @user.valid?, "#{valid_address.inspect} should be valid"
```

(Здесь использован метод `inspect`, упоминавшийся в разделе 4.3.3.) Включение адреса, вызвавшего ошибку, в текст сообщения особенно полезно в тестах с циклом `each`; иначе любая ошибка будет просто сообщать номер строки, одинаковый для всех адресов электронной почты, а этого недостаточно для определения источника проблемы.

Теперь добавим тест на *недопустимость* со множеством недопустимых адресов, таких как *user@example,com* (запятая вместо точки) и *user_at_foo.org* (отсутствует знак `@`). Так же, как в предыдущем тесте, определим собственное сообщение, показывающее адрес, вызвавший ошибку.

Листинг 6.19 ❖ Тесты для реализации проверки формата электронной почты **КРАСНЫЙ** (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_foo.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
```

Сейчас набор тестов должен быть **КРАСНЫМ**:

Листинг 6.20 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Проверка формата адреса электронной почты выполняется с применением параметра `format`, как показано ниже:

```
validates :email, format: { with: /<регулярное выражение>/ }
```

Он осуществляет проверку допустимости атрибута с помощью *регулярного выражения* – мощного (и зачастую загадочного) языка сопоставления строк с шаблонами. Это значит, что нужно сконструировать регулярное выражение, которому будут соответствовать допустимые адреса электронной почты и *не* соответствовать недопустимые.

На самом деле существует полное регулярное выражение для проверки адресов электронной почты, которое соответствует официальным стандартам, но слишком большое, непонятное, оно вполне может привести к неверным результатам[1]. Поэтому здесь мы используем более простое регулярное выражение, которое на практике дает более надежные результаты. Оно выглядит так:

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
```

Чтобы было понятнее, в табл. 6.1 приводится это же выражение, разбитое на небольшие фрагменты[2].

---

[1] Например, знаете ли вы, что `"Michael Hartl"@example.com` с кавычками и пробелом в середине является допустимым адресом электронной почты, согласно стандарту? Невероятно, но так и есть.

[2] Обратите внимание, что в табл. 6.1 под «буквой» подразумевается «строчная буква», но символ `i` в конце выражения обеспечивает нечувствительность к регистру.

Таблица 6.1 ❖ Разобранное на части регулярное выражение для проверки адресов электронной почты

Выражение	Значение
`/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i`	Полное регулярное выражение
`/`	Начало выражения
`\A`	Соответствует началу строки
`[\w+\-.]+`	По крайней мере, один символ слова, плюс, дефис или точка
`@`	Символ «@»
`[a-z\d\-.]+`	По крайней мере, одна буква, цифра, дефис или точка
`\.`	Символ точки
`[a-z]+`	По крайней мере, одна буква
`\z`	Соответствует концу строки
`/`	Конец регулярного выражения
`i`	Нечувствительность к регистру

Из табл. 6.1 можно многое узнать, но, чтобы действительно понять регулярные выражения, я предлагаю использовать интерактивный редактор, такой как Rubular (рис. 6.7)[1]. На сайте Rubular (http://www.rubular.com/) реализован ин-

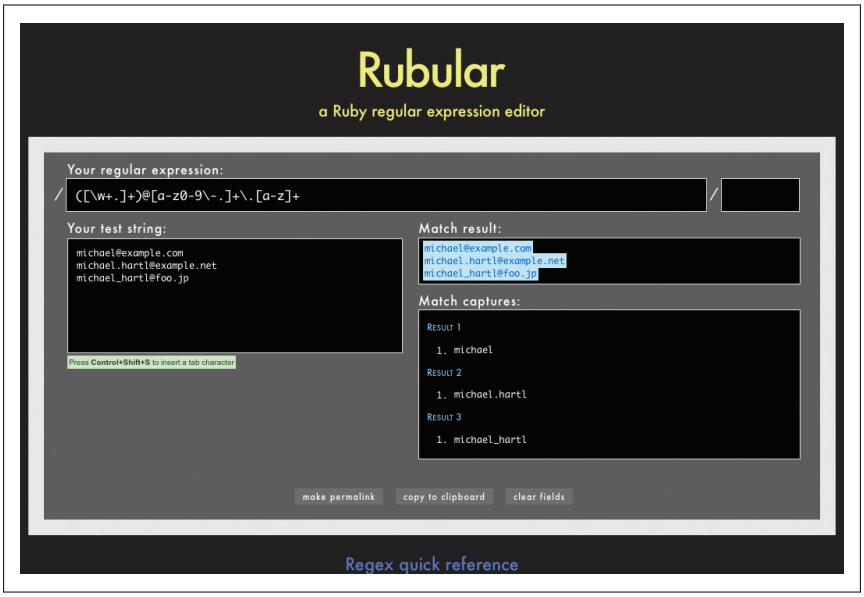


Рис. 6.7 ❖ Замечательный редактор регулярных выражений Rubular

[1] Если вы сочтете его столь же полезным, как и я, пожертвуйте небольшую сумму (http://bit.ly/donate-to-rubular), чтобы вознаградить разработчика Rubular – Майкла Ловитта (Michael Lovitt) – за его замечательную работу.

терактивный интерфейс для создания регулярных выражений, а также имеется удобная справка. Исследуйте содержимое табл. 6.1 в окне веб-браузера с открытым интерфейсом Rubular – никакое руководство по регулярным выражениям не сможет заменить интерактивную игру с ними. (*Примечание*: если вы решите поэкспериментировать с регулярным выражением из табл. 6.1 в Rubular, уберите символы `\A` и `\z`, потому что в Rubular отсутствует четкое определение начала и конца входной строки.)

Применим регулярное выражение из табл. 6.1 для проверки формата `email`, как показано в листинге 6.21.

Листинг 6.21 ❖ Проверка формата email с помощью регулярного выражения **ЗЕЛЕНЫЙ** (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
```

Регулярное выражение `VALID_EMAIL_REGEX` – это *константа*. В Ruby константам даются имена, начинающиеся с заглавной буквы. Следующий код:

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
validates :email, presence: true, length: { maximum: 255 },
                  format:   { with: VALID_EMAIL_REGEX }
```

гарантирует, что допустимыми будут считаться адреса электронной почты, только соответствующие шаблону. (В этом выражении есть один заметный недостаток: оно считает допустимыми адреса, содержащие несколько точек подряд, например `foo@bar..com`. Устранение этого недочета требует более сложного регулярного выражения, которое я оставляю для самостоятельного упражнения (раздел 6.5).)

Сейчас набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 6.22 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test:models
```

Это означает, что осталось реализовать единственное ограничение: уникальность адресов электронной почты.

6.2.5. Проверка уникальности

Чтобы гарантировать уникальность адресов электронной почты (и возможность их использования в качестве имен пользователей), мы передадим методу `validates` параметр `:unique`. Но предупреждаю: с проверкой уникальности связан один *важный* аспект, поэтому не просто просмотрите раздел, а прочитайте его внимательно.

Начнем с нескольких коротких тестов. В предыдущих тестах мы главным образом использовали метод `User.new`, который просто создает объект в памяти, но

для проверки уникальности требуется поместить запись в базу данных[1]. В листинге 6.23 представлен первый вариант проверки уникальности.

Листинг 6.23 ❖ Тест проверки отклонения повторяющихся адресов электронной почты **КРАСНЫЙ** (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
```

Здесь выполняется попытка создать второго пользователя с таким же адресом электронной почты, как и у `@user`, вызвав метод `@user.dup` и создав таким способом дубликат пользователя с такими же атрибутами. Так как после этого мы сохраняем `@user`, к моменту проверки адрес электронной почты второго пользователя уже будет храниться в базе данных, следовательно, он должен быть опознан как недопустимый.

Чтобы новый тест из листинга 6.23 завершился успехом, добавим параметр `uniqueness: true` в вызов метода проверки `email`, как показано в листинге 6.24.

Листинг 6.24 ❖ Проверка уникальности адресов электронной почты **ЗЕЛЕНЫЙ** (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
```

Но это еще не все. Адреса электронной почты обычно обрабатываются, как если бы они были нечувствительны к регистру, то есть `foo@bar.com`, `FOO@BAR.COM` и `FoO@BAr.coM` считаются одинаковыми, поэтому проверка должна учитывать этот слу-

[1] Как отмечалось в начале раздела, роль тестовой базы данных играет `db/test.sqlite3`.

чай[1]. А раз нечувствительность к регистру так важна, проверим ее, как показано в листинге 6.25.

Листинг 6.25 ❖ Тестирование нечувствительности к регистру проверки уникальности адресов электронной почты **КРАСНЫЙ** (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
    duplicate_user.email = @user.email.upcase
    @user.save
    assert_not duplicate_user.valid?
  end
end
```

Здесь добавлен вызов метода `upcase` (кратко описан в разделе 4.3.2). Этот тест делает то же самое, что и первоначальный вариант, но пытается создать пользователя с адресом, записанным прописными буквами. Если этот тест кажется вам немного абстрактным, запустите консоль:

```
$ railsconsole --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
```

---

[1] Формально только имя домена в адресе является нечувствительным к регистру: на самом деле адрес `foo@bar.com` отличается от `Foo@bar.com`. На практике, однако, полагаться на этот факт – плохая идея; как отмечено на сайте about.com (http://email.about.com/od/emailbehindthescenes/f/email_case_sens.htm): «Поскольку чувствительность к регистру может создать массу проблем, было бы глупо требовать от пользователей вводить адреса электронной почты с соблюдением регистра символов. Вряд ли какая-либо из служб электронной почты или кто-то из поставщиков услуг Интернета обращает внимание на регистр символов в адресах, возвращая электронные письма, в которых адрес получателя набран неправильно (например, в верхнем регистре)». Спасибо читателю Райли Мозес (Riley Moses) за это замечание.

Метод duplicate_user.valid? сейчас возвращает true, потому что пока проверка уникальности различает регистр символов, но нам требуется получить false. К счастью, :uniqueness принимает дополнительный параметр :case_sensitive как раз для этих целей (листинг 6.26).

Листинг 6.26 ❖ Проверка уникальности адресов электронной почты, нечувствительная к регистру **ЗЕЛЕНЫЙ** (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
```

Обратите внимание, что мы просто заменили true (в листинге 6.24) на case_sensitive: false (в листинге 6.26). Соответственно, Rails делает вывод, что уникальность должна быть истинной.

Теперь наше приложение гарантирует уникальность адресов электронной почты – с важной оговоркой – и набор тестов должен завершаться успехом:

Листинг 6.27 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Осталась еще одна маленькая проблема: *проверка уникальности в Active Record не гарантирует уникальности на уровне базы данных*. Чтобы понять, в чем дело, рассмотрим следующую ситуацию:

1. Алиса регистрируется в нашем приложении и указывает адрес alice@wonderland.com.
2. Она случайно *дважды* щелкает на кнопке **Submit** (Отправить), посылая два запроса друг за другом.
3. Затем происходит следующее: первый запрос создает в памяти пользователя, который проходит проверку, второй запрос делает то же самое, первый пользователь сохраняется в базе, и следом за ним сохраняется второй пользователь.
4. Результат: в базе данных хранятся две учетные записи с одинаковыми адресами электронной почты, несмотря на наличие проверки уникальности.

Если описанная последовательность кажется вам невероятной, поверьте мне, это может произойти на любом Rails-сайте со значительным трафиком (я это усвоил однажды на собственном горьком опыте). К счастью, проблема имеет довольно простое решение: достаточно обеспечить уникальность не только на уровне модели, но и на уровне базы данных. Для этого нужно создать в базе данных *индекс* для столбца email (блок 6.2) и потребовать его уникальности.

Блок 6.2 ❖ Индексы базы данных

При создании столбца в базе данных важно определить, понадобится ли когда-нибудь выполнять *поиск* записей по нему. Рассмотрим, например, атрибут `email`, созданный миграцией из листинга 6.2. Когда в главе 7 мы приступим к реализации процедуры входа пользователя, нам понадобится найти учетную запись, соответствующую предоставленному адресу электронной почты. К сожалению, если основываться исключительно на модели данных, единственный способ найти пользователя адресу будет – просмотреть все записи в базе данных и сравнить атрибут `email` в каждой из них с искомым адресом электронной почты – в худшем случае придется проверить *все* строки (пользователь может оказаться последним в базе данных). Это способ известен как *полный просмотр таблицы* и никуда не годится для настоящих сайтов с тысячами пользователей.

Добавление индекса по столбцу `email` решает проблему. Чтобы понять, как работает индекс в базе данных, полезно рассмотреть аналогию с алфавитным указателем. Чтобы найти все упоминания некоторого слова, например «foobar», вам придется просмотреть каждую страницу – это бумажная версия полного просмотра таблицы. С другой стороны, если в книге есть алфавитный указатель, можно просто найти в нем «foobar», чтобы узнать, на каких страницах встречается это слово. Индекс базы данных работает почти так же.

Индекс по столбцу `email` представляет обновление требований к модели данных, что (как обсуждалось в разделе 6.1.1) делается в Rails посредством миграций. Мы видели в разделе 6.1.1, что создание модели User автоматически привело к созданию новой миграции (листинг 6.2); в данном случае мы добавляем структуру к существующей модели, поэтому необходимо создать миграцию непосредственно, используя генератор `migration`:

```
$ rails generate migration add_index_to_users_email
```

В отличие от миграции, связанной с созданием таблицы пользователей, миграция ввода уникальности адреса электронной почты не предопределена, поэтому нужно заполнить ее содержимым листинга 6.28.

Листинг 6.28 ❖ Миграция для реализации уникальности адреса электронной почты (db/migrate/[timestamp]_add_index_to_users_email.rb)

```
class AddIndexToUsersEmail < ActiveRecord::Migration
  def change
    add_index :users, :email, unique: true
  end
end
```

Добавление индекса осуществляется вызовом Rails-метода `add_index`. Индекс сам по себе не обеспечивает уникальности, это делает параметр `unique: true`.

Заключительный шаг – выполнить миграцию базы данных:

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

(Если эта команда завершилась неудачей, попробуйте закрыть все консольные сеансы, которые могут блокировать базу данных и препятствовать миграции.)

Сейчас набор тестов должен стать **КРАСНЫМ** из-за нарушения уникальности в *испытательном окружении* с примерами данных для тестов. Испытательные данные были созданы автоматически в листинге 6.1, и листинг 6.29 показывает, что адреса электронной почты в них не уникальны. (Кроме того, они *недопустимы*, но испытательные данные не проходят проверку допустимости.)

Листинг 6.29 ❖ Исходные испытательные данные **КРАСНЫЙ** (test/fixtures/users.yml)

```
# Дополнительные сведения об испытательном окружении ищите на сайте:
# http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
  name: MyString
  email: MyString

two:
  name: MyString
  email: MyString
```

Так как испытательные данные не понадобятся до главы 8, мы пока просто удалим их, оставив пустой файл (листинг 6.30).

Листинг 6.30 ❖ Пустой файл с испытательными данными **ЗЕЛЕНЫЙ** (test/fixtures/users.yml)

```
# пустой
```

Мы разобрались с реализацией уникальности, но для полной уверенности в уникальности адресов осталась сделать еще кое-что. Некоторые средства доступа к базам данных используют чувствительные к регистру индексы; они считают разными строки «Foo@ExAMPle.CoM» и «foo@example.com». Но для нашего приложения эти адреса одинаковы. Чтобы избежать подобной несовместимости, мы стандартизируем все адреса и перед сохранением будем переводить все символы адреса в нижний регистр («Foo@ExAMPle.CoM» в «foo@example.com»). Это можно реализовать с помощью *обратного вызова* – метода, вызываемого в определенный момент в жизненном цикле объекта ActiveRecord. В данном случае это момент перед сохранением объекта, поэтому для перевода адресов в нижний регистр будем использовать функцию обратного вызова `before_save`[1]. Результат показан в листинге 6.31. (Это лишь первая реализация; мы еще вернемся к этой теме в разделе 8.4, где для определения обратных вызовов будем использовать более удобный вариант *со ссылкой на метод*.)

[1] За дополнительной информацией обращайтесь к статье с описанием обратных вызовов в Rails API (http://api.rubyonrails.org/v4.2.0/classes/ActiveRecord/Callbacks.html).

Листинг 6.31 ❖ Гарантия уникальности атрибута email за счет перевода в нижний регистр **ЗЕЛЕНЫЙ** (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
```

Код в листинге 6.31 передает блок в `before_save` и преобразует адрес электронной почты в нижний регистр с применением строкового метода `downcase`. (Создание тестов для метода преобразования в нижний регистр я оставлю вам в качестве упражнения (раздел 6.5).)

В листинге 6.31 присваивание можно было бы записать так:

```
self.email = self.email.downcase
```

(где `self` ссылается на текущего пользователя), но внутри модели User ключевое слово `self` в правой части присваивания можно опустить:

```
self.email = email.downcase
```

Мы коротко обсуждали эту идею в контексте применения метода `reverse` в методе `palindrome` (раздел 4.4.2), где также указывалось, что `self` обязательно в присваивании, поэтому

```
email = email.downcase
```

не будет работать. (Подробнее мы обсудим эту тему в разделе 8.4.)

Теперь в ситуации с Алисой, описанной выше, ничего страшного не произойдет: база данных сохранит учетную запись из первого запроса и отклонит второй запрос из-за нарушения уникальности. (Ошибка появится в журнале Rails, но в этом нет ничего плохого.) Более того, добавление индекса для атрибута адреса электронной почты преследует вторую цель, упоминавшуюся в разделе 6.1.4: как отмечалось блоке 6.2, индекс для атрибута `email` решает потенциальную проблему эффективности, предотвращая полный просмотр таблицы при поиске пользователя по адресу электронной почты.

6.3. Добавление безопасного пароля

После реализации проверки допустимости значений полей `name` и `email` можно добавить последний из основных атрибутов модели User: безопасный пароль. Мы будем запрашивать у каждого пользователя пароль (с подтверждением) и сохранять в базе данных *хэшированную* версию. (Обратите внимание: в данном случае под *хэшем* понимается не структура данных Ruby, а результат применения необра-

тимой хэш-функции[1] к исходным данным.) Попутно мы реализуем метод *аутентификации* пользователя на основе введенного пароля, который задействуем в главе 8.

Метод аутентификации будет принимать введенный пароль, хэшировать его и сравнивать с хэшированным значением в базе данных. Если они совпадают, значит, введен верный пароль, и пользователя можно считать аутентифицированным. Использование хэшированных значений позволяет устанавливать подлинность пользователей без хранения самих паролей в базе данных. То есть даже если база данных будет взломана, пароли пользователей все равно останутся в безопасности.

6.3.1. Хэшированный пароль

Практически весь механизм поддержки безопасных паролей будет реализован на основе единственного Rails-метода `has_secure_password`, который мы включим в модель User:

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

Вместе с этим методом в модель будут добавлены:

- атрибут `password_digest` для хранения хэшированной версии пароля;
- пара виртуальных атрибутов[2] (`password` и `password_confirmation`) вместе с проверками наличия в них значений на этапе создания объекта и их совпадения;
- метод `authenticate`, возвращающий объект пользователя (в ответ на ввод верного пароля) или `false`.

Единственное, что действительно необходимо для `has_secure_password`, – это наличие атрибута `password_digest`. (Название *digest* (дайджест) пришло из криптографии[3] – это значение хэш-суммы, вычисленное хэш-функцией. В данном случае хэшированный пароль и дайджест-пароль являются синонимами[4].) В результате модель User приобретает вид, как показано на рис. 6.8.

---

1 https://ru.wikipedia.org/wiki/Хеширование.

2 В данном случае слово *виртуальные* означает, что атрибуты существуют в объекте модели, но соответствуют базе данных.

3 https://ru.wikipedia.org/wiki/Хеш-сумма.

4 Дайджесты хэшированных паролей часто ошибочно называют *зашифрованными паролями*. Например, в исходном коде `has_secure_password` допущена эта ошибка, как и в первых двух изданиях данной книги. Эта терминология ошибочна, так как, по определению, шифрование *обратимо* – шифрование предполагает возможность расшифровывания. Смысл вычисления дайджеста пароля, напротив, заключается в *необратимости* – невозможности получить исходный пароль из дайджеста. (Спасибо читателю Энди Филипсу (Andy Philips) за это замечание, подтолкнувшее меня на исправление неточной терминологии.)

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string

Рис. 6.8 ❖ Модель данных User с добавленным атрибутом password_digest

Для реализации модели на рис. 6.8 создадим сначала соответствующую миграцию, добавляющую столбец password_digest. Миграции можно дать любое название, но по соглашению завершим его окончанием to_users, потому что миграция описывает добавление столбцов в таблицу users. В результате получаем имя миграции add_password_digest_to_users:

```
$ rails generate migration add_password_digest_to_users password_digest:string
```

Мы передали в команду аргумент password_digest:string с именем и типом атрибута, который нужно создать. (Сравните с первой командой создания таблицы users в листинге 6.1, которой передавались аргументы name:string и email:string.) Аргумент password_digest:string содержит достаточно информации, чтобы Rails смог создать миграцию, это видно в листинге 6.32.

Листинг 6.32 ❖ Миграция для добавления столбца password_digest в таблицу users (db/migrate/[timestamp]_add_password_digest_to_users.rb)

```
class AddPasswordDigestToUsers < ActiveRecord::Migration
  def change
    add_column :users, :password_digest, :string
  end
end
```

Здесь метод add_column добавляет в таблицу users столбец password_digest. Чтобы применить изменения, достаточно запустить миграцию базы данных:

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

Чтобы создать дайджест пароля, метод has_secure_password использует хэш-функцию bcrypt. Хэшируя пароли с помощью bcrypt[1], можно быть уверенными, что злоумышленник не сможет войти на сайт, даже если ему удастся получить копию базы данных. Чтобы получить доступ к bcrypt, необходимо добавить в файл Gemfile приложения гем bcrypt (листинг 6.33)

Листинг 6.33 ❖ Включение гема bcrypt в файл Gemfile

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

gem 'rails',                '4.2.0'
```

[1] https://ru.wikipedia.org/wiki/Bcrypt.

```
gem 'bcrypt',              '3.1.7'
.
.
.
```

и запустить `bundle install`, как обычно:

```
$ bundle install
```

6.3.2. Метод has_secure_password

После добавления в модель User атрибута `password_digest` и получения доступа к функции `bcrypt` можно приступать к определению метода `has_secure_password` в модели User, как показано в листинге 6.34.

Листинг 6.34 ❖ Добавление has_secure_password в модель User **КРАСНЫЙ** (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
```

Индикатор **КРАСНЫЙ** в заголовке листинга 6.34 указывает, что попытки тестирования оканчиваются неудачей, в чем можно убедиться, выполнив команду:

Листинг 6.35 ❖ **КРАСНЫЙ**

```
$ bundle exec rake test
```

Причина в том, что `has_secure_password` проверяет значения виртуальных атрибутов `password` и `password_confirmation`, упоминавшихся в разделе 6.3.1, но в комплекте тестов (листинг 6.25) переменная `@user` создается без этих атрибутов:

```
def setup
  @user = User.new(name: "Example User", email: "user@example.com")
end
```

Чтобы обеспечить безошибочное выполнение тестов, достаточно добавить эти два атрибута, как показано в листинге 6.36.

Листинг 6.36 ❖ Добавление пароля и его подтверждения **ЗЕЛЕНЫЙ** (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
```

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 6.37 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Преимущества от добавления `has_secure_password` будут показаны чуть ниже (раздел 6.3.4), а пока реализуем минимальные требования к паролям.

6.3.3. Минимальная длина пароля

В целом желательно принять некоторые минимальные требования к паролям и применять их, усложняя их угадывание. В Rails существует множество средств обеспечения надежности пароля, но для простоты мы ограничимся только минимальной длиной и потребуем, чтобы пароль не был пустым. Выбрав длину в 6 символов как разумный минимум, получаем тест для проверки допустимости, как показано в листинге 6.38.

Листинг 6.38 ❖ Тесты для минимальной длины пароля **КРАСНЫЙ** (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 have a minimum length" do
    @user.password = @user.password_confirmation = "a" * 5
    assert_not @user.valid?
  end
end
```

Обратите внимание на компактную форму записи множественного присваивания:

```
@user.password = @user.password_confirmation = "a" * 5
```

Эта инструкция присваивает конкретное значение сразу обоим атрибутам, `password` и `password_confirmation`, – в данном случае строку длиной в 5 символов, построенную с помощью операции умножения строки, как в листинге 6.14.

Нетрудно догадаться, как будет выглядеть код, ограничивающий минимальную длину, если вспомнить соответствующую проверку максимальной длины имени пользователя (листинг 6.16):

```
validates :password, length: { minimum: 6 }
```

В результате получается модель User, как показано в листинге 6.39.

Листинг 6.39 ❖ Законченная реализация безопасных паролей **ЗЕЛЕНЫЙ** (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, length: { minimum: 6 }
end
```

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 6.40 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test:models
```

6.3.4. Создание и аутентификация пользователя

Сейчас, закончив модель User, добавим пользователя в базу данных, чтобы приготовиться к созданию страницы, отображающей информацию о пользователе (раздел 7.1). Также более детально рассмотрим результаты добавления `has_secure_password` в модель User и исследуем важнейший метод `authenticate`.

Поскольку пользователи пока не могут регистрироваться в учебном приложении через веб-интерфейс – эта операция является целью главы 7, – создадим нового пользователя вручную, в Rails-консоли. Для удобства мы будем использовать метод `create`, обсуждавшийся в разделе 6.1.3, но на этот раз операция будет выполняться уже *не* в песочнице, чтобы сохранить пользователя в базе данных. Запустите обычный сеанс командой `rails console` и создайте пользователя:

```
$ 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$10$sLcMI2f8VglgirzjSJOln.Fv9NdLMbqmR4rdTWIXY1G...">
```

Чтобы проверить результат, откройте таблицу `users` в обозревателе SQLite, как показано на рис. 6.9[1]. (Пользователи облачной IDE должны загрузить файл базы данных, рис. 6.5.) Определения столбцов, соответствующих атрибутам модели данных, можно увидеть на рис. 6.8.

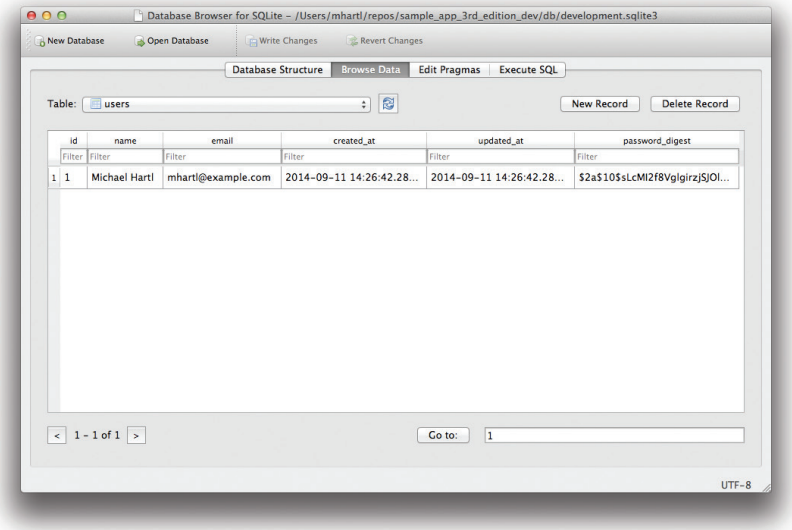


Рис. 6.9 ❖ Запись с информацией о пользователе в базе данных SQLite db/development.sqlite3

Вернувшись в консоль и проверив атрибут `password_digest`, можно увидеть результат работы `has_secure_password` из листинга 6.39:

```
>> user = User.find_by(email: "mhartl@example.com")
>> user.password_digest
=> "$2a$10$YmQTuuDNOszvu5yi7auOC.F4G//FGhyQSWCpghqRWQWITUYlG3XVy"
```

Это хэшированная версия пароля (`"foobar"`), использовавшегося при создании объекта User. Так как она сконструирована с помощью `bcrypt`, по ней невозможно определить исходный пароль[2].

Как отмечалось в разделе 6.3.1, `has_secure_password` автоматически добавляет в соответствующую модель метод `authenticate`. Этот метод выясняет, является

---

[1] Если что-то пошло не так, всегда можно перезагрузить базу данных:
1. Выйти из консоли.
2. Выполнить команду `rm -f development.sqlite3`, чтобы удалить базу данных. (Более изящный метод будет описан в главе 7.)
3. Выполнить миграции `bundle exec rake db:migrate`.
4. Запустить консоль.

[2] Алгоритм bcrypt выдает хэш с «солью» (https://ru.wikipedia.org/wiki/Соль_(криптография)), защищенный от двух классов атак – по словарю и по радужным таблицам.

ли данный пароль допустимым для этого пользователя, вычисляя его дайджест и сравнивая результат с password_digest в базе данных. В случае с только что созданным пользователем можно попробовать несколько неверных паролей:

```
>> user.authenticate("not_the_right_password")
false
>> user.authenticate("foobaz")
false
```

Здесь user.authenticate возвращает false для ошибочного пароля. Если указать верный пароль, authenticate вернет объект пользователя:

```
>> 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...">
```

В главе 8 мы будем использовать метод authenticate в реализации входа зарегистрированных пользователей на сайт. На самом деле не так уж важно, что authenticate возвращает пользователя; главное, что в логическом контексте возвращаемое значение интерпретируется как true. Так как объект user не равен ни nil, ни false, этот прием прекрасно работает[1]:

```
>> !!user.authenticate("foobar")
=> true
```

6.4. Заключение

Начав с нуля, мы создали в этой главе рабочую модель User с необходимыми атрибутами и проверками, накладывающими несколько важных ограничений. Кроме того, реализовали возможность безопасной аутентификации пользователей по указанным ими паролям. Такое значительное количество функциональности заняло всего двенадцать строк кода.

В следующей главе мы создадим форму регистрации для создания новых пользователей, а также страницу для отображения информации об отдельном пользователе. А в главе 8 используем механизм аутентификации из раздела 6.3, дав пользователям возможность входить на сайт.

Если вы используете Git, зафиксируйте внесенные изменения, если вы этого еще не сделали:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Make a basic User model (including secure passwords)"
```

Затем слейте с основной ветвью и отправьте в удаленный репозиторий:

---

[1] Как вы наверняка помните из раздела 4.2.3, оператор !! преобразует объект в соответствующее ему логическое значение.

```
$ git checkout master
$ git merge modeling-users
$ git push
```

Чтобы модель User заработала на действующем сайте, выполните миграции в Heroku с помощью `heroku run`:

```
$ bundle exec rake test
$ git push heroku
$ heroku run rake db:migrate
```

и проверьте, запустив консоль:

```
$ 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$10$IViF0Q5j3hsEVgHgrrKH3uDou86Ka2lEPz8zkwQopwj...">
```

6.4.1. Что мы узнали в этой главе

- Миграции позволяют изменять модели данных в приложении.
- Библиотека Active Record включает множество методов для создания и управления моделями данных.
- Механизмы проверки в Active Record позволяют устанавливать ограничения на данные в моделях.
- Основными проверками являются: наличие, длина, формат.
- Регулярные выражения сложны для понимания, но обладают широкими возможностями.
- Добавление индексов в базу данных увеличивает эффективность поиска и позволяет гарантировать уникальность на уровне базы данных.
- Мы смогли добавить в модели безопасный пароль с помощью встроенного метода `has_secure_password`.

6.5. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Добавьте тест, проверяющий преобразование адреса электронной почты в нижний регистр из листинга 6.31, как показано в листинге 6.41. Здесь метод `reload` извлекает значение из базы данных, а метод `assert_equal` проверяет равенство. Чтобы убедиться, что тест в листинге 6.41 выполняет тестирова-

ние правильно, закомментируйте строку `before_save`, чтобы получить **КРАСНЫЙ** тест, затем раскомментируйте ее, чтобы получить **ЗЕЛЕНЫЙ**.

2. Запустите набор тестов и убедитесь, что функцию обратного вызова `before_save` можно написать с применением «bang»-метода `email.downcase!` для непосредственного изменения атрибута `email`, как показано в листинге 6.42.
3. Как отмечалось в разделе 6.2.4, регулярное выражение из листинга 6.21 пропускает недопустимые адреса с несколькими точками в названии домена, идущими подряд, например `foo@bar..com`. Добавьте этот адрес в список недопустимых в листинге 6.19, чтобы получить **КРАСНЫЙ** тест, а затем используйте более сложное регулярное выражение из листинга 6.43, добившись безошибочного выполнения тестов.

Листинг 6.41 ❖ Тест для функции преобразования адреса электронной почты в нижний регистр из листинга **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
    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
```

Листинг 6.42 ❖ Альтернативная реализация функции обратного вызова `before_save` **ЗЕЛЕНЫЙ** (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, length: { minimum: 6 }
end
```

Листинг 6.43 ❖ Запрет последовательностей точек в именах доменов **ЗЕЛЕНЫЙ** (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, length: { minimum: 6 }
end
```

Глава 7

Регистрация

Теперь, когда у нас есть рабочая модель User, пришло время добавить возможность, без которой могут существовать очень немногие сайты: регистрацию пользователей на сайте. Для отправки приложению регистрационной информации о пользователе мы будем использовать *HTML-форму* (раздел 7.2), на основе этой информации будет создаваться новая учетная запись, а ее атрибуты – сохраняться в базе данных (раздел 7.4). По окончании процесса регистрации важно отобразить страницу профиля с информацией о вновь созданной учетной записи. Поэтому сначала мы создадим страницу *отображения* профиля, которая станет первым шагом на пути реализации REST-архитектуры для поддержки пользователей (раздел 2.2.2). Заодно продолжим работу, начатую в разделе 5.3.4, и добавим в наш набор ещё несколько коротких и выразительных интеграционных тестов.

В этой главе мы будем полагаться на проверки допустимости модели User из главы 6, чтобы увеличить шансы, что пользователи будут указывать действительные адреса электронной почты. В главе 10 мы точно *удостоверимся* в правильности адреса, добавив в процедуру регистрации пользователя дополнительный шаг *активации учетной записи*.

7.1. Страница профиля пользователя

В этом разделе мы сделаем первый шаг на пути к созданию страницы профиля, добавив в приложение страницу, отображающую имя и фотографию пользователя, как показано на рис. 7.1[1]. Наша конечная цель – реализовать отображение фотографии пользователя, основных его данных и списка микросообщений, как показано на рис. 7.2[2]. (На рис. 7.2 показан первый пример использования шаблонного текста *lorem ipsum*, имеющего весьма захватывающую историю[3], с которой определенно стоит познакомиться на досуге.) Окончательно эта страница, а вместе с ней

---

[1] Mockingbird (https://gomockingbird.com/) не поддерживает пользовательских изображений, таких как фотографии, как показано на рис. 7.1; я добавил его вручную с помощью GIMP (http://www.gimp.org/).

[2] Изображение бегемота было взято на сайте http://www.flickr.com/photos/43803060@N00/24308857/.

[3] http://lpgenerator.ru/blog/2015/05/12/istoriya-proishozhdeniya-lorem-ipsum/.

и учебное приложение будут закончены в главе 12.

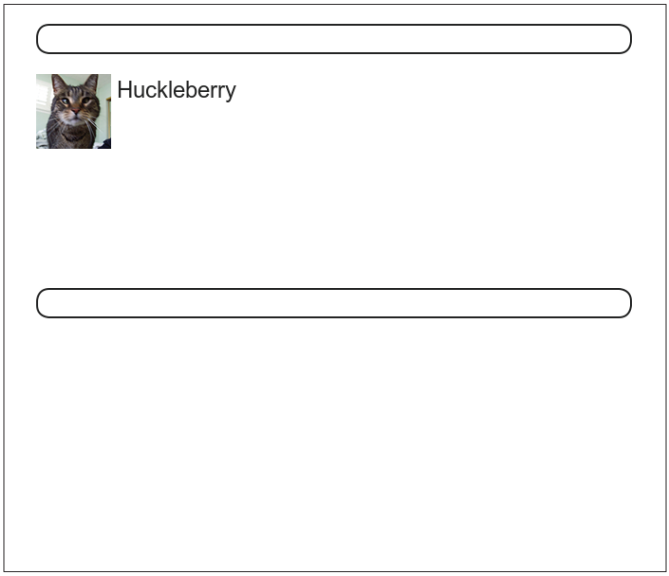


Рис. 7.1 ❖ Макет страницы профиля пользователя, которая будет реализована в этом разделе

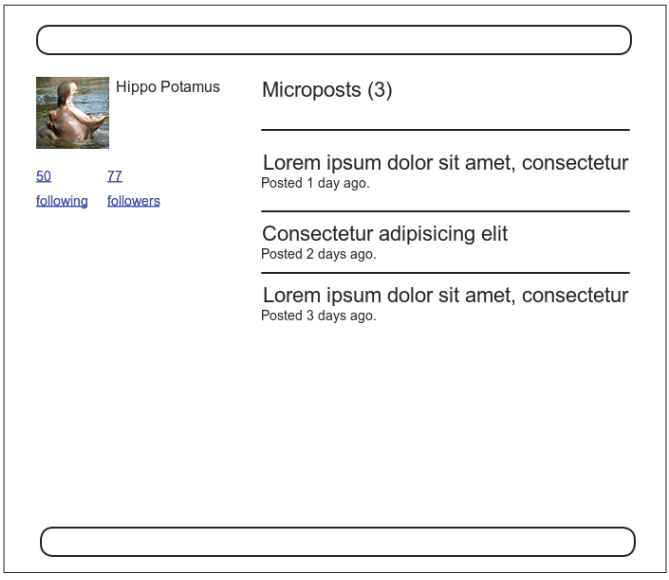


Рис. 7.2 ❖ Макет наших лучших ожиданий от законченной страницы профиля

Если вы используете систему управления версиями, создайте рабочую ветку, как обычно:

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

7.1.1. Отладка и окружение Rails

Профили в этом разделе станут первыми, по-настоящему динамическими страницами в нашем приложении. Хотя представление будет существовать в виде единственной страницы, в каждом профиле будет использована его собственная информация, полученная из базы данных приложения. Готовясь к созданию динамических страниц, добавим в шаблон сайта вывод отладочной информации (листинг 7.1). Вывод этой полезной информации о каждой странице будет осуществляться с помощью встроенного метода `debug` и переменной `params` (мы узнаем о ней больше в разделе 7.1.2).

Листинг 7.1 ❖ Добавление вывода отладочной информации в шаблон сайта (app/views/layouts/application.html.erb)

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

Чтобы отладочная информация не отображалась перед пользователями развернутого приложения, в листинге 7.1 имеется инструкция

```
if Rails.env.development?
```

разрешающая вывод отладочной информации только в *окружении разработки* – одном из трех окружений, по умолчанию определенных в Rails (блок 7.1)[1]. В частности, `Rails.env.development?` возвращает `true` только в окружении разработки, поэтому код на встроенном Ruby

```
<%= debug(params) if Rails.env.development? %>
```

[1] Также можно определить собственное окружение; подробности можно найти на сайте RailsCasts (http://railscasts.com/episodes/72-adding-an-environment).

не будет добавляться в развернутое приложение или в тесты. (Вывод отладочной информации в тестах не навредит, но и не даст ничего хорошего, поэтому лучше ограничиться ее выводом только в окружении разработки.)

Блок 7.1 ❖ Окружения Rails

Rails поставляется с тремя настроенными окружениями: `test` (тестовое), `development` (разработки) и `production` (промышленное). Окружением по умолчанию для консоли Rails является окружение разработки:

```
$ rails console
Loading development environment
>> Rails.env
=> "development"
>> Rails.env.development?
=> true
>> Rails.env.test?
=> false
```

Как видите, в Rails имеется объект `Rails` с атрибутом `env` и логическими методами, среди которых имеется метод `Rails.env.test?`, возвращающий `true` в тестовом окружении и `false` в остальных.
Если понадобится запустить консоль в другом окружении (например, для отладки теста), можно передать окружение сценарию `console` в виде параметра:

```
$ rails console test Loading test environment
>> Rails.env
=> "test"
>> Rails.env.test?
=> true
```

Сервер Rails, так же как консоль, по умолчанию выполняется в окружении разработки, и для него тоже можно изменить окружение:

```
$ rails server --environment production
```

Если попробовать запустить приложение в промышленном окружении, оно не будет работать без настроенной базы данных, которую можно создать, выполнив команду `rake db:migrate` в промышленном окружении:

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

(Я считаю, что три разных взаимоисключающих способа переопределения окружения в консоли, на сервере и для команд миграции могут сбить с толку кого угодно, поэтому я потрудился показать здесь все три.)
Кстати, если вы развернули учебное приложение на Heroku, определить его окружение можно командой `heroku run console`:

```
$ heroku run console
>> Rails.env
=> "production"
```

```
>> Rails.env.production?
=> true
```

Естественно, поскольку Heroku является платформой для развертывания сайтов, она запускает каждое приложение в промышленном окружении.

Чтобы вывод отладочной информации выглядел более опрятно, добавим несколько правил в таблицу стилей, созданную в главе 5 (листинг 7.2).

Листинг 7.2 ❖ Добавление правил оформления блока с отладочной информацией, включая примесь Sass (app/assets/stylesheets/custom.css.scss)

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* примеси, переменные и пр.*/

$gray-medium-light: #eaeaea;

@mixin box_sizing {
  -moz-box-sizing:    border-box;
  -webkit-box-sizing: border-box;
  box-sizing:         border-box;
}
.
.
.
/* разное */

.debug_dump {
  clear: both;
  float: left;
  width: 100%;
  margin-top: 45px;
  @include box_sizing;
}
```

Этот код вводит новый для нас инструмент – Sass-примесь, в данном случае box_sizing. Примесь позволяет сгруппировать CSS-правила, чтобы их могли использовать несколько элементов, превращая

```
.debug_dump {
  .
  .
  .
  @include box_sizing;
}
```

в

```
.debug_dump {
  .
  .
```

```
  .
  -moz-box-sizing:    border-box;
  -webkit-box-sizing: border-box;
  box-sizing:         border-box;
}
```

Мы еще раз задействуем эту примесь в разделе 7.2.1. Результат ее применения к блоку с отладочной информацией показан на рис. 7.3.

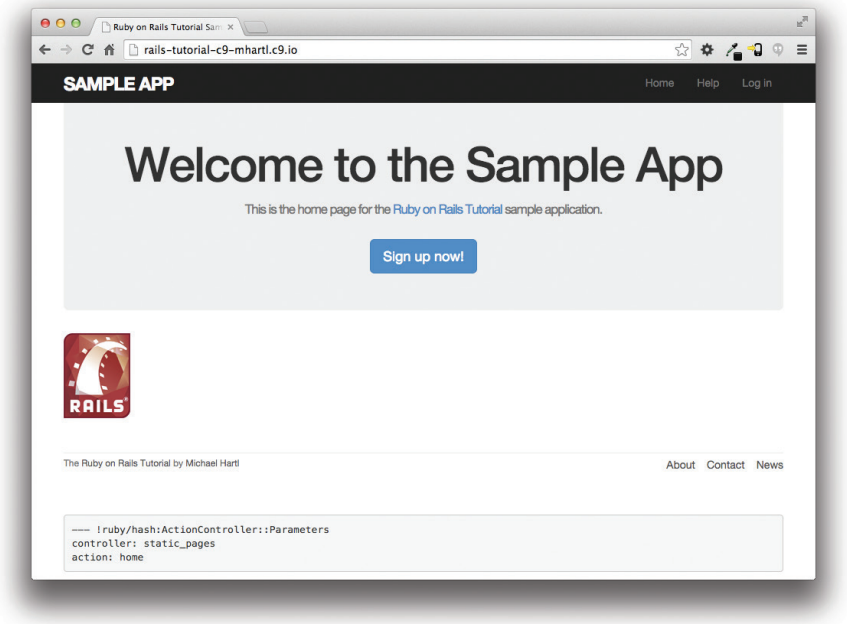


Рис. 7.3 ❖ Главная страница учебного приложения с отладочной информацией

Блок с отладочной информацией на рис. 7.3 содержит потенциально полезные сведения об отображаемой странице:

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

Это представление `params` на языке YAML[1], который, по сути, является хэшем и в данном случае идентифицирует контроллер и его метод. Еще один пример мы увидим в разделе 7.1.2.

---

[1] Отладочная информация в Rails отображается в формате YAML (рекурсивный акроним от «YAML Ain't Markup Language», – не язык разметки) – дружественного формата данных, удобочитаемого и для компьютеров, и для людей (https://ru.wikipedia.org/wiki/YAML).

7.1.2. Ресурс Users

Для создания страницы профиля, необходимо, чтобы в базе данных имелась информация о пользователе. В результате возникает проблема яйца и курицы: как создать пользователя до появления действующей страницы регистрации? К счастью, эта проблема уже решена: в разделе 6.3.4 мы создали запись User вручную с помощью консоли Rails, поэтому в базе данных должен быть один пользователь:

```
$ 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...">
```

(Если у вас сейчас нет ни одного пользователя, вернитесь к разделу 6.3.4 и завершите его.) Из вывода консоли видно, что этот пользователь имеет идентификатор id, равный 1, и теперь можно переходить к созданию страницы для отображения информации об этом пользователе. Мы будем следовать соглашениям REST-архитектуры (блок 2.2), что означает представление данных в виде *ресурсов*, которые могут создаваться, отображаться, изменяться и удаляться – четыре действия, соответствующих четырем основным операциям – `POST`, `GET`, `PATCH` и `DELETE` – в стандарте HTTP (блок 3.2).

В соответствии с принципами REST для ссылок на ресурсы обычно используются их имена и уникальные идентификаторы. Что это означает в контексте пользователей, о которых мы теперь думаем как о *ресурсах* Users, – мы должны показать пользователя с идентификатором 1, послав запрос `GET` по адресу URL `/users/1`. Тип запроса *неявно* подразумевает вызов метода действия `show` – когда используется поддержка REST, встроенная в Rails, `GET`-запросы автоматически обрабатываются действием `show`.

В разделе 2.2.1 мы видели, что URL страницы пользователя с идентификатором 1 имеет вид: /users/1. К сожалению, обращение к этому адресу прямо сейчас приведет к выводу сообщения об ошибке в журнал сервера (рис. 7.4).

```
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'
```

Рис. 7.4 ❖ Сообщение об ошибке в журнале сервера при попытке обратиться по адресу /users/1

Чтобы настроить маршрутизацию для /users/1, достаточно добавить всего одну строку в файл маршрутов (config/routes.rb):

```
resources :users
```

как показано в листинге 7.3.

Листинг 7.3 ❖ Добавление ресурса Users в файл маршрутов (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
```

Несмотря на то что сейчас мы хотели лишь создать страницу для вывода информации о пользователе, строка `resources :users` не просто добавляет действующий адрес /users/1, – она обеспечивает автоматический вызов *всех* методов действий для RESTful-ресурса Users[1] наряду с поддержкой большого количества именованных маршрутов (раздел 5.3.3). Получившийся комплекс URL, действий и именованных маршрутов показан в табл. 7.1. (Сравните с табл. 2.2.) В течение следующих трех глав мы охватим остальные записи в табл. 7.1, по мере реализации всех действий, необходимых для превращения Users в полноценный RESTful-ресурс.

Таблица 7.1 ❖ RESTful-маршруты для ресурса Users

HTTP-запрос	URL	Метод	Именованный маршрут	Назначение
GET	/users	index	users_path	Страница со списком всех пользователей
GET	/users/1	show	user_path(user)	Страница пользователя с идентификатором 1
GET	/users/new	new	new_user_path	Страница создания нового пользователя (регистрации)
POST	/users	create	users_path	Создание нового пользователя
GET	/users/1/edit	edit	edit_user_path(user)	Страница редактирования пользователя с идентификатором 1
PATCH	/users/1	update	user_path(user)	Изменение пользователя с идентификатором 1
DELETE	/users/1	destroy	user_path(user)	Удаление пользователя с идентификатором 1

[1] Мы получили действующую *маршрутизацию*, но соответствующие страницы пока могут не работать. Например, обращение к адресу /users/1/edit правильно будет передаваться в метод edit контроллера Users, но так как он пока отсутствует, попытка вызвать его завершится ошибкой.

Теперь у нас есть действующие маршруты, но страница по-прежнему отсутствует (рис. 7.5). Чтобы исправить это, создадим минимальную версию страницы профиля, которую дополним в разделе 7.1.4.

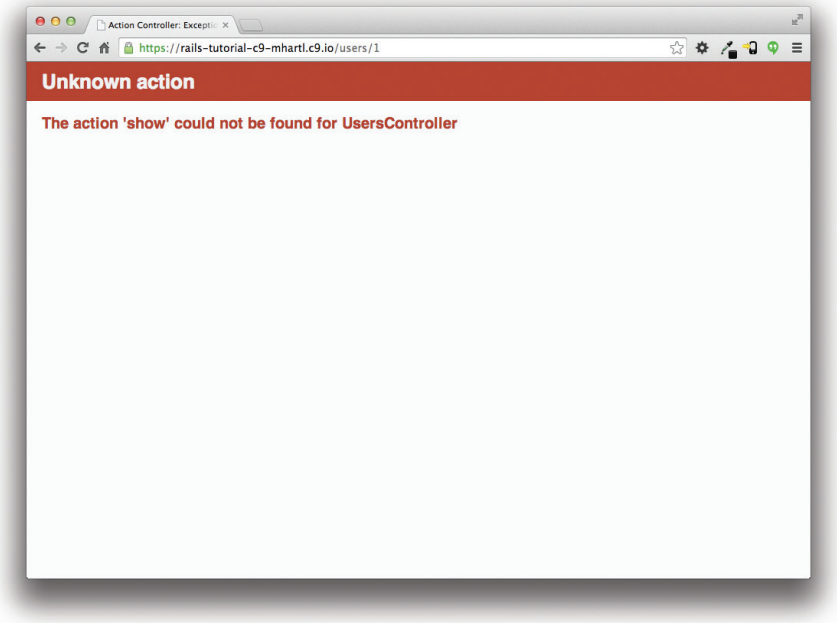


Рис. 7.5 ❖ Для URL /users/1 имеется маршрут, но отсутствует страница

Мы будем использовать стандартное для Rails размещение страницы, реализующей вывод информации о пользователе: `app/views/users/show.html.erb`. В отличие от представления `new.html.erb`, созданного генератором в листинге 5.28, файл `show.html.erb` в данный момент отсутствует, поэтому его нужно создать вручную и заполнить содержимым из листинга 7.4.

Листинг 7.4 ❖ Представление-заглушка для отображения информации о пользователе (app/views/users/show.html.erb)

```
<%= @user.name %>, <%= @user.email %>
```

В этом представлении используется встроенный Ruby для отображения имени и адреса электронной почты пользователя, и предполагается наличие переменной экземпляра `@user`. Конечно же, окончательная страница будет выглядеть совершенно иначе (и не будет публично демонстрировать адрес электронной почты).

Чтобы заставить представление работать, необходимо определить переменную `@user` в соответствующем методе действия `show` контроллера Users. Здесь, для извлечения пользователя из базы данных, используется метод `find` модели User (раздел 6.1.4), как показано в листинге 7.5.

Листинг 7.5 ❖ Контроллер Users с действием show (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
  end
end
```

Для получения идентификатора пользователя мы использовали переменную `params`. В результате передачи запроса контроллеру Users элемент `params[:id]` получит значение `1`, в итоге данный вызов метода `find` эквивалентен вызову `User.find(1)`, который мы видели в разделе 6.1.4. (Технически выражение `params[:id]` возвращает строку `"1"`, но `find` достаточно сообразителен, чтобы преобразовать ее в целое число.)

После создания представления и добавления метода маршрут /users/1 работает замечательно, как видно на рис. 7.6. (Если вы не перезапустили сервер Rails после добавления bcrypt, сделайте это сейчас.) Обратите внимание, что отладочная информация на рис. 7.6 подтверждает значение `params[:id]`:

```
---
action: show
controller: users
id: '1'
```

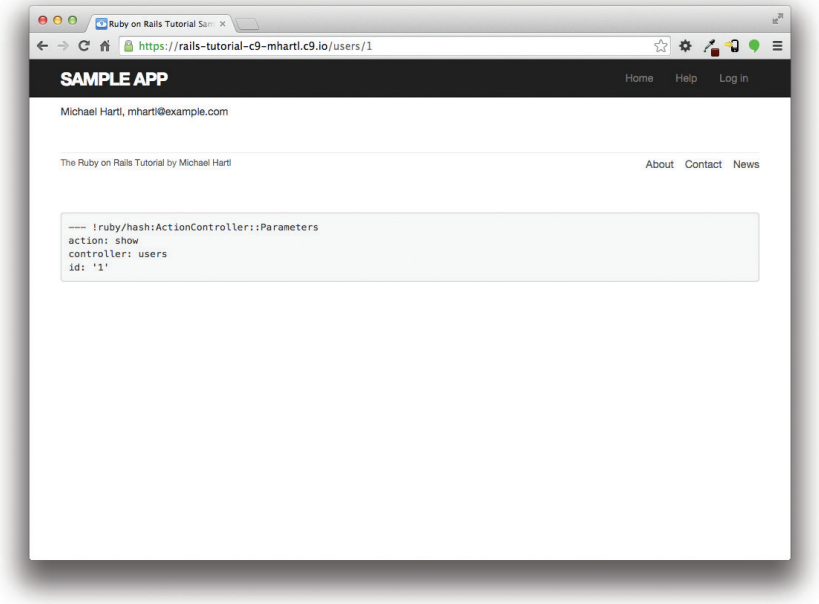


Рис. 7.6 ❖ Страница с информацией о пользователе после добавления ресурса Users

Именно поэтому вызов

```
User.find(params[:id])
```

в листинге 7.5 обнаруживает пользователя с идентификатором 1.

7.1.3. Отладчик

В разделе 7.1.2 мы видели, как отладочная информация помогает понять, что происходит в приложении. Начиная с Rails 4.2 появился еще более непосредственный способ получения отладочной информации с помощью гема byebug (листинг 3.2). Чтобы увидеть, как он работает, просто добавьте строку debugger в приложение, как показано в листинге 7.6.

Листинг 7.6 ❖ Контроллер Users с отладчиком (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
    debugger
  end

  def new
  end
end
```

Если теперь посетить адрес /users/1, сервер Rails вернет приглашение byebug:

```
(byebug)
```

Его можно рассматривать как аналог консоли Rails и выполнять команды, чтобы узнать состояние приложения:

```
(byebug) @user.name
"Example User"
(byebug) @user.email
"example@railstutorial.org"
(byebug) params[:id]
"1"
```

Чтобы выйти и продолжить выполнение приложения, нажмите **Ctrl-D**, затем удалите строку debugger из метода show (листинг 7.7).

Листинг 7.7 ❖ Контроллер Users с удаленной строкой отладчика (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
  end
end
```

Всякий раз, когда появляются какие-то сомнения, добавьте вызов `debugger` близко к той части кода, которая, по вашему мнению, вызывает проблемы. Исследование состояния системы с помощью `byebug` – весьма мощный метод выявления ошибок и интерактивной отладки приложения.

7.1.4. Аватар и боковая панель

Определив в предыдущем разделе заготовку страницы пользователя, теперь немного улучшим ее, добавив изображение пользователя и начальную реализацию боковой панели. Начнем с добавления «глобально распознаваемого аватара», или *граватара* (http://gravatar.com), к профилю пользователя[1]. Gravatar – это бесплатная служба, позволяющая пользователям загружать изображения и связывать их со своими адресами электронной почты. То есть это удобный способ добавить изображение пользователя, не связываясь с проблемами выгрузки изображений, их обрезкой и хранением; все, что нам нужно, – это создать правильный адрес URL изображения в службе Gravatar, используя адрес электронной почты, и соответствующий аватар появится автоматически. (Как реализовать выгрузку собственных изображений, рассказывается в разделе 11.4.)

Нам нужно определить вспомогательную функцию `gravatar_for`, которая будет возвращать аватар для данного пользователя, как показано в листинге 7.8.

Листинг 7.8 ❖ Представление, отображающее имя пользователя с и его аватар (app/views/users/show.html.erb)

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

По умолчанию методы, определенные в любом вспомогательном файле, автоматически доступны в любом представлении, но для удобства мы поместим `gravatar_for` в файл для вспомогательных функций, связанных с контроллером Users. Как отмечено в документации к Gravatar (http://en.gravatar.com/site/implement/hash/), адреса URL для доступа к аватарам основаны на MD5-хэше (https://ru.wikipedia.org/wiki/MD5) адреса электронной почты пользователя. В Ruby алгоритм MD5-хэширования реализуется с помощью метода `hexdigest`, который является частью библиотеки Digest:

```
>> email = "MHARTL@example.COM".
>> Digest::MD5::hexdigest(email.downcase)
=> "1fda4469bcbec3badf5418269ffc5968"
```

Поскольку адреса электронной почты нечувствительны к регистру (раздел 6.2.4), в отличие от MD5-хэшей, мы использовали метод `downcase`, чтобы

[1] В индуизме аватаром называют проявление божества в человеческом или животном образе. В более широком смысле термин аватар используется для обозначения некоего представления личности, особенно в виртуальной сфере.

передать аргумент в hexdigest в нижнем регистре. (Это не будет иметь значения в нашей книге из-за функции обратного вызова в листинге 6.31, которая уже преобразует адрес электронной почты в нижний регистр, но это необходимо в случае с gravatar_for, которая может получать адреса электронной почты из других источников.) Получившаяся вспомогательная функция gravatar_for представлена в листинге 7.9.

Листинг 7.9 ❖ Определение вспомогательного метода gravatar_for (app/helpers/users_helper.rb)

```
module UsersHelper

  # Возвращает граватар для данного пользователя.
  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
```

Код в листинге 7.9 возвращает тег img граватара с CSS-классом gravatar и атрибутом alt с именем пользователя (это особенно удобно для слабовидящих пользователей, которые пользуются экранным диктором).

Страница профиля изображена на рис. 7.7, и на ней показан граватар по умолчанию, так как адрес user@example.com не является действительным адресом электронной почты. (На самом деле, если зайти на example.com, можно увидеть, что этот домен зарезервирован для таких примеров, как этот.)

Чтобы отобразить собственный граватар в приложении, задействуем update_attributes (раздел 6.1.5), для замены адреса электронной почты пользователя другим, которым я могу управлять:

```
$ rails console
>> user = User.first
>> user.update_attributes(name: "Example User",
?>                        email: "example@railstutorial.org",
?>                        password: "foobar",
?>                        password_confirmation: "foobar")
=> true
```

Мы назначили пользователю адрес электронной почты example@railstutorial.org, который я связал с логотипом Rails Tutorial, как это видно на рис. 7.8.

Последний элемент, необходимый для завершения страницы, – это начальная версия боковой панели. Мы реализуем ее с помощью тега aside, он используется для элементов, дополняющих страницу, но также может использоваться отдельно. Добавим классы row и col-md-4, которые являются частью фреймворка Bootstrap. Код измененной страницы профиля пользователя представлен в листинге 7.10.

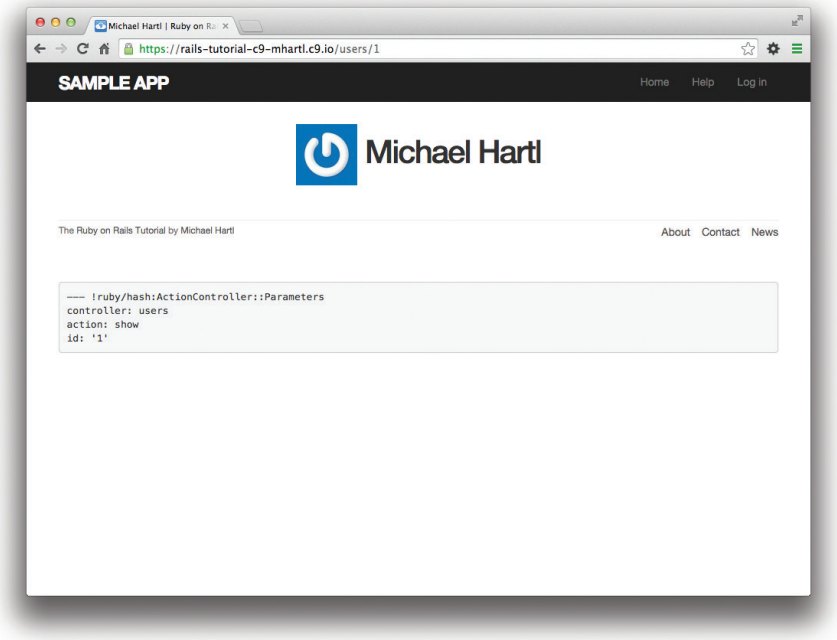


Рис. 7.7 ❖ Страница профиля пользователя с граватаром по умолчанию

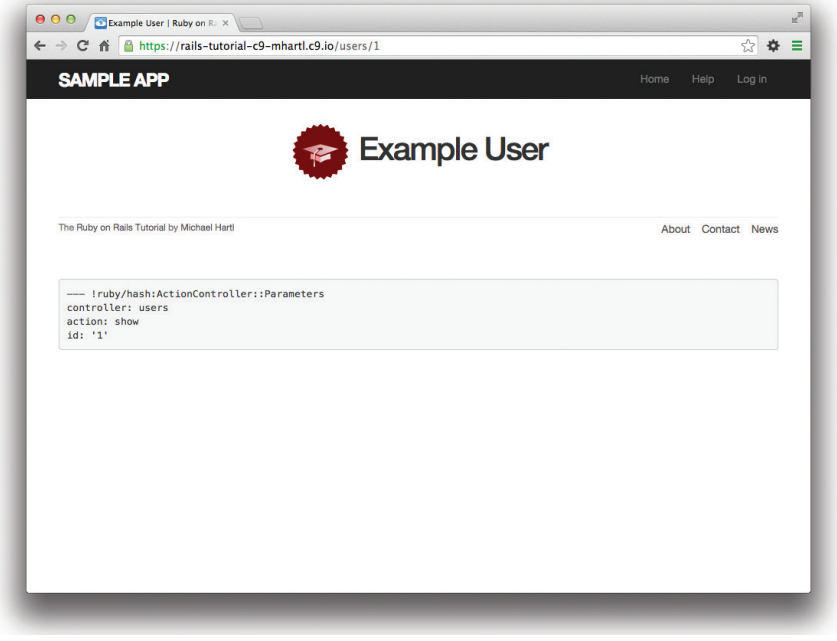


Рис. 7.8 ❖ Страница профиля пользователя с собственным граватаром

Листинг 7.10 ❖ Добавление боковой панели в представление show (app/views/users/show.html.erb)

```
<% provide(:title, @user.name) %>
<div class="row">
  <aside class="col-md-4">
    <section class="user_info">
      <h1>
        <%= gravatar_for @user %>
        <%= @user.name %>
      </h1>
    </section>
  </aside>
</div>
```

После размещения HTML-элементов и CSS-классов по своим местам можно применить стили к странице профиля (в том числе к боковой панели и граватару), добавив код SCSS из листинга 7.11[1]. (Обратите внимание на вложение CSS-правил в таблице, такое возможно благодаря препроцессору Sass, который обрабатывает ресурсы.) Получившаяся страница показана на рис. 7.9.

Листинг 7.11 ❖ SCSS-код с определением стилей для страницы профиля пользователя и боковой панели (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* боковая панель */

aside {
  section.user_info {
    margin-top: 20px;
  }
  section {
    padding: 10px 0;
    margin-top: 20px;
    &:first-child {
      border: 0;
      padding-top: 0;
    }
    span {
      display: block;
      margin-bottom: 3px;
      line-height: 1;
    }
    h1 {
      font-size: 1.4em;
```

[1] В листинге 7.11 определяется класс `.gravatar_edit`, который будет задействован в главе 9.

```
      text-align: left;
      letter-spacing: -1px;
      margin-bottom: 3px;
      margin-top: 0px;
    }
  }
}

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

.gravatar_edit {
  margin-top: 15px;
}
```

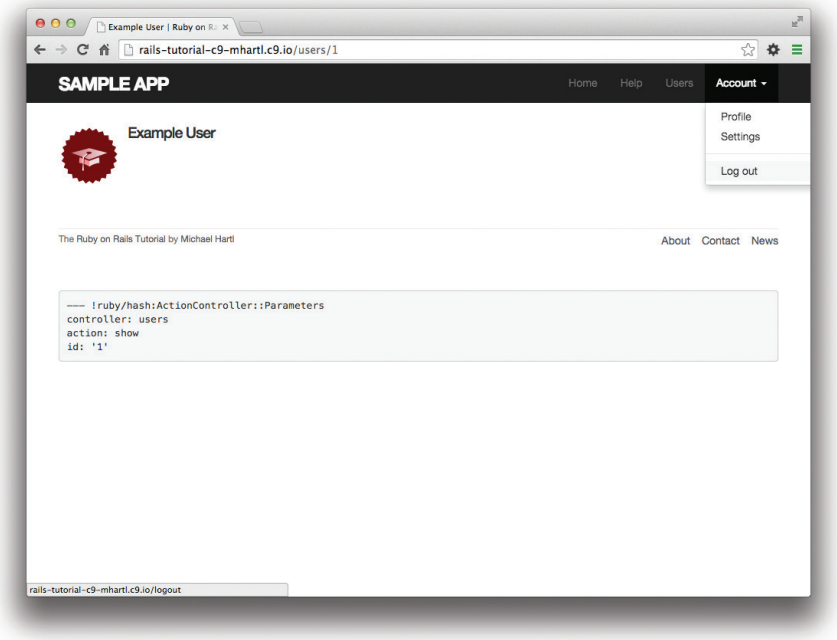


Рис. 7.9 ❖ Страница профиля пользователя с боковой панелью и CSS

7.2. Форма регистрации

Теперь, получив действующую (хоть и незавершенную) страницу профиля пользователя, можно приступить к созданию формы для регистрации на нашем сайте. Мы видели на рис. 7.10, что страница регистрации в настоящий момент пуста и совершенно бесполезна. Цель этого раздела – начать менять это печальное положение дел и создать форму, изображенную на рис. 7.11.

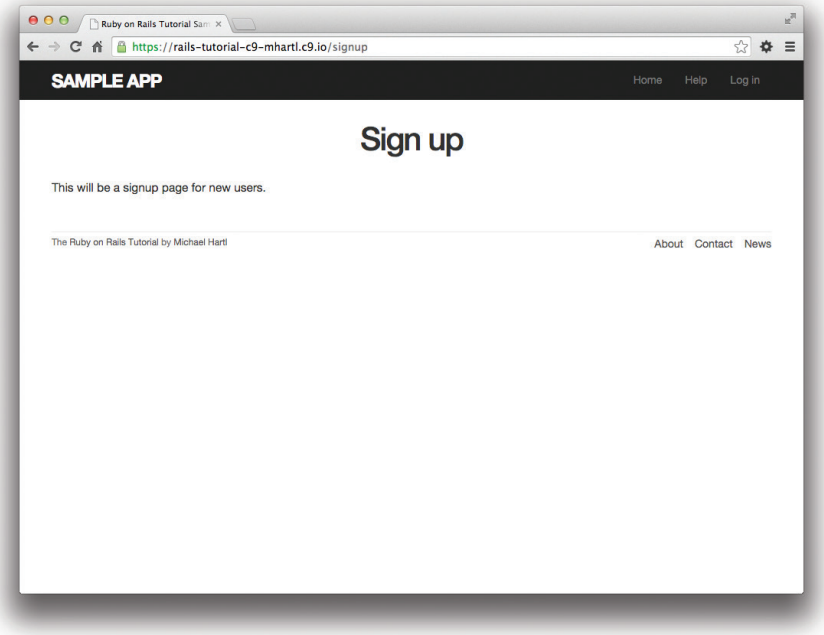


Рис. 7.10 ❖ Текущее состояние страницы регистрации /signup

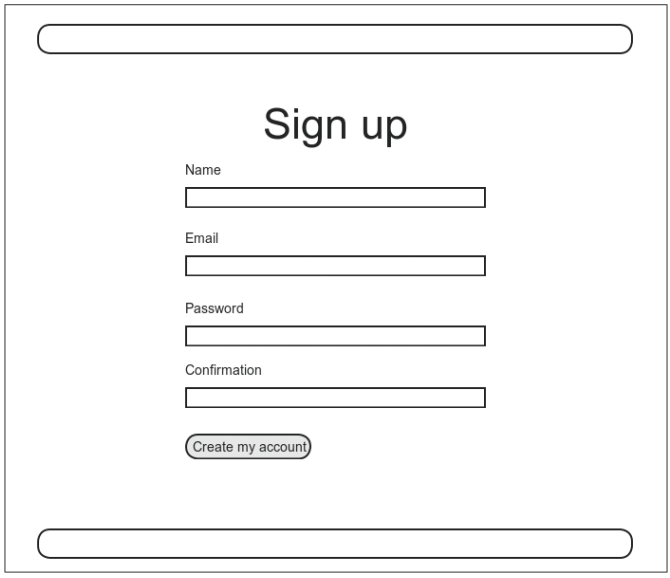


Рис. 7.11 ❖ Макет страницы регистрации пользователей

Так как мы собираемся добавить возможность создания новых пользователей через веб-интерфейс, удалим пользователя, созданного с помощью консоли в разделе 6.3.4. Для этого просто очистим базу данных с помощью Rake-задачи `db:migrate:reset`:

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

В некоторых системах может потребоваться перезагрузить сервер (**Ctrl-C**), чтобы изменения вступили в силу.

7.2.1. Применение form_for

Основой страницы регистрации является *форма* для отправки информации (имя, электронная почта, пароль, подтверждение). В Rails для этого существует вспомогательный метод `form_for`, принимающий объект Active Record и конструирующий форму на основе его атрибутов.

Напоминаю, что запросы к странице регистрации /signup передаются в метод `new` контроллера Users (листинг 5.33), поэтому сначала создадим объект User, который будет затем использоваться в качестве аргумента для `form_for`. Соответствующее определение переменной `@user` показано в листинге 7.12.

Листинг 7.12 ❖ Добавление переменной @user в метод new (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

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

Сама форма представлена в листинге 7.13. Мы подробно обсудим ее в разделе 7.2.2, а для начала придадим ей чуть более привлекательный вид с помощью стилей SCSS (листинг 7.14). Обратите внимание на повторное использование примеси `box_sizing` из листинга 7.2. После применения этих правил страница регистрации будет выглядеть, как на рис. 7.12.

Листинг 7.13 ❖ Форма регистрации новых пользователей (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>
```

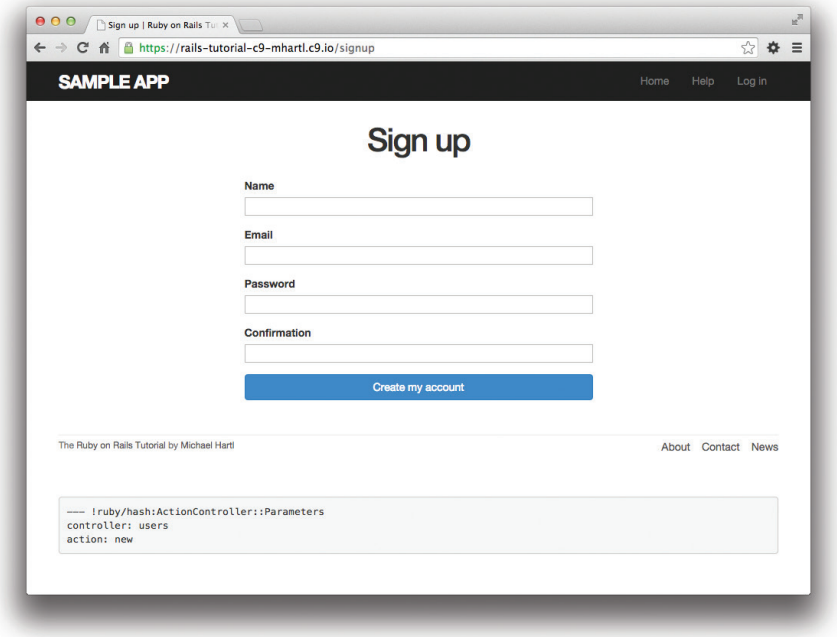


Рис. 7.12 ❖ Форма регистрации пользователя

Листинг 7.14 ❖ Стили CSS для формы регистрации (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* формы */

input, textarea, select, .uneditable-input {
  border: 1px solid #bbb;
  width: 100%;
  margin-bottom: 15px;
```

```
  @include box_sizing;
}

input {
  height: auto !important;
}
```

7.2.2. Разметка HTML формы регистрации

Чтобы понять, как действует форма в листинге 7.13, полезно разбить код на небольшие части. Сначала посмотрим на общую структуру, которая начинается с вызова `form_for` и заканчивается вызовом `end`:

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

Ключевое слово `do` указывает, что `form_for` принимает блок с одной переменной, которую мы назвали `f` (от слова «form»).

Как это зачастую происходит со вспомогательными функциями в Rails, нам не требуется знать подробности реализации, но нам нужно знать, что *делает* объект `f`: когда вызывается метод с именем, соответствующим элементу формы (http://www.w3schools.com/html/html_forms.asp), такому как текстовое поле, переключатель или поле ввода пароля, `f` возвращает код этого элемента, специально предназначенный для установки атрибута объекта `@user`. Другими словами, строки

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

создают HTML-элемент «текстовое поле с заголовком» для изменения атрибута `name` модели User.

Если посмотреть на исходный код созданной формы (для этого необходимо в браузере щелкнуть правой кнопкой мыши на странице и выбрать в контекстном меню пункт **Inspectelement** (Исследовать элемент)), можно увидеть разметку HTML, подобную той, что представлена в листинге 7.15. Давайте уделим немного времени обсуждению ее структуры.

Листинг 7.15 ❖ Разметка HTML для формы на рис. 7.12

```
<form accept-charset="UTF-8" action="/users" class="new_user"
      id="new_user" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />

  <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>
```

Начнем с внутренней структуры документа. Сравнивая листинги 7.13 и 7.15, можно заметить, что код на встроенном Ruby

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

производит разметку HTML:

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

Код

```
<%= f.label :email %>
<%= f.email_field :email %>
```

производит разметку HTML:

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

И код

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

производит разметку HTML:

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

Как показано на рис. 7.13, поля ввода имени и адреса электронной почты (`type="text"` и `type="email"`) просто отображают свое содержимое, в то время как поля ввода пароля (`type="password"`) скрывают ввод в целях безопасности, как это видно на рис. 7.13. (Преимущество поля ввода адреса электронной почты – в том, что некоторые системы обрабатывают его не так, как текстовое; например, на некоторых мобильных устройствах переход в поле `type="email"` вызовет появление специальной клавиатуры, оптимизированной для ввода адресов электронной почты.)

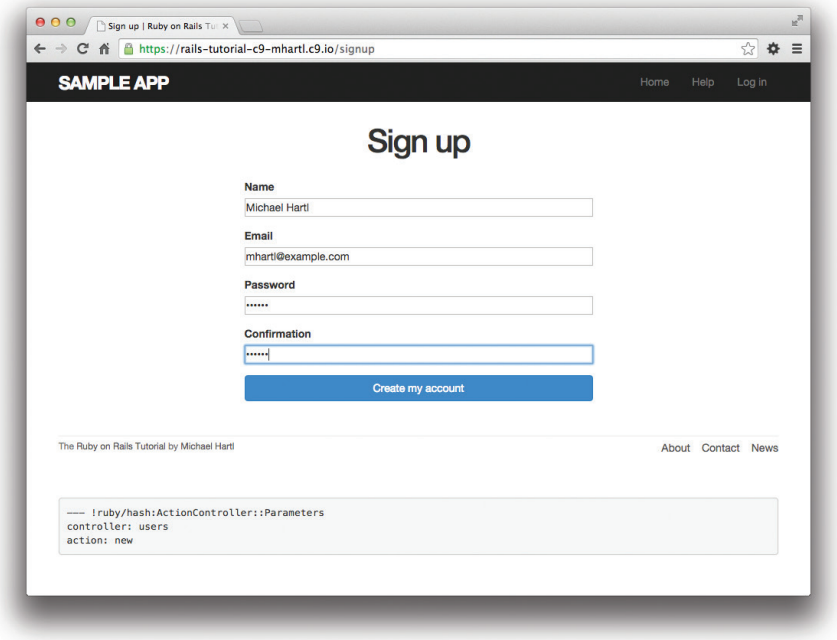


Рис. 7.13 ❖ Заполненная форма с полями text и password

Как мы увидим в разделе 7.4, ключом к созданию пользователя является специальный атрибут `name` в каждом теге `input`:

```
<input id="user_name" name="user[name]" - - - />
.
.
.
<input id="user_password" name="user[password]" - - - />
```

Значения `name` позволяют Rails построить хэш инициализации (через переменную `params`) для создания пользователя с использованием введенных значений, как будет показано в разделе 7.3.

Второй важный элемент – сам тег `form`. При его создании Rails использует объект `@user`: поскольку каждый объект в Ruby знает свой класс (раздел 4.4.1), Rails понимает, что `@user` принадлежит классу `User`; более того, так как `@user` – это *новый* пользователь, Rails знает, что необходимо построить форму с методом `post`, наиболее подходящим глаголом для создания нового объекта (блок 3.2):

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

Атрибуты `class` и `id`, по большому счету, не имеют особого значения; более важными являются `action="/users"` и `method="post"`. Вместе они формируют инструкцию для отправки HTTP-запроса `POST` по адресу URL `/users`. В следующих двух разделах мы увидим, к чему это приводит.

(Вероятно, вы также заметили код, который присутствует только внутри тега `form`:

```
<div style="display:none">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />
</div>
```

Этот код не отображается в браузере, он предназначен для внутреннего пользования фреймворком Rails, поэтому для нас не так важно понимать, что он делает. Вкратце: с помощью символа Юникода `✓` (галочка ✓) он заставляет браузеры принимать данные в правильной кодировке, кроме того, он содержит *аутентификационный токен*, который Rails использует для защиты от атак под названием cross-site request forgery (CSRF, подделка межсайтовых запросов)[1].)

7.3. Неудачная регистрация

Мы рассмотрели разметку HTML-формы на рис. 7.12 (листинг 7.15), но пока не охватили всех подробностей, и, чтобы лучше понять все тонкости, рассмотрим ситуацию неудачной регистрации. В этом разделе мы создадим форму, которая принимает недопустимые данные и вновь отображает страницу регистрации со списком ошибок, как для примера показано на рис. 7.14.

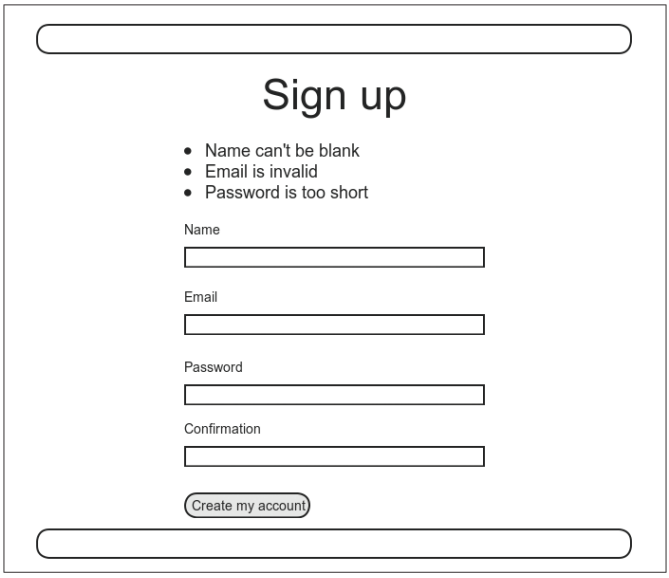


Рис. 7.14 ❖ Макет страницы со списком ошибок при неудачной регистрации

[1] См. статью на сайте Stack Overflow (http://stackoverflow.com/questions/941594/understanding-the-rails-authenticity-token), если вам интересны подробности.

7.3.1. Действующая форма

Как рассказывалось в разделе 7.1.2, добавление `resources :users` в файл `routes.rb` (листинг 7.3) автоматически обеспечивает Rails-приложение возможностью откликаться на адреса URL из табл. 7.1. В частности, запрос `POST` по адресу `/users` обрабатывается методом `create`. Этот метод должен использовать форму, отправленную посетителем, для создания нового объекта пользователя вызовом `User.new`, попытаться сохранить этого пользователя и в случае неудачи отобразить страницу регистрации для возможной повторной отправки формы. Начнем с того, что еще раз взглянем на код для формы регистрации:

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

Как было отмечено в разделе 7.2.2, этот код HTML посылает запрос `POST` по адресу URL /users.

Нашим первым шагом на пути к действующей форме регистрации будет добавление кода из листинга 7.16. В нем второй раз используется метод `render`, который мы впервые видели при обсуждении частичных шаблонов (раздел 5.1.3); как видите, `render` также может использоваться в методах контроллеров. Обратите внимание, что мы воспользовались этой возможностью, чтобы реализовать структуру `if-else` для раздельной обработки случаев удачной и неудачной регистраций в зависимости от значения, возвращаемого методом `@user.save`, – `true` или `false`, как мы видели в разделе 6.1.3.

Листинг 7.16 ❖ Метод create, способный обрабатывать неудачную регистрацию (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])     # Не окончательная реализация!
    if @user.save
      # Обработать успешное сохранение.
    else
      render 'new'
    end
  end
end
```

Обратите внимание на комментарий: это не окончательная реализация, но этого достаточно для начала. Мы закончим реализацию в разделе 7.3.2.

Лучший способ понять, как действует код в листинге 7.16, – *отправить* форму с недопустимыми регистрационными данными. Результат такой попытки показан

на рис. 7.15, а полная отладочная информация (с увеличенным размером шрифта) – на рис. 7.16. (На рис. 7.15 также показана *веб-консоль*, которая открывает Rails-консоль в браузере для содействия в отладке. Ее можно использовать, например, для изучения модели User, но в данном случае нам нужно исследовать переменную `params`, недоступную в веб-консоли.)

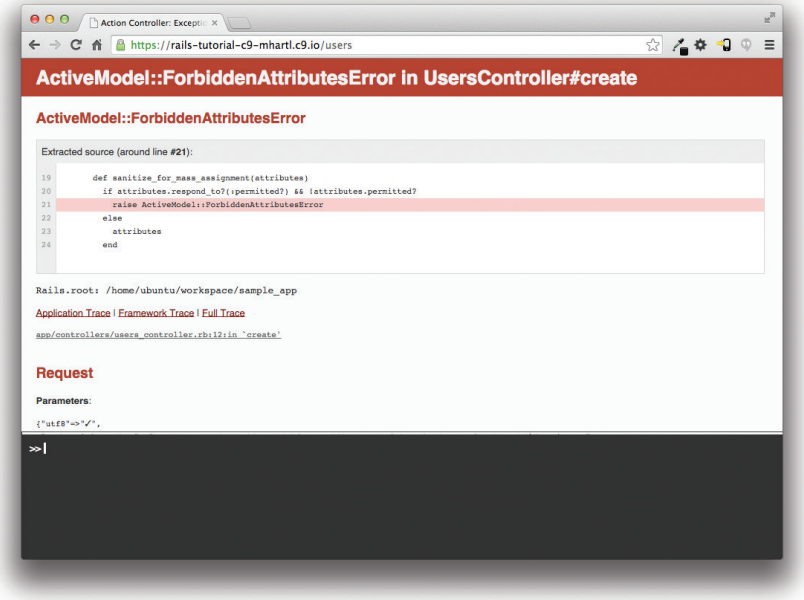


Рис. 7.15 ❖ Неудачная попытка регистрации

Чтобы получить более полное представление, как Rails обрабатывает регистрационные данные, давайте поближе познакомимся с элементом `user` в хэше с отладочной информацией (рис. 7.16):

```
"user" => { "name" => "Foo Bar",
            "email" => "foo@invalid",
            "password" => "[FILTERED]",
            "password_confirmation" => "[FILTERED]"
          }
```

Этот хэш передается в контроллер Users как часть хэша `params`, которая, как мы видели в разделе 7.1.2, содержит информацию о каждом запросе. Для адресов URL, таких как /users/1, в `params[:id]` хранится идентификатор пользователя (в этом примере число 1). Когда посылается регистрационная форма, хэш `params` содержит хэш хэшей – конструкцию, которую мы впервые увидели в разделе 4.3.3, где впервые познакомились с переменной `params`. Отладочная информация на рис. 7.16 показывает, что отправка формы дает в результате хэш `user` с атрибутами, соответствующими полям ввода формы, ключами в котором служат значения атрибутов `name` в тегах `input` (листинг 7.13); например, следующий тег `input`

```
<input id="user_email" name="user[email]" type="email" />
```

с атрибутом `name="user[email]"` соответствует атрибуту `email` в хэше `user`.

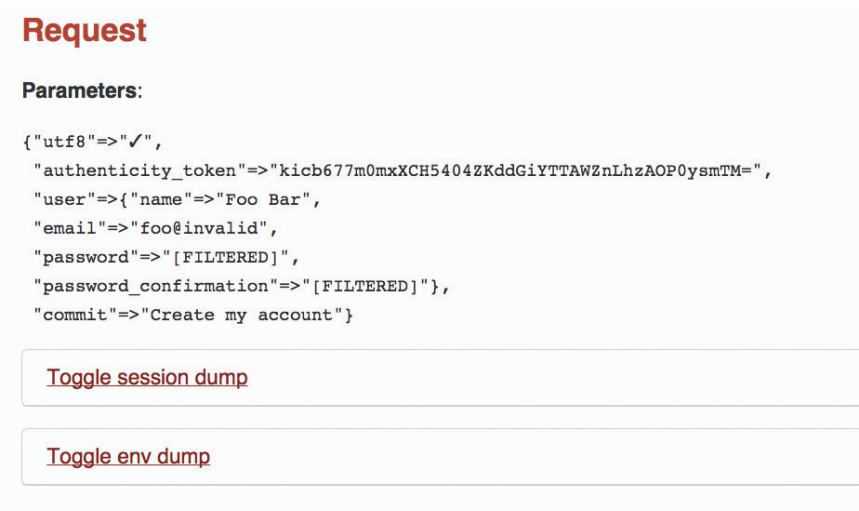


Рис. 7.16 ❖ Отладочная информация о неудачной попытке регистрации

В отладочном выводе хэш-ключи показаны в виде строк, но в контроллере Users к ним можно обращаться как к символам, то есть `params[:user]` на самом деле является хэшем атрибутов пользователя, который можно передать в вызов `User.new`, как впервые было показано в разделе 4.4.5 и затем в листинге 7.16. Это значит, что строка

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

практически эквивалентна вызову:

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

В предыдущих версиях Rails также можно было использовать инструкцию:

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

но подобный прием был небезопасным по умолчанию и требовал приложения немалых усилий, чтобы защитить базу данных приложения от возможного изменения злонамеренными пользователями. В версиях Rails выше 4.0 этот код вызывает ошибку (как показано на рис. 7.15 и рис. 7.16), то есть он безопасен по умолчанию.

7.3.2. Строгие параметры

В разделе 4.4.5 упоминалась идея *массового присваивания*, подразумевающая инициализацию Ruby-переменной хэшем значений, например:

```
@user = User.new(params[:user])    # Не окончательная реализация!
```

Комментарий в листинге 7.16 указывает, что это не окончательная реализация. Причина в том, что инициализация всего хэша `params` является *чрезвычайно* опасной – она вызывает передачу в `User.new` *всех* данных, отправленных пользователем. Например, представьте, что помимо текущих атрибутов модель User содержит атрибут `admin`, используемый для идентификации пользователей-администраторов сайта. (Мы реализуем такой атрибут в разделе 9.4.1.) Чтобы присвоить этому атрибуту значение `true`, нужно передать значение `admin='1'` в `params[:user]`, что легко выполнимо с помощью такого консольного HTTP-клиента, как `curl`. То есть, передавая весь хэш `params` в `User.new`, мы позволим любому пользователю получить права администратора, включив в веб-запрос параметр `admin='1'`.

Для решения этой проблемы предыдущие версии Rails использовали метод `attr_accessible` на уровне *модели*, и вы все еще можете видеть его в некоторых старых Rails-приложениях, но в Rails 4.0 предпочтительнее использовать так называемые *строгие параметры* на уровне контроллера. В этом случае можно указать, какие параметры являются *обязательными*, а какие – *разрешенными*. Кроме того, прямая передача хэша `params` (как в примере выше) приведет к ошибке, то есть теперь Rails-приложения по умолчанию невосприимчивы к уязвимостям массового присваивания. В данном случае мы можем потребовать наличия атрибута `:user` в хэше `params` и разрешить атрибуты `name`, `email`, `password` и `password_confirmation` (но только их). Это делается так:

```
params.require(:user).permit(:name, :email, :password, :password_confirmation)
```

Эта инструкция вернет версию хэша `params`, содержащую только разрешенные атрибуты (при этом вызовет ошибку при отсутствии атрибута `:user`).

Чтобы упростить использование данных параметров, введем вспомогательный метод `user_params` (возвращающий соответствующий инициализирующий хэш) и используем его вместо `params[:us- er]`:

```
@user = User.new(user_params)
```

Поскольку `user_params` будет использоваться только внутри контроллера Users и нет никакой надобности открывать к нему доступ через веб для внешних пользователей, мы сделаем его приватным, добавив ключевое слово `private`, как показано в листинге 7.17. (Подробнее спецификатор `private` обсуждается в разделе 8.4.)

Листинг 7.17 ❖ Использование строгих параметров вместо decreate (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      # Обработать успешное сохранение.
```

```
    else
      render 'new'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

Кстати, увеличенный отступ в определении метода `user_params` помогает визуально выделить методы, следующие за спецификатором `private`. (Опыт показывает, что это благоразумная привычка; в классах со множеством методов легко по ошибке сделать метод приватным, а это приведет к немалой путанице, когда он станет недоступным для вызова в соответствующем объекте.)

Теперь форма регистрации работает, по крайней мере она не сообщает об ошибках при попытке отправить ее. С другой стороны, как показано на рис. 7.17, она не дает никакой обратной связи при отправке недопустимых данных (исключая область отладки в среде разработки), а это может приводить в замешательство. Да и нового пользователя она до сих пор не создает. Мы решим первую проблему в разделе 7.3.3, а вторую – в разделе 7.4.

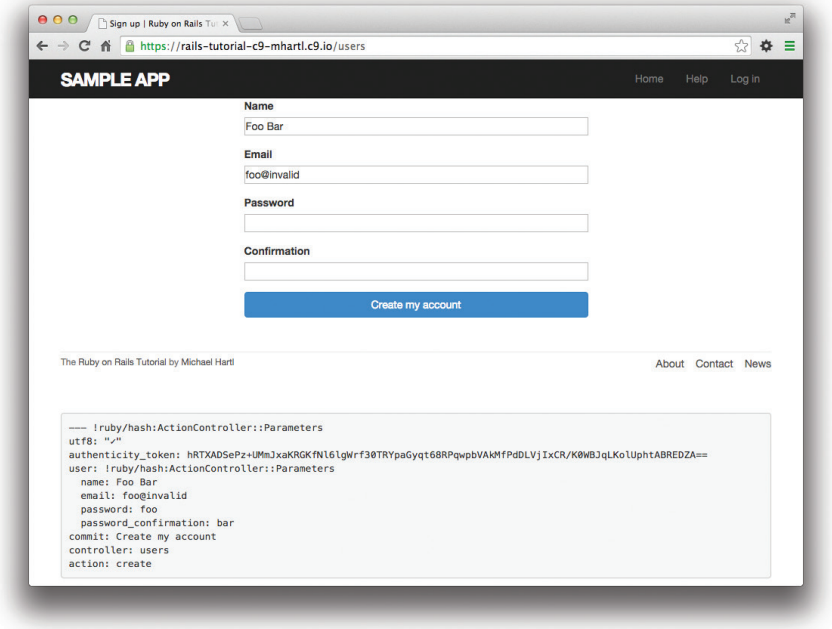


Рис. 7.17 ❖ Результат отправки формы регистрации с недопустимыми данными

7.3.3. Сообщения об ошибках при регистрации

В качестве финального шага в реализации сценария неудачной регистрации нового пользователя добавим сообщения об ошибках, описывающие проблемы, помешавшие регистрации. Для удобства Rails предоставляет такие сообщения автоматически, основываясь на проверках в модели User. Рассмотрим, например, попытку сохранения пользователя с неправильным адресом электронной почты и слишком коротким паролем:

```
$ 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)"]
```

Объект `errors.full_messages` (мы видели его краем глаза в разделе 6.2.2) содержит массив сообщений об ошибках.

Так же как в консольном сеансе выше, в случае неудачной попытки регистрации сохранения листинг 7.16 генерирует список сообщений об ошибках, связанных с объектом `@user`. Для их отображения в браузере мы организуем вывод частичного шаблона `error_messages`, добавив CSS-класс `form-control` (имеющий особое значение в Bootstrap) к каждому полю записи, как показано в листинге 7.18. Стоит отметить, что этот шаблон – только первая попытка; окончательная версия будет представлена в разделе 11.3.2.

Листинг 7.18 ❖ Отображение сообщений об ошибках в форме регистрации (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>
```

Обратите внимание, что здесь методу render передается частичный шаблон с именем 'shared/error_messages' – это общепринятое соглашение по размещению частичных шаблонов в выделенном каталоге shared/, которые планируется использовать в представлениях многих контроллеров. (В разделе 9.1.1 мы увидим, где используется это соглашение.) Значит, необходимо создать новый каталог app/views/shared командой mkdir (табл. 1.1):

```
$ mkdir app/views/shared
```

Затем в обычном текстовом редакторе следует создать файл шаблона _error_messages.html.erb. Содержимое шаблона приводится в листинге 7.19.

Листинг 7.19 ❖ Шаблон для отображения сообщений об ошибках (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 %>
```

Этот шаблон содержит несколько новых для нас конструкций, включая два метода объекта errors. Первый из них – count, который просто возвращает количество ошибок:

```
>> user.errors.count
=> 2
```

Второй новый метод – any?, который (вместе с empty?) составляет пару взаимодополняющих методов:

```
>> user.errors.empty?
=> false
>> user.errors.any?
=> true
```

Метод empty?, который мы впервые увидели в разделе 4.2.3, когда говорили о строках, также поддерживается объектом errors, возвращая true, когда объект пуст, и false в противном случае. Метод any? действует как простая противоположность методу empty?, он возвращает true при наличии элементов в объекте и false в противном случае. (Кстати, все эти методы – count, empty? и any? – также реализованы в массивах Ruby. Мы найдем применение этому факту в разделе 11.2.)

Другой новинкой является вспомогательная функция pluralize. По умолчанию она недоступна в консоли, но мы можем явно добавить ее, подключив модуль ActionView::Helpers::TextHelper[1]:

```
>> include ActionView::Helpers::TextHelper
>> pluralize(1, "error")
=> "1 error"
>> pluralize(5, "error")
=> "5 errors"
```

Функция pluralize принимает целочисленный аргумент и возвращает правильную версию множественного числа для слова во втором аргументе. В основе этого метода лежит мощный механизм преобразования слов во множественное число (в том числе слов, являющихся исключением из правил):

```
>> pluralize(2, "woman")
=> "2 women"
>> pluralize(3, "erratum")
=> "3 errata"
```

В результате код

```
<%= pluralize(@user.errors.count, "error") %>
```

возвращает "0 errors", "1 error", "2 errors" и т. д., в зависимости от количества ошибок, позволяя избежать грамматически неверных фраз вроде "1 errors" (удручающе распространенная ошибка в приложениях и в сети).

Обратите внимание, что листинг 7.19 содержит CSS-идентификатор error_explanation для оформления сообщений об ошибках. (Как рассказывалось в разделе 5.1.2, для применения стиля по атрибуту id в CSS используется знак решетки #.) Кроме того, после неудачной попытки отправить форму Rails автоматически помещает поля с ошибками в элементы div с CSS-классом field_with_errors. Это позволяет определить свой стиль оформления сообщений об ошибках с помощью разметки SCSS, показанной в листинге 7.20, где используется Sass-функция @extend, включающая поддержку Bootstrap-класса has-error.

Листинг 7.20. Стили CSS для оформления сообщений об ошибках (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* формы */
.
.
.
#error_explanation {
```

[1] Я выяснил это, изучив описание pluralize в справочнике по Rails API (http://api.rubyonrails.org/v4.2.0/classes/ActionView/Helpers/TextHelper.html#method-i-pluralize).

```
  color: red;
  ul {
    color: red;
    margin: 0 0 30px 0;
  }
}

.field_with_errors {
  @extend .has-error;
  .form-control {
    color: $state-danger-text;
  }
}
```

Теперь при попытке отправить недопустимую регистрационную информацию будут появляться сообщения об ошибках (рис. 7.18). Поскольку сообщения генерируются средствами проверки модели, они автоматически изменятся, если вы когда-нибудь поменяете свое мнение о, скажем, формате адресов электронной почты или минимальной длине паролей. (Примечание: так как обе проверки – `presence` и `has_secure_password` – определяют пустые пароли, сейчас форма регистрации выводит два сообщения об ошибке при отправке пустого пароля. Мы могли бы поработать с сообщениями напрямую, чтобы устранить эту проблему, но, к счастью, она автоматически будет решена при добавлении `allow_nil:true` в разделе 9.1.4.)

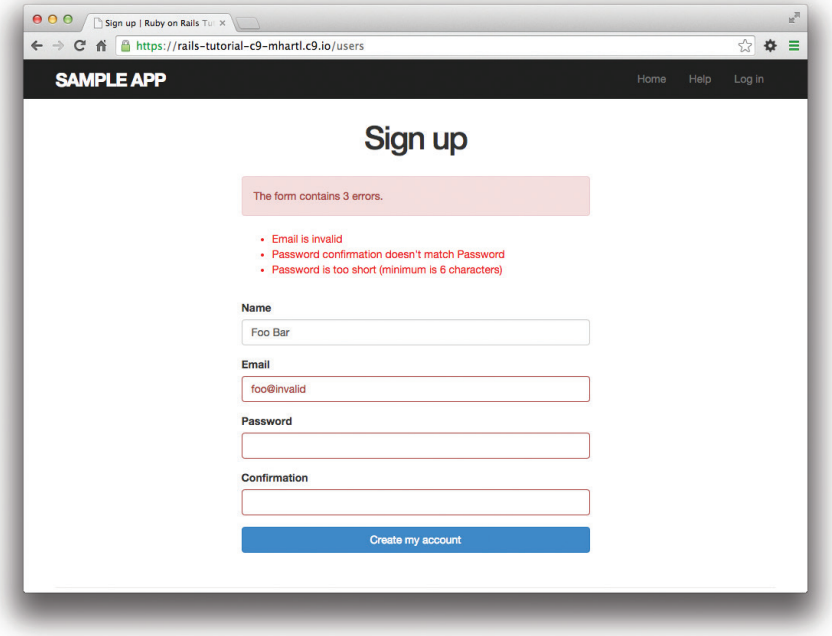


Рис. 7.18 ❖ Сообщения об ошибках в случае неудачной попытки регистрации

7.3.4. Тесты для неудачной регистрации

До появления мощных веб-фреймворков с полным набором средств тестирования разработчикам часто приходилось тестировать формы вручную. Например, для тестирования страницы регистрации нам пришлось бы открыть страницу в браузере, заполнить форму допустимыми и недопустимыми данными и проверить поведение приложения в каждом случае. Кроме того, нам пришлось бы проделывать это каждый раз при изменении приложения. Этот процесс весьма утомителен и чреват ошибками.

К счастью, в Rails можно писать тесты для автоматического тестирования форм. В этом разделе мы напишем один такой тест для проверки правильного поведения при отправке недопустимой формы; в разделе 7.4.4 мы напишем аналогичный тест для допустимой формы.

Сначала сгенерируем файл интеграционных тестов для процедуры регистрации пользователей и назовем его `users_signup` (в соответствии с соглашением о выборе имен ресурсов во множественном числе):

```
$ rails generate integration_test users_signup
      invoke  test_unit
      create    test/integration/users_signup_test.rb
```

(Этот же файл мы будем использовать в разделе 7.4.4 для тестирования успешной регистрации.)

Основная цель теста – убедиться, что щелчок на кнопке отправки формы не создаст нового пользователя, если форма содержит недопустимую информацию. (Тестирование сообщений об ошибках я оставляю вам в качестве самостоятельного упражнения (раздел 7.7).) Для этого нужно проверить *количество* пользователей, об этом внутри тестов позаботится метод `count`, доступный для любого класса Active Record, в том числе и User:

```
$ rails console
>> User.count
=> 0
```

Метод `User.count` возвращает 0, так как мы сбросили базу данных в начале раздела 7.2. Как и в разделе 5.3.4, применим для тестирования элементов HTML метод `assert_select`, позаботившись о проверке только тех элементов, которые вряд ли изменятся в будущем.

Начнем с получения страницы регистрации с помощью `get`:

```
get signup_path
```

Чтобы протестировать отправку формы, необходимо отправить запрос `POST` по маршруту `users_path` (табл. 7.1), эту операцию выполняет функция `post`:

```
assert_no_difference 'User.count' do
  post users_path, user: { name: "",
                           email: "user@invalid",
                           password: "foo",
                           password_confirmation: "bar" }
end
```

Мы включили хэш `params[:user]`, необходимый для вызова `User.new` в методе `create` (листинг 7.17). Завернув вызов `post` в метод `assert_no_difference` со строковым аргументом `'User.count'`, мы организовали сравнение значения `User.count` до и после выполнения блока `assert_no_difference`. Это эквивалентно сохранению количества пользователей, отправке данных и сравнению нового и прежнего количества пользователей:

```
before_count = User.count
post users_path, ...
after_count = User.count
assert_equal before_count, after_count
```

Хотя эти две последовательности равнозначны, применение `assert_no_difference` позволяет получить более ясный и идиоматически правильный код Ruby.

Стоит отметить, что функции `get` и `post`, использованные выше, технически не связаны между собой, и на самом деле не обязательно получать страницу регистрации перед отправкой формы. Тем не менее я предпочитаю указывать оба запроса для концептуальной ясности и перепроверки безошибочного отображения формы регистрации.

Собрав все вместе, получим тест, представленный в листинге 7.21. В нем также содержится вызов `assert_template` для проверки повторного обращения к методу `new` при ошибочной регистрации. Проверку появления сообщений об ошибках я оставляю вам в качестве самостоятельного упражнения (раздел 7.7).

Листинг 7.21 ❖ Тест провальной регистрации **ЗЕЛЕНЫЙ**
(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'
  end
end
```

Так как код приложения был написан до интеграционных тестов, набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 7.22 ❖. **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

7.4. Успешная регистрация

Реализовав сценарий отправки формы с недопустимой информацией, можно перейти к заключительному этапу и реализовать сохранение нового пользователя (в случае, если была отправлена допустимая информация). Сначала попробуем сохранить пользователя; если все пройдет успешно, информация автоматически будет записана в базу данных; затем *переадресуем* браузер на страницу профиля пользователя (с дружеским приветствием), как показано на рис. 7.19. Если попытка сохранения потерпит неудачу, просто отступим к сценарию, разработанному в разделе 7.3.

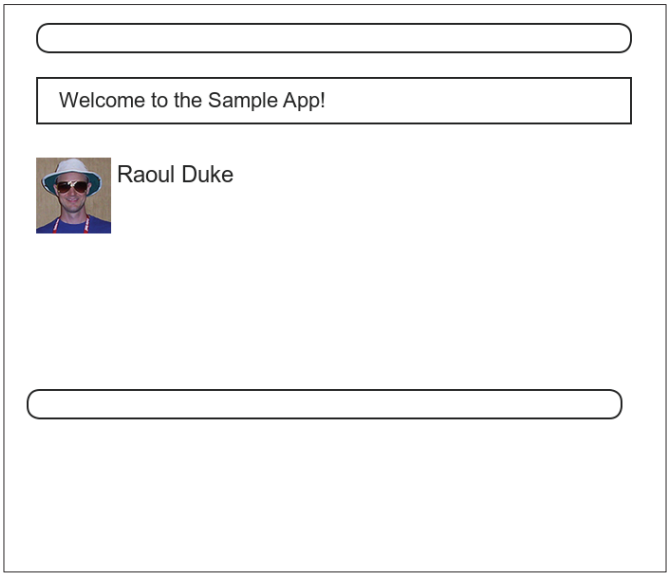


Рис. 7.19 ❖ Макет страницы, отображаемой после успешной регистрации

7.4.1. Окончательная форма регистрации

Чтобы закончить работу с формой регистрации, необходимо заполнить закомментированный раздел в листинге 7.17 соответствующим кодом. Сейчас форма терпит неудачу, даже при отправке допустимой информации. Как показано на рис. 7.20, это связано с тем, что по умолчанию Rails стремится отобразить соответствующее представление, но шаблона представления для метода `create` сейчас не существует.

Вместо того чтобы отображать страницу, соответствующую успешной регистрации пользователя, мы будем *переадресовывать* браузер на другую страницу. Для этого последуем общему правилу переадресации на страницу профиля вновь созданного пользователя, хотя вполне можно переадресовать и на главную страницу. Переадресацию будет выполнять метод `redirect_to`, как показано в листинге 7.23.

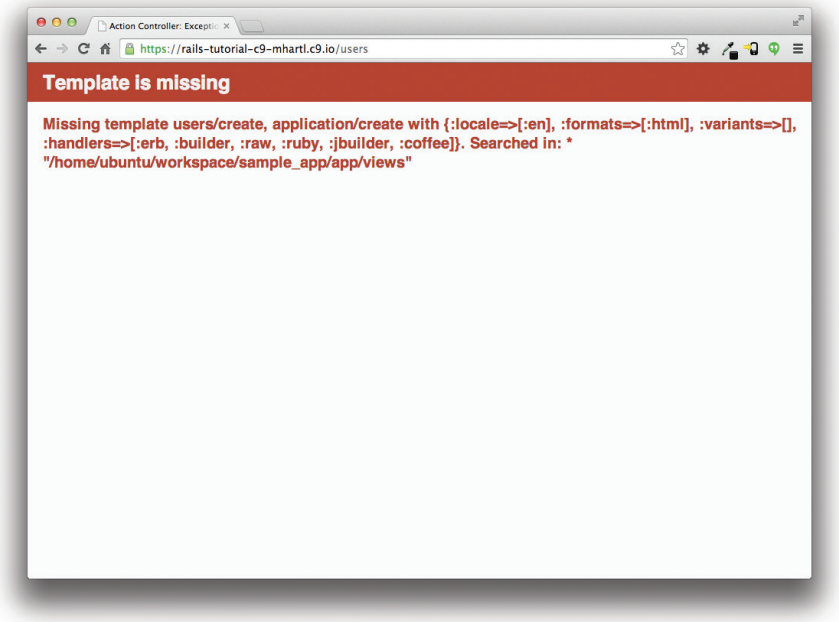


Рис. 7.20 ❖ Страница с ошибками, отображаемая при попытке отправить форму с допустимой информацией

Листинг 7.23 ❖ Метод create с сохранением и переадресацией (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
```

Мы написали

```
redirect_to @user
```

хотя могли использовать эквивалентную инструкцию

```
redirect_touser_url(@user)
```

Такое возможно, потому что Rails автоматически определяет, что перенаправление `redirect_to @user` должно быть выполнено в `user_url(@user)`.

7.4.2. Кратковременные сообщения

Теперь форма регистрации действительно работает, но перед отправкой регистрационной информации добавим одну маленькую деталь, широко используемую в веб-приложениях: сообщение, которое появляется на следующей странице (в данном случае приветствие новому пользователю) и затем исчезает либо при переходе к следующей странице, либо при перезагрузке текущей.

В Rails для отображения кратковременных сообщений используется специальный метод `flash`, с которым можно обращаться как с обычным хэшем. Rails придерживается соглашения, согласно которому применяет ключ `:success` к сообщениям об успехе операции (листинг 7.24).

Листинг 7.24 ❖ Добавление всплывающего сообщения в процедуру регистрации пользователя (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
```

После создания кратковременного сообщения его можно отобразить на первой странице после перенаправления. Для этого необходимо перебрать сообщения

в хэше flash и вставить необходимые в макет сайта. В разделе 4.3.3 мы уже видели, что обойти значения хэша можно с помощью переменной 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.
```

Следуя этому шаблону, можно организовать отображение содержимого хэша flash на всем сайте:

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

(Получилась неудобоваримая комбинация HTML и ERb; превращение ее в более ясную и понятную конструкцию я оставляю в качестве самостоятельного упражнения (раздел 7.7).) Следующий код на встроенном Ruby

```
alert-<%= message_type %>
```

создает CSS-класс в соответствии с типом сообщения. То есть для сообщения :success будет создан класс:

```
alert-success
```

(Ключ :success – это символ, но встроенный Ruby автоматически преобразует его в строку "success" перед вставкой в шаблон.) Используя разные классы для каждого ключа, можно применять разные стили к сообщениям разных типов. Например, в разделе 8.1.4 мы будем использовать flash[:danger], чтобы сообщить о неудачной попытке входа на сайт[1]. (По сути, однажды мы уже использовали alert-danger для оформления элемента div с сообщением об ошибке в листинге 7.19.) Bootstrap CSS поддерживает стили для четырех таких flash-классов (success, info, warning и danger), и у нас будет повод использовать все их в процессе разработки учебного приложения.

Так как сообщение уже вставлено в шаблон, в результате выполнения

```
flash[:success] = "Welcome to the Sample App!"
```

получается следующая разметка HTML:

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

---

[1] На самом деле будем использовать тесно связанный метод flash.now, но мы отложим эти тонкости, пока они нам не понадобятся.

Добавив код на встроенном Ruby, о котором говорилось выше, в шаблон сайта, получаем результат, представленный в листинге 7.25.

Листинг 7.25 ❖ Добавление содержимого переменной flash в шаблон сайта (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. Первая регистрация

Результат проделанной работы можно увидеть, зарегистрировав первого пользователя с именем «Rails Tutorial» и адресом электронной почты «example@railstutorial.org» (рис. 7.21). Получившаяся страница (рис. 7.22) отображает приветственное сообщение после успешной регистрации, оформленное в приятных зеленых тонах, определяемых классом `success`, добавленное CSS-фреймворком Bootstrap в разделе 5.1.2. (Если вместо этого вы получили сообщение об ошибке, указывающее, что адрес электронной почты уже занят, выполните Rake-задачу `db:migrate:reset`, как описывалось в разделе 7.2, перезапустите сервер разработки и повторите процедуру регистрации.) После перезагрузки страницы профиля кратковременное сообщение исчезнет, как и было обещано (рис. 7.23).

Теперь можно проверить базу данных, просто чтобы еще раз убедиться, что новый пользователь действительно создан:

```
$ 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$10$zthScEx9x6EkuLa4NolGye6OOZgrkp1B6LQ12pTHlNB...">
```

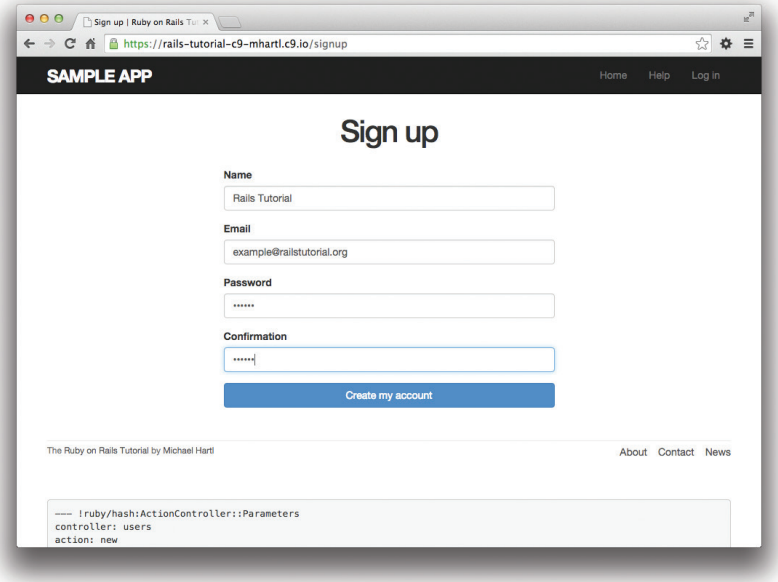


Рис. 7.21 ❖ Ввод информации для регистрации первого пользователя

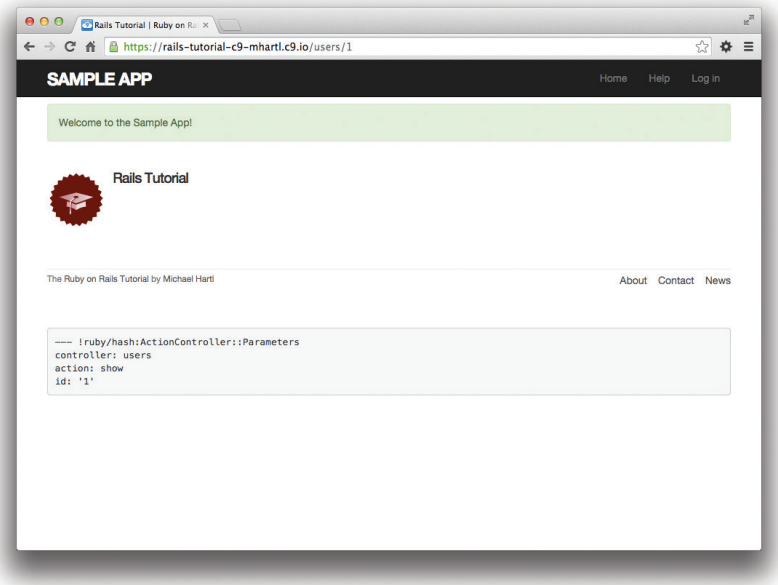


Рис. 7.22 ❖ Результат успешной регистрации пользователя с кратковременным сообщением

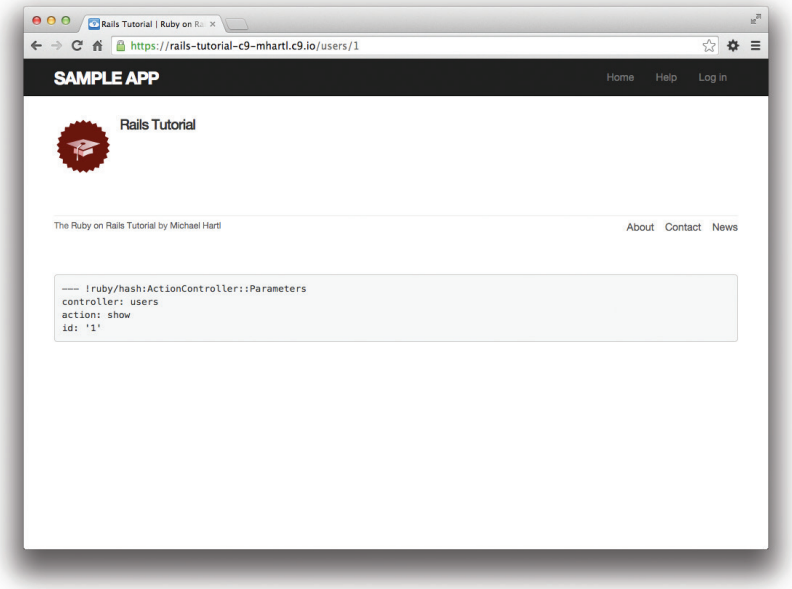


Рис. 7.23 ❖ Профиль пользователя с исчезнувшим кратковременным сообщением

7.4.4. Тесты для успешной отправки формы

Прежде чем продолжить, напишем тесты для сценария успешной отправки формы, чтобы проверить поведение приложения и предотвратить регрессии. По аналогии с тестами для сценария неудачной регистрации в разделе 7.3.4, основная цель – проверить содержимое базы данных. Сейчас мы проверяем ситуацию отправки допустимой информации, и нам нужно убедиться, что пользователь был создан. По аналогии с тестом в листинге 7.21, где использовалась инстукция:

```
assert_no_difference 'User.count' do
  post signup_path, ...
end
```

мы будем использовать здесь похожий метод `assert_difference`:

```
assert_difference 'User.count', 1 do
  post_via_redirect users_path, ...
end
```

Так же как `assert_no_difference`, метод `assert_difference` принимает в первом аргументе строку `'User.count'` и сравнивает значение `User.count` до и после выполнения блока. Второй (необязательный) аргумент определяет величину разности (в данном случае 1).

Заменив `assert_no_difference` на `assert_difference` в листинге 7.21, мы получим тест в листинге 7.26. Обратите внимание на использование варианта `post_via_`

redirect для передачи запроса post по маршруту users_path. Так мы организовали переадресацию к отображению шаблона 'users/show' после отправки формы. (Неплохо было бы написать тест для кратковременного сообщения, но я оставлю это вам в качестве самостоятельного упражнения (раздел 7.7).)

Листинг 7.26 ❖ Тест успешной регистрации **ЗЕЛЕНЫЙ**
(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'
  end
end
```

Обратите внимание, что тест в листинге 7.26 также проверяет отображение шаблона 'users/show' после успешной регистрации. Для прохождения этого теста необходима корректная работа маршрутов Users (листинг 7.3), метода show (листинг 7.5) и представления show.html.erb (листинг 7.8). Как результат строка

```
assert_template 'users/show'
```

надежно проверяет практически все, что имеет отношение к странице профиля пользователя. Подобный вид сквозного покрытия важных функций приложения показывает, почему интеграционные тесты так полезны.

7.5. Профессиональное развертывание

Теперь, когда у нас имеется действующая страница регистрации, самое время развернуть приложение на действующем сервере. Первые попытки развернуть приложение мы предприняли еще в главе 3, но сейчас впервые оно *действительно* умеет что-то делать, поэтому воспользуемся этой возможностью и развернем его как настоящие профессионалы. В частности, добавим в приложение важную функцию и сделаем процедуру регистрации защищенной, а также заменим первоначальный веб-сервер на тот, который подходит для реального использования.

В качестве подготовки к развертыванию объединим изменения с основной ветвью:

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

7.5.1. Поддержка SSL

При отправке формы регистрации, разработанной в этой главе, имя, адрес электронной почты и пароль пересылаются по сети и, следовательно, могут быть перехвачены. Это серьезная брешь в системе безопасности приложения, а устранить ее можно при помощи поддержки защищенных сокетов (Secure Sockets Layer, SSL)[1] для шифрования всей уязвимой информации до того, как она покинет локальный браузер. Хотя мы можем использовать поддержку SSL только для страницы регистрации, проще будет задействовать ее по всему сайту, получив заодно дополнительные преимущества в виде защищенного входа пользователя на сайт (глава 8) и иммунитета к критичной уязвимости *session hijacking* (перехват сеанса), которую мы рассмотрим в разделе 8.4.

Чтобы включить поддержку SSL, достаточно раскомментировать одну строку в `production.rb`, конфигурационном файле для приложений, развертываемых в эксплуатационном окружении. Как показано в листинге 7.27, нам требуется лишь настроить переменную `config`.

Листинг 7.27 ❖ Включение поддержки SSL в приложении (config/environments/production.rb)

```
Rails.application.configure do
  .
  .
  .
  # Включить доступ к приложению только через SSL, использовать защищенный
  # транспортный уровень и защищенные сокеты.
  config.force_ssl = true
  .
  .
  .
end
```

Теперь необходимо настроить поддержку SSL на удаленном сервере. Включение поддержки SSL на сайте предполагает покупку и настройку *SSL-сертификата* для вашего домена. Это весьма большой объем работы, но, к счастью, здесь он нам не понадобится: приложения, работающие в домене Heroku (такие как учебное приложение), могут «упасть на хвост» компании Heroku и использовать ее SSL-сертификат. Поэтому, когда мы развернем приложение в разделе 7.5.2, поддержка SSL автоматически будет доступна. (Если вы захотите включить поддержку SSL в собственном домене, например `www.example.com`, зайдите на страницу: https://www.nic.ru/dns/service/ssl/get_certificate.html.)

---

[1] Технически правильнее называть SSL как TLS (Transport Layer Security – безопасность транспортного уровня), но все, кого я знаю, по-прежнему говорят «SSL».

7.5.2. Действующий веб-сервер

После включения поддержки SSL необходимо настроить приложение для использования действующего веб-сервера. По умолчанию Heroku пользуется сервером WEBrick, написанном на языке Ruby, который легко настраивается и запускается, но он плохо справляется с большими нагрузками (https://devcenter.heroku.com/articles/ruby-default-web-server). Поэтому WEBrick не подходит для целей эксплуатации, и мы заменим его на Puma – HTTP-сервер, способный обрабатывать большое количество входящих запросов (https://devcenter.heroku.com/articles/deploying-rails-applications-with-the-puma-web-server).

Чтобы добавить новый веб-сервер, просто последуем инструкциям в документации Heroku. Сначала добавим в `Gemfile` гем `puma` (листинг 7.28). Так как он нам не нужен на локальном компьютере, поместим его в группу `:production`.

Листинг 7.28 ❖ Добавление Puma в 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
```

Мы настроили компоновщик так, что он не будет устанавливать гемы для эксплуатационного окружения (раздел 3.1), поэтому код в листинге 7.28 не добавит никаких гемов в окружение разработки, но нам нужно запустить его, чтобы обновить `Gemfile.lock`:

```
$ bundle install
```

Далее создадим файл `config/puma.rb` и заполним его содержимым из листинга 7.29. Этот код взят прямо из документации Heroku[1], и совершенно нет нужды его понимать.

Листинг 7.29 ❖ Файл настройки веб-сервера для эксплуатационного окружения (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'
```

[1] В листинге 7.29 немного изменено форматирование, чтобы уместить код по ширине страницы.

```
on_worker_boot do
  # Настройки, специфические для Rails 4.1+ см.: https://devcenter.heroku.com/articles/
  # deploying-rails-applications-with-the-puma-web-server#on-worker-boot
  ActiveRecord::Base.establish_connection
end
```

Наконец, создадим так называемый файл процессов `Procfile`, чтобы Heroku запускала процесс Puma в эксплуатационном окружении (листинг 7.30). Файл процессов `Procfile` должен быть создан в корневом каталоге приложения (то есть там же, где и `Gemfile`).

Листинг 7.30 ❖ Procfile для запуска веб-сервера Puma (./Procfile)

```
web: bundle exec puma -C config/puma.rb
```

После настройки веб-сервера можно отправлять изменения в репозиторий и приступать к развертыванию приложения[1]:

```
$ 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
```

Теперь мы имеем форму регистрации, действующую в эксплуатационном окружении, и результат успешной регистрации можно увидеть на рис. 7.24. Обратите внимание на тип протокола `https://` и значок замка в адресной строке, все это указывает на действующую поддержку SSL.

7.5.3. Номер версии Ruby

В процессе развертывания приложения в среде Heroku вы могли увидеть предупреждение:

```
###### 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'
(
###### ВНИМАНИЕ:
       Вы не объявили версию Ruby в своем Gemfile.
       Чтобы объявить номер версии Ruby,
       добавьте следующую строку в Gemfile:
       ruby '2.1.5'
)
```

[1] Мы не изменяли модели данных в этой главе, поэтому нет необходимости запускать миграцию в Heroku, но только если вы четко следовали указаниям в разделе 6.4. Так как некоторые читатели жаловались на появление проблем, я на всякий случай добавил `heroku run rake db:migrate` в качестве заключительного шага.

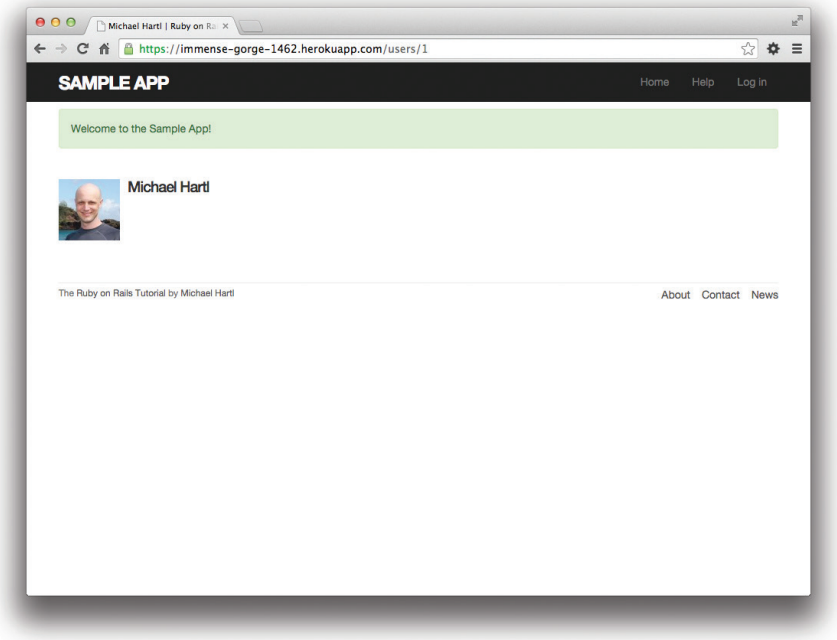


Рис. 7.24 ❖ Регистрация в Интернете

Как показывает опыт, на уровне этой книги издержки от явного объявления номера версии Ruby перевешивают (незначительные) преимущества, поэтому пока стоит игнорировать это предупреждение. Основная проблема – в том, что поддержание синхронизации учебного приложения и вашей системы с последней версией Ruby может стать огромным неудобством[1], тем более что номер используемой версии практически никогда не бывает важным. Тем не менее следует помнить, что для критически важных приложений, развертываемых на Heroku, рекомендуется указывать конкретную версию Ruby в `Gemfile`, чтобы обеспечить максимальную совместимость между средой разработки и эксплуатации.

7.6. Заключение

Возможность регистрировать пользователей – важная веха для нашего приложения. Хотя учебное приложение до сих пор не делает ничего полезного, мы заложили необходимый фундамент для последующей разработки. В главе 8 мы завершим разработку механизма аутентификации, позволив пользователям входить и выходить из приложения. В главе 9 разрешим всем пользователям изменять инфор-

[1] Например, после нескольких часов безуспешных попыток установить Ruby 2.1.4 на локальный компьютер я обнаружил, что накануне был выпущен Ruby 2.1.5. Попытка установить 2.1.5 также не удалась.

мацию в своих учетных записях, а администраторам дадим возможность удалять пользователей, тем самым завершив полный набор REST-операций с ресурсом Users из табл. 7.1.

7.6.1. Что мы узнали в этой главе

- Rails выводит полезную отладочную информацию при помощи метода `debug`.
- Sass-примеси дают возможность группировать CSS-правила и использовать их многократно.
- Rails поставляется с поддержкой трех стандартных окружений: разработки, тестирования и эксплуатации.
- Мы можем взаимодействовать с пользователями как с *ресурсом* через стандартный набор REST-адресов URL.
- Граватар – удобный способ демонстрации пиктограмм, представляющих пользователей.
- Вспомогательная функция `form_for` применяется для создания форм и взаимодействий с объектами Active Record.
- В случае неудачной регистрации вновь отображается страница регистрации и выводятся сообщения об ошибках, автоматически определяемые в Active Record.
- В Rails предусмотрен хэш `flash` как стандартный способ отображения кратковременных сообщений.
- В случае успешной регистрации в базе данных создается запись для пользователя, и браузер переадресуется на страницу профиля пользователя с приветственным сообщением.
- Для проверки поведения формы и предотвращения регрессий можно использовать интеграционные тесты.
- В эксплуатационном окружении можно настроить поддержку SSL для защищенной передачи данных и Puma – для высокой производительности.

7.7. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Убедитесь, что код в листинге 7.31 позволяет вспомогательной функции `gravatar_for` (была определена в разделе 7.1.4) принимать необязательный параметр `size`, делая допустимым в представлении, например, такой код: `gravatar_for user, size: 50`. (Мы воспользуемся этой улучшенной функцией в разделе 9.3.1.)
2. Напишите тесты для сообщений об ошибках, реализованных в листинге 7.18. Насколько детальными будут эти тесты, зависит от вас; шаблон предложен в листинге 7.32.

3. Напишите тесты для кратковременных сообщений, реализованных в разделе 7.4.2. Насколько детальными будут эти тесты, зависит от вас; минимальный шаблон предложен в листинге 7.33, его необходимо завершить, заменяя `FILL_IN` подходящим кодом. (Даже тестирование наличия верного ключа, не говоря уже о тексте, скорее всего, будет хрупким, поэтому я предпочитаю просто проверить, что хэш `flash` непустой.)
4. Как рассказывалось в разделе 7.4.2, разметка HTML, использующая `flash` в листинге 7.25, выглядит безобразно. Проверьте с помощью набора тестов, что более ясный код из листинга 7.34 с функцией `content_tag` работает точно так же.

Листинг 7.31 ❖ Добавление хэша параметров в функцию gravatar_for (app/helpers/users_helper.rb)

```
module UsersHelper

  # Возвращает граватар для указанного пользователя.
  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
```

Листинг 7.32 ❖ Шаблон для тестирования сообщений об ошибках (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#<CSS id for error explanation>'
    assert_select 'div.<CSS class for field with error>'
  end
  .
  .
  .
end
```

Листинг 7.33 ❖ Шаблон для тестирования хэша 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
```

Листинг 7.34 ❖ Применение content_tag в шаблоне сайта (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>
```

Глава 8

Вход и выход

Теперь, когда новые пользователи могут регистрироваться на нашем сайте (глава 7), пришло время дать им возможность входить на сайт и выходить. Мы реализуем все три наиболее распространенные модели поведения входа/выхода в Интернете: «забыть» аутентификационные данные пользователя при закрытии браузера (разделы 8.1 и 8.2), *автоматически* запомнить (раздел 8.4) и дать возможность *выбора* в виде флажка «запомнить меня» (раздел 8.4.5)[1].

Система аутентификации (подтверждения подлинности), которую мы разработаем в этой главе, позволит настроить сайт для каждого пользователя и реализовать модель авторизации, основанную на статусе входа и личности текущего пользователя. Например, в этой главе мы добавим в заголовок сайта ссылки «войти»/«выйти» и ссылку на профиль пользователя. В главе 9 мы реализуем модель защиты данных, чтобы только зарегистрированные пользователи могли просматривать список пользователей. Пользователь будет иметь доступ к редактированию лишь своей информации, и только администраторы смогут удалять других пользователей из базы данных. И наконец, в главе 11 мы используем идентификатор вошедшего в систему пользователя для создания микросообщений, связанных с этим пользователем, а в главе 12 позволим текущему пользователю следовать за другими пользователями приложения (и получать поток их микросообщений).

В этой длинной и сложной главе подробно раскрываются многие аспекты организации входа, поэтому я рекомендую сосредоточиться на последовательном ее чтении, раздел за разделом. Кроме того, многие читатели сообщают, что им было полезно прочитать главу второй раз.

8.1. Сеансы

HTTP – это *протокол без сохранения состояния*, он обрабатывает каждый запрос как независимую транзакцию, которая не может использовать информацию из любого предыдущего запроса. Это означает, что с его помощью невозможно за-

---

[1] Еще одна распространенная модель поведения – завершение сеанса по прошествии определенного времени. Это особенно уместно на сайтах с конфиденциальной информацией, например на сайтах, связанных с банковской или другой финансовой деятельностью или торговлей.

помнить личность пользователя при переходе между страницами; вместо этого веб-приложениям, где требуется регистрация пользователей, приходится использовать *сеансы*, полупостоянное соединение между двумя компьютерами (такими как клиентский компьютер с веб-браузером и сервер с Rails).

Чаще всего поддержку сеансов в Rails реализуют с применением *cookies*, небольших фрагментов текста, хранящихся в браузере пользователя. Так как cookies сохраняются при переходе между страницами, они могут хранить информацию (идентификатор пользователя, к примеру), которую приложение использует для получения информации о пользователе из базы данных. В этом разделе и в разделе 8.2 мы задействуем Rails-метод `session` для создания временных сеансов, которые *завершаются* в момент закрытия браузера[1], а затем в разделе 8.4 добавим более долговременные сеансы, применив другой Rails-метод: `cookies`.

Сеансы удобнее моделировать как ресурсы RESTful: страница входа будет отображать *новый* сеанс, в результате входа будет *создаваться* сеанс, а в результате выхода – *уничтожаться*. В отличие от ресурса Users, который хранится в базе данных (в виде модели User), ресурс Sessions будет храниться в cookies, поэтому основная работа над системой входа будет связана с построением механизма аутентификации, опирающегося на cookies. В этом и в последующих разделах мы будем заниматься подготовительной работой: создадим форму входа, контроллер Sessions и соответствующие методы этого контроллера. Затем мы завершим реализацию процедуры входа, написав код управления сеансами в разделе 8.2.

Как и в предыдущих главах, мы будем добавлять код в новую ветвь и объединим изменения в конце:

```
$ git checkout master
$ git checkout -b log-in-log-out
```

8.1.1. Контроллер Sessions

Элементы системы входа и выхода соответствуют определенным действиям REST – методам контроллера Sessions: форма входа обрабатывается методом `new` (рассмотрим в этом разделе), сам вход обрабатывается передачей запроса `POST` методу `create` (раздел 8.2), выход обрабатывается передачей запроса `DELETE` методу `destroy` (раздел 8.3). (Вспомните о соответствии между глаголами HTTP и действиями REST из табл. 7.1.)

Для начала сгенерируем контроллер Sessions с методом `new`:

```
$ rails generate controller Sessions new
```

(Добавление имен методов, таких как `new` в команде выше, вызывает создание *представлений*, поэтому здесь не были указаны методы `create` и `destroy`, у которых нет представлений.) Следуя модели страницы регистрации из раздела 7.2, создадим форму входа для создания новых сеансов, как показано на рис. 8.1.

---

[1] Некоторые браузеры имеют возможность восстанавливать такие сеансы, но, как вы понимаете, Rails никак не контролирует подобного поведения.

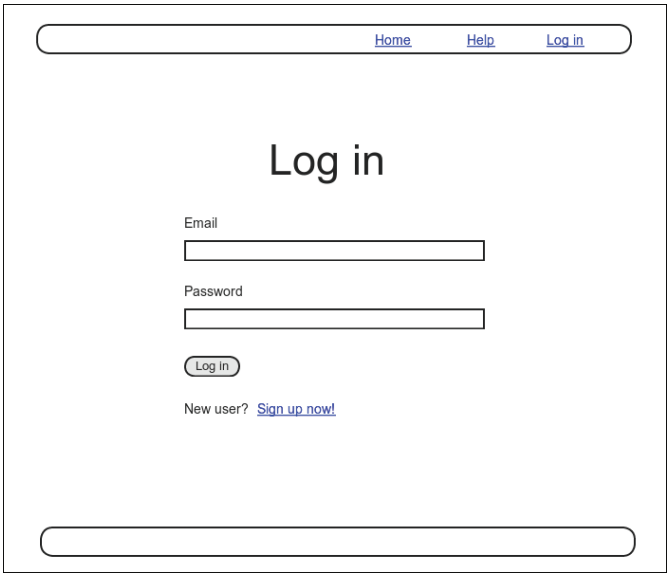


Рис. 8.1 ❖ Макет формы входа

В отличие от ресурса Users, где использовался специальный метод `resources`, чтобы автоматически получить полный набор RESTful-маршрутов (листинг 7.3), в ресурсе Sessions нам нужны только именованные маршруты, обрабатывающие запросы `GET` и `POST` к маршруту `login` и запрос `DELETE` к маршруту `logout`. Результат показан в листинге 8.1 (в нем заодно удалены ненужные маршруты, созданные командой `rails generate controller`).

Таблица 8.1 ❖ Маршруты, соответствующие правилами для сеансов из листинга 8.1

HTTP-запрос	URL	Именованный маршрут	Действие	Предназначение
`GET`	/login	`login_path`	`new`	Страница представления нового сеанса (вход)
`POST`	/login	`login_path`	`create`	Создание нового сеанса (вход)
`DELETE`	/logout	`logout_path`	`destroy`	Удаление сеанса (выход)

Листинг 8.1 ❖ Добавление ресурса со стандартными RESTful-действиями для работы с сеансами (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
```

Маршруты в листинге 8.1 определяют соответствие между адресами URL и методами контроллера, так же, как это было сделано для пользователей (табл. 7.1), как показано в табл. 8.1.

Так как мы только что добавили несколько именованных маршрутов, будет полезно взглянуть на полный список маршрутов в приложении, он формируется командой `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
```

Необязательно понимать этот вывод во всех деталях, но возможность получить его позволяет иметь общее представление о действиях, поддерживаемых приложением.

8.1.2. Форма входа

Определив контроллер и маршруты, заполним представление нового сеанса, то есть форму входа. Сравнив рис. 8.1 с рис. 7.11, можно заметить, что форма входа внешне выглядит почти так же, как форма регистрации, кроме того что в ней всего два поля (адрес электронной почты и пароль) вместо четырех.

Как показано на рис. 8.2, при вводе неверной информации нужно вновь вывести страницу входа с сообщениями об ошибках. Для отображения сообщений в разделе 7.3.3 мы использовали частичный шаблон и в том же разделе узнали, что текст ошибок автоматически предоставляется механизмом Active Record. Однако такой подход не годится для сеансов, так как сеанс не является объектом Active Record, поэтому мы будем отображать ошибки с помощью хэша `flash`.

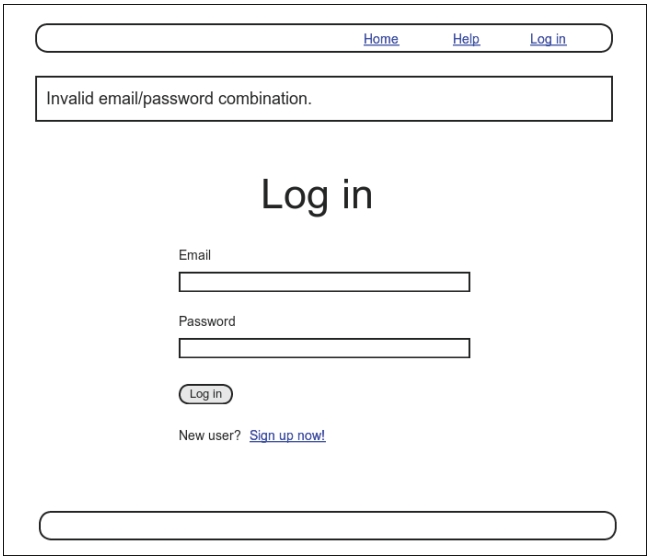


Рис. 8.2 ❖ Макет формы для сценария неудачного входа

Вспомните, как в форме регистрации применялась вспомогательная функция `form_for` (листинг 7.13), которая принимала переменную экземпляра `@user`:

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

Основное отличие между формами входа и регистрации заключается в отсутствии модели Session и, как следствие, аналога переменной `@user`. Это означает, что при конструировании формы представления нового сеанса необходимо передать функции `form_for` чуть больше информации; в частности, вызов

```
form_for(@user)
```

позволяет Rails сделать вывод, что атрибут `action` формы должен иметь значение с адресом URL `/users` и атрибут `method` – значение `POST`. В случае с сеансами требуется явно указать имя ресурса и соответствующий URL[1]:

```
form_for(:session, url: login_path)
```

Зная, как выглядит правильный вызов `form_for`, легко сделать форму входа, соответствующую макету на рис. 8.1, используя в качестве модели форму регистрации (листинг 7.13), как показано в листинге 8.2.

---

[1] Второй способ – использовать `form_tag` вместо `form_for`, и это было бы даже более идиоматически правильным решением с точки зрения Rails, но оно имело бы мало общего с формой регистрации, а на этом этапе я хочу подчеркнуть параллельность структуры.

Листинг 8.2 ❖ Реализация формы входа (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 %>

    <p>New user? <%= link_to "Sign up now!", signup_path %></p>
  </div>
</div>
```

Обратите внимание на ссылку на страницу регистрации, добавленную для удобства. Код в листинге 8.2 создает форму входа, изображенную на рис. 8.3. (Так как навигационная ссылка **Login** (Вход) пока не заполнена, необходимо ввести адрес URL `/login` прямо в адресной строке браузера. Этот недостаток мы исправим в разделе 8.2.3.)

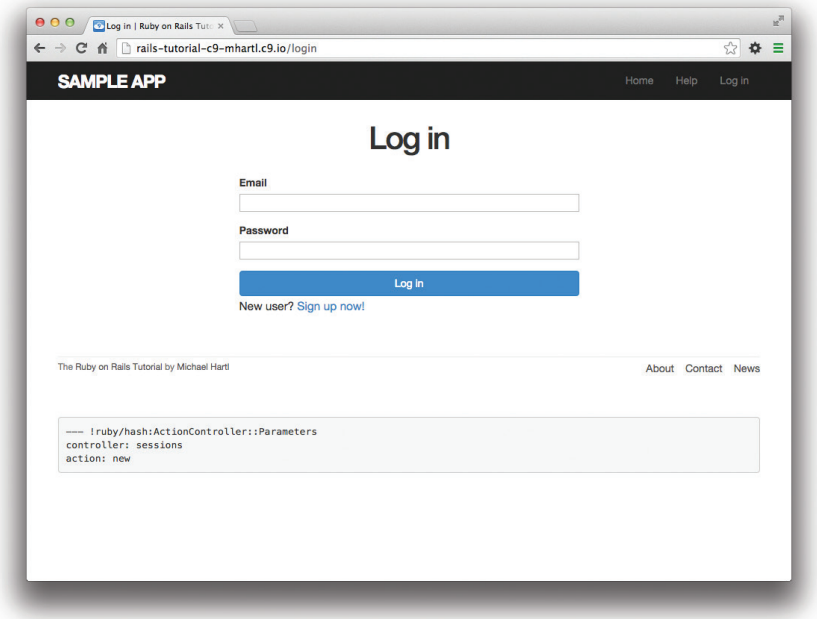


Рис. 8.3 ❖ Форма входа

Сгенерированная разметка HTML-формы показана в листинге 8.3.

Листинг 8.3 ❖ Разметка HTML-формы входа, сгенерированная кодом из листинга 8.2

```
<form accept-charset="UTF-8" action="/login" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden"
         value="NNb6+J/j46LcrgYUC60wQ2titMuJQ5lLqyAbnbAUkdo=" />
  <label for="session_email">Email</label>
  <input class="form-control" id="session_email"
         name="session[email]" type="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>
```

Сравнивая листинги 8.3 и 7.15, нетрудно догадаться, что в результате отправки этой формы будет создан хэш `params`, в котором `params[:session][:email]` и `params[:session][:password]` соответствуют полям **Email** (Адрес электронной почты) и **Password** (Пароль).

8.1.3. Поиск и аутентификация пользователя

Как и в случае с созданием (регистрацией) пользователей, сначала реализуем обработку неудачного сценария. Сперва разберемся, что происходит при отправке формы, а затем организуем вывод сообщений об ошибках (рис. 8.2.) Затем заложим основу успешного входа (раздел 8.2), научив приложение оценивать каждую попытку входа, опираясь на верность комбинации адреса электронной почты и пароля.

Начнем с определения минимального метода `create` в контроллере Sessions, а также пустых методов `new` и `destroy` (листинг 8.4). Метод `create` пока ничего не делает, кроме отображения представления `new`, но для начала этого хватит. Отправка формы `/sessions/new` приведет к появлению страницы, изображенной на рис. 8.4.

Листинг 8.4 ❖ Предварительная версия метода create (app/controllers/sessions_controller.rb)

```
class SessionsController < ApplicationController

  def new
  end

  def create
    render 'new'
  end

  def destroy
  end
end
```

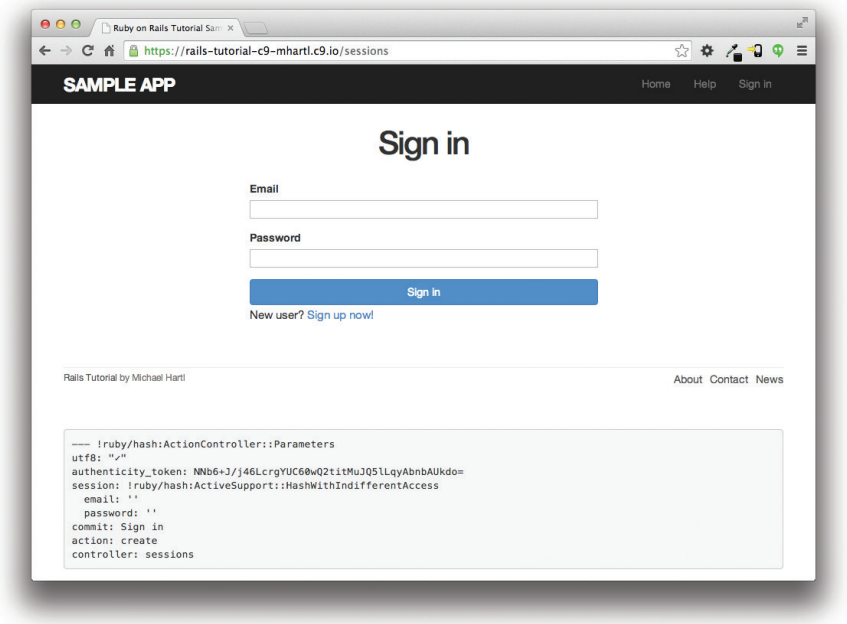


Рис. 8.4 ❖ Неудачный вход с текущим методом create из листинга 8.4

Внимательное изучение отладочной информации на рис. 8.4 показывает, как отмечалось в конце раздела 8.1.2, что в результате отправки формы создается хэш `params`, содержащий элементы `email` и `password` в элементе с ключом `session` (опустим некоторые неважные детали, необходимые только для внутреннего пользования Rails):

```
---
session:
  email: 'user@example.com'
  password: 'foobar'
  commit: Log in
  action: create
  controller: sessions
```

Как и в случае с регистрацией пользователя (рис. 7.15), эти параметры образуют вложенный хэш, как тот, что мы видели в листинге 4.10. В частности, `params` содержит вложенный хэш формы

```
{ session: { password: "foobar", email: "user@example.com" } }
```

Это означает, что элемент

```
params[:session]
```

сам является хэшем:

```
{ password: "foobar", email: "user@example.com" }
```

Соответственно,

```
params[:session][:email]
```

хранит отправленный адрес электронный почты, а

```
params[:session][:password]
```

хранит пароль.

Другими словами, внутри метода `create` хэш `params` хранит всю информацию, необходимую для аутентификации пользователя по электронной почте и паролю. Не случайно, что у нас уже есть необходимые нам методы: `User.find_by`, предоставленный Active Record (раздел 6.1.4), и `authenticate`, предоставленный `has_secure_password` (раздел 6.3.4). Напомню, что `authenticate` возвращает `false` при ошибке аутентификации (раздел 6.3.4), поэтому нашу стратегию входа можно обобщить, как показано в листинге 8.5.

Листинг 8.5 ❖ Поиск и аутентификация пользователей (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])
      # Осуществить вход пользователя и переадресовать на страницу профиля.
    else
      # Создать сообщение об ошибке.
      render 'new'
    end
  end

  def destroy
  end
end
```

Первая выделенная строка извлекает информацию о пользователе из базы данных по адресу электронной почты. (Напомню, что все адреса сохраняются в нижнем регистре, поэтому, чтобы обеспечить соответствие, здесь вызывается метод `downcase`.) Следующая строка может немного смутить, но она довольно распространена в идиоматике Rails-программирования:

```
user && user.authenticate(params[:session][:password])
```

Здесь используется оператор `&&` (логическое *И*) для определения верности полученного пользователя. Учитывая, что любой объект, кроме `nil` и `false`, является

в логическом смысле true (раздел 4.2.3), возможные результаты этого выражения показаны в табл. 8.2. В ней видно, что выражение в инструкции if возвращает true, только если пользователь с указанным адресом электронной почты существует в базе данных и его пароль соответствует указанному.

Таблица 8.2 ❖ Возможные значения выражения user && user.authenticate(...)

Пользователь	Пароль	a &&b
Не существует	*Любой*	`(nil && [любой]) ==false`
Существует	Неверный	`(true && false) ==false`
Существует	Верный	`(true && true) ==true`

8.1.4. Отображение кратковременных сообщений

В разделе 7.3.3 было реализовано отображение сообщений об ошибках регистрации с использованием модели User. Ошибки были связаны с конкретным объектом Active Record, но эта стратегия здесь неприменима, потому что сеансы не являются моделью Active Record. Вместо этого мы поместим сообщение в хэш flash, чтобы оно отображалось в случае ошибки входа. Первая, не совсем правильная попытка представлена в листинге 8.6.

Листинг 8.6 ❖ Не совсем правильная попытка обработки ошибки входа (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])
      # Осуществить вход пользователя и переадресовать на страницу профиля.
    else
      flash[:danger] = 'Invalid email/password combination'    # Неправильно!
      render 'new'
    end
  end

  def destroy
  end
end
```

Поскольку кратковременное сообщение отображается в шаблоне сайта (листинг 7.25), сообщение flash[:danger] будет отображено автоматически; кроме того, благодаря Bootstrap CSS к нему автоматически будет применено соответствующее оформление (рис. 8.5).

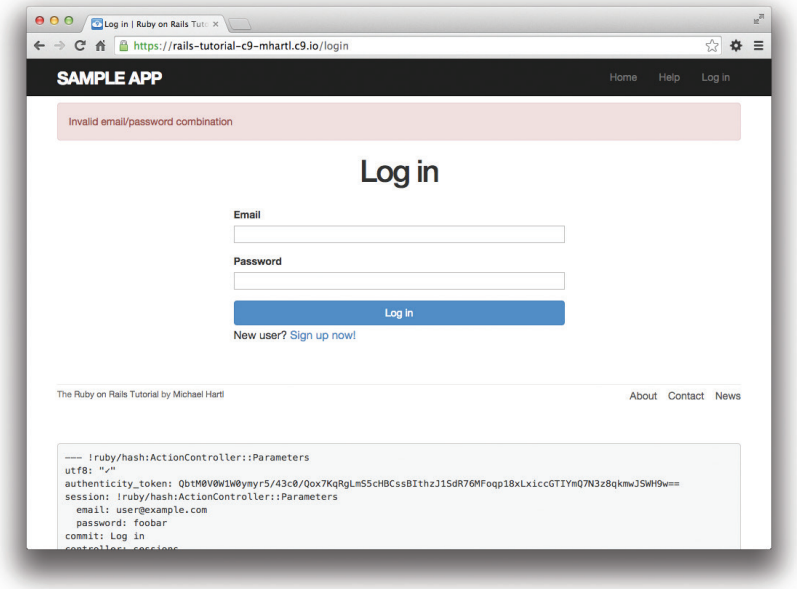


Рис. 8.5 ❖ Кратковременное сообщение об ошибке входа

К сожалению, как было отмечено в тексте и в комментарии в листинге 8.6, этот код не совсем верный. Однако страница выглядит нормально, так в чем же подвох? Проблема в том, что содержимое хэша `flash` существует в течение *одного запроса*, но, в отличие от переадресации, которая выполняется в листинге 7.24, повторное отображение шаблона вызовом метода `render` не считается запросом. В результате сообщение существует на один запрос дольше, чем нам необходимо. Например, если в форме входа отправить неправильную информацию, а затем перейти на главную страницу, кратковременное сообщение отобразится повторно (рис. 8.6). Устранением этого недостатка мы займемся в разделе 8.1.5.

8.1.5. Тест кратковременного сообщения

Неверное поведение хэша `flash` – это лишь небольшая ошибка в нашем приложении. Согласно рекомендациям по тестированию (блок 3.3), это именно та ситуация, когда стоит написать тест, выявляющий эту ошибку, чтобы она не появилась снова. Поэтому, прежде чем продолжить, напишем короткий интеграционный тест, отправляющий форму входа. Он не только задокументирует ошибку и предотвратит регрессию, но также даст хорошую основу для дальнейших интеграционных тестов входа и выхода.

Начнем с создания интеграционного теста для проверки поведения входа в приложение:

```
$ rails generate integration_test users_login
      invoke  test_unit
      create    test/integration/users_login_test.rb
```

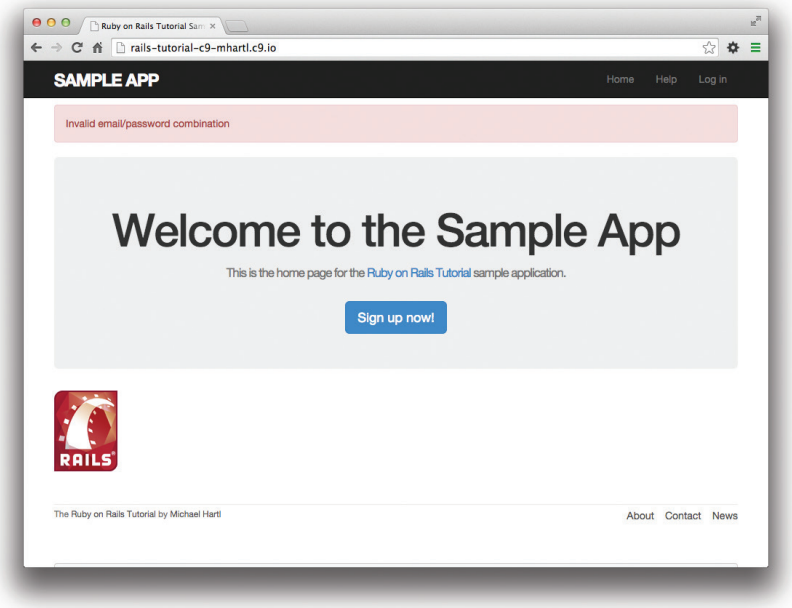


Рис. 8.6 ❖ Пример сохранения кратковременного сообщения

Итак, нам нужен тест, выявляющий последовательность, изображенную на рис. 8.5 и 8.6. Выражаясь простым языком, тест должен:

1) открыть страницу входа;
2) убедиться, что форма, представляющая новый сеанс, отображается верно;
3) отправить хэш `params` с ошибочной информацией в составе запроса по маршруту `login_path`;
4) убедиться, что сервер вновь вернул форму, но уже с сообщением об ошибке;
5) перейти на другую страницу (например, на главную);
6) убедиться, что сообщение отсутствует на ней.

Тест, реализующий эти шаги, показан в листинге 8.7.

Листинг 8.7 ❖ Тест, выявляющий нежелательное сохранение хэша flash **КРАСНЫЙ** (test/integration/users_login_test.rb)

```
require 'test_helper'

class UsersLoginTest < ActionDispatch::IntegrationTest

  test "login with invalid information" do
    get login_path
    assert_template 'sessions/new'
    post login_path, session: { email: "", password: "" }
    assert_template 'sessions/new'
    assert_not flash.empty?
    get root_path
```

```
    assert flash.empty?
  end
end
```

На данном этапе этот тест должен быть **КРАСНЫМ**:

Листинг 8.8 ❖ КРАСНЫЙ

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb
```

Эта команда демонстрирует, как с использованием аргумента `TEST` и полного пути к файлу выполнить один (и только один) файл теста.

Чтобы обеспечить успех тесту из листинга 8.7, заменим `flash` специальным вариантом `flash.now`, созданным именно для вывода кратковременных сообщений на отображаемой странице. В отличие от `flash`, содержимое `flash.now` исчезает сразу после дополнительного запроса – именно такое поведение мы тестировали в листинге 8.7. Верный код приложения, с учетом этой замены, показан в листинге 8.9.

Листинг 8.9 ❖ Правильный способ обработки ошибки входа ЗЕЛЕНЫЙ (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])
      # Осуществить вход пользователя и переадресовать на страницу профиля.
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
  end
end
```

Теперь интеграционный тест входа выполняется успешно, как и весь набор тестов:

Листинг 8.10 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb
$ bundle exec rake test
```

8.2. Вход

Теперь, когда наша форма входа научилась обрабатывать ошибку входа, необходимо реализовать отправку верных данных и впустить пользователя на сайт. В этом разделе пользователь будет входить с помощью временного сеансового cookie,

срок хранения которого автоматически истекает после закрытия браузера. В разделе 8.4 мы добавим сеансы, продолжающие существовать даже после закрытия браузера.

Реализация сеансов связана с необходимостью определения множества связанных функций для использования в разных контроллерах и представлениях. Как рассказывалось в разделе 4.2.5, Ruby предоставляет поддержку *модулей* для объединения таких функций в один пакет. Когда мы создавали контроллер Sessions, автоматически был создан вспомогательный модуль Sessions (раздел 8.1.1). Более того, такие модули автоматически подключаются всеми Rails-представлениями; а если подключить модуль в базовом классе всех контроллеров (контроллер Application), он также станет доступен во всех контроллерах (листинг 8.11).

Листинг 8.11 ❖ Подключение вспомогательного модуля Sessions в контроллере Application (app/controllers/application_controller.rb)

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper
end
```

После завершения настройки можно переходить к коду, реализующему вход пользователей.

8.2.1. Метод log_in

Вход пользователя проще всего реализовать с помощью метода `session` из фреймворка Rails. (Это отдельный метод, и он не имеет отношения к контроллеру Sessions из раздела 8.1.1.) Метод `session` можно интерпретировать как хэш и присваивать ему значения:

```
session[:user_id] = user.id
```

Эта инструкция передаст браузеру пользователя cookie с зашифрованной версией идентификатора пользователя, что позволит получить его на следующей странице через `session[:user_id]`. В отличие от постоянных cookies, создаваемых методом `cookies` (раздел 8.4), временные cookies уничтожаются сразу после закрытия браузера.

Так как один и тот же подход к организации входа требуется применить в двух разных местах, мы определим функцию `log_in` в модуле Sessions, как показано в листинге 8.12.

Листинг 8.12 ❖ Функция log_in (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end
end
```

Так как временные cookies, создаваемые методом `session`, автоматически шифруются, у злоумышленников не будет возможности использовать информацию о сеансе, чтобы осуществить вход под видом зарегистрированного пользователя. Это относится только к временным сеансам, запущенным методом `session`, но *не* относится к постоянным, запущенным методом `cookies`. Постоянные cookies уязвимы для атак типа *session hijacking* (перехват сеанса), поэтому в разделе 8.4 нам придется быть более осторожными с информацией, передаваемой браузеру.

Теперь мы готовы закончить метод `create`, реализовав вход пользователя и переадресовав его на страницу профиля (листинг 8.13)[1].

Листинг 8.13 ❖ Вход пользователя (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
  end
end
```

Обратите внимание на компактную форму переадресации

```
redirect_to user
```

которую мы уже видели в разделе 7.4.1. Rails автоматически преобразует ее в маршрут к странице профиля пользователя:

```
user_url(user)
```

После определения метода `create` форма входа из листинга 8.2 должна заработать. Пока в результате входа ничего не происходит, и, кроме прямого изучения состояния сеанса в браузере, нельзя сказать, вошли вы или нет. В качестве первого шага к более заметным изменениям мы научимся в разделе 8.2.2 получать информацию о текущем пользователе из базы данных по его идентификатору в сеансе. В разделе 8.2.3 мы изменим ссылки в шаблоне приложения, в том числе адрес URL профиля текущего пользователя.

---

[1] Метод `log_in` доступен в контроллере Sessions благодаря подключению модуля в листинге 8.11.

8.2.2. Текущий пользователь

Разместив идентификатор пользователя в защищенном временном сеансе, мы теперь можем получить его на следующей странице, определив метод current_user для поиска в базе данных пользователя с идентификатором, соответствующим идентификатору в сеансе. Метод current_user нужен для построения, например, таких конструкций:

```
<%= current_user.name %>
```

и

```
redirect_to current_user
```

Чтобы найти текущего пользователя, можно применить метод find, как на странице профиля пользователя (листинг 7.5):

```
User.find(session[:user_id])
```

Но, как рассказывалось в разделе 6.1.4, find вызывает исключение, если идентификатор пользователя отсутствует в базе данных. Такое поведение допустимо на странице профиля пользователя, так как произойти подобное может только в случае неверного значения идентификатора, но в нашем случае session[:user_id] часто будет иметь значение nil (если пользователь не вошел на сайт). Учитывая это обстоятельство, мы будем использовать тот же метод find_by, который применяется для поиска по адресу электронной почты в методе create, но с идентификатором пользователя вместо адреса электронной почты:

```
User.find_by(id: session[:user_id])
```

Если идентификатор окажется ошибочным, вместо исключения этот метод вернет nil (указывая на отсутствие пользователя).

Теперь можно определить метод current_user:

```
def current_user
  User.find_by(id: session[:user_id])
end
```

Это вполне работоспособное решение, но оно предполагает многократное обращение к базе данных, если, например, метод current_user будет вызван несколько раз на странице. Вместо этого мы последуем распространенному в Ruby соглашению о хранении результатов User.find_by в переменной экземпляра, при этом обращение к базе данных происходит только в первый раз, а при последующих вызовах немедленно возвращается переменная экземпляра[1]:

```
if @current_user.nil?
  @current_user = User.find_by(id: session[:user_id])
else
```

[1] Такая практика сохранения значения вызова метода называется мемоизация. (Это технический термин, а не ошибочное написание слова «memorization» (запоминание).)

```
  @current_user
end
```

Вспомнив про оператор || (*ИЛИ*) из раздела 4.2.3, это выражение можно переписать иначе:

```
@current_user = @current_user ||
                User.find_by(id: session[:user_id])
```

Так как в логическом контексте объект User является истинным, вызов find_by будет выполняться, только если переменной @current_user еще не было присвоено значение.

Это действующий код, но идиоматически не совсем правильный; ниже приводится более правильный способ записи:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Здесь использован немного странный, но весьма распространенный оператор ||= («или равно») (блок 8.1).

Блок 8.1 ❖ Что за *$@! этот ваш ||=?

Оператор присваивания ||= («или равно») – широко известная идиома Ruby, поэтому начинающим Rails-разработчикам так важно ее знать и уметь применять. На первый взгляд он кажется странным, но его легко понять по аналогии.
Начнем с общепринятого шаблона записи приращения переменной:

```
x = x + 1
```

Многие языки программирования поддерживают сокращенную форму записи этой операции; в Ruby (а также в C, C++, Perl, Python, Java и т. д.) она выглядит следующим образом:

```
x += 1
```

Аналогичные конструкции существуют и для других операторов:

```
$ rails console
>> x = 1
=> 1
>> x +=1
=> 2
>> x *=3
=> 6
>> x -= 8
=> -2
>> x /= 2
=> -1
```

Для каждого из этих случаев подходит шаблон x = x ○ y, и x ○= y будет верным для любого оператора ○.

Другим распространенным шаблоном в Ruby является присваивание переменной, только если она имеет значение nil, иначе ее оставляют в покое. Применив описанную аналогию к оператору || (*ИЛИ*) из раздела 4.2.3, получим вот такую запись:

```
>> @foo
=> nil
>> @foo = @foo || "bar"
=> "bar"
>> @foo = @foo || "baz"
=> "bar"
```

Поскольку в логическом контексте nil интерпретируется как ложное значение, первое выражение присваивания интерпретируется как nil || "bar", то есть как "bar". По аналогии второе присваиваниеинтерпретируется как "bar" || "baz", и также в результате получается "bar". Это связано с тем, что любые значения, кроме nil и false, в логическом контексте интерпретируются как истинные, а вычисление последовательности операторов || прерывается после обнаружения первого истинного значения. (Эта практика оценки выражений || слева направо и остановки на первом истинном значении известна как *вычисление по короткой схеме*. Тот же принцип применяется к оператору &&, только в этом случае вычисления прерываются на первом значении false.)

Сравнив разные операторы, мы увидели, что @foo = @foo || "bar" соответствует шаблону x = x ○ y с оператором || вместо ○:

```
x    =   x   +   1   -> x     +=   1
x    =   x   *   3   -> x     *=   3
x    =   x   -   8   -> x     -=   8
x    =   x   /   2   -> x     /=   2
@foo = @foo || "bar" -> @foo ||= "bar"
```

Теперь понятно, что выражения @foo = @foo || "bar" и @foo ||= "bar" эквивалентны. В случае с текущим пользователем это означает вот такую конструкцию:

```
@current_user ||= User.find_by(id: session[:user_id])
```

Вуаля!

(Кстати, внутренне Ruby фактически оценивает выражение @foo || @foo = "bar", что позволяет избежать ненужного присваивания, когда @foo имеет значение nil или false. Но это выражение не так хорошо объясняет оператор ||=, поэтому в обсуждении я использовал приблизительный эквивалент @foo = @foo || "bar".)

Применив идеи, изложенные выше, мы получили сжатый метод current_user (листинг 8.14).

Листинг 8.14 ❖ Поиск текущего пользователя в сеансе (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
```

```
  def log_in(user)
    session[:user_id] = user.id
  end

  # Возвращает текущего вошедшего пользователя (если есть).
  def current_user
    @current_user ||= User.find_by(id: session[:user_id])
  end
end
```

После добавления метода `current_user` можно вносить в приложение изменения, связанные с определением статуса пользователя.

8.2.3. Изменение ссылок шаблона

Первое практическое применение системы регистрации на нашем сайте выражается в добавлении ссылок, изменяющихся в зависимости от статуса пользователя. В частности, как показано на рис. 8.7[1], добавим ссылки для выхода, перехода к настройкам пользователя, на список всех пользователей и на страницу профиля текущего пользователя. Обратите внимание, что ссылки для выхода и на профиль находятся в раскрывающемся меню **Account** (Учетная запись); как сделать такое меню с помощью Bootstrap, мы увидим в листинге 8.16.

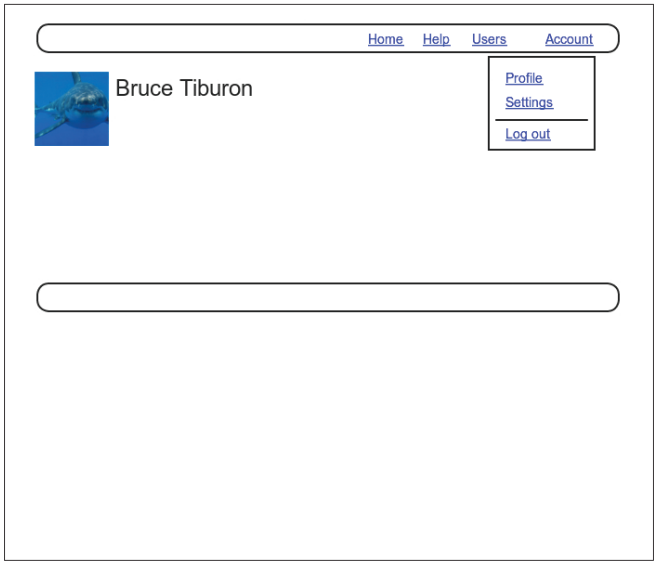


Рис. 8.7 ❖ Макет страницы профиля пользователя после успешного входа

В данный момент я бы предпочел написать интеграционные тесты, чтобы проверить вышеописанное поведение. Как говорилось в блоке 3.3, когда поближе по-

[1] Картинка взята по адресу: http://www.flickr.com/photos/hermanusbackpackers/3343254977/.

знакомитесь с инструментами тестирования Rails, вы будете более расположены сначала писать тесты. Однако в данном случае тесты содержат несколько новых идей, поэтому будет лучше перенести их в отдельный раздел (раздел 8.2.4).

Смена ссылок в шаблоне сайта подразумевает использование структуры if-else, чтобы зарегистрированному пользователю показывать один набор ссылок, а незарегистрированному – другой:

```
<% if logged_in? %>
  # Ссылки для зарегистрированного пользователя
<% else %>
  # Ссылки для незарегистрированного пользователя
<% end %>
```

Этот код требует наличия логического метода `logged_in?`, который мы сейчас определим.

Пользователь считается зарегистрированным, если в сеансе существует текущий пользователь, то есть если `current_user` имеет значение, отличное от `nil`. Эта проверка требует применения оператора «не» (раздел 4.2.3), который записывается как восклицательный знак `!`. Требуемый метод `logged_in?` представлен в листинге 8.15.

Листинг 8.15 ❖ Вспомогательный метод logged_in? (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Возвращает текущего вошедшего пользователя (если есть).
  def current_user
    @current_user ||= User.find_by(id: session[:user_id])
  end

  # Возвращает true, если пользователь зарегистрирован, иначе возвращает false.
  def logged_in?
    !current_user.nil?
  end
end
```

Теперь мы готовы изменить ссылки для зарегистрированного пользователя. Это четыре новые ссылки, две из которых пока останутся заглушками (мы доработаем их в главе 9):

```
<%= link_to "Users",    '#' %>
<%= link_to "Settings", '#' %>
```

Ссылка для выхода использует маршрут, который был определен в листинге 8.1:

```
<%= link_to "Log out", logout_path, method: :delete %>
```

Обратите внимание, что ссылка на выход передает в аргументе хэш, который указывает, что он должен отправляться HTTP-запросом DELETE[1]. Также добавим ссылку на профиль:

```
<%= link_to "Profile", current_user %>
```

Мы могли бы написать

```
<%= link_to "Profile", user_path(current_user) %>
```

но, как обычно, Rails позволяет ссылаться непосредственно на пользователя благодаря автоматическому преобразованию current_user в user_path(current_user). Наконец, если пользователь *не* вошел, создается ссылка на форму входа с использованием маршрута входа:

```
<%= link_to "Log in", login_path %>
```

Собрав все вместе, получим обновленный частичный шаблон заголовка, как показано в листинге 8.16.

Листинг 8.16 ❖ Изменение ссылок шаблона для зарегистрированных пользователей (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 %>
```

[1] Веб-браузеры на самом деле не умеют отправлять запрос DELETE, Rails имитирует их с помощью JavaScript.

```
      </ul>
    </nav>
  </div>
</header>
```

При размещении новых ссылок в шаблоне мы сразу же воспользовались поддержкой раскрывающихся меню в Bootstrap[1]. В частности, обратите внимание на включение специальных CSS-классов Bootstrap, таких как `dropdown`, `dropdown-menu` и т. д. Чтобы активировать раскрывающееся меню, необходимо в файле `application.js` указать собственную JavaScript-библиотеку Bootstrap (листинг 8.17).

Листинг 8.17 ❖ Добавление JavaScript-библиотеки Bootstrap в application.js (app/assets/javascripts/application.js)

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= requireturbolinks
//= require_tree .
```

Теперь откройте форму входа и войдите, введя верные данные, таким способом вы эффективно протестируете предыдущие три раздела[2]. После добавления кода из листингов 8.16 и 8.17 должны появиться раскрывающиеся меню и ссылки для зарегистрированных пользователей, как на рис. 8.8. Закрыв браузер, вы сможете убедиться, что приложение забыло ваш статус и требует войти снова, чтобы вы могли увидеть вышеописанные изменения.

8.2.4. Тестирование изменений в шаблоне

После проверки поведения приложения после входа и прежде чем двигаться дальше, напишем интеграционный тест, который поможет не допустить регрессии. Мы продолжим тест из листинга 8.7 и добавим в него следующую последовательность операций:

1) открыть форму входа;
2) отправить верную информацию методом `post`;
3) убедиться, что ссылка на форму входа исчезла;
4) убедиться, что появилась ссылка для выхода;
5) убедиться, что появилась ссылка на профиль.

Чтобы увидеть эти изменения, тест должен выполнять вход с учетными данными уже зарегистрированного пользователя, а это значит, что пользователь должен существовать в базе данных. По умолчанию для этого в Rails используются тестовые данные, предназначенные для загрузки в тестовую базу данных. В разделе 6.2.5 мы столкнулись с необходимостью удаления тестовых данных для тес-

---

[1] Более подробную информацию ищите по адресу: http://getbootstrap.com/components/.

[2] Если вы работаете в облачной IDE, я рекомендую использовать другой браузер для тестирования поведения формы входа, чтобы не пришлось закрывать браузер с запущенной IDE.

тирования уникальности адреса электронной почты (листинг 6.30). Теперь мы можем заполнить этот пустой файл собственными данными.

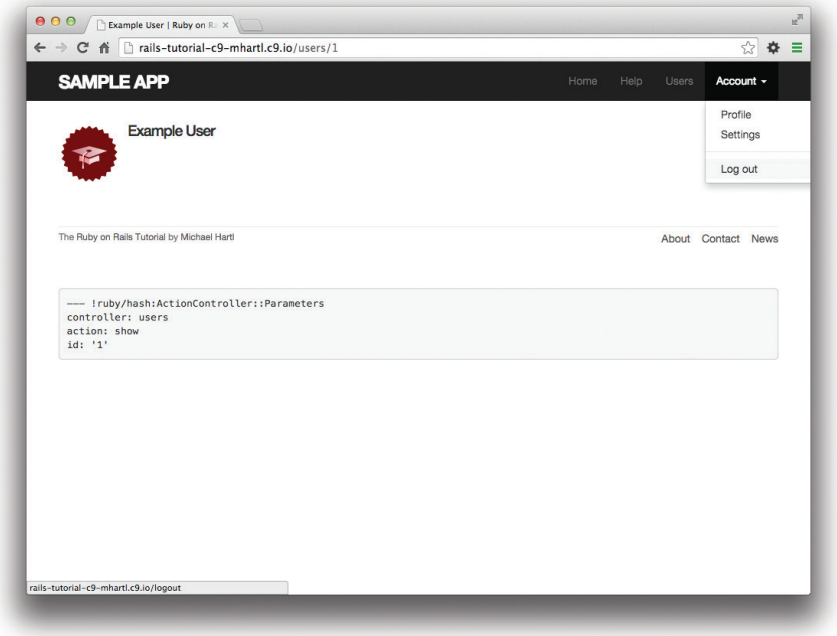


Рис. 8.8 ❖ Для зарегистрированного пользователя отображаются новые ссылки и раскрывающиеся меню

Сейчас нам нужен только один пользователь, информация о котором должна включать имя и адрес электронной почты. Так как нам требуется осуществить вход на сайт, этому пользователю также понадобится верный пароль для сравнения с паролем, который будет передан методу `create` контроллера Sessions. Взглянув на модель данных на рис. 6.8, можно заметить, что нам необходим атрибут `password_digest`, для создания которого определим метод `digest`.

Как обсуждалось в разделе 6.3.1, шифрованную версию пароля создает функция `bcrypt` (через `has_secure_password`), поэтому создадим пароль в тестовых данных тем же методом. Мы нашли этот метод в реализации надежных паролей (https://github.com/rails/rails/blob/master/activemodel/lib/active_model/secure_password.rb):

```
BCrypt::Password.create(string, cost: cost)
```

где `string` – это строка, которую нужно хэшировать, и `cost` – параметр, определяющий максимальные вычислительные затраты на расчет хэша. Использование больших значений `cost` уменьшает вероятность вычисления исходного пароля по хэшу, а это очень важная мера предосторожности в эксплуатационном окружении, но в тестах хотелось бы, чтобы метод `digest` работал настолько быстро, насколько это возможно. В реализации надежных паролей есть строка и для этого:

```
cost = ActiveModel::SecurePassword.min_cost ?
       BCrypt::Engine::MIN_COST :
       BCrypt::Engine.cost
```

Этот туманный код, который сейчас необязательно понимать в деталях, обеспечивает поведение, которое было описано выше: использует минимальный параметр `cost` в тестах и нормальный (высокий) в эксплуатационном окружении. (Подробнее о странной конструкции `?:` рассказывается в разделе 8.4.5.)

Мы могли бы поместить полученный метод `digest` куда угодно, но в разделе 8.4.1 он нам понадобится в модели User. Поэтому поместим его в `user.rb`. Так как при вычислении дайджеста доступ к объекту пользователя не требуется, включим метод `digest` непосредственно в сам класс User, сделав его *методом класса* (как рассказывалось в разделе 4.4.1). Результат можно увидеть в листинге 8.18.

Листинг 8.18 ❖ Добавление метода digest для использования в тестах (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, length: { minimum: 6 }

  # Возвращает дайджест для указанной строки.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end
end
```

Теперь можно определить тестовые данные для проверки входа пользователя (листинг 8.19).

Листинг 8.19 ❖ Тестовые данные для проверки входа пользователя (test/fixtures/users.yml)

```
michael:
  name: Michael Example
  email: michael@example.com
  password_digest: <%= User.digest('password') %>
```

Обратите внимание, что в определениях тестовых данных допускается использовать инструкции на встроенном Ruby, благодаря чему можно записать:

```
<%= User.digest('password') %>
```

чтобы создать дайджест пароля для тестового пользователя.

Несмотря на то что мы определили атрибут password_digest, необходимый для has_secure_password, иногда бывает удобно сослаться на простой (виртуальный) пароль. Увы, сделать это через механизм тестовых данных невозможно – добавление атрибута password в листинг 8.19 заставит Rails пожаловаться на отсутствие соответствующего столбца в базе данных (и это правда). Далее мы договоримся, что все тестовые пользователи будут иметь один и тот же пароль ('password').

Теперь можно обратиться к созданному пользователю в тестах:

```
user = users(:michael)
```

Здесь users соответствует имени файла с тестовыми данными users.yml, а символ :michael ссылается на пользователя с ключом, указанным в листинге 8.19.

Теперь можно написать тест, проверяющий ссылки в шаблоне, превратив в код последовательность действий, что была описана в начале раздела (листинг 8.20).

Листинг 8.20 ❖ Тест входа пользователя с верной информацией **ЗЕЛЕНЫЙ** (test/integration/users_login_test.rb)

```
require 'test_helper'

class UsersLoginTest < 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
```

Здесь мы использовали

```
assert_redirected_to @user
```

чтобы проверить переадресацию, а также

```
follow_redirect!
```

чтобы открыть эту страницу. Тест в листинге 8.20 также проверяет удаление ссылки для входа – на странице должно быть *ноль* ссылок login_path:

```
assert_select "a[href=?]", login_path, count: 0
```

Включив дополнительный параметр `count: 0`, мы сообщили методу `assert_select`, что ожидается ноль ссылок, соответствующих шаблону. (Сравните с `count: 2` в листинге 5.25, который проверяет наличие ровно двух ссылок, соответствующих шаблону.)

Так как код приложения уже работает, набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 8.21 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test TEST=test/integration/users_login_test.rb \
>                       TESTOPTS="--name test_login_with_valid_information"
```

Здесь показано, как запустить определенный тест из файла, передав параметр

```
TESTOPTS="--name test_login_with_valid_information"
```

с названием теста. (Слово «test» и слова из описания теста через символ подчеркивания.) Символ > на второй строке – это символ «продолжения строки», командная строка добавляет его автоматически, и его не нужно набирать буквально.

8.2.5. Вход после регистрации

Мы закончили с аутентификацией, но вновь зарегистрировавшиеся пользователи могут оказаться сбитыми с толку, обнаружив, что они не вошли на сайт по умолчанию. Это довольно странно – заставлять пользователя входить на сайт сразу после регистрации, поэтому реализуем автоматический вход новых пользователей как часть процесса регистрации. Для этого нужно лишь добавить вызов `log_in` в метод `create` контроллера Users, как показано в листинге 8.22[1].

Листинг 8.22 ❖ Вход пользователя после регистрации (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
```

[1] Метод `log_in` доступен не только в контроллере Sessions, но и в Users, благодаря подключению модуля в листинге 8.11.

```
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

Чтобы протестировать новое поведение, добавим строку в тест из листинга 7.26, проверяющую вход. В связи с этим полезно определить вспомогательную функцию `is_logged_in?`, соответствующую функции `logged_in?` из листинга 8.15 и возвращающую `true` при наличии пользователя с указанным идентификатором в (тестовом) сеансе и `false` в противном случае (листинг 8.23). (Так как вспомогательные функции недоступны в тестах, мы не можем использовать `current_user`, как в листинге 8.15, но нам доступен метод `session`, который мы будем использовать вместо него.) Здесь мы вызываем `is_logged_in?` вместо `logged_in?`, чтобы не перепутать вспомогательные функции друг с другом[1].

Листинг 8.23 ❖ Логический метод определения статуса входа внутри тестов (test/test_helper.rb)

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all

  # Возвращает true, если тестовый пользователь вошел.
  def is_logged_in?
    !session[:user_id].nil?
  end
end
```

Теперь можно проверить вход пользователя после регистрации в тесте, дописав строку в листинг 8.24.

Листинг 8.24 ❖ Тест входа после регистрации **ЗЕЛЕНЫЙ** (test/integration/users_signup_test.rb)

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest
  .
  .
  .
```

[1] Например, однажды я видел **ЗЕЛЕНЫЙ** набор тестов даже после случайного удаления основного метода `log_in` из вспомогательного модуля Sessions. Причина в том, что тесты благополучно использовали тестовую вспомогательную функцию с таким же именем, а приложение отказывалось работать. Похожую проблему мы решим при определении тестовой вспомогательной функции `log_in_as` в листинге 8.50.

```
  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
```

Набор тестов должен остаться **ЗЕЛЕНЫМ**:

Листинг 8.25 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

8.3. Выход

Как рассказывалось в разделе 8.1, наша модель аутентификации предполагает сохранение пользователей вошедшими, пока они явно не выйдут из системы. В этом разделе мы добавим эту необходимую возможность выхода. Так как ссылка **Log-out** (Выход) уже определена (листинг 8.16), нам достаточно лишь написать метод контроллера, удаляющий сеанс пользователя.

До сих пор, определяя методы контроллера Sessions, мы следовали соглашениям RESTful, использовав имя `new` для страницы входа и `create` для его реализации. Мы продолжим эту тенденцию и дадим методу удаления сеанса имя `destroy`. В отличие от реализации входа, которая понадобилась нам в листингах 8.13 и 8.22, реализация выхода нужна только в одном месте, поэтому мы поместим ее прямо в метод `destroy`. Как будет показано в разделе 8.4.6, такое решение (с небольшими изменениями) заодно облегчит нам тестирование механизма аутентификации.

Выход подразумевает отмену изменений, произведенных методом `log_in` из листинга 8.12, а именно удаление идентификатора пользователя из сеанса[1]. Для этого воспользуемся методом `delete`:

```
session.delete(:user_id)
```

Мы также присвоим текущему пользователю значение `nil`, хотя в данном случае это не имеет значения из-за немедленного перенаправления к корневому URL[2].

---

[1] Некоторые браузеры поддерживают возможность автоматического восстановления сеанса, поэтому убедитесь, что она у вас выключена, прежде чем выйти.

[2] Присваивание значения `nil` переменной `@current_user` будет иметь значение, только если `@current_user` была создана до вызова `destroy` (а это не так) *и* не предполагается немедленной переадресации (а она предполагается). Это маловероятная комбинация событий, и в нашем приложении такой порядок действий совершенно необязателен, но он связан с обеспечением безопасности, поэтому я включил его для полноты картины.

По аналогии с методом log_in и другими родственными методами мы поместим метод log_out во вспомогательный модуль Sessions (листинг 8.26).

Листинг 8.26 ❖ Метод log_out (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход указанного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end
  .
  .
  .
  # Осуществляет выход текущего пользователя.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

Теперь поместим вызов метода log_out в метод destroy контроллера Sessions (листинг 8.27).

Листинг 8.27 ❖ Удаление сеанса (выход пользователя) (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
```

Чтобы протестировать механизм выхода, добавим несколько строк в тест входа из листинга 8.20. После входа вызовем delete, чтобы послать запрос DELETE по

маршруту выхода (табл. 8.1), и убедимся, что пользователь вышел, и произошел переход по корневому адресу URL. Также убедимся, что снова появилась ссылка для входа, а ссылки для выхода и на профиль пропали, как показано в листинге 8.28.

Листинг 8.28 ❖ Тест выхода пользователя **ЗЕЛЕНЫЙ**
(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
```

(Теперь, когда `is_logged_in?` доступен в тестах, мы добавили проверку `assert is_logged_in?` сразу после отправки верной информации.)

Реализовав и протестировав метод `destroy`, первоначальный триумвират регистрация/вход/выход завершен, а набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 8.29 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

8.4. Запомнить меня

Система входа, реализованная в разделе 8.2, вполне автономна и полностью функциональна, но у большинства веб-сайтов есть дополнительная способность запоминания сеанса пользователя, даже после закрытия браузера. В этом разделе мы сначала реализуем сохранение сеансов пользователей и будем завершать их толь-

ко в случае явного выхода. В разделе 8.4.5 мы воплотим другую распространенную модель, флажок «Запомнить меня», который позволит пользователям автоматически входить на сайт при следующем посещении. Обе эти модели – профессионального уровня, первая реализована на таких сайтах, как GitHub и Bitbucket, вторая – в Facebook и Twitter.

8.4.1. Узелок на память

В разделе 8.2 мы использовали Rails-метод `session`, чтобы сохранить идентификатор пользователя, но эта информация пропадает при закрытии браузера. В этом разделе мы сделаем первый шаг в направлении постоянных сеансов, сгенерировав специальный токен для создания постоянных cookies методом `cookies`, вместе с защищенным дайджестом, подтверждающим подлинность токена.

Как отмечалось в разделе 8.2.1, информация, сохраненная с помощью метода `session`, защищена по умолчанию, но это не относится к информации, сохраненной с помощью метода `cookies`. В частности, постоянные cookies уязвимы для атак вида «перехвата сеанса», когда злоумышленник использует украденный токен для входа под видом определенного пользователя. Есть четыре основных способа кражи cookies: (1) с помощью анализатора пакетов, когда cookies передаются по небезопасной сети[1], (2) взлом базы данных, где хранятся токены, (3) межсайтовый скриптинг (Cross-Site Scripting, XSS)[2] и (4) получение физического доступа к машине, с которой пользователь осуществил вход. Первую угрозу мы предотвратили в разделе 7.5, включив поддержку протокола защищенных сокетов (Secure Sockets Layer, SSL) на всем сайте, который защищает сетевые данные от анализаторов пакетов. Вторую угрозу мы предотвратим за счет хранения дайджеста токена вместо него самого, так же, как мы храним дайджест пароля вместо самого пароля (раздел 6.3). Rails автоматически борется с третьей угрозой, экранируя все, что попадает в шаблоны представлений. Наконец, несмотря на отсутствие непреодолимого способа остановить злоумышленника, получившего физический доступ к компьютеру, откуда осуществлен вход, мы минимизируем четвертую проблему за счет изменения токенов при каждом выходе-входе и *криптографического шифрования* всей возможной конфиденциальной информации, хранимой в браузере.

С учетом всего вышесказанного получается следующий план создания постоянных сеансов:

1) создать строку из случайных цифр для использования в качестве токена;
2) поместить токен в cookie и указать дату окончания действия далеко в будущем;
3) сохранить дайджест токена в базе данных;

---

[1] Перехват сеанса получил широкую огласку благодаря приложению Firesheep, которое показало видимость токенов на многих сайтах при подключении к публичным Wi-Fi-сетям.

[2] https://ru.wikipedia.org/wiki/Межсайтовый_скриптинг.

4) поместить зашифрованную версию идентификатора пользователя в cookie;
5) при получении cookie с постоянным идентификатором необходимо найти пользователя в базе данных и убедиться, что токен в cookie совпадает со связанным дайджестом в базе данных.

Обратите внимание, насколько последний этап похож на реализацию входа пользователя, когда мы извлекали информацию о пользователе по адресу электронной почты, а затем проверяли (методом `authenticate`) совпадение пароля с его дайджестом в базе данных (листинг 8.5). В результате такого совпадения наша реализация будет соответствовать аспектам `has_secure_password`.

Начнем с добавления атрибута `remember_digest` в модель User (рис. 8.9).

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string

Рис. 8.9 ❖ Модель User с дополнительным атрибутом `remember_digest`

Чтобы добавить эту модель данных в приложение, нужно сгенерировать миграцию:

```
$ rails generate migration add_remember_digest_to_users \
>remember_digest:string
```

(Сравните с миграцией после добавления дайджеста пароля в разделе 6.3.1.) Как и в прошлый раз, дадим миграции имя, оканчивающееся на `_to_users`, чтобы указать Rails на ее предназначение – изменить таблицу `users` в базе данных. Поскольку мы также указали атрибут (`remember_digest`) и его тип (`string`), Rails создаст миграцию по умолчанию (листинг 8.30).

Листинг 8.30 ❖ Миграция для дайджеста (db/migrate/[timestamp]_add_remember_digest_to_users.rb)

```
class AddRememberDigestToUsers < ActiveRecord::Migration
  def change
    add_column :users, :remember_digest, :string
  end
end
```

Так как мы не собираемся извлекать данные о пользователях по этому атрибуту, нет необходимости добавлять индекс для столбца `remember_digest`, и мы можем просто применить эту миграцию:

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

Теперь необходимо решить, что использовать в качестве токена. Существует множество практически эквивалентных подходов – нам подойдет любая длинная случайная строка. Метод `urlsafe_base64` из модуля `SecureRandom` в стандартной библиотеке Ruby вполне соответствует нашим требованиям[1]: он возвращает случайную строку длиной в 22 символа, составленную из знаков A–Z, a–z, 0–9, «-» и «_» (всего 64 возможных знака, поэтому «base64»). Типичная строка base64[2] выглядит вот так:

```
$ rails console
>> SecureRandom.urlsafe_base64
=> "q5lt38hQDc_959PVoo6b7A"
```

Как нет ничего страшного в одинаковом пароле у двух пользователей[3], так же нет необходимости требовать уникальности токенов, но при ее наличии безопасность будет обеспечена лучше[4]. В случае со строками base64 каждый из 22 символов может принять 64 возможных значения, поэтому вероятность совпадения двух токенов ничтожна мала: $1/64^{22} = 2^{-132} \times 10^{-40}$. В качестве бонуса, если использовать строки base64, которые специально спроектированы безопасными для URL (об этом говорит их название `urlsafe_base64`), мы сможем с помощью того же генератора токена создавать ссылки для активации учетной записи и сброса пароля в главе 10.

Запоминание пользователей подразумевает создание токена и сохранение дайджеста токена в базе данных. Мы уже определили метод `digest` в тестовых данных (листинг 8.18) и можем использовать результаты нашего обсуждения для создания метода `new_token`. Как и `digest`, он не требует объекта пользователя, поэтому сделаем его методом класса[5]. В результате получим модель User, представленную в листинге 8.31.

Листинг 8.31 ❖ Добавление метода создания токена (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 }
```

---

[1] Мой выбор опирается на статью по теме «Запомнить меня» (http://railscasts.com/episodes/274-remember-me-reset-password).

[2] https://ru.wikipedia.org/wiki/Base64.

[3] На самом деле благодаря хэшированию с солью в bcrypt нет никакой возможности определить совпадение двух паролей.

[4] Если токен уникален, для перехвата сеанса злоумышленнику всегда будут необходимы две вещи – идентификатор пользователя и токен из cookie.

[5] Как правило, если методу не требуется экземпляр объекта, его следует определить как метод класса. Это решение окажется действительно важным в разделе 10.1.2.

```
  has_secure_password
  validates :password, length: { minimum: 6 }

  # Возвращает дайджест для указанной строки.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def User.new_token
    SecureRandom.urlsafe_base64
  end
end
```

Наша следующая задача – определить метод `user.remember`, который свяжет токен с пользователем и сохранит соответствующий дайджест в базу данных. Благодаря миграции из листинга 8.30 в модели User уже имеется атрибут `remember_digest`, но пока нет атрибута `remember_token`. Нам нужно сделать токен доступным через `user.remember_token` (чтобы сохранить в cookie) *без* хранения его в базе данных. Мы решили подобную проблему с безопасными паролями в разделе 6.3, создав виртуальный атрибут `password`, парный для защищенного атрибута `password_digest` в базе данных. Тогда виртуальный атрибут `password` автоматически создавался методом `has_secure_password`, но код для `remember_token` нам придется написать самим. Для создания доступного атрибута нужно использовать `attr_accessor`, который до этого мы видели в разделе 4.4.5:

```
class User < ActiveRecord::Base
  attr_accessor :remember_token
  .
  .
  .
  def remember
    self.remember_token = ...
    update_attribute(:remember_digest, ...)
  end
end
```

Обратите внимание на форму присваивания в первой строке метода `remember`. Из-за особенностей присваивания внутри объектов без ссылки `self` эта операция создаст локальную переменную `remember_token`, которая нам совсем не нужна. Ссылка `self` гарантирует, что присваивание будет выполнено пользовательскому атрибуту `remember_token`. (Теперь вы знаете, почему в функции обратного вызова `before_save`, в листинге 6.31, стоит `self.email`, а не просто `email`.) Во второй строке метода `remember` вызывается метод `update_attribute`, обновляющий дайджест токена. (Как отмечалось в разделе 6.1.5, он минует проверки, а это необходимо в данном случае, так как у нас нет доступа к паролю пользователя или его подтверждению.)

Из всего вышесказанного следует, что токен и связанный с ним дайджест можно создать, сначала вызвав `User.new_token`, а затем обновив дайджест вызовом `User.digest`. Эту процедуру выполняет метод `remember`, показанный в листинге 8.32.

Листинг 8.32 ❖ Метод remember в модели User **ЗЕЛЕНЫЙ** (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, length: { minimum: 6 }

  # Возвращает дайджест для указанной строки.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Запоминает пользователя в базе данных для использования в постоянных сеансах.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end
end
```

8.4.2. Вход с запоминанием

После создания действующего метода `user.remember` можно приступать к реализации поддержки постоянных сеансов, сохраняя идентификаторы пользователей (зашифрованные) и токены в постоянных cookies в браузере. Для этого нам понадобится метод `cookies`, с ним (как и с `session`) можно обращаться, как с хэшем. Каждый cookie состоит из двух частей, значения и необязательного срока действия. Например, можно создать постоянный сеанс, если создать cookie с токеном и установить срок действия 20 лет:

```
cookies[:remember_token] = { value:   remember_token,
                             expires: 20.years.from_now.utc }
```

(Здесь использована одна из удобных вспомогательных функций для работы со временем, имеющихся в Rails, о которых рассказывается в блоке 8.2.) Шаблон

установки срока действия cookie на 20 лет настолько распространен, что в Rails есть специальный метод permanent для его реализации, поэтому можно просто написать

```
cookies.permanent[:remember_token] = remember_token
```

В этом случае Rails установит срок действия на 20.years.from_now автоматически.

Блок 8.2 ❖ Срок действия cookie истекает через 20 лет

Вы наверняка помните, как рассказывалось в разделе 4.4.2, что Ruby позволяет добавлять методы в любые, даже встроенные, классы. В том разделе мы добавляли метод palindrome? в класс String (и в результате обнаружили, что "deified" является палиндромом). Там же мы видели, что Rails добавляет метод blank? в класс Object (благодаря чему все три вызова – "".blank?, " ".blank? и nil.blank? – возвращают true). Метод cookies.permanent, который создает «постоянные» cookies со сроком действия 20.years.from_now, – еще один пример этой практики – посредством одной из вспомогательных функций для работы со временем, которые Rails добавляет в класс Fixnum (базовый класс для чисел):

```
$ 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 добавляет и другие функции:

```
>> 1.kilobyte
=> 1024
>> 5.megabytes
=> 5242880
```

Их удобно использовать для проверки объема выгружаемых данных и ограничивать, например, выгрузку изображений размером в 5.megabytes.
Эту возможность следует использовать с осторожностью, но гибкость добавления методов во встроенные классы позволяет создавать чрезвычайно естественные дополнения к обычному Ruby. В конце концов, элегантность Rails в значительной степени обусловлена податливостью лежащего в его основе языка Ruby.

Для сохранения идентификатора пользователя в cookie можно последовать за шаблоном применения метода session (листинг 8.12) и получить что-то вроде:

```
cookies[:user_id] = user.id
```

Этот метод сохраняет идентификатор в простом текстовом виде, поэтому он оставляет незащищенным формат cookie приложения и облегчает злоумышленникам взлом учетных записей. Чтобы устранить эту проблему, мы будем использовать *подписанные* cookies, которые шифруются перед отправкой браузеру:

```
cookies.signed[:user_id] = user.id
```

Так как нам требуется, чтобы идентификатор пользователя шел в паре с постоянным токеном, его тоже необходимо сделать постоянным, связав методы `signed` и `permanent`:

```
cookies.permanent.signed[:user_id] = user.id
```

После создания cookie в представлениях других страниц мы сможем извлечь информацию о пользователе посредством такого кода:

```
User.find_by(id: cookies.signed[:user_id])
```

при этом `cookies.signed[:user_id]` автоматически расшифрует идентификатор пользователя из cookie. Затем вызовом `bcrypt` можно убедиться в совпадении `cookies[:remember_token]` с `remember_digest`, созданным в листинге 8.32. (Для тех, кто удивляется, почему нельзя просто использовать зашифрованный идентификатор пользователя без токена, отмечу, что это позволит злоумышленнику при наличии зашифрованного идентификатора войти под видом пользователя на неограниченный срок. А в данном решении злоумышленник сможет войти под видом пользователя, только когда пользователь выйдет из системы.)

Последним кусочком мозаики является проверка совпадения данного токена с пользовательским, и в данном случае есть несколько равнозначных путей использования `bcrypt` для этого. Заглянув в исходный код реализации защищенных паролей (https://github.com/rails/rails/blob/master/activemodel/lib/active_model/secure_password.rb), можно увидеть такой способ сравнения[1]:

```
BCrypt::Password.new(password_digest) == unencrypted_password
```

В нашем случае аналогичный код будет выглядеть вот так:

```
BCrypt::Password.new(remember_digest) == remember_token
```

Если задуматься, он действительно выглядит необычно: кажется, что он сравнивает дайджест напрямую с токеном, а это означает *расшифровку* дайджеста для сравнения через ==. Но, как мы знаем, хэширование является необратимой операцией, поэтому так быть не может. В самом деле, если углубиться в исходный код гема `bcrypt`, можно обнаружить *переопределение* оператора сравнения ==, и внутри гема вышеуказанное сравнение эквивалентно следующему:

```
BCrypt::Password.new(remember_digest)
                .is_password?(remember_token)
```

Вместо == в нем использован логический метод `is_password?`, выполняющий сравнение. Так как его смысл немного яснее, мы задействуем в приложении именно эту, вторую форму сравнения.

Вышесказанное предполагает размещение сравнения дайджеста с токеном в методе `authenticated?` модели User, где он будет играть ту же роль, что и метод

---

[1] Как отмечалось в разделе 6.3.1, термин «шифрование» здесь не совсем подходит, так как защищенные пароли не шифруются, а хэшируются.

authenticate из has_secure_password в подтверждении подлинности пользователя (листинг 8.13). Реализация показана в листинге 8.33. (Метод authenticated? в листинге 8.33 привязан конкретно к дайджесту, но он будет не менее полезен и в других случаях, поэтому в главе 10 мы сделаем его более обобщенным.)

Листинг 8.33 ❖ Добавление метода authenticated? в модель 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, length: { minimum: 6 }

  # Возвращает дайджест для указанной строки.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Запоминает пользователя в базе данных для использования в постоянных сеансах.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end

  # Возвращает true, если указанный токен соответствует дайджесту.
  def authenticated?(remember_token)
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end
end
```

Обратите внимание: аргумент remember_token в методе authenticated? – это не то же самое средство доступа, что мы определили в листинге 8.32 при помощи attr_accessor :remember_token; это локальная переменная метода. (Так как аргумент относится к токену, нет ничего странного в использовании такого же названия.) Также обратите внимание на атрибут remember_digest – это то же самое, что и self.remember_digest, и, так же как name и email в главе 6, он автоматически создается

механизмом Active Record на основании имени соответствующего столбца в базе данных (листинг 8.30).

Теперь мы готовы запомнить вошедшего пользователя, добавив вызов вспомогательной функции `remember` вслед за `log_in`, как показано в листинге 8.34.

Листинг 8.34 ❖ Вход и запоминание пользователя (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

  def destroy
    log_out
    redirect_to root_url
  end
end
```

По аналогии с методом `log_in`, вся основная работа передается кодом из листинга 8.34 вспомогательному модулю SessionsHelper, где определен метод `remember`, вызывающий `user.remember`, который, в свою очередь, генерирует токен и сохраняет в базу данных его дайджест. Затем он вызывает метод `cookies`, чтобы создать постоянный cookie, как было описано выше. Результат показан в листинге 8.35.

Листинг 8.35 ❖ Запоминание пользователя (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Запоминает пользователя в постоянном сеансе.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end
```

```
  # Возвращает текущего вошедшего пользователя (если есть).
  def current_user
    @current_user ||= User.find_by(id: session[:user_id])
  end

  # Возвращает true, если пользователь зарегистрирован, иначе возвращает false.
  def logged_in?
    !current_user.nil?
  end

  # Осуществляет выход текущего пользователя.
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

Теперь вход пользователя будет запоминаться, в том смысле, что браузер получит соответствующий токен, но пока от этого мало пользы, так как метод `current_user` из листинга 8.14 знает только о временном сеансе:

```
@current_user ||= User.find_by(id: session[:user_id])
```

При наличии поддержки постоянных сеансов нам требуется сначала попытаться получить пользователя из временного сеанса и, только если отсутствует `session[:user_id]`, выполнить поиск по `cookies[:user_id]`. Сделать это можно так:

```
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
```

(Здесь все происходит по схеме `user && user.authenticated` из листинга 8.5.) Этот код будет работать, но обратите внимание на повторение вызовов `session` и `cookies`. Это дублирование можно устранить:

```
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
```

Здесь использована распространенная, но, возможно, не очень понятная конструкция

```
if (user_id = session[:user_id])
```

Несмотря на определенное сходство, это *не* сравнение (сравнение выполняется двумя знаками равенства ==), а присваивание. Если попробовать прочесть это выражение, получится никак не «Если значение id пользователя равно значению id в сеансе...», а нечто вроде «Если в сеансе существует id пользователя (при этом присвоим его переменной user_id) ...»[1].

Определение вспомогательной функции current_user, которое мы обсудили выше, приводит нас к реализации в листинге 8.36.

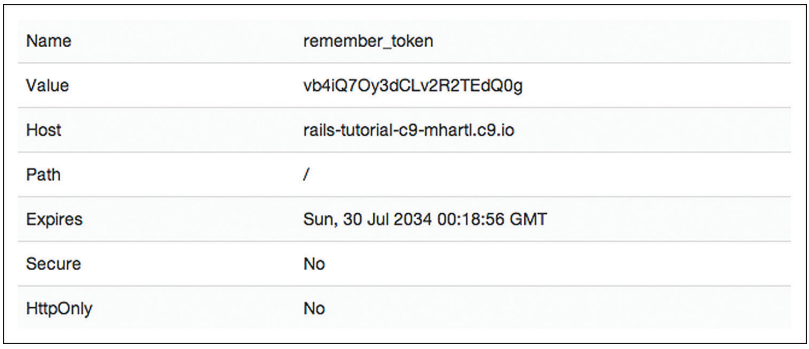
Листинг 8.36 ❖ Обновленный метод current_user с поддержкой постоянных сеансов **КРАСНЫЙ** (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Запоминает пользователя в постоянном сеансе.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Возвращает пользователя, соответствующего токену в 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

  # Возвращает true, если пользователь зарегистрирован, иначе возвращает false.
  def logged_in?
    !current_user.nil?
  end

  # Осуществляет выход текущего пользователя.
```

[1] Как правило, я следую соглашению и помещаю такое присваивание в скобки, которые визуально напоминают, что это не сравнение.

```
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end
end
```

Теперь вновь вошедшие пользователи запоминаются, в чем легко убедиться: войдите на сайт, закройте браузер, перезапустите учебное приложение и снова откройте страницу сайта – вы по-прежнему должны быть в статусе «вход выполнен». Если хотите, можете даже изучить cookies браузера, чтобы увидеть результат непосредственно (рис. 8.10)[1].



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

Рис. 8.10. Cookie с токеном в браузере

Сейчас в приложении осталась нерешенной только одна проблема: очистка cookies в браузере при необходимости, потому что это единственная возможность выйти. Такие вещи должен обнаруживать набор тестов, и действительно, в данный момент он **КРАСНЫЙ**:

Листинг 8.37 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

8.4.3. Забыть пользователя

Чтобы дать пользователям возможность выхода, определим методы, с помощью которых будем забывать пользователя, по аналогии с методами для запоминания. Получившийся метод `user.forget` просто отменяет все сделанное методом `user.remember`, присваивая дайджесту значение `nil`, как показано в листинге 8.38.

Листинг 8.38 ❖ Добавление метода forget в модели User (app/models/user.rb)

```
class User < ActiveRecord::Base
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
```

[1] Погуглите по фразе «<название браузера> изучить cookies», чтобы узнать, как исследовать cookies в вашей системе.

```
  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 }

  # Возвращает дайджест данной строки.
  def User.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def User.new_token
    SecureRandom.urlsafe_base64
  end

  # Запоминает пользователя в базе данных для использования в постоянном сеансе.
  def remember
    self.remember_token = User.new_token
    update_attribute(:remember_digest, User.digest(remember_token))
  end

  # Возвращает true, если токен совпадает с дайджестом.
  def authenticated?(remember_token)
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end

  # Забывает пользователя
  def forget
    update_attribute(:remember_digest, nil)
  end
end
```

Теперь, после добавления вспомогательного метода `forget` и его вызова в методе `log_out` (листинг 8.39), все готово к закрытию постоянного сеанса. Метод `forget` вызывает `user.forget`, а затем удаляет cookies`user_id` и `remember_token`.

Листинг 8.39 ❖ Выход из постоянного сеанса (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end
  .
  .
  .
```

```
  # Закрывает постоянный сеанс.
  def forget(user)
    user.forget
    cookies.delete(:user_id)
    cookies.delete(:remember_token)
  end

  # Осуществляет выход текущего пользователя.
  def log_out
    forget(current_user)
    session.delete(:user_id)
    @current_user = nil
  end
end
```

8.4.4. Две тонкости

Теперь осталось разобраться с двумя тесно связанными тонкостями. Первая имеет отношение к ссылке **Logout** (Выход): даже при том, что она появляется только после входа, может так случиться, что у пользователя сайт будет открыт сразу в нескольких окнах браузера. Если в таком случае пользователь выполнит выход в одном окне, сбросив `current_user` в `nil`, тогда щелчок на этой ссылке в другом окне приведет к ошибке из-за вызова `forget(current_user)` в методе `log_out` (листинг 8.39)[1]. Этого можно избежать, если осуществлять выход, только когда пользователь является вошедшим в систему.

Вторая проблема может возникнуть, если пользователь попытается выполнить вход (с сохранением сеанса) в нескольких браузерах, например в Chrome и Firefox, затем выйти в одном из них и после этого попытаться закрыть сеанс в другом[2]. Предположим, что пользователь осуществил выход в Firefox, в результате чего дайджесту токена было присвоено значение `nil` (вызовом `user.forget` в листинге 8.38). Приложение по-прежнему будет работать в Firefox: так как метод `log_out` в листинге 8.39 удалит идентификатор пользователя, переменная `user` будет иметь значение `nil` в методе `current_user`:

```
# Возвращает пользователя, соответствующего токену в 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
```

---

[1] Спасибо читателю Пауло Селио (Paulo Celio) за замечание.

[2] Спасибо читателю Нильсу де Рону (Niels de Ron) за замечание.

Как результат выражение:

```
user && user.authenticated?(cookies[:remember_token])
```

вернет `false` из-за вычислений по короткой схеме. (Переменная `user` содержит значение `nil`, которое в логическом контексте интерпретируется как `false`, поэтому второе выражение вычисляться не будет.) В Chrome, напротив, идентификатор пользователя *не* будет удален, поэтому переменная `user` не получит значения `nil` в

```
user && user.authenticated?(cookies[:remember_token])
```

и второе выражение будет вычислено. Это означает, что в методе `authenticated?` (листинг 8.33) произойдет вызов

```
def authenticated?(remember_token)
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

со значением `nil` в `remember_digest`, что приведет к ошибке в вызове `BCrypt::Password.new(remember_digest)`. Вместо этого нам нужно, чтобы в таком случае `authenticated?` возвращал `false`.

Это как раз тот самый случай, когда очень полезно вести разработку через тестирование, поэтому, прежде чем корректировать ошибки, мы напишем тесты для их выявления. Сначала мы добьемся неудачи в интеграционном тесте из листинга 8.28, как показано в листинге 8.40.

Листинг 8.40 ❖ Проверка выхода пользователя **КРАСНЫЙ**
(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_toroot_url
    # Сымитировать щелчок на ссылке для выхода во втором окне.
    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
```

Второй вызов `delete logout_path` в листинге 8.40 должен вызывать ошибку из-за отсутствия `current_user`, а значит, тест теперь **КРАСНЫЙ**:

Листинг 8.41 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

В код приложения необходимо просто добавить условие вызова `log_out`: только если `logged_in?` возвращает `true` (листинг 8.42).

Листинг 8.42 ❖ Выполнять выход, только когда пользователь определяется как вошедший **ЗЕЛЕНЫЙ** (app/controllers/sessions_controller.rb)

```
class SessionsController < ApplicationController
  .
  .
  .
  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

Второй случай с двумя разными браузерами сложнее сымитировать в интеграционных тестах, но его легко проверить прямо в тестах модели User. Нужно лишь начать с пользователя, у которого нет дайджеста (это будет верно для переменной `@user`, определенной в методе `setup`), а затем вызвать `authenticated?` (листинг 8.43). (Обратите внимание, что токен мы оставили пустым; его значение не играет роли, так как ошибка возникает еще до его использования.)

Листинг 8.43 ❖ Проверка вызова authenticated? с несуществующим дайджестом **КРАСНЫЙ** (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
```

Так как `BCrypt::Password.new(nil)` вызовет ошибку, набор тестов в данный момент **КРАСНЫЙ**:

Листинг 8.44 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Чтобы исправить ошибку и обеспечить успешное выполнение тестов, нужно лишь вернуть `false`, если дайджест равен `nil` (листинг 8.45).

Листинг 8.45 ❖ Добавление в authenticated? обработки несуществующего дайджеста **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
class User<ActiveRecord::Base
  .
  .
  .
  # Возвращает true, если указанный токен соответствует дайджесту.
  def authenticated?(remember_token)
    return false if remember_digest.nil?
    BCrypt::Password.new(remember_digest).is_password?(remember_token)
  end
end
```

Чтобы немедленно выйти, если дайджест равен `nil`, здесь использовано ключевое слово `return`. Это распространенный способ подчеркнуть, что остальная часть метода игнорируется в этом случае. Также можно использовать равнозначный код

```
if remember_digest.nil?
  false
else
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

но я предпочитаю ясность версии из листинга 8.45 (которая заодно получилась немного короче).

Теперь наш набор тестов должен быть **ЗЕЛЕНЫМ**, а это значит, что мы решили обе проблемы:

Листинг 8.46 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

8.4.5. Флажок «Запомнить меня»

Код, написанный в разделе 8.4.3, реализует в нашем приложении законченную систему аутентификации профессионального уровня. В качестве последнего шага мы посмотрим, как сделать необязательным сохранение статуса пользователя с помощью флажка «Запомнить меня». Макет формы входа с таким флажком изображен на рис. 8.11.

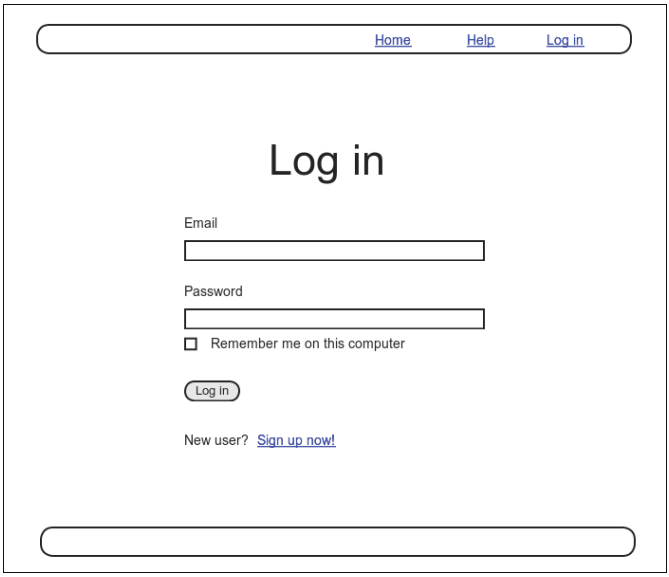


Рис. 8.11 ❖ Макет формы входа с флажком «Запомнить меня»

Начнем реализацию с добавления флажка в форму входа из листинга 8.2. Так же как метки, текстовые поля, поля паролей и кнопки отправки, флажки можно создавать с помощью вспомогательного метода Rails. Чтобы правильно применить стили, необходимо вложить флажок внутрь метки:

```
<%= f.label :remember_me, class: "checkboxinline" do %>
  <%= f.check_box :remember_me %>
  <span>Remember me on this computer</span>
<% end %>
```

Добавив этот код, мы получим форму входа, как показано в листинге 8.47.

Листинг 8.47 ❖ Добавление флажка «Запомнить меня» в форму входа (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>
```

В листинге 8.47 использованы CSS-классы `checkbox` и `inline`, которые в Bootstrap служат для размещения флажка и текста («Remember me on this computer» – «Запомнить меня на этом компьютере») в одну линию. Чтобы придать форме окончательное оформление, достаточно определить лишь несколько CSS-правил (листинг 8.48). Получившаяся форма входа показана на рис. 8.12.

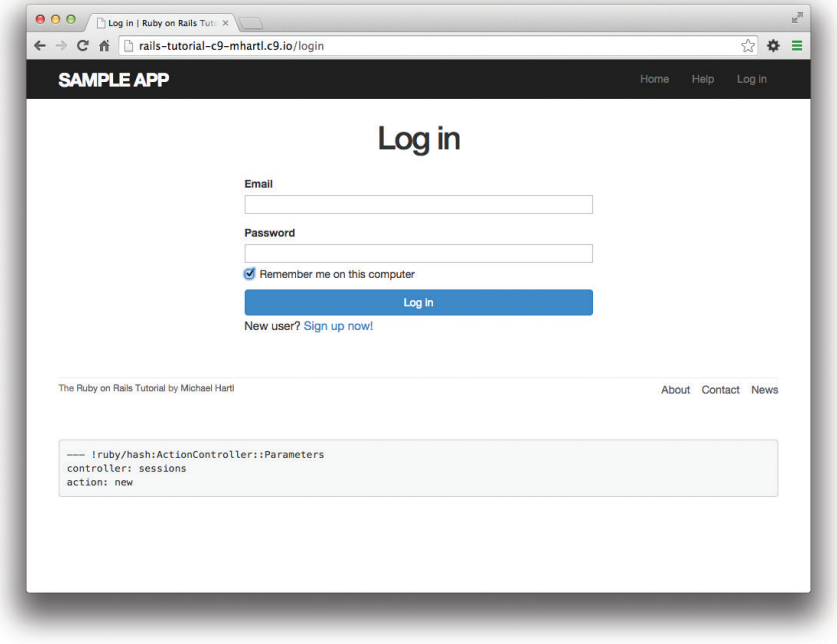


Рис. 8.12 ❖ Форма входа с флажком «Запомнить меня»

Листинг 8.48 ❖ Правила CSS для флажка «Запомнить меня» (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* формы */
.
.
```

```
.
.checkbox {
  margin-top: -10px;
  margin-bottom: 10px;
  span {
    margin-left: 20px;
    font-weight: normal;
  }
}

#session_remember_me {
  width: auto;
  margin-left: 0;
}
```

Покончив с формой, можно приступать к реализации запоминания пользователей при установленном флажке и забывать их в противном случае. Невероятно, но благодаря работе, проделанной в предыдущих разделах, вся реализация может уместиться в одной строке. Для начала отметим, что хэш `params` для отправленной формы входа теперь содержит значение флажка (убедиться в этом можно, отправив форму с недопустимой информацией и изучив значения в отладочном разделе страницы). В частности, значение

```
params[:session][:remember_me]
```

равно `'1'`, если флажок установлен, и `'0'` – в противном случае.

Проверив соответствующее значение в хэше `params`, мы можем теперь запоминать или забывать пользователя[1]:

```
if params[:session][:remember_me] == '1'
  remember(user)
else
  forget(user)
end
```

Как рассказывается в блоке 8.3, такой вариант ветвления `if-then` можно записать в одну строку с помощью *тернарного оператора*[2]:

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

Добавив это выражение в метод `create` контроллера Sessions, получим на удивление компактный код (листинг 8.49). (Теперь вам должен быть понятен код из

---

[1] Обратите внимание, что снятие флажка означает выход пользователя во всех браузерах на всех компьютерах. Альтернативный вариант независимого запоминания пользователя в каждом браузере был бы более удобен для пользователей, но он менее безопасен и сложнее в реализации. Предлагаю амбициозным читателям самим попробовать свои силы в реализации этого варианта.

[2] До этого мы писали `remember user` без скобок, но если их опустить в тернарном операторе, получится синтаксическая ошибка.

листинга 8.18, где тернарный оператор используется для определения bcrypt-переменной cost.)

Листинг 8.49 ❖ Обработка флажка «Запомнить меня» (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
```

Теперь система входа полностью завершена, в чем можно убедиться, снимая и устанавливая флажок в браузере.

Блок 8.3 ❖ 10 типов людей

Одна старая шутка гласит, что в мире есть 10 типов людей: те, кто понимает двоичную систему счисления, и те, кто не понимает (10, конечно же, – это 2 в двоичном счислении). В этом же духе мы можем сказать, что в мире есть 11 типов людей: те, кому нравится тернарный оператор, те, кому он не нравится, и те, кто пока не знает о нем. (Если вам случилось быть в третьей категории, скоро вы ее покинете.)
Когда вы много программируете, вы быстро узнаете, что на практике для управления потоком выполнения часто используется такой прием:

```
if boolean?
  do_one_thing
else
  do_something_else
end
```

Ruby, как и многие другие языки (включая C/C++, Perl, PHP и Java), позволяет заменить этот код более компактным выражением с тернарным оператором (называется так потому, что состоит из трех частей):

```
boolean? ? do_one_thing : do_something_else
```

Тернарный оператор можно также использовать для замены присваивания:

```
if boolean?
  var = foo
else
  var = bar
end
```

выражением следующего вида:

```
var = boolean? ? foo : bar
```

Наконец, тернарный оператор часто бывает удобно использовать для возвращения значения из функции:

```
def foo
  do_stuff
  boolean? ? "bar" : "baz"
end
```

Поскольку Ruby неявно возвращает значение последнего выражения в функции, здесь метод `foo` возвращает `"bar"` или `"baz"` в зависимости от значения `boolean?` – `true` или `false`.

8.4.6. Проверка запоминания

Функция «Запомнить меня» готова, и теперь важно написать тесты для проверки ее поведения. Одна из причин – выявление ошибок реализации, и мы обсудим ее чуть ниже. Но еще более важная причина состоит в том, что в данный момент основной код запоминания пользователя фактически не охвачен тестами. Исправление этих недостатков потребует некоторой хитрости, но в результате мы получим гораздо более мощный набор тестов.

Тестирование флажка «Запомнить меня»

Когда я первоначально реализовал обработку флажка в листинге 8.49, вместо правильного выражения

```
params[:session][:remember_me] == '1' ? remember(user) : forget(user)
```

я написал

```
params[:session][:remember_me] ? remember(user) : forget(user)
```

В данном контексте `params[:session][:remember_me]` равно либо `'0'`, либо `'1'`, и оба эти значения являются истинными в логическом смысле, поэтому выражение *является истинным всегда*, и приложение будет работать так, как будто флажок всегда установлен. Это как раз та ошибка, которую могут выявить тесты.

Так как запоминание пользователей требует, чтобы они вошли в систему, первым делом определим вспомогательный метод, осуществляющий вход пользова-

теля внутри тестов. В листинге 8.20 мы делали это с помощью метода post и хэша session, но делать так каждый раз весьма обременительно. Чтобы избежать ненужного повторения, напишем вспомогательный метод log_in_as.

Метод, реализующий вход пользователя, зависит от типа тестов. Внутри интеграционных тестов мы можем отправить запрос post, как в листинге 8.20, но в других тестах (например, контроллера и модели) это не сработает, и необходимо напрямую вызывать метод session. Итак, log_in_as должен определить тип теста и выполнить соответствующие операции. Определить разницу между интеграционными и всеми остальными тестами можно с помощью удобного Ruby-метода defined?, который возвращает true, если его аргумент определен, и false в противном случае. В данной ситуации метод post_via_redirect (мы видели его в листинге 7.26) доступен только в интеграционных тестах, поэтому вызов

```
defined?(post_via_redirect) ...
```

вернет true внутри интеграционных тестов и false в любых других. Это предполагает определение логического метода integration_test? и использование конструкции if-then:

```
ifintegration_test?
  # Выполнить вход отправкой запроса post
else
  # Выполнить вход с помощью метода session
end
```

Заместив строки с комментариями соответствующим кодом, получим вспомогательный метод log_in_as (листинг 8.50). (Это довольно сложный метод, поэтому будет очень здорово, если вы сможете сами разобраться в нем.)

Листинг 8.50 ❖ Вспомогательный метод log_in_as (test/test_helper.rb)

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
fixtures :all

  # Возвращает true, если тестовый пользователь вошел.
  def is_logged_in?
    !session[:user_id].nil?
  end

  # Выполняет вход тестового пользователя.
  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

    # Возвращает true внутри интеграционного теста.
    def integration_test?
      defined?(post_via_redirect)
    end
end
```

Обратите внимание, что для максимальной гибкости метод `log_in_as` принимает хэш `options` (как в листинге 7.31) с параметрами, хранящими значения по умолчанию пароля и флажка «Запомнить меня»: `'password'` и `'1'`, соответственно. В частности, так как хэши возвращают `nil` для несуществующих ключей, код

```
remember_me = options[:remember_me] || '1'
```

присвоит значение указанного параметра при его наличии либо значение по умолчанию (вычисление выполняется по короткой схеме, как описано в блоке 8.1).

Чтобы проверить поведение флажка «Запомнить меня», напишем два теста: один – для случая установки флажка и один – для случая его отсутствия. Это легко сделать с помощью метода входа из листинга 8.50:

```
log_in_as(@user, remember_me: '1')
```

и

```
log_in_as(@user, remember_me: '0')
```

(Так как `'1'` – это значение по умолчанию для `remember_me`, этот параметр можно опустить в первом случае, но я включил его, чтобы параллельность структуры была более очевидной.)

После входа пользователя можно проверить его запоминание по ключу `remember_token` в `cookies`. В идеале следовало бы проверить значение cookie на равенство токену, но в настоящей реализации тестов у нас нет никакой возможности получить к нему доступ: в переменной `user` в контроллере есть нужный атрибут с токеном, но (так как `remember_token` является виртуальным) в переменной `@user` в тестах такого атрибута нет. Исправление этого незначительного недостатка я оставляю в качестве упражнения (раздел 8.6), но сейчас мы можем просто проверить, является ли соответствующий cookie значением `nil`.

Есть одна небольшая проблема – по некоторым причинам метод `cookies` не работает внутри тестов с символами в качестве ключей, поэтому выражение

```
cookies[:remember_token]
```

всегда будет возвращать `nil`. К счастью, `cookies` работает со строковыми ключами, поэтому выражение

```
cookies['remember_token']
```

вернет нужное нам значение. Получившиеся тесты представлены в листинге 8.51. (Напомню, как было показано в листинге 8.20, `users(:michael)` ссылается на тестовые данные с информацией о пользователе из листинга 8.19.)

Листинг 8.51 ❖ Тест флажка «Запомнить меня» **ЗЕЛЕНЫЙ** (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
```

Предполагая, что вы не допустили в реализации той же ошибки, что и я, набор тестов теперь должен быть **ЗЕЛЕНЫМ**:

Листинг 8.52 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Тестирование восстановления сеанса

В разделе 8.4.2 мы вручную проверили работоспособность постоянного сеанса, реализованного в предыдущих разделах, но соответствующая ветка в методе `current_user` на данный момент совершенно не протестирована. В подобных ситуациях я предпочитаю вызвать исключения в блоках кода, не охваченных тестированием: если код не охвачен тестами, они все так же будут выполняться; в противном случае полученная ошибка укажет на соответствующий тест. Результат применения такого подхода показан в листинге 8.53.

Листинг 8.53 ❖ Вызов исключения в непроверенной ветке **ЗЕЛЕНЫЙ** (app/helpers/sessions_helper.rb)

```
module SessionsHelper
  .
  .
  .
  # Возвращает пользователя, соответствующего токену в 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    # Если тест выполнится успешно, значит, эта ветвь не охвачена тестированием.
      user = User.find_by(id: user_id)
      if user && user.authenticated?(cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

И в данный момент тесты выполняются успешно:

Листинг 8.54 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Конечно же, это проблема, так как код в листинге 8.53 не работает. Кроме того, проверять постоянные сеансы вручную очень утомительно, поэтому если предполагается позднее провести рефакторинг метода `current_user` (мы сделаем это в главе 10), очень важно протестировать его.

Так как вспомогательный метод `log_in_as` (листинг 8.50) автоматически устанавливает `session[:user_id]`, тестирование ветки восстановления в методе `current_user` довольно сложно организовать в интеграционных тестах. К счастью, это ограничение можно обойти, протестировав метод `current_user` прямо в тестах для вспомогательного модуля Sessions. Для этого необходимо создать файл:

```
$ touch test/helpers/sessions_helper_test.rb
```

Последовательность тестирования проста:

1) определить переменную `user` с использованием тестовых данных;
2) вызвать метод `remember`, чтобы запомнить данного пользователя;
3) убедиться, что `current_user` возвращает данного пользователя.

Так как метод `remember` не устанавливает `session[:user_id]`, эта операция проверит нужную ветку «восстановления». Результат представлен в листинге 8.55.

Листинг 8.55 ❖ Тест постоянного сеанса (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
```

Обратите внимание, что мы добавили еще один тест, сравнивающий текущего пользователя со значением `nil`, если дайджест не соответствует токену, и тем самым протестировали выражение `authenticated?` во вложенной инструкции `if`:

```
if user && user.authenticated?(cookies[:remember_token])
```

Между прочим, мы могли написать

```
assert_equalcurrent_user, @user
```

и получили бы тот же результат, но (как упоминалось в разделе 5.6) общепринятый порядок аргументов для `assert_equal` – *ожидаемое*, *действительное*:

```
assert_equal <expected>, <actual>
```

поэтому для примера в листинге 8.55 получим

```
assert_equal @user, current_user
```

Теперь набор тестов **КРАСНЫЙ**, как и требовалось:

Листинг 8.56 ❖ КРАСНЫЙ

```
$ bundle exec rake test TEST=test/helpers/sessions_helper_test.rb
```

Мы можем обеспечить успешное выполнение тестов, удалив `raise` и восстановив исходный метод `current_user` (листинг 8.57). (Убедиться, что второй тест в листинге 8.55 терпит неудачу, а значит, он тестирует все совершенно правильно, можно, удалив выражение `authenticated?`.)

Листинг 8.57 ❖ Удаление вызова исключения **ЗЕЛЕНЫЙ** (app/helpers/sessions_helper.rb)

```
module SessionsHelper
  .
  .
  .
  # Возвращает пользователя, соответствующего токену в 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
```

Теперь набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 8.58 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Теперь, когда ветвь восстановления в `current_user` протестирована, можно не сомневаться, что любые регрессии будут выявлены без проверки вручную.

8.5. Заключение

В последних двух главах мы многое узнали, превратив наше перспективное, но бесформенное приложение в сайт с полноценной поддержкой регистрации и входа/выхода. Чтобы получить законченную функциональность, нам осталось лишь реализовать проверку аутентичности – ограничить доступ к страницам для незарегистрированных пользователей. Мы решим эту задачу и заодно дадим пользователям возможность редактировать информацию о себе в главе 9.

Прежде чем продолжить, слейте изменения с основной ветвью:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Finish log in/log out"
$ git checkout master
$ git merge log-in-log-out
```

Затем отправьте в удаленный репозиторий и на сервер:

```
$ bundle exec rake test
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

Обратите внимание, что на короткое время после отправки приложение станет недоступным, пока не закончится миграция базы данных. На сайтах со значительным трафиком неплохо включать *режим обслуживания* (https://devcenter.heroku.com/articles/maintenance-mode) перед внесением изменений:

```
$ heroku maintenance:on
$ git push heroku
```

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

При этом в процессе развертывания и миграции будет показана стандартная страница с сообщением об ошибке. (Мы больше не будем с этим возиться, но неплохо увидеть такой этап хотя бы один раз.) За более подробной информацией обращайтесь к разделу с описанием *страниц ошибок* (https://devcenter.heroku.com/articles/error-pages) в документации Heroku.

8.5.1. Что мы узнали в этой главе

- Rails может сохранять состояние при переходе к следующей странице с помощью временных и постоянных cookies.
- Форма входа предназначена для создания нового сеанса и дает пользователю возможность выполнить вход.
- Метод `flash.now` применяется для вывода кратковременных сообщений.
- Разработка через тестирование особенно полезна при отладке, когда в тесте воспроизводится ошибка.
- Метод `session` позволяет безопасно передать идентификатор пользователя в браузер для создания временного сеанса.
- Мы можем изменять элементы сайта (такие как ссылки в шаблоне) в зависимости от статуса входа.
- Интеграционные тесты могут проверить правильность маршрутов, обновления в базе данных и корректность изменения шаблона.
- Каждому пользователю присваивается случайный токен и соответствующий дайджест для использования в постоянном сеансе.
- Метод `cookies` позволяет создать постоянный сеанс путем размещения постоянного cookie с токеном в браузере.
- Статус входа текущего пользователя определяется наличием его идентификатора во временном сеансе или уникального токена в постоянном сеансе.
- Приложение осуществляет выход пользователя путем удаления идентификатора из сеанса и постоянного cookie из браузера.
- Тернарный оператор – это компактный способ записи простых выражений `if-then`.

8.6. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. В листинге 8.32 мы определили методы класса `new_token` и `digest` путем явного добавления префикса `User`. Это отличное решение, и так как они действи-

тельно *вызываются* как `User.new_token` и `User.digest`, это, пожалуй, наиболее понятный способ для их определения. Но существуют два более корректных способа, один из них немного непонятный, а второй крайне непонятный. Запустив набор тестов, убедитесь в правильности реализаций из листинга 8.59 (немного непонятный) и листинга 8.60 (крайне непонятный). (Обратите внимание, что в данном случае `self` – это класс `User`, тогда как в других методах `self` ссылается на *экземпляр* объекта пользователя. Именно поэтому эти способы реализации не совсем понятны.)

2. Как было показано в разделе 8.4.6, в настоящее время приложение не имеет доступа к виртуальному атрибуту `remember_token` внутри интеграционных тестов из листинга 8.51. Однако такой доступ можно организовать с помощью специального тестового метода `assigns`. Внутри теста можно получить доступ к переменной *экземпляра*, определенной в контроллере, используя `assigns` с соответствующим символом. Например, если в методе `create` определить переменную `@user`, мы сможем обратиться к ней через `assigns(:user)`. Сейчас в методе `create` контроллера Sessions определена обычная переменная `user`, но если заменить ее переменной экземпляра, мы сможем протестировать наличие пользовательского токена в cookie. Заполните пропущенные элементы в листингах 8.61 и 8.62 (отмечены знаком вопроса `?` и `FILL_IN`) и завершите усовершенствованный тест флажка «Запомнить меня».

Листинг 8.59 ❖ Определение методов new_token и digest с использованием ссылки self **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
classUser<ActiveRecord::Base
  .
  .
  .
  # Возвращает хэш – дайджест указанной строки.
  def self.digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def self.new_token
    SecureRandom.urlsafe_base64
  end
.
.
.
end
```

Листинг 8.60 ❖ Определение методов new_token и digest с использованием class<<self **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  class << self
  # Возвращает хэш - дайджест указанной строки.
  def digest(string)
    cost = ActiveModel::SecurePassword.min_cost ?
           BCrypt::Engine::MIN_COST :
           BCrypt::Engine.cost
    BCrypt::Password.create(string, cost: cost)
  end

  # Возвращает случайный токен.
  def new_token
    SecureRandom.urlsafe_base64
  end
end
.
.
.
```

Листинг 8.61 ❖ Шаблон для использования переменной экземпляра вместо decreate (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
```

Листинг 8.62 ❖ Шаблон усовершенствованного теста «Запомнить меня» **ЗЕЛЕНЫЙ** (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 assigns(:user).FILL_IN, FILL_IN
  end

  test "login without remembering" do
    log_in_as(@user, remember_me: '0')
    assert_nil cookies['remember_token']
  end
  .
  .
  .
end
```

Глава 9

Обновление, отображение и удаление пользователей

В этой главе мы завершим реализацию REST-интерфейса для ресурса Users (табл. 7.1), добавив методы `edit`, `update`, `index` и `destroy`. Для начала дадим пользователям возможность обновлять свои профили, для чего, вполне естественно, будем вынуждать их авторизоваться (с использованием механизмов, реализованных в главе 8). Затем создадим список всех пользователей (также требующий аутентификации), что послужит причиной внесения тестовых данных в базу и реализации постраничного вывода. В заключение мы добавим возможность удаления пользователей, стирая информацию о них в базе данных. Так как мы не можем позволить любому пользователю обладать такими опасными полномочиями, позаботимся о создании привилегированного класса пользователей-администраторов, уполномоченных для удаления других пользователей.

9.1. Обновление пользователей

Механизм редактирования информации о пользователе имеет много общего с механизмом создания новых пользователей (глава 7). Вместо метода `new`, отображающего представление с формой для создания нового пользователя, у нас будет метод `edit` и представление с формой для редактирования информации о существующих пользователях; вместо `create`, отвечающего на запрос `POST`, у нас будет метод `update`, отвечающий на запрос `PATCH` (блок 3.2). Основное отличие заключается в том, что зарегистрироваться может любой человек, но только текущий пользователь должен иметь возможность изменять информацию о себе. Механизм аутентификации из главы 8 позволит использовать *предварительный фильтр*, чтобы гарантировать именно такое положение вещей.

Для начала перейдем в рабочую ветку `updating-users`:

```
$ git checkout master
$ git checkout -b updating-users
```

9.1.1. Форма редактирования

Сначала разберемся с формой редактирования, макет которой представлен на рис. 9.1[1]. Чтобы превратить ее в действующую страницу, необходимо создать метод `edit` в контроллере Users и представление для редактирования информации о пользователе. Начнем с метода `edit`, который извлекает информацию о соответствующем пользователе из базы данных. В табл. 7.1 указано, что страница редактирования информации о пользователе имеет URL: `/users/1/edit` (предполагается, что идентификатор пользователя равен `1`). Как вы наверняка помните, идентификатор пользователя доступен в переменной `params[:id]`, а это означает, что пользователя можно найти, как показано в листинге 9.1.

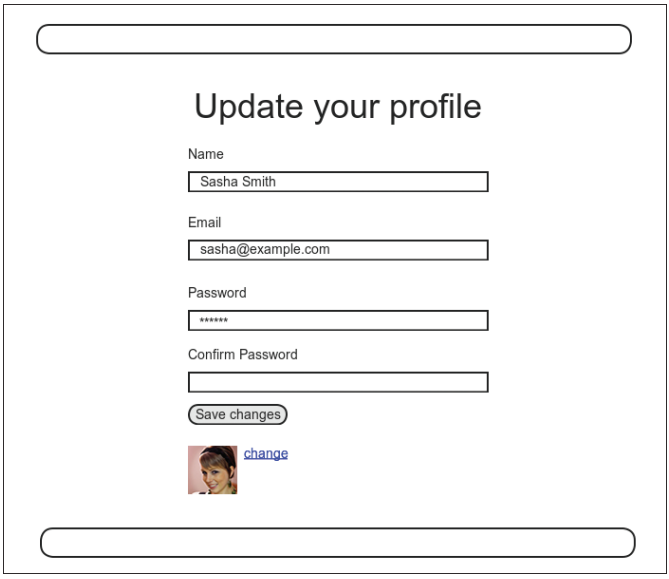


Рис. 9.1 ❖ Макет страницы редактирования пользователя

Листинг 9.1 ❖ Метод edit (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)
```

[1] Картинка взята со страницы: http://www.flickr.com/photos/sashawolff/4598355045/.

```
    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
```

Соответствующее представление (которое нужно создать вручную) показано в листинге 9.2. Обратите внимание, насколько близко оно похоже на представление для создания нового пользователя из листинга 7.13; значительное совпадение предполагает вынесение повторяющегося кода в частичный шаблон, но мы оставим это в качестве упражнения (раздел 9.6).

Листинг 9.2 ❖ Представление для редактирования пользователя (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>
```

Здесь мы повторно использовали частичный шаблон `error_messages` из раздела 7.3.3. Кстати, `target="_blank"` в ссылке на граватар – это изящный фокус, заставляющий браузер открыть страницу в новом окне или вкладке, что очень удобно, когда осуществляется переход по ссылке на сторонний сайт.

После определения переменной экземпляра `@user` в листинге 9.1 страница редактирования должна отображаться правильно, как показано на рис. 9.2. Поля **Name** и **Email** демонстрируют прием автоматического заполнения с использованием атрибутов существующей переменной `@user`.

Заглянув в исходный код HTML, можно увидеть тег формы, как показано в листинге 9.3 (мелкие детали могут отличаться).

Листинг 9.3 ❖ Разметка HTML-формы редактирования из листинга 9.2 и рис. 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>
```

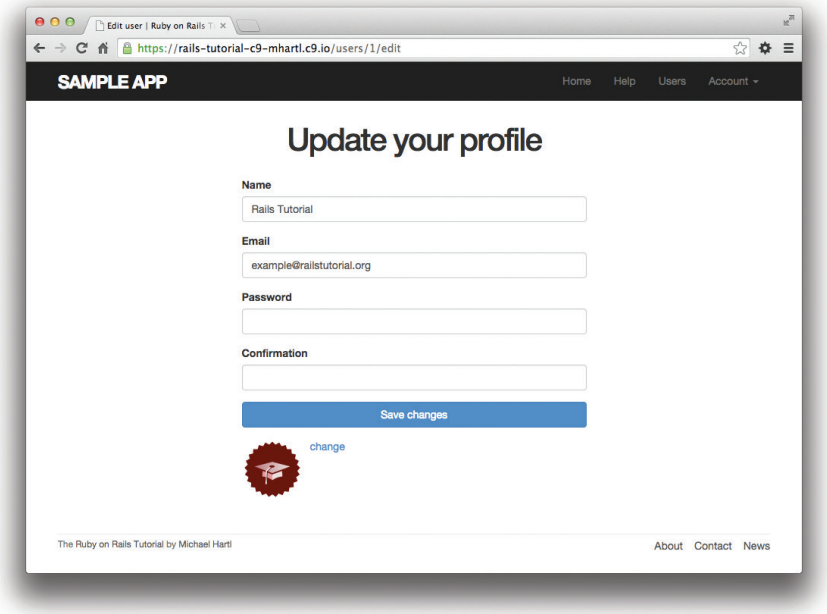


Рис. 9.2 ❖ Начальная страница редактирования пользователя с заполненными полями имени и электронной почты

Обратите внимание на скрытое поле ввода

```
<input name="_method" type="hidden" value="patch" />
```

Поскольку веб-браузеры сами не могут отправлять запросы PATCH (как требует от них REST- интерфейс, описанный в табл. 7.1), Rails подделывает их при помощи запроса POST и скрытого поля input[1].

Стоит также упомянуть еще одну тонкость: вызов form_for(@user) в листинге 9.2 *абсолютно* совпадает с вызовом в листинге 7.13. Но как же Rails узнает, что нужно использовать запрос POST для создания новых пользователей и PATCH для редактирования? Ответ кроется в логическом методе new_record?, с помощью которого можно определить, является пользователь новым или он уже существует в базе данных:

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

Конструируя форму с помощью form_for(@user), Rails использует POST, если @user.new_record? возвращает true, и PATCH, если возвращает false.

В качестве финального штриха добавим ссылку на настройки профиля пользователя в меню сайта. Это просто сделать, зная именованный маршрут edit_user_path из табл. 7.1 и имея удобный вспомогательный метод current_user из листинга 8.36:

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

Полный код приложения представлен в листинге 9.4.

Листинг 9.4 ❖ Добавление ссылки на настройки профиля пользователя в меню сайта (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>
```

[1] Не беспокойтесь о том, как это работает; детали представляют интерес для разработчиков самого фреймворка Rails, но, по их замыслу, не имеют значения для разработчиков Rails-приложений.

```
          </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. Неудача при редактировании

В этом разделе мы разберемся с ситуацией неудачи при редактировании, следуя идее неудачного входа (раздел 7.3). Начнем с создания метода `update`, который использует `update_attributes` (раздел 6.1.5) для обновления информации о пользователе на основе полученного хэша `params`, как показано в листинге 9.5. При получении недопустимой информации попытка обновления вернет `false`, и ветка `else` повторно отобразит страницу редактирования. Мы уже видели такое раньше; структура очень похожа на первую версию метода `create` (листинг 7.16).

Листинг 9.5 ❖ Первая версия метода update (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)
      # Обработать успешное изменение.
    else
      render 'edit'
    end
  end

  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end
end
```

Обратите внимание на использование `user_params` в вызове `update_attributes`, это означает использование строгих параметров для предотвращения уязвимости массового присваивания (как описано в разделе 7.3.2).

Благодаря наличию проверок в модели User и частичного шаблона `error_messages` в листинге 9.2 отправка недопустимой информации приведет к появлению сообщений об ошибках (рис. 9.3).

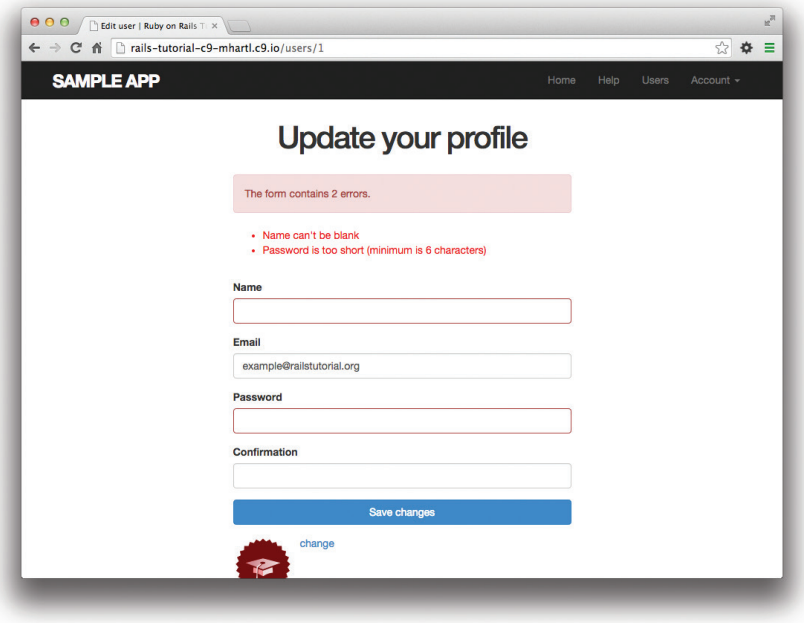


Рис. 9.3 ❖ Сообщение об ошибке после отправки формы редактирования

9.1.3. Тестирование неудачной попытки редактирования

В разделе 9.1.2 мы создали действующую форму редактирования. Следуя рекомендациям из блока 3.3, сейчас мы напишем интеграционный тест для выявления регрессий. Как обычно, сначала сгенерируем его:

```
$ rails generate integration_test users_edit
      invoke  test_unit
      create    test/integration/users_edit_test.rb
```

Затем напишем простой тест, определяющий неудачную попытку редактирования (листинг 9.6). Он контролирует поведение приложения, проверяя, отображается ли шаблон редактирования вновь после отправки недопустимой информации. Обратите внимание на метод `patch`, посылающий запрос `PATCH`, который работает по тому же принципу, что и методы `get`, `post` и `delete`.

Листинг 9.6 ❖ Тест для неудачной попытки редактирования **ЗЕЛЕНЫЙ** (test/integration/users_edit_test.rb)

```
require 'test_helper'

class UsersEditTest < 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
```

Сейчас набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 9.7 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

9.1.4. Успешная попытка редактирования (с TDD)

Теперь пришло время заставить работать форму редактирования. Редактирование изображения для профиля уже работает, поскольку мы переложили загрузку изображений на Gravatar; мы можем изменить аватар, щелкнув на ссылке «change», показанной на рис. 9.2, и перейти на страницу редактирования изображения, как показано на рис. 9.4. Давайте заставим так же хорошо работать остальной функционал редактирования пользователя.

Рис. 9.4 ❖ Интерфейс обрезки изображений на сайте Gravatar с фоткой какого-то чувака

Освоившись с тестированием, вы наверняка обнаружите, что интеграционные тесты полезно писать до прикладного кода, а не после. Такие тесты иногда называют *приемочными*, так как они определяют момент, когда конкретная функция может быть принята как завершенная. Чтобы понять суть, завершим разработку редактирования пользователей через тестирование.

Мы проверим правильность изменения информации о пользователе, написав тест, похожий на тот, что представлен в листинге 9.6, только на этот раз отправим допустимую информацию. Затем убедимся в наличии кратковременного сообщения и успешной переадресации на страницу профиля, а также удостоверимся в корректном изменении информации о пользователе в базе данных. Результат представлен в листинге 9.8. Обратите внимание, что параметры `password` и `password_confirmation` не заполнены, это удобно для пользователей, которые не хотят обновлять пароль каждый раз при изменении имени или адреса электронной почты. Метод `@user.reload` (впервые мы видели его в разделе 6.1.5) используется для повторного извлечения информации о пользователе из базы данных и подтверждения их успешного обновления. (Это та самая деталь, которую легко забыть на начальном этапе, поэтому приемочное тестирование (и вообще TDD) требует определенного опыта работы.)

Листинг 9.8 ❖ Тест успешной попытки редактирования **КРАСНЫЙ** (test/integration/users_edit_test.rb)

```
require 'test_helper'

class UsersEditTest < 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 @user.name, name
    assert_equal @user.email, email
  end
end
```

Метод `update`, необходимый для успешного прохождения тестов, очень похож на окончательный вариант метода `create` (листинг 8.22), как показано в листинге 9.9.

Листинг 9.9 ❖ Метод update **КРАСНЫЙ** (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
    else
      render 'edit'
    end
```

```
  end
  .
  .
  .
end
```

Как указано в заголовке листинга 9.9, набор тестов все еще **КРАСНЫЙ**, это объясняется проверкой длины пароля (листинг 6.39), она дает отрицательный результат для пустых параметров `password` и `password_confirmation` в листинге 9.8. Чтобы обеспечить успешное прохождение тестов, в проверке необходимо сделать исключение для пустых паролей. Реализовать это можно передачей в `validates` параметра `allow_nil: true` (листинг 9.10).

Листинг 9.10 ❖ Разрешение пустых паролей при обновлении **ЗЕЛЕНЫЙ** (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, length: { minimum: 6 }, allow_blank: true
  .
  .
  .
end
```

Если вам кажется, что этот код позволит новым пользователям регистрироваться с пустым паролем, напомню, что `has_secure_password` (раздел 6.3.3) содержит независимую проверку, которая как раз определяет пустые пароли. (Так как пустые пароли теперь пропускаются основной проверкой, но определяются методом `has_secure_password`, мы заодно решили проблему с дублированием сообщений об ошибке, упомянутую в разделе 7.3.3.)

Теперь страница редактирования информации о пользователе должна работать (рис. 9.5). Убедиться в этом можно, запустив набор тестов, который в данный момент должен быть **ЗЕЛЕНЫМ**:

Листинг 9.11 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

9.2. Авторизация

В мире веб-приложений *аутентификация* позволяет идентифицировать пользователей на сайте, а *авторизация* – ограничивать круг допустимых действий. Благодаря созданию механизма аутентификации в главе 8 мы теперь готовы к реализации авторизации.

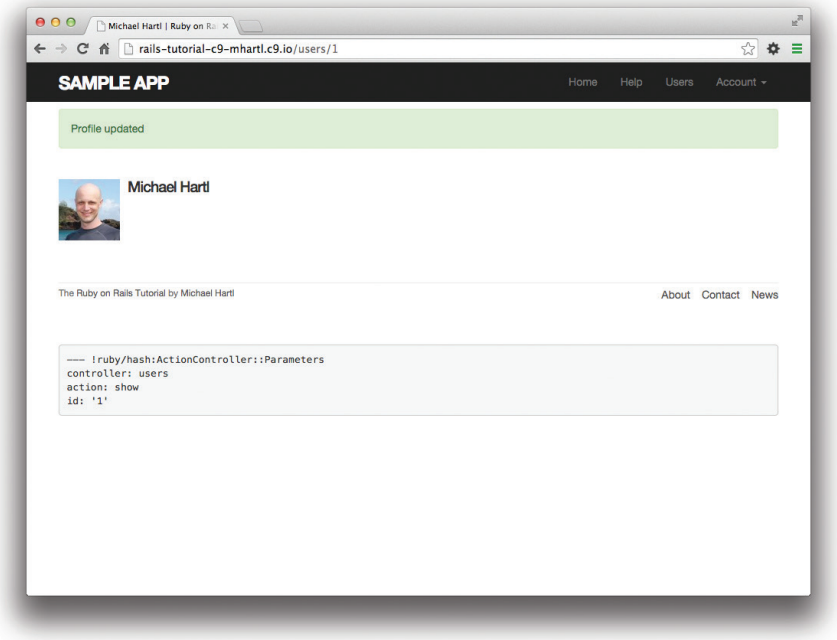


Рис. 9.5 ❖ Результат успешной попытки редактирования

Хотя методы `edit` и `update` из раздела 9.1 функционально завершены, они создают нелепую брешь в системе безопасности, позволяя любым пользователям (даже незарегистрированным) выполнять любые действия, а любой зарегистрированный пользователь может изменять информацию любого другого пользователя. В этом разделе мы реализуем модель безопасности, которая будет требовать от пользователей входа в систему и предотвращать обновление ими любой информации, кроме их собственной.

В разделе 9.2.1 мы разберемся с ситуацией, когда незарегистрированный пользователь пытается посетить защищенную страницу. Так как это может легко произойти в нормальном процессе использования приложения, такой пользователь будет направлен на страницу входа с соответствующим сообщением, как показано на рис. 9.6. С другой стороны, если пользователь пытается посетить страницу, на которую у него никогда не было разрешения (например, вошедший пользователь пытается попасть на страницу редактирования профиля другого пользователя), он должен быть переадресован на главную страницу (раздел 9.2.2).

9.2.1. Требование входа пользователей

Для реализации поведения, описанного выше, воспользуемся *предварительным фильтром* в контроллере Users. Предварительные фильтры применяют команду `before_action`, чтобы обеспечить вызов конкретного метода до определенных дей-

ствий[1]. Чтобы потребовать от пользователей выполнить вход, мы определим метод `logged_in_user` и вызовем его через `before_action :logged_in_user`, как показано в листинге 9.12.

Рис. 9.6 ❖ Макет страницы, отображаемой при попытке посетить защищенную страницу

Листинг 9.12 ❖ Добавление предварительного фильтра logged_in_user **КРАСНЫЙ** (app/controllers/users_controller.rb)

```
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

    # Предварительные фильтры
```

[1] Раньше команда установки предварительного фильтра называлась `before_filter`, но основная группа разработчиков Rails решила переименовать ее, чтобы подчеркнуть, что фильтр выполняется перед определенным действием контроллера.

```
    # Подтверждает вход пользователя.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end
end
```

По умолчанию предварительные фильтры применяются ко всем методам контроллера, поэтому мы явно ограничили фильтр только методами `:edit` и `:update`, передав соответствующий хэш `:only`.

Чтобы увидеть результат работы фильтра, выйдите с сайта и попытайтесь посетить страницу редактирования пользователя `/users/1/edit` (рис. 9.7).

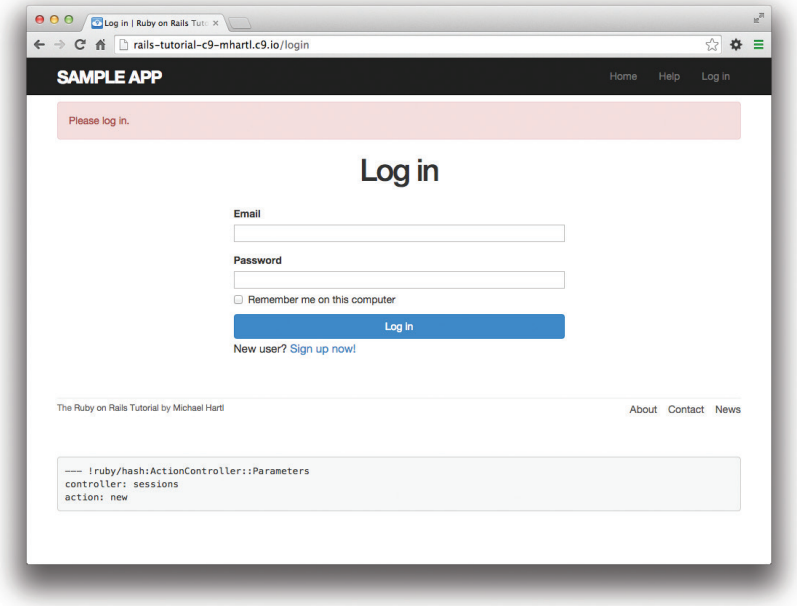


Рис. 9.7 ❖ Форма входа после попытки посетить защищенную страницу

Как указано в заголовке листинга 9.12, сейчас набор тестов **КРАСНЫЙ**:

Листинг 9.13 ❖ **КРАСНЫЙ**

```
$ bundle exec rake test
```

Причина в том, что теперь методы `edit` и `update` требуют осуществить вход, но в соответствующих тестах нет ни одного вошедшего на сайт пользователя.

Мы исправим тесты, осуществив вход пользователя до вызова методов `edit` и `update`. Это легко сделать во вспомогательной функции `log_in_as` из раздела 8.4.6 (листинг 8.50), как показано в листинге 9.14.

Листинг 9.14 ❖ Вход тестового пользователя **ЗЕЛЕНЫЙ** (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
```

(Мы могли бы избежать дублирования, поместив вызов `log_in_as` в метод `setup`, но в разделе 9.2.3 мы изменим один из тестов для посещения страницы редактирования до входа, а это будет невозможно, если вход будет происходить в момент подготовки тестов.)

Сейчас набор тестов должен быть **ЗЕЛЕНЫМ**.

Листинг 9.15 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Несмотря на успешное прохождение набора тестов, работа над предварительными фильтрами еще не закончена, так как тесты будут все такими же **ЗЕЛЕНЫМИ**, даже если удалить модель безопасности, – в этом можно убедиться, закомментировав ее (листинг 9.16). Это очень плохо – из всех регрессий, которые наши тесты должны определять, массивная дыра в безопасности стоит, пожалуй, на первом месте, и код в листинге 9.16 должен несомненно приводить к неудаче во время тестирования. Давайте напишем тесты для этого.

Листинг 9.16 ❖ Закомментированный предварительный фильтр для проверки модели безопасности **ЗЕЛЕНЫЙ** (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController
  # before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

Так как предварительный фильтр выполняется перед вызовами методов контроллера, мы поместим соответственные тесты в комплект тестов для контроллера Users. План заключается в том, чтобы попасть в методы `edit` и `update`, послав соответствующие запросы, проверить наличие кратковременных сообщений и переадресовать пользователя на страницу входа. Согласно табл. 7.1, правильными будут запросы `GET` и `PATCH`, соответственно. Значит, внутри тестов нужно использовать методы `get` и `patch`, как показано в листинге 9.17.

Листинг 9.17 ❖ Тестирование защищенности методов edit и update **КРАСНЫЙ** (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

  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
```

Обратите внимание на аргументы в вызовах методов `get`:

```
get :edit, id: @user
```

и `patch`:

```
patch :update, id: @user, user: { name: @user.name, email: @user.email }
```

Здесь применено соглашение Rails о `id: @user`, которое автоматически преобразуется в `@user.id`. Чтобы получить верные маршруты, во втором случае необходимо передать дополнительный хэш `user`. (Если заглянуть в сгенерированные тесты для контроллера Users из мини-приложения в главе 2, можно увидеть код, показанный выше.)

Теперь набор тестов должен быть **КРАСНЫМ**, как и требовалось. Чтобы он стал **ЗЕЛЕНЫМ**, просто раскомментируйте предварительный фильтр (листинг 9.18).

Листинг 9.18 ❖ Раскомментированный предварительный фильтр **ЗЕЛЕНЫЙ** (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:edit, :update]
  .
  .
  .
end
```

И тесты снова выполняются успешно:

Листинг 9.19 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Теперь любое случайное обращение к методам редактирования со стороны неуполномоченных пользователей будет немедленно обнаружено набором тестов.

9.2.2. Требование наличия прав у пользователя

Конечно, требования выполнить вход недостаточно; пользователи должны иметь право редактировать только собственную информацию. Как мы видели в разделе 9.2.1, набор тестов легко может пропустить существенную брешь в безопасности, поэтому мы продолжим разработку через тестирование, чтобы убедиться в корректной реализации модели безопасности. Для этого дополним тесты контроллера Users из листинга 9.17.

Чтобы убедиться, что пользователь не может редактировать информацию другого пользователя, нужно иметь возможность осуществить вход второго пользователя. А это означает, что нужно добавить второго пользователя в файл тестовых данных, как показано в листинге 9.20.

Листинг 9.20 ❖ Добавление второго пользователя в файл с тестовыми данными (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') %>
```

Применив метод `log_in_as` из листинга 8.50, можно проверить методы `edit` и `update` (листинг 9.21). Обратите внимание, что в данном случае ожидается переадресация пользователя на главную страницу вместо страницы входа, так как он пытается отредактировать информацию другого пользователя, уже выполнив вход.

Листинг 9.21 ❖ Тесты попытки редактирования профиля другого пользователя **КРАСНЫЙ** (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
```

Для переадресации пользователя, пытающегося отредактировать чужой профиль, добавим второй метод, `correct_user`, вместе с предварительным фильтром для его вызова (листинг 9.22). Предварительный фильтр `correct_user` определяет переменную `@user`, поэтому можно убрать присваивание переменной `@user` в методах `edit` и `update`.

Листинг 9.22 ❖ Предварительный фильтр correct_user для защиты страниц edit/update **ЗЕЛЕНЫЙ** (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
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password, :password_confirmation)
    end

    # Предварительные фильтры

    # Подтверждает вход пользователя.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Подтверждает права пользователя.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless @user == current_user
    end
end
```

Теперь набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 9.23 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

В качестве заключительного шага последуем за распространенным соглашением и определим во вспомогательном модуле Sessions логический метод current_user? для использования в предварительном фильтре correct_user (листинг 9.24). Этот метод будет использоваться для замены программного кода, такого как

```
unless @user == current_user
```

на (немного) более выразительный:

```
unless current_user?(@user)
```

Листинг 9.24 ❖ Метод current_user? (app/helpers/sessions_helper.rb)

```
module SessionsHelper

  # Осуществляет вход данного пользователя.
  def log_in(user)
    session[:user_id] = user.id
  end

  # Запоминает пользователя в постоянном сеансе.
  def remember(user)
    user.remember
    cookies.permanent.signed[:user_id] = user.id
    cookies.permanent[:remember_token] = user.remember_token
  end

  # Возвращает true, если данный пользователь является текущим.
  def current_user?(user)
    user == current_user
  end
  .
  .
  .
end
```

Заменив прямое сравнение логическим методом, мы получим код, как показано в листинге 9.25.

Листинг 9.25 ❖ Окончательный предварительный фильтр correct_user **ЗЕЛЕНЫЙ** (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
  .
  .
  .
  private

    def user_params
      params.require(:user).permit(:name, :email, :password,
                                   :password_confirmation)
    end

    # Предварительные фильтры

    # Подтверждает вход пользователя.
    def logged_in_user
      unless logged_in?
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Подтверждает права пользователя.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

9.2.3. Дружелюбная переадресация

Реализация системы авторизации на нашем сайте завершена, но есть один небольшой недостаток: когда пользователь пытается получить доступ к защищенной странице, он переадресуется на страницу профиля независимо от того, куда пытался попасть. Другими словами, если пользователь, не выполнивший входа, попытается посетить страницу редактирования, после входа он будет переадресован на страницу `/users/1` вместо `/users/1/edit`. Было бы намного дружелюбнее все же переадресовывать его в нужный пункт назначения.

Код приложения в этом случае получится достаточно сложным, но мы можем написать невероятно простые тесты для дружелюбной переадресации, просто изменив порядок входа и посещения страницы редактирования в листинге 9.14. Как показано в листинге 9.26, тест пытается посетить страницу редактирования, затем осуществляет вход, убеждается, что пользователь переадресован на страницу *редактирования* вместо страницы профиля, заданной по умолчанию. (Также здесь удалена проверка отображения шаблона редактирования, так как это уже не ожидаемое поведение.)

Листинг 9.26 ❖ Тест дружелюбной переадресации **КРАСНЫЙ** (test/integration/users_edit_test.rb)

```
require 'test_helper'

class UsersEditTest < 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:              "foobar",
                                    password_confirmation: "foobar" }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal @user.name,  name
    assert_equal @user.email, email
  end
end
```

Написав тест, терпящий неудачу, можно приступать к реализации дружелюбной переадресации[1]. Чтобы отправить пользователя в нужный ему пункт назначения, нужно где-то сохранить адрес запрашиваемой страницы, а затем переадресовать пользователя туда вместо страницы по умолчанию. Добиться этого можно с помощью пары методов, `store_location` и `redirect_back_or`, которые определим во вспомогательном модуле Sessions (листинг 9.27).

Листинг 9.27 ❖ Реализация дружелюбной переадресации (app/helpers/sessions_helper.rb)

```
module SessionsHelper
  .
  .
  .
  # Перенаправить по сохраненному адресу или на страницу по умолчанию.
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
```

[1] Код в данном разделе – это адаптированный гем Clearance от thoughtbot.

```
  end

  # Запоминает URL.
  def store_location
    session[:forwarding_url] = request.url if request.get?
  end
end
```

В роли механизма хранения URL для переадресации выступает тот же самый объект `session`, который использовался в разделе 8.2.1 для осуществления входа пользователя. Здесь также используется объект `request` (вызов `request.url`) для получения URL запрашиваемой страницы.

Метод `store_location` сохраняет запрашиваемый URL в переменной `session` с ключом `:forwarding_url`, но только в случае запроса `GET`. Это предотвращает сохранение URL, если пользователь, скажем, отправляет форму, не зарегистрировавшись на сайте (что, пусть и является крайним случаем, но все же может случиться, если, например, пользователь удалил cookies сеанса вручную перед отправкой формы). В таком случае при переадресации будет отправлен запрос `GET` к URL, ожидающему `POST`, `PATCH` или `DELETE`, что приведет к ошибке. Включение `if request.get?` предотвращает это[1].

Чтобы использовать `store_location`, необходимо добавить его в предварительный фильтр `logged_in_user`, как показано в листинге 9.28.

Листинг 9.28 ❖ Добавление store_location в предварительный фильтр logged-in (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

    # Предварительные фильтры

    # Подтверждает вход пользователя.
```

[1] Спасибо читателю Йолу Адлеру (Yoel Adler) за подмеченную проблему и предоставленное решение.

```
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end

    # Подтверждает права пользователя.
    def correct_user
      @user = User.find(params[:id])
      redirect_to(root_url) unless current_user?(@user)
    end
end
```

Собственно переадресацию выполняет метод `redirect_back_or`, перенаправляющий пользователя по запрошенному адресу URL, если он существует, или URL по умолчанию в противном случае. Добавим вызов этого метода в метод `create` контроллера Sessions для переадресации после успешного входа (листинг 9.29). В нем используется оператор ИЛИ `||`:

```
session[:forwarding_url] || default
```

Это выражение возвращает `session[:forwarding_url]`, если это значение не равно `nil`, в противном случае возвращается заданный по умолчанию URL `default`. Листинг 9.27 удаляет URL для переадресации (вызовом `session.delete(:forwarding_url)`), иначе последующие попытки входа приводили бы к переадресации на защищенную страницу, пока пользователь не закроет браузер. (Тестирование этого поведения оставлено в качестве упражнения (раздел 9.6).) Кроме того, удаление сеанса происходит даже несмотря на то, что строка, реализующая переадресацию, находится выше; переадресация не произойдет до явного вызова инструкции `return` или окончания метода, поэтому весь код, написанный ниже вызова переадресации, будет выполнен.

Листинг 9.29 ❖ Метод create контроллера Sessions с дружелюбной переадресацией (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
```

Теперь интеграционный тест дружелюбной переадресации из листинга 9.26 должен выполняться успешно, и базовую реализацию аутентификации пользователя и защиты страниц можно считать законченной. Как обычно, прежде чем продолжить, хорошо бы убедиться, что набор тестов **ЗЕЛЕНЫЙ**:

Листинг 9.30 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

9.3. Вывод списка всех пользователей

В этом разделе мы добавим предпоследний метод, `index` (список), предназначенный для отображения *всех* пользователей вместо одного-единственного. Попутно мы узнаем, как заполнить базу данных образцами пользователей и организовать *постраничный вывод* списка, чтобы получить возможность отображения большого количества пользователей. Макет страницы со списком пользователей, ссылками постраничного просмотра и навигационной ссылкой «Users» показан на рис. 9.8[1]. В разделе 9.4 мы добавим в список пользователей административный интерфейс, чтобы дать возможность удалять учетные записи.

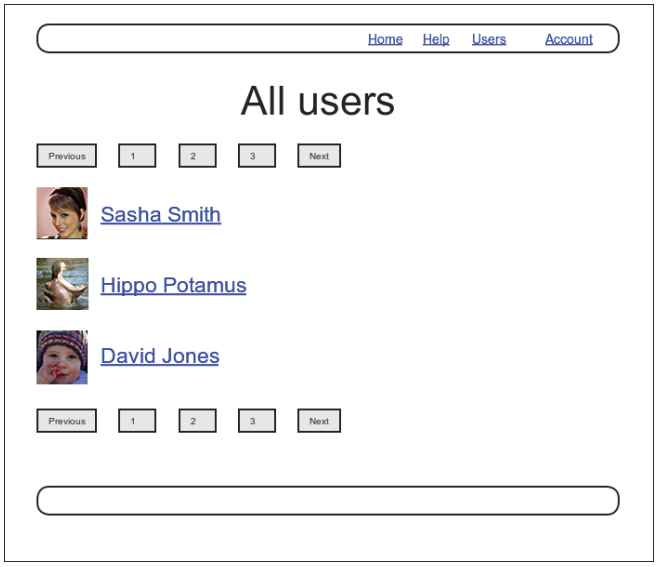


Рис. 9.8 ❖ Макет страницы со списком пользователей

[1] Фото ребенка взято по адресу: http://www.flickr.com/photos/glasgows/338937124/.

9.3.1. Список пользователей

Для начала реализуем модель безопасности. Страницы отдельных пользователей мы сохраним доступными для всех посетителей сайта, но страницу со списком мы сделаем видимой только для вошедших пользователей, чтобы ограничить возможности незарегистрированных пользователей[1].

Для защиты страницы со списком от несанкционированного доступа сначала добавим короткий тест, проверяющий переадресацию пользователя в методе `index` (листинг 9.31).

Листинг 9.31 ❖ Тестирование переадресации в методе index **КРАСНЫЙ** (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
```

Теперь нужно просто добавить метод `index` и внести его в список защищенных предварительным фильтром `logged_in_user` (листинг 9.32).

Листинг 9.32 ❖ Требование входа пользователя для метода index **ЗЕЛЕНЫЙ** (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
```

[1] Аналогичная модель авторизации используется в Twitter.

Для отображения нам потребуется создать переменную со списком всех пользователей сайта, а затем отобразить каждого из них в представлении index путем перебора в цикле. Если вспомнить соответствующий метод в мини-приложении (листинг 2.5), для извлечения всех пользователей из базы данных можно применить метод `User.all` и присвоить его результат переменной экземпляра `@users` в представлении (листинг 9.33). (Если отображение всех пользователей за раз покажется вам плохой идеей, вы правы, но мы избавимся от этого недостатка в разделе 9.3.3.)

Листинг 9.33 ❖ Метод index (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController
  before_action :logged_in_user, only: [:index, :edit, :update]
  .
  .
  .
  def index
    @users = User.all
  end
  .
  .
  .
end
```

Чтобы получить страницу со списком, создадим представление (его файл нужно создать вручную), которое перебирает всех пользователей и заключает каждого из них в тег `li`. В этом нам поможет метод `each`, который будет отображать аватар и имя каждого пользователя. Весь список при этом упакован в тег `ul` (листинг 9.34).

Листинг 9.34 ❖ Представление для списка пользователей (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>
```

Здесь использовано решение упражнения, представленное в листинге 7.31 (раздел 7.7), которое позволяет передавать вспомогательному методу параметр с желаемым размером, если он отличается от размера по умолчанию. Если вы не решили это упражнение, добавьте в файл со вспомогательным модулем Users содержимое из листинга 7.31, прежде чем двигаться дальше.

Давайте также добавим несколько правил CSS (или, скорее, SCSS) для оформления (листинг 9.35).

Листинг 9.35 ❖ Правила CSS для оформления списка пользователей (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* Список пользователей */
.users {
  list-style: none;
  margin: 0;
  li {
    overflow: auto;
    padding: 10px 0;
    border-bottom: 1px solid $gray-lighter;
  }
}
```

Наконец, добавим ссылку на список пользователей в меню навигации с помощью `users_path`, задействовав тем самым последний неиспользованный именованный маршрут из табл. 7.1. Результат показан в листинге 9.36.

Листинг 9.36 ❖ Добавление ссылки на список пользователей (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>
```

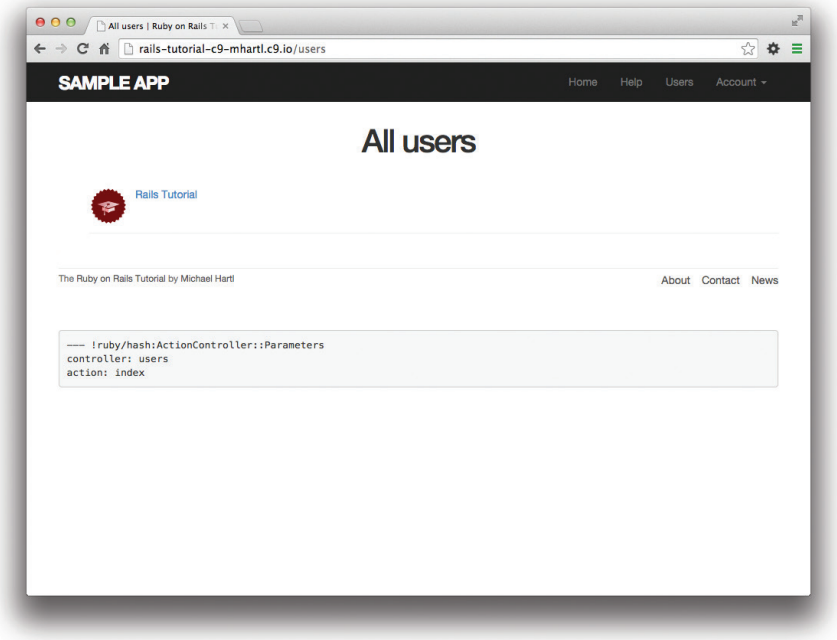


Рис. 9.9 ❖ Список пользователей с единственной записью

Теперь список пользователей стал полностью функциональным, и весь набор тестов должен выполняться успешно:

Листинг 9.37 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Но, с другой стороны, как показано на рис. 9.9, он выглядит... несколько пустовато. Давайте исправим эту печальную ситуацию.

9.3.2. Образцы пользователей

В этом разделе наш одинокий пользователь обзаведется небольшой компанией. Конечно, чтобы создать достаточное количество пользователей для приличного списка, можно было бы использовать веб-браузер, чтобы посетить страницу регистрации и создать новых пользователей по одному, но Ruby (и Rake) дают более простую альтернативу.

Сначала добавим гем *Faker* в `Gemfile`, он позволит создавать образцы пользователей с почти реалистичными именами и адресами электронной почты (листинг 9.38).

Листинг 9.38 ❖ Добавление гема Faker в Gemfile

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

gem 'rails',                '4.2.0'
gem 'bcrypt',               '3.1.7'
gem 'faker',                '1.4.2'
.
.
.
```

Затем установим его, как обычно:

```
$ bundle install
```

Далее добавим Rake-задачу для заполнения базы данных образцами пользователей, поместив ее в файл `db/seeds.rb`. Результат представлен в листинге 9.39. (Это довольно сложный код, поэтому не стоит особо беспокоиться о деталях.)

Листинг 9.39 ❖ Rake-задача для заполнения базы данных образцами пользователей (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
```

Этот код создает образец пользователя с именем и адресом электронной почты, который дублирует существующего пользователя, а затем создает еще 99 образцов. Метод `create!` действует так же, как `create`, только вызывает исключение (раздел 6.1.4) для недопустимого пользователя вместо простого возврата значения `false`. Такое поведение облегчает отладку, помогая обнаруживать малозаметные ошибки.

Теперь можно очистить базу данных, а затем вызвать Rake-задачу `db:seed`[1]:

[1] В принципе, обе эти задачи можно было бы объединить в `rake db:reset`, но на момент написания этого текста данная команда не работала с последней версией Rails.

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

Заполнение базы может протекать медленно и в некоторых системах занимает несколько минут. Кроме того, некоторые читатели сообщали, что не могут запустить команду `reset`, когда работает сервер Rails, поэтому вам, быть может, придется сначала остановить его.

После запуска задачи `db:seed` в приложении появилось 100 образцов пользователей. Как видно на рис. 9.10, я взял на себя смелость связать первые несколько образцов электронных адресов с граватарами, поэтому не все изображения совпадают с картинками по умолчанию. (Вам может понадобиться перезапустить вебсервер в этой точке.)

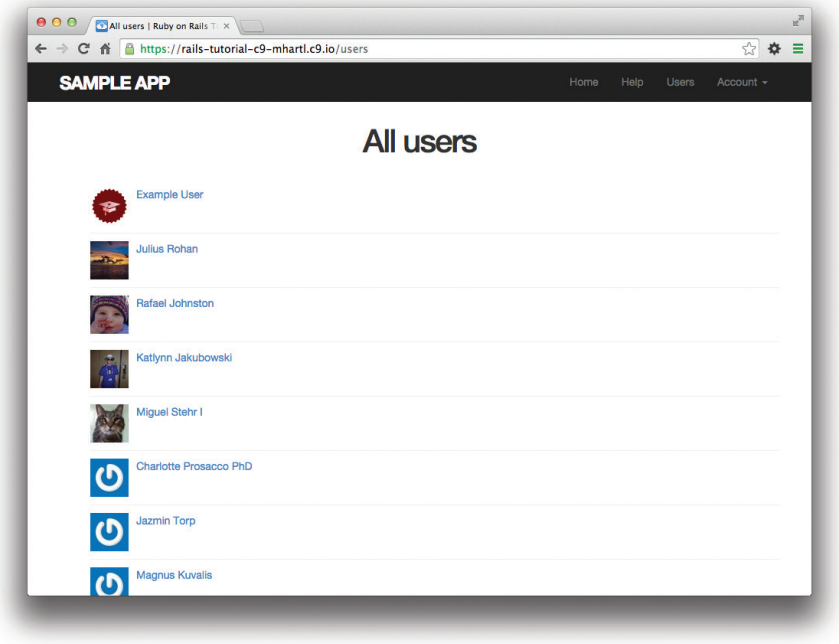


Рис. 9.10 ❖ Страница с сотней пользователей

9.3.3. Постраничный просмотр

Наш исходный пользователь более не страдает от одиночества, но теперь появилась другая проблема: у него *слишком большая компания*, и вся она расположилась на одной странице. Сейчас это сотня, что уже довольно много, а на реальном сайте это могут быть и тысячи. Решение заключается в *разбиении списка пользователей на страницы*, чтобы показывать единовременно только (например) 30 записей на странице.

В Rails поддерживается несколько способов разбиения на страницы, и мы будем использовать один из самых простых и надежных, `will_paginate` (https://github.

com/mislav/will_paginate/wiki). Чтобы воспользоваться им, нам потребуются два гема: `will_paginate` и `bootstrap-will_paginate`, который настраивает `will_paginate` на использование стилей из Bootstrap. Дополненный `Gemfile` показан в листинге 9.40.

Листинг 9.40 ❖ Добавление will_paginate в Gemfile

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

gem 'rails',                  '4.2.0'
gem 'bcrypt',                 '3.1.7'
gem 'faker',                  '1.4.2'
gem 'will_paginate',          '3.0.7'
gem 'bootstrap-will_paginate', '0.0.10'
.
.
.
```

Выполним `bundle install`:

```
$ bundle install
```

Также следует перезапустить веб-сервер, чтобы корректно загрузить новые гемы.

Чтобы разбиение на страницы заработало, нужно добавить немного кода в представление `index`, а также заменить вызов `User.all` в методе `index` на объект, реализующий разбиение на страницы. Для начала добавим в представление специальный метод `will_paginate` (листинг 9.41); мы вскоре увидим, почему код был добавлен выше и ниже списка пользователей.

Листинг 9.41 ❖ Список пользователей с разбиением на страницы (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 %>
```

Метод `will_paginate` немного необычный; внутри представления `users` он автоматически находит объект `@users`, а затем отображает ссылки для доступа к страницам. И все же представление пока не работает, так как в данный момент `@users` содержит результаты вызова `User.all` (листинг 9.33), в то время как `will_paginate` требует постраничного вывода результатов с явным применением метода `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 [#<User id: 1,...
```

Обратите внимание, что `paginate` принимает в качестве аргумента хэш с ключом `:page` и номером запрашиваемой страницы. `User.paginate` извлекает пользователей из базы данных блоками (30 по умолчанию), основываясь на параметре `:page`. Так, например, страница 1 содержит пользователей с 1 по 30, страница 2 – с 31 по 60 и т. д. Если значение ключа `:page` равно `nil`, `paginate` просто вернет первую страницу.

Используя `paginate` вместо `all` в методе `index` (листинг 9.42), мы можем организовать постраничный вывод пользователей в учебном приложении. Параметр `page` берется из `params[:page]`, который автоматически генерируется `will_paginate`.

Листинг 9.42 ❖ Постраничный вывод списка пользователей в методе 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
```

Страница со списком пользователей теперь должна выглядеть, как показано на рис. 9.11. (На некоторых системах может потребоваться перезапустить сервер Rails.) Так как мы включили `will_paginate` выше и ниже списка пользователей, ссылки на страницы появились в обоих местах.

Если сейчас щелкнуть на ссылке **2** или **Next** (Следующая), будет выполнен переход на вторую страницу списка, как показано на рис. 9.12.

9.3.4. Тестирование страницы со списком пользователей

Теперь, когда страница со списком пользователей заработала, напишем упрощенные тесты для нее, в том числе тесты для проверки разбиения на страницы. Суть в том, чтобы войти на сайт, открыть страницу со списком, проверить наличие первой страницы и ссылок на другие страницы. Для этого нам необходимо иметь в тестовой базе данных достаточное количество пользователей, то есть их должно быть больше 30.

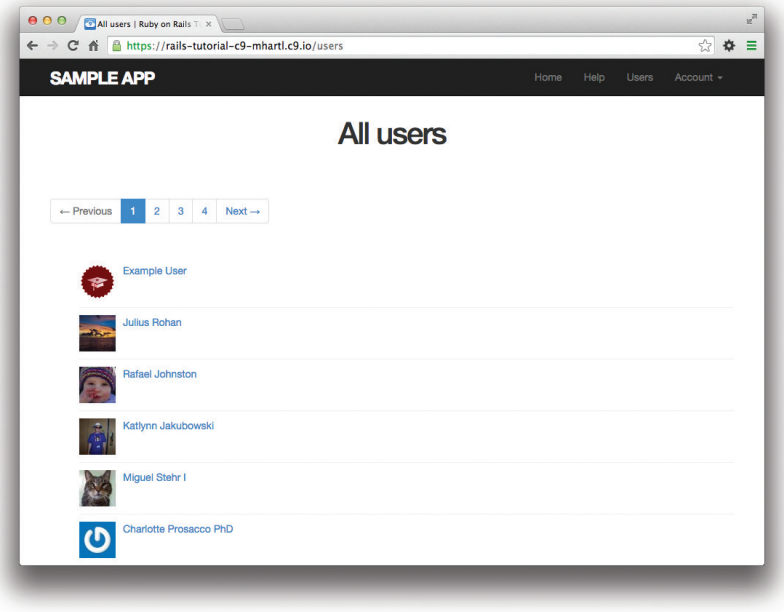


Рис. 9.11 ❖ Страница со списком пользователей и поддержкой постраничного просмотра

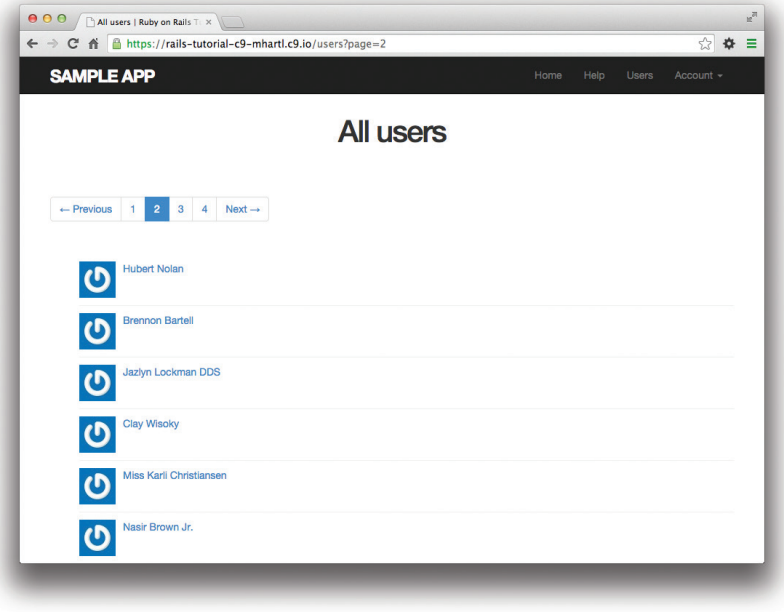


Рис. 9.12 ❖ Страница 2 списка пользователей

В листинге 9.20 мы создали второго пользователя, но создавать вручную 30 и более пользователей довольно тяжело. К счастью, как мы уже видели, работая с атрибутом пользователя password_digest, в файле с тестовыми данными можно использовать программный код на встроенном Ruby, а значит, 30 дополнительных пользователей мы можем создать так, как показано в листинге 9.43. (Здесь также создается еще пара именованных пользователей для будущих нужд.)

Листинг 9.43 ❖ Добавление 30 дополнительных пользователей в тестовые данные (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 %>
```

Теперь мы готовы написать тест для списка пользователей. Сначала сгенерируем заготовку для теста:

```
$ rails generate integration_test users_index
      invoke  test_unit
      create    test/integration/users_index_test.rb
```

Тест будет проверять наличие тега div с классом pagination, а также наличие первой страницы списка пользователей. Результат представлен в листинге 9.44.

Листинг 9.44 ❖ Тест для списка пользователей и постраничного просмотра **ЗЕЛЕНЫЙ** (test/integration/users_index_test.rb)

```
require 'test_helper'

class UsersIndexTest < 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
```

Набор тестов должен быть **ЗЕЛЕНЫМ**.

Листинг 9.45 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

9.3.5. Частичный рефакторинг

Мы закончили разбивку списка пользователей на страницы, но есть одно улучшение, от которого я не могу удержаться: в Rails имеется несколько невероятно интересных инструментов создания компактных представлений, и в этом разделе мы займемся рефакторингом страницы со списком пользователей с их применением. Так как наш код хорошо протестирован, мы можем с уверенностью приступить к рефакторингу, не опасаясь, что можем что-то нарушить.

Сначала заменим тег `li` из листинга 9.41 вызовом `render` (листинг 9.46).

Листинг 9.46 ❖ Первая попытка рефакторинга представления со списком (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 %>
```

Здесь методу `render` передается не строка с именем частичного шаблона, а переменная `user` класса `User`[1]; Rails автоматически найдет шаблон `_user.html.erb`, который мы должны создать (листинг 9.47).

---

[1] Имя `user` не принципиально – мы могли бы написать `@users.each do |foobar|` и затем `render foobar`. Ключом является класс объекта – в данном случае это `User`.

Листинг 9.47 ❖ Шаблон для отображения отдельно взятого пользователя (app/views/users/_user.html.erb)

```
<li>
  <%= gravatar_for user, size: 50 %>
  <%= link_to user.name, user %>
</li>
```

Это явное улучшение, но можно сделать еще лучше: передать методу `render` переменную `@users` непосредственно (листинг 9.48).

Листинг 9.48 ❖ Полностью реорганизованный список пользователей **ЗЕЛЕНЫЙ** (app/views/users/index.html.erb)

```
<% provide(:title, 'All users') %>
<h1>All users</h1>

<%= will_paginate %>

<ul class="users">
  <%= render @users %>
</ul>

<%= will_paginate %>
```

Rails обнаруживает, что `@users` – это список объектов `User`; более того, при вызове с коллекцией пользователей Rails автоматически перебирает ее и отображает каждый элемент коллекции с помощью шаблона `_user.html.erb`. В результате получается впечатляюще компактный код, как показано в листинге 9.48.

Как и при любом рефакторинге, обязательно следует убедиться, что набор тестов по-прежнему **ЗЕЛЕНЫЙ**, несмотря на изменившийся код приложения:

Листинг 9.49 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

9.4. Удаление пользователей

Теперь, закончив работу над списком пользователей, нам осталось реализовать последнюю каноническую операцию REST: `destroy`. В этом разделе мы добавим ссылки для удаления пользователей, как показано на рис. 9.13, и определим метод `destroy`, осуществляющий удаление. Но сначала создадим класс уполномоченных на это пользователей-администраторов.

9.4.1. Администраторы

Мы будем определять привилегированных пользователей с правами администратора с помощью логического атрибута `admin` в модели User, что автоматически приводит нас к необходимости создания логического метода `admin?`, проверяющего наличие административного статуса. Полученная модель данных представлена на рис. 9.14.

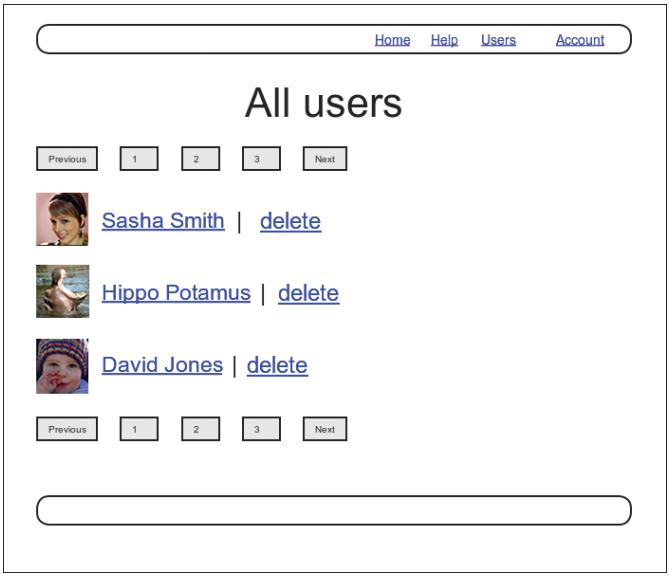


Рис. 9.13 ❖ Макет списка пользователей со ссылками для удаления

users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime
password_digest	string
remember_digest	string
admin	boolean

Рис. 9.14 ❖ Модель User с новым логическим атрибутом admin

Добавим атрибут `admin`, как обычно, выполнив миграцию и указав тип `boolean` в командной строке:

```
$ rails generate migration add_admin_to_users admin:boolean
```

Миграция добавит столбец `admin` в таблицу `users`, как показано в листинге 9.50. Обратите внимание на аргумент `default: false` в `add_column`, это значит, что по умолчанию пользователи не будут иметь административных привилегий. (Без аргумента `default: false` атрибут `admin` по умолчанию получит значение `nil`, что также является `false`, так что этот шаг не является строго обязательным. Однако это более ясно и четко сообщает о наших намерениях и фреймворку Rails, и тем, кто будет читать наш код.)

Листинг 9.50 ❖ Миграция для добавления логического атрибута admin в модель User (db/migrate/[timestamp]_add_admin_to_users.rb)

```
class AddAdminToUsers < ActiveRecord::Migration
  def change
    add_column :users, :admin, :boolean, default: false
  end
end
```

Запускаем миграцию, как обычно:

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

Как и ожидалось, Rails догадался о логическом характере атрибута `admin` и автоматически добавил метод со знаком вопроса `admin?`:

```
$ rails console --sandbox
>> user = User.first
>> user.admin?
=> false
>> user.toggle!(:admin)
=> true
>> user.admin?
=> true
```

Мы использовали метод `toggle!` для переключения атрибута `admin` между значениями `false` и `true`.

В качестве последнего шага обновим код, заполняющий базу тестовыми данными, чтобы сделать первого пользователя администратором по умолчанию (листинг 9.51).

Листинг 9.51 ❖ Rake-задача для заполнения базы данных образцами пользователей, из которых первый – администратор (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
```

Затем перезапустим базу данных:

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

Вновь о строгих параметрах

Вы могли заметить, что в листинге 9.51 административный пользователь создается за счет включения `admin: true` в хэш с параметрами. Это подчеркивает опасность незащищенности объектов в Интернете: если просто передавать хэш с параметрами в произвольном веб-запросе, злоумышленник сможет отправить вот такой запрос `PATCH`[1]:

```
patch /users/17?admin=1
```

Этот запрос сделает администратором пользователя с идентификатором 17, а это является серьезной потенциальной угрозой.

Из-за этой угрозы важно обновлять только те атрибуты, которые безопасны для редактирования через Интернет. Как рассказывалось в разделе 7.3.2, сделать это можно при помощи *строгих параметров*, посредством вызова методов `require` и `permit` хэша `params`:

```
def user_params
  params.require(:user).permit(:name, :email, :password,
                               :password_confirmation)
end
```

В частности, обратите внимание, что атрибут `admin` не включен в список разрешенных атрибутов. Именно это не позволит произвольному пользователю получить административный доступ к нашему приложению. Поскольку это очень важный нюанс, желательно написать тесты для всех нередактируемых атрибутов, и такой тест для атрибута `admin` оставлен в качестве упражнения (раздел 9.6).

9.4.2. Метод destroy

Последний шаг, необходимый для завершения ресурса Users, заключается в добавлении ссылок для удаления и метода `destroy`. Начнем со ссылки для каждого пользователя на странице со списком, разрешив доступ к ним только администраторам. То есть ссылки `"delete"` будут отображаться, только если текущий пользователь является администратором (листинг 9.52).

Листинг 9.52 ❖ Ссылки для удаления пользователей (доступны только администраторам) (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>
```

[1] Запросы `PATCH` подобного типа можно отправлять с помощью инструментов командной строки, таких как `curl`.

Аргумент `method: :delete` обеспечивает отправку запроса `DELETE` в случае щелчка на ссылке. Кроме того, каждая ссылка упакована в инструкцию `if`, поэтому только администраторы смогут их видеть. Результат, видимый администраторам, представлен на рис. 9.15.

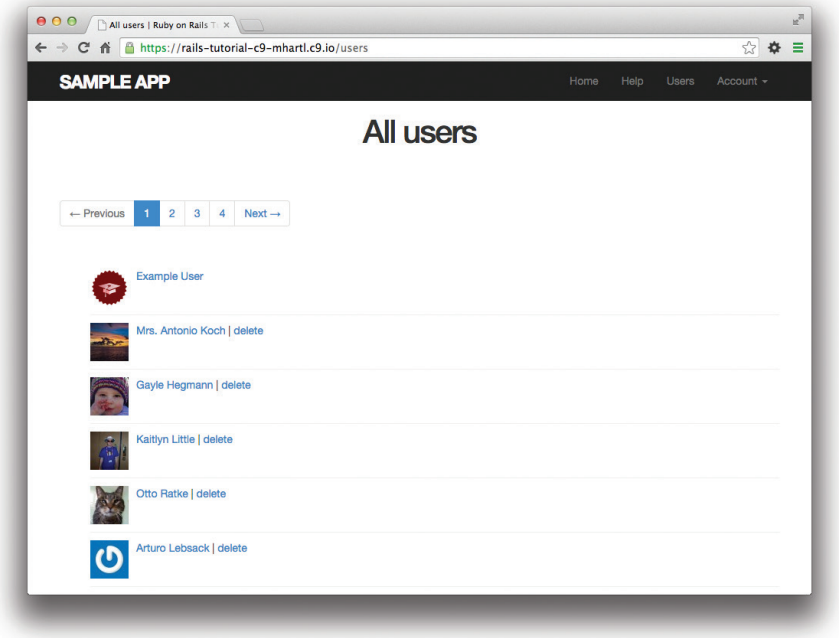


Рис. 9.15 ❖ Список пользователей со ссылками для удаления

Браузеры не могут отправлять запросы `DELETE`, поэтому Rails подделывает их с помощью JavaScript. То есть ссылки, реализующие удаление, не будут работать, если у пользователя отключен JavaScript. Если необходимо обеспечить поддержку браузеров с отключенным JavaScript, можете подделать запрос `DELETE` через запрос `POST`, этот подход будет работать даже без JavaScript[1].

Чтобы ссылки заработали, нужно добавить метод `destroy` (табл. 7.1), который будет находить пользователя и удалять его вызовом метода `destroy` механизма Active Record, а затем переадресовывать пользователя на страницу со списком, как показано в листинге 9.53. Так как для удаления пользователь сначала должен выполнить вход, мы также добавим `:destroy` в предварительный фильтр `logged_in_user`.

[1] Более подробно об этом можно узнать в статье «Destroy Without JavaScript» (http://railscasts.com/episodes/77-destroy-without-javascript).

Листинг 9.53 ❖ Добавление метода destroy (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
```

Метод `destroy` использует прием составления цепочек из вызовов методов, чтобы объединить вызовы `find` и `destroy` в одну строку:

```
User.find(params[:id]).destroy
```

Как и было задумано, удалить пользователя из веб-браузера сможет *только* администратор, так как только ему видны ссылки для удаления, но у нас все еще остается ужасная дыра в безопасности: любой опытный злоумышленник сможет просто послать запрос `DELETE` из командной строки и удалить любого пользователя на сайте. Чтобы обезопасить сайт, необходимо ограничить доступ к методу `destroy`, чтобы только администраторы могли удалять пользователей.

Так же как в разделах 9.2.1 и 9.2.2, реализуем такое ограничение (в данном случае к методу `destroy`) при помощи предварительного фильтра. Получившийся фильтр `admin_user` представлен в листинге 9.54.

Листинг 9.54 ❖ Предварительный фильтр, ограничивающий доступ к методу destroy (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
    .
    .
    .
    # Подтверждает наличие административных привилегий.
    def admin_user
```

```
      redirect_to(root_url) unless current_user.admin?
    end
end
```

9.4.3. Тесты для проверки удаления пользователя

Когда имеешь дело с чем-то, настолько опасным, как удаление пользователей, очень важно хорошо протестировать ожидаемое поведение. Для начала сделаем администратором одного из наших пользователей в тестовых данных (листинг 9.55).

Листинг 9.55 ❖ Превращение одного из пользователей в администратора (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 %>
```

Следуя опыту, полученному в разделе 9.2.1, поместим тесты, работающие на уровне методов, в файл тестов контроллера Users. Как и в случае с тестами, проверяющими выход (листинг 8.28), будем использовать тестовый метод `delete` для отправки запроса `DELETE` напрямую в метод `destroy`. Нам нужно проверить два случая: невошедший пользователь должен быть переадресован на страницу входа, а вошедший пользователь-неадминистратор должен быть переадресован на главную страницу. Результат представлен в листинге 9.56.

Листинг 9.56 ❖ Тесты на уровне действия для контроля доступа **ЗЕЛЕНЫЙ** (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
```

Кроме того, при помощи метода `assert_no_difference` (мы уже видели его в листинге 7.21) тест проверяет, что количество пользователей не изменилось.

Тесты в листинге 9.56 проверяют поведение приложения, когда действует неуполномоченный пользователь (не администратор), но нам также нужно убедиться, что администратор может использовать ссылку для успешного удаления пользователя. Так как ссылки для удаления находятся на странице списка пользователей, внесем дополнения в тесты из листинга 9.44. Самым сложным будет проверить факт удаления пользователя после щелчка на ссылке, и мы сделаем это вот так:

```
assert_difference 'User.count', -1 do
  delete user_path(@other_user)
end
```

Здесь использован метод `assert_difference`, который впервые мы увидели в листинге 7.26, когда создавали пользователя, на этот раз с его помощью мы проверяем удаление пользователя – `User.count` должен измениться на –1.

Если собрать все вместе, мы получим тесты проверки разбиения на страницы и удаления (листинг 9.57), в которых есть проверки как для администраторов, так и для простых пользователей.

Листинг 9.57 ❖ Интеграционные тесты для ссылок для удаления и собственно удаления пользователей **ЗЕЛЕНЫЙ** (test/integration/users_index_test.rb)

```
require 'test_helper'

class UsersIndexTest < 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',
                                                  method: :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
```

Этот код проверяет наличие ссылок для удаления, пропуская проверку ссылок для удаления администраторов (у которых нет такой ссылки благодаря особенностям реализации в листинге 9.52).

Теперь механизм удаления протестирован, и набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 9.58 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

9.5. Заключение

Мы прошли долгий путь с момента создания контроллера Users давным-давно, в разделе 5.4. Те пользователи даже не могли зарегистрироваться; теперь же у них

есть такая возможность, а кроме того, они могут входить в систему, выходить из нее, просматривать свои профили, редактировать свои данные и видеть список всех пользователей, а некоторые могут даже удалять других пользователей.

Учебное приложение в своем нынешнем виде представляет надежную основу для любого веб-сайта, где требуется система аутентификации и авторизации пользователей. В главе 10 мы добавим еще пару усовершенствований: ссылку для активации учетной записи вновь зарегистрированного пользователя (чтобы заодно проверить действительность адреса электронной почты) и сброс пароля для забывчивых пользователей.

Прежде чем двигаться дальше, объедините все изменения с основной веткой:

```
$ git add -A
$ git commit -m "Finish user edit, update, index, and destroy actions"
$ git checkout master
$ git merge updating-users
$ git push
```

Вы также можете развернуть приложение и даже заполнить базу данных образцами пользователей на действующем сайте (использовав для ее очистки задачу `pg:reset`).

Конечно, на действующем сайте вы вряд ли захотите заполнять его тестовыми данными, но здесь эта процедура демонстрируется исключительно для иллюстрации (рис. 9.16). Порядок пользователей может отличаться – в моей системе он не совпадает с локальной версией на рис. 9.11; это связано с тем, что мы не указали порядок пользователей по умолчанию при извлечении из базы данных, поэтому сейчас порядок зависит только от нее. Это не особенно важно для пользователей, но будет иметь значение для микросообщений, и позднее, в разделе 11.1.4, мы решим эту проблему.

9.5.1. Что мы изучили в этой главе

- Информация о пользователях может изменяться при помощи формы редактирования, которая отправляет запрос `PATCH` в метод `update`.
- Безопасное обновление через Интернет обеспечивается строгими параметрами.
- Предварительные фильтры предоставляют стандартный способ выполнения некоторых методов перед вызовом определенного метода контроллера.
- Мы реализовали авторизацию с помощью предварительных фильтров.
- Тесты авторизации содержат как низкоуровневые команды, отправляющие HTTP-запросы напрямую в методы контроллера, так и высокоуровневые интеграционные тесты.
- Дружелюбная переадресация перенаправляет пользователей после входа на ту страницу, куда они хотели попасть.
- Страница со списком пользователей поддерживает постраничное отображение.
- Для заполнения базы данных образцами через `rake db:seed` в Rails используется стандартный файл `db/seeds.rb`.

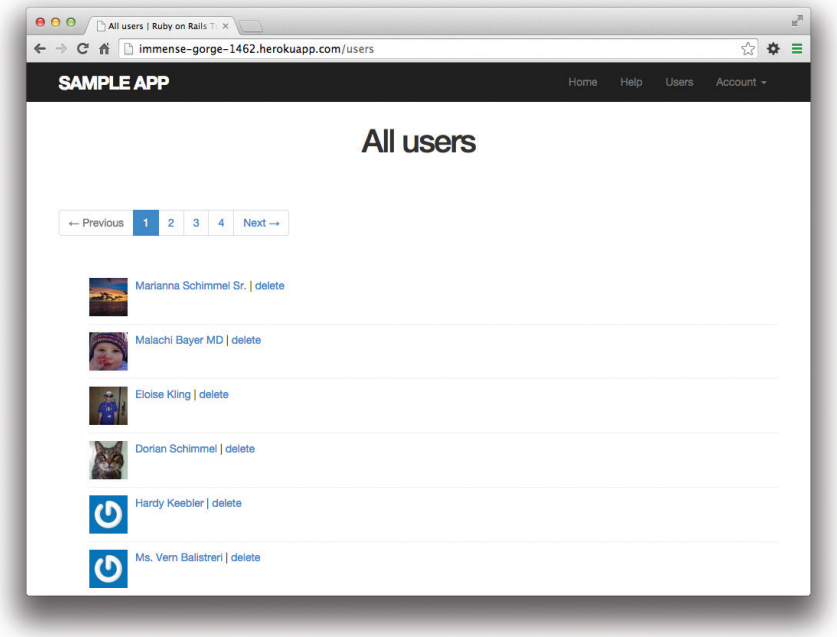


Рис. 9.16 ❖ Образцы пользователей на действующем сервере

- Вызов `render @users` автоматически вызывает частичный шаблон `_user.html.erb` для каждого пользователя в коллекции.
- Добавление логического атрибута `admin` в модель пользователей автоматически дает логический метод `user.admin?`.
- Администраторы могут удалять пользователей через Интернет, щелкнув на ссылке, в результате чего посылается запрос `DELETE` в метод `destroy` контроллера Users.
- Мы можем создать множество образцов пользователей благодаря поддержке встроенного Ruby в тестовых данных.

9.6. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Напишите тест, чтобы убедиться, что дружелюбная переадресация выполняет перенаправление на заданный адрес только в первый раз. При последующих попытках входа URL переадресации должен вернуться к значению

по умолчанию (то есть переадресация должна выполняться на страницу профиля). *Подсказка:* дополните тесты в листинге 9.26 проверкой значения `session[:forwarding_url]`.

2. Напишите интеграционные тесты для всех ссылок в шаблоне, в том числе для проверки поведения в случае вошедшего и невошедшего пользователя. *Подсказка:* дополните тесты в листинге 5.25, применив вспомогательный метод `log_in_as`.
3. Посылая запрос `PATCH` напрямую в метод `update`, как показано в листинге 9.59, убедитесь, что атрибут `admin` невозможно изменить из веб-браузера. Чтобы быть уверенными, что вы тестируете именно то, что нужно, сначала *добавьте* `admin` в список разрешенных параметров в `user_params`, чтобы первоначально тест был **КРАСНЫМ**.
4. Удалите повторяющийся код в форме, проведя рефакторинг представлений `new.html.erb` и `edit.html.erb`, использовав частичные шаблоны из листинга 9.60, как показано в листингах 9.61 и 9.62. Обратите внимание на использование метода `provide`, который уже применялся в разделе 3.4.3 для устранения повторений в шаблоне[1].

Листинг 9.59 ❖ Тестирование недоступности атрибута admin для редактирования (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?
```

[1] Спасибо читателю Хосе Карлосу Монтеро Гомесу (Jose Carlos Montero Gomez) за предложение по рефакторингу представлений `new` и `edit`.

```
  end
  .
  .
  .
end
```

Листинг 9.60 ❖ Частичный шаблон для форм new и edit (app/views/users/_form.html.erb)

```
<%= 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' %>
```

Листинг 9.61 ❖ Представление регистрации с частичным шаблоном (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 'fields', f: f %>
      <%= f.submit "Create my account", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Глава 10

Активация учетной записи и сброс пароля

В главе 9 мы закончили создание базового ресурса Users (реализовав все стандартные методы REST из табл. 7.1) вместе с гибкой системой аутентификации (проверки подлинности пользователя) и авторизации (проверки прав доступа). В этой главе мы добавим последние штрихи – две тесно связанные функции: активацию учетной записи (проверку адреса электронной почты нового пользователя) и сброс пароля (для забывчивых пользователей). Каждая из них предполагает создание нового ресурса, что дает нам шанс увидеть дополнительные примеры контроллеров, маршрутизации и миграций базы данных. Попутно мы узнаем, как отправлять электронные письма. Наконец, эти функции здорово дополняют друг друга: при сбросе пароля происходит отправка ссылки на электронную почту пользователя, правильность которой подтверждается при первоначальной активации учетной записи[1].

10.1. Активация учетной записи

Сейчас вновь зарегистрированные пользователи незамедлительно получают полный доступ к своему профилю (глава 7). В этом разделе мы реализуем этап активации, чтобы убедиться, что указанный адрес электронной почты действительно принадлежит пользователю. Для этого потребуется связать с пользователем токен активации и дайджест, отправив в письме ссылку с токеном, при переходе по которой и будет происходить активация.

---

[1] Технически пользователь может изменить свои данные и ввести неверный адрес электронной почты с помощью функции редактирования профиля из раздела 9.1, но в этой реализации мы просто воспользуемся всеми выгодами проверки электронной почты, не утруждая себя лишними проверками.

Реализация активации очень похожа на функцию входа пользователя (раздел 8.2) и особенно на функцию запоминания (раздел 8.4). Базовая последовательность выглядит так:

1) первоначально учетная запись пользователя «неактивна»;
2) в процессе регистрации пользователя генерируются токен активации и соответствующий дайджест;
3) затем дайджест сохраняется в базе данных, а пользователю отправляется электронное письмо со ссылкой, содержащей токен активации и адрес электронной почты пользователя[1];
4) когда пользователь перейдет по ссылке, мы найдем его по адресу электронной почты и установим подлинность токена, сравнив его с дайджестом;
5) если пользователь подтвердил регистрацию, мы поменяем статус учетной записи с «неактивная» на «активная».

Благодаря сходству с токеном паролей и токеном в реализации запоминания, мы можем повторно использовать эти идеи для реализации активации учетной записи (и для сброса пароля тоже), в том числе методами `User.digest` и `User.new_token`, и немного измененной версией метода `user.authenticated?`. В табл. 10.1 иллюстрируется упомянутое сходство (в том числе и для сброса пароля из раздела 10.2). Мы определим обобщенную версию метода `authenticated?` из табл. 10.1 в разделе 10.1.3.

Таблица 10.1 ❖ Сходство между реализациями входа, запоминания пользователя, активации учетной записи и сброса пароля

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)

Как обычно, для новой функции создадим в репозитории новую ветку. Как будет показано в разделе 10.3, активация учетной записи и сброс пароля содержат некоторые общие настройки электронной почты, которые будет необходимо применить до слияния с основной ветвью. Поэтому будет удобно создать общую рабочую ветку:

```
$ git checkout master
$ git checkout -b account-activation-password-reset
```

[1] Вместо него можно использовать идентификатор пользователя, так как он уже присутствует в адресах URL приложения, но электронный адрес предпочтительнее на тот случай, если у нас появится желание скрыть пользовательские идентификаторы по какой-то причине (например, чтобы помешать конкурентам узнать количество пользователей приложения).

10.1.1. Ресурс AccountActivations

Так же как в случае с сеансами (раздел 8.1), мы реализуем активацию учетной записи в виде ресурса, который, впрочем, не будет связан с моделью Active Record. Вместо этого соответствующие данные (токен и признак активации) мы добавим в модель User. Однако взаимодействовать с этим ресурсом мы будем через стандартные методы REST; так как ссылка активации будет менять статус пользователя, нам понадобится метод `edit`[1]. Для этого нужен контроллер AccountActivations, который можно создать так[2]:

```
$ rails generate controller AccountActivations --no-test-framework
```

Обратите внимание на флаг, требующий пропустить создание тестов. Это связано с тем, что нам не нужны тесты контроллера (вместо них будут использоваться интеграционные тесты (раздел 10.1.4), поэтому лучше сразу пропустить их.

В письме со ссылкой на активацию учетной записи пользователю будет посылаться адрес URL примерно такого вида:

```
edit_account_activation_url(activation_token, ...)
```

а это значит, что понадобится именованный маршрут для метода `edit`. Мы можем организовать его посредством строки `resources`, как показано в листинге 10.1.

Листинг 10.1 ❖ Добавление ресурса активации учетных записей (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
```

Далее нам нужен уникальный токен для активации учетных записей. Сейчас безопасность беспокоит нас гораздо меньше, чем в случае с паролями, токенами в реализации запоминания или (как мы увидим в разделе 10.2) сбросом паролей – когда злоумышленник имел возможность получить полный контроль над учетной записью, – но все же есть несколько вариантов, когда незащищенный токен акти-

---

[1] Возможно, разумнее было бы использовать метод `update`, но ссылку активации нужно отправить в электронном письме, что предполагает стандартный щелчок на ней в браузере, который вызовет отправку запроса `GET` вместо `PATCH`, требуемого методом `update`.

[2] Так как предполагается использовать метод `edit`, мы могли бы включить имя `edit` в командную строку, но в этом случае будут сгенерированы одноименное представление и тесты, которые нам совершенно не нужны.

вации может скомпрометировать учетную запись[1]. Поэтому, следуя примеру с токеном для запоминания из раздела 8.4, мы составим пару из открытого виртуального токена и его хэшированного дайджеста в базе данных. При этом мы сможем получить доступ к токену активации с использованием

```
user.activation_token
```

и подтвердить подлинность пользователя с помощью

```
user.authenticated?(:activation, token)
```

(Потребуется изменить метод `authenticated?` из листинга 8.33.) Мы также добавим логический атрибут `activated` для проверки статуса учетной записи с помощью такого же автоматически созданного логического метода, как в разделе 9.4.1:

```
if user.activated? ...
```

Наконец, запишем время и дату регистрации на тот случай, если они понадобятся нам в будущем. Полная модель данных показана на рис. 10.1.

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

Рис. 10.1 ❖ Модель User с атрибутами для активации учетной записи

Вот так выглядит миграция, которая добавляет сразу все три атрибута в модель данных:

```
$ rails generate migration add_activation_to_users \
> activation_digest:string activated:boolean activated_at:datetime
```

(Значок > во второй строке означает «продолжение строки», он появляется автоматически, и его вводить не надо.) Так же как в случае с атрибутом `admin` (листинг 9.50), добавим в определение атрибута `activated` значение по умолчанию `false` (листинг 10.2).

---

[1] Например, если злоумышленник имеет доступ к базе данных, он сможет активировать вновь созданную учетную запись и таким способом осуществить вход от лица пользователя, а затем изменить пароль для получения полного контроля.

Листинг 10.2 ❖ Миграция для активации аккаунта (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
```

Затем, как обычно, произведем миграцию:

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

Так как вновь зарегистрировавшийся пользователь должен пройти процедуру активации, соответствующему объекту User необходимо присвоить токен активации и его дайджест. Нечто подобное мы видели в разделе 6.2.5, где преобразовывали адрес электронной почты в нижний регистр перед сохранением в базе данных. Тогда мы применили функцию обратного вызова `before_save` вместе с методом `downcase` (листинг 6.31). Функция `before_save` автоматически вызывается перед сохранением объекта как при создании, так и при обновлении, однако дайджест токена активации должен сохраняться только при создании пользователя. Поэтому здесь следует использовать функцию `before_create`:

```
before_create :create_activation_digest
```

Этот код называется *ссылкой на метод*, заставляет Rails найти метод `create_activation_digest` и вызвать перед созданием пользователя. (В листинге 6.31 мы передавали явный блок в `before_save`, но обычно предпочтительнее использовать ссылки на методы.) Так как сам метод `create_activation_digest` используется только внутри модели User, нет нужды выносить его за ее пределы; как мы видели в разделе 7.3.2, в Ruby для этого есть ключевое слово `private`:

```
private

  def create_activation_digest
    # Создать токен и дайджест.
  end
```

Все методы, следующие за ключевым словом `private`, автоматически скрываются, как можно видеть в консольном сеансе:

```
$ rails console
>> User.first.create_activation_digest
NoMethodError: private method `create_activation_digest' called for #<User>
```

Цель функции `before_create` – присвоить токен и соответствующий дайджест:

```
self.activation_token  = User.new_token
self.activation_digest = User.digest(activation_token)
```

Здесь просто повторно использованы методы создания токена и дайджеста из реализации запоминания пользователя, в чем легко убедиться, сравнив этот код с методом `remember` из листинга 8.32:

```
# Запоминает пользователя в базе данных для использования в постоянных сеансах.
def remember
  self.remember_token = User.new_token
  update_attribute(:remember_digest, User.digest(remember_token))
end
```

Основное отличие во втором случае заключается в вызове `update_attribute`. Причина в том, что токен и дайджест в реализации запоминания создаются для уже существующего пользователя, тогда как `before_create` вызывается *до* создания учетной записи (то еть когда пользователя еще не существует). В результате вызова этой функции, когда новый пользователь создается посредством `User.new` (как при регистрации в листинге 7.17), он автоматически получает оба атрибута – `activation_token` и `activation_digest`; так как последний связан со столбцом в базе данных (рис. 10.1), он будет автоматически записан при сохранении пользователя.

Соединив все вышесказанное, мы получим модель User, представленную в листинге 10.3. В соответствии с виртуальной природой токена активации мы добавили второй атрибут в `attr_accessor`. Обратите внимание, что заодно мы воспользовались возможностью и применили ссылку на метод для перевода адреса электронной почты в нижний регистр.

Листинг 10.3 ❖ Добавление поддержки активации в модель User **ЗЕЛЕНЫЙ** (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

    # Преобразует адрес электронной почты в нижний регистр.
    def downcase_email
      self.email = email.downcase
    end

    # Создает и присваивает токен активации и его дайджест.
    def create_activation_digest
      self.activation_token = User.new_token
      self.activation_digest = User.digest(activation_token)
    end
end
```

Прежде чем двигаться дальше, нужно обновить тестовые данные, чтобы тестовые пользователи активировались автоматически (листинги 10.4 и 10.5). (Метод `Time.zone.now` – это встроенный вспомогательный метод Rails, возвращающий текущую отметку времени с учетом часового пояса сервера.)

Листинг 10.4 ❖ Активация образцов пользователей по умолчанию (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
```

Листинг 10.5 ❖ Активация пользователей в тестовых данных (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 %>
```

Чтобы изменения вступили в силу, нужно очистить базу данных и снова заполнить ее:

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

10.1.2. Отправка письма для активации

Завершив подготовку модели, можно приступать к реализации отправки электронного письма со ссылкой на страницу активации учетной записи. Для этого с помощью библиотеки Action Mailer добавим объект, осуществляющий рассылку, который будем использовать в методе `create` контроллера Users для отправки электронного письма со ссылкой на страницу активации. Объект рассылки имеет ту же структуру, что и методы контроллера, то есть шаблоны писем в нем определяются в виде представлений. В этом разделе мы создадим такой объект рассылки и представление со ссылкой, содержащей токен активации и адрес электронной почты учетной записи.

Так же как модель или контроллер, объект рассылки можно создать командой `rails generate`:

```
$ rails generate mailer UserMailer account_activation password_reset
```

Здесь мы сразу добавили необходимые методы `account_activation` и `password_reset`, последний понадобится в разделе 10.2.

Попутно Rails создаст по два шаблона представления для каждого из методов, один в текстовом формате, второй в формате HTML. Представления для метода `account_activation` представлены в листингах 10.6 и 10.7.

Листинг 10.6 ❖ Текстовое представление для метода account_activation (app/views/user_mailer/account_activation.text.erb)

```
UserMailer#account_activation

<%= @greeting %>, find me in app/views/user_mailer/account_activation.text.erb
```

Листинг 10.7 ❖ Сгенерированное HTML-представление для метода account_activation (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>
```

Давайте рассмотрим сгенерированный объект рассылки, чтобы понять, как он работает (листинги 10.8 и 10.9). В листинге 10.8 мы видим адрес по умолчанию `from` (от кого), общий для всех объектов рассылки в приложении, а в каждом методе в листинге 10.9 есть адрес получателя. (В листинге 10.8 также указан шаблон `'mailer'`, определяющий формат электронного письма; в данной книге мы не будем пользоваться шаблонами, тем не менее вы сможете найти их в `app/views/layouts`.) Сгенерированный код включает также переменную экземпляра (`@greeting`), доступную в представлениях объекта рассылки, так же как переменные экземпляров контроллеров доступны в обычных представлениях.

Листинг 10.8 ❖ Сгенерированный объект рассылки для приложения (app/mailers/application_mailer.rb)

```
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer'
end
```

Листинг 10.9 ❖ Сгенерированный объект рассылки для модели User (app/mailers/user_mailer.rb)

```
class UserMailer < ActionMailer::Base

  # Тема письма может быть указана в I18n-файле
  # config/locales/en.yml,как:
  #
  # en.user_mailer.account_activation.subject
  #
  def account_activation
    @greeting = "Hi"

    mail to: "to@example.org"
  end

  # Тема письма может быть указана в I18n-файле
  # config/locales/en.yml,как:
  #
  # en.user_mailer.password_reset.subject
  #
  def password_reset
    @greeting = "Hi"

    mail to: "to@example.org"
  end
end
```

Для создания письма со ссылкой на страницу активации нужно настроить сгенерированные шаблоны (листинг 10.10), затем создать переменную экземпляра с пользователем (чтобы использовать ее в представлениях) и послать письма по адресу `user.email` (листинг 10.11). Как можно заметить в листинге 10.11, метод `email` принимает также ключ `subject`, значение которого используется в качестве темы электронного письма.

Листинг 10.10 ❖ Объект рассылки для приложения с новым адресом отправителя по умолчанию (app/mailers/application_mailer.rb)

```
class ApplicationMailer < ActionMailer::Base
  default from: "noreply@example.com"
  layout 'mailer'
end
```

Листинг 10.11 ❖ Отправка ссылки на страницу активации учетной записи (app/mailers/user_mailer.rb)

```
class UserMailer < ActionMailer::Base

  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
```

Как и в обычных представлениях, для настройки шаблона можно воспользоваться встроенным Ruby, в данном случае мы добавим приветствие пользователя по имени и включим уникальную ссылку на страницу активации. Наш план состоит в том, чтобы найти пользователя по адресу электронной почты и проверить его подлинность по токену активации, поэтому в ссылке должны быть оба этих параметра. Так как активация реализована в виде ресурса AccountActivations, сам токен может быть аргументом именованного маршрута, определенного в листинге 10.1:

```
edit_account_activation_url(@user.activation_token, ...)
```

Напомню, что

```
edit_user_url(user)
```

возвращает URL вида:

```
http://www.example.com/users/1/edit
```

поэтому соответствующий URL со ссылкой на страницу активации будет выглядеть вот так:

```
http://www.example.com/account_activations/q5lt38hQDc_959PVoo6b7A/edit
```

q5lt38hQDc_959PVoo6b7A – это строка в формате base64, созданная методом new_token (листинг 8.31). Она играет ту же роль, что идентификатор пользователя в /users/1/edit. В частности, в методе edit контроллера Activations токен будет доступен в хэше params как params[:id].

Чтобы включить сюда еще и адрес электронной почты, нам понадобится *параметр запроса*, который в URL выглядит как пара ключ/значение после знака вопроса[1]:

```
account_activations/q5lt38hQDc_959PVoo6b7A/edit?email=foo%40example.com
```

Обратите внимание, что знак «@» в адресе электронной почты выглядит в URL как %40, то есть он «экранирован» для гарантии допустимости URL. В Rails, чтобы передать параметры запроса, нужно в именованный маршрут включить хэш:

```
edit_account_activation_url(@user.activation_token, email: @user.email)
```

При таком способе определения параметров запроса в именованных маршрутах Rails автоматически экранирует все специальные символы. Получившийся адрес электронной почты автоматически будет восстановлен в контроллере и доступен в элементе params[:email].

Теперь, когда в листинге 10.11 была определена переменная экземпляра @user, можно создать необходимые ссылки с помощью именованного маршрута edit и встроенного Ruby, как показано в листингах 10.12 и 10.13. Для конструирования ссылки в HTML-шаблоне использован метод link_to.

Листинг 10.12 ❖ Текстовое представление для активации учетной записи (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) %>
```

Листинг 10.13 ❖ HTML-представление для активации учетной записи (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) %>
```

Чтобы увидеть результат определения этих двух шаблонов, можно использовать *предварительный просмотр электронных писем* – специальный URL, пре-

[1] Адреса URL могут содержать множество параметров запроса в виде нескольких пар ключ/значение, разделенных знаком амперсанда &, например /edit?name=Foo%20Bar&email=foo%40example.com.

доставляемый фреймворком Rails, чтобы мы могли видеть, как выглядят наши электронные письма. Но прежде нужно добавить кое-какие настройки в окружение разработки приложения, как показано в листинге 10.14.

Листинг 10.14 ❖ Настройки электронных писем в среде разработки (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
```

Здесь указано имя хоста `'example.com'`, но вам нужно указать действительное имя хоста вашего окружения разработки. Например, в мой системе используются (в зависимости от того, использую я облачную IDE или локальный сервер):

```
host = 'rails-tutorial-c9-mhartl.c9.io'   # Облачная IDE
```

или

```
host = 'localhost:3000'                   # Локальный сервер
```

После перезапуска сервера разработки, чтобы эти настройки вступили в силу, нужно обновить файл предварительного просмотра объекта рассылки, который автоматически был создан в разделе 10.1.2, как показано в листинге 10.15.

Листинг 10.15 ❖ Сгенерированный файл предварительного просмотра объекта рассылки (test/mailers/previews/user_mailer_preview.rb)

```
# Предварительный просмотр всех писем по адресу:
# http://localhost:3000/rails/mailers/user_mailer
class UserMailerPreview <ActionMailer::Preview

  # Предварительный просмотр этого письма:
  # http://localhost:3000/rails/mailers/user_mailer/account_activation
  def account_activation
    UserMailer.account_activation
  end

  # Предварительный просмотр этого письма:
  # http://localhost:3000/rails/mailers/user_mailer/password_reset
  def password_reset
    UserMailer.password_reset
  end

end
```

Так как метод account_activation из листинга 10.11 ожидает получить допустимый объект пользователя, этот код не будет работать. Чтобы исправить проблему, определим переменную user с информацией о первом пользователе в базе данных разработки и передадим ее в вызов UserMailer.account_activation (листинг 10.16). Обратите внимание, что здесь также присваивается значение атрибуту user.activation_token, так как шаблонам активации в листингах 10.12 и 10.13 необходим токен для включения в URL ссылки. (У пользователя в базе данных нет атрибута activation_token, так как он виртуальный (раздел 10.1.1).)

Листинг 10.16 ❖ Действующий метод предварительного просмотра для активации учетной записи (test/mailers/previews/user_mailer_preview.rb)

```
# Предварительный просмотр всех писем по адресу:
# http://localhost:3000/rails/mailers/user_mailer
class UserMailerPreview <ActionMailer::Preview

  # Предварительный просмотр этого письма:
  # 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

  # Предварительный просмотр этого письма:
  # http://localhost:3000/rails/mailers/user_mailer/password_reset
  def password_reset
    UserMailer.password_reset
  end
end
```

Теперь можно перейти по предложенному адресу URL и увидеть, как будет выглядеть электронное письмо со ссылкой на страницу активации учетной записи. (Если вы работаете в облачной IDE, вам нужно заменить localhost:3000 на соответствующий базовый URL.) Получившиеся письма в формате HTML и в виде простого текста представлены на рис. 10.2 и 10.3.

И наконец, напишем пару тестов для проверки только что увиденного. Это не столь сложно, как кажется, потому что Rails создает полезные заготовки тестов (листинг 10.17).

Листинг 10.17 ❖ Автоматически сгенерированные тесты объекта рассылки (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
    assert_equal ["to@example.org"], mail.to
```

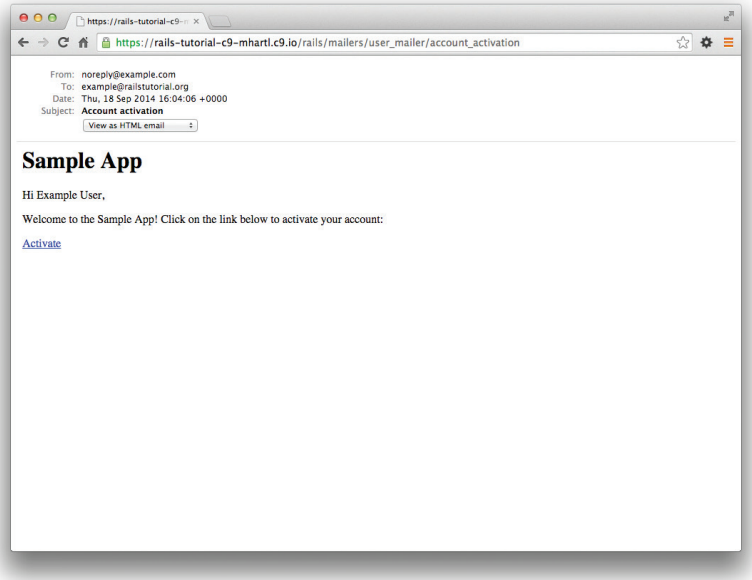


Рис. 10.2 ❖ HTML-версия электронного письма для активации учетной записи

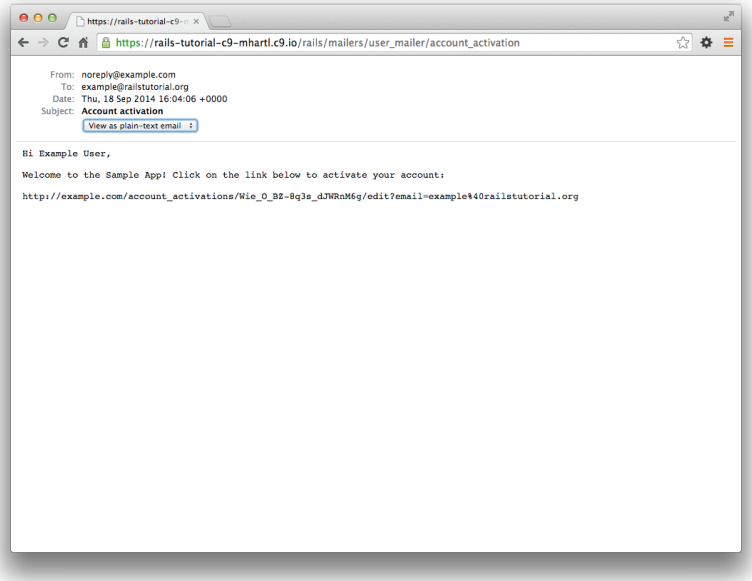


Рис. 10.3 ❖ Текстовая версия электронного письма для активации учетной записи

```
    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
```

Здесь применен весьма мощный метод `assert_match`, который может принимать не только строки, но и регулярные выражения:

```
assert_match 'foo', 'foobar'    # true
assert_match 'baz', 'foobar'    # false
assert_match /\w+/, 'foobar'    # true
assert_match /\w+/, '$#!*+@'    # false
```

Тесты в листинге 10.18 с помощью `assert_match` проверяют наличие имени, токена активации и экранированного адреса электронной почты в теле письма. Обратите внимание на использование

```
CGI::escape(user.email)
```

для маскировки адреса электронной почты тестового пользователя[1].

Листинг 10.18 ❖ Тест текущей реализации электронного письма **КРАСНЫЙ** (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
```

[1] Чтобы понять, как осуществить что-то подобное, поищите в Google по фразе: «ruby rails экранирование url». Вы найдете две основные возможности (http://stackoverflow.com/questions/6714196/ruby-url-encoding-string): `URI::encode(str)` и `CGI::escape(str)`. Попробовав их обе, вы поймете, что работает только последняя. (На самом деле есть еще и третья: библиотека `ERB::Util`, включающая метод `url_encode` (http://apidock.com/ruby/ERB/Util/url_encode), который действует точно так же.)

```
    assert_match CGI::escape(user.email), mail.body.encoded
  end
end
```

Мы позаботились, чтобы добавить токен активации в определение тестового пользователя, иначе он был бы пустым.

Чтобы обеспечить успешное прохождение тестов, нужно указать правильный хост домена в файле с настройками тестовой среды (листинг 10.19).

Листинг 10.19 ❖ Настройка хоста в файле тестовой среды (config/environments/test.rb)

```
Rails.application.configure do
  .
  .
  .
  config.action_mailer.delivery_method = :test
  config.action_mailer.default_url_options = { host: 'example.com' }
  .
  .
  .
end
```

Теперь тест должен быть **ЗЕЛЕНЫМ**:

Листинг 10.20 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test:mailers
```

Чтобы задействовать объект рассылки в приложении, необходимо добавить пару строк в метод `create`, используемый для регистрации пользователей (листинг 10.21). Обратите внимание, что в листинге 10.21 мы изменили адрес перенаправления после регистрации. Раньше выполнялся переход на страницу профиля пользователя (раздел 7.4), но теперь это не имеет смысла, так как необходимо сначала активировать учетную запись. Вместо этого новый пользователь переадресуется на главную страницу.

Листинг 10.21 ❖ Добавление активации учетной записи в процедуру регистрации пользователя. **КРАСНЫЙ** (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
```

Так как теперь вместо страницы профиля пользователь переадресуется по корневому адресу URL и вход не осуществляется автоматически, как раньше, набор тестов стал **КРАСНЫМ**, хотя приложение работает именно так, как задумано. Мы исправим это, временно закомментировав строки, вызывающие неудачу, как показано в листинге 10.22. Позже, в разделе 10.1.4, мы раскомментируем их и напишем правильные тесты.

Листинг 10.22 ❖ Временное исключение неудачных тестов **ЗЕЛЕНЫЙ** (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
```

Если сейчас попытаться зарегистрировать нового пользователя, вы будете переадресованы, как показано на рис. 10.4, и на ваш адрес будет отправлено электронное письмо, как показано в листинге 10.23. Имейте в виду, что в окружении

разработки вы *не* получите настоящего письма, но упоминание о нем появится в журнале сервера. (Может понадобиться немного прокрутить журнал, чтобы увидеть нужную запись.) О том, как отправить настоящее электронное письмо в эксплуатационном окружении, рассказывается в разделе 10.3.

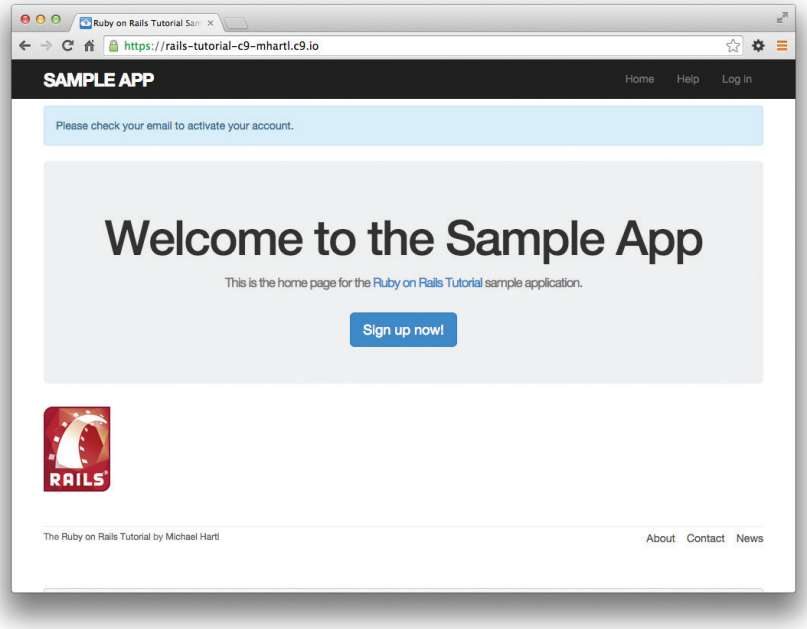


Рис. 10.4 ❖ Главная страница с сообщением о необходимости активации после регистрации

Листинг 10.23 ❖ **Пример** письма со ссылкой на страницу активации учетной записи в журнале сервера

```
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?ema
il=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. Активация учетной записи

Теперь, когда у нас имеется готовое письмо (листинг 10.23), нужно добавить метод `edit` в контроллер AccountActivations, выполняющий активацию учетной записи. В разделе 10.1.2 говорилось, что токен активации и адрес электронной почты доступны в виде `params[:id]` и `params[:email]` соответственно. Следуя модели обслуживания паролей (листинг 8.5) и токенов запоминания (листинг 8.36), найдем пользователя и подтвердим его подлинность, как показано ниже:

```
user = User.find_by(email: params[:email])
if user && user.authenticated?(:activation, params[:id])
```

(Как будет показано чуть ниже, в этом выражении должно быть еще одно логическое условие. Сможете ли вы догадаться, какое именно?)

В примере выше для проверки совпадения дайджеста с токеном используется метод `authenticated?`, но сейчас он работать не будет, так как специализирован для токена запоминания (листинг 8.33):

```
# Возвращает true, если указанный токен соответствует дайджесту.
def authenticated?(remember_token)
  return false if remember_digest.nil?
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

Здесь `remember_digest` – это атрибут модели User, и внутри модели его можно переписать так:

```
self.remember_digest
```

Каким-то образом нам нужно сделать его *переменной*, чтобы можно было вызвать

```
self.activation_token
```

вместо того, чтобы передавать соответствующий параметр в `authenticated?`.

Решение предполагает наш первый пример *метапрограммирования*, который, по сути, является программой, порождающей другую программу. (Метапрограммирование – одна из самых сильных особенностей Ruby, и «волшебство» многих функций Rails обусловлено его использованием.) Ключом в данном случае является могущественный метод `send`, который позволяет вызывать метод по его названию, «посылая сообщение» указанному объекту. Например, в следующем консольном сеансе метод `send` посылает сообщение встроенному Ruby-объекту, чтобы узнать длину массива:

```
$ rails console
>> a = [1, 2, 3]
>> a.length
=> 3
>> a.send(:length)
=> 3
>> a.send('length')
=> 3
```

Передача символа `:length` или строки `'length'` методу `send` эквивалента вызову метода `length` данного объекта. В качестве второго примера получим доступ к атрибуту `activation_digest` первого пользователя в базе данных:

```
>> 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"
```

В последнем примере мы определили переменную `attribute` и присвоили ей символ `:activation`, а затем использовали интерполяцию строк, чтобы сконструировать аргумент для `send`. Здесь можно также использовать строку `'activation'`, но обычно принято использовать символы, и в любом случае

```
"#{attribute}_digest"
```

после интерполяции строки превращается в

```
"activation_digest"
```

(Мы уже знаем, что символы интерполируются в строки, как было показано в разделе 7.4.2.) Опираясь на обсуждение метода send, текущий метод authenticated? можно переписать, как показано ниже:

```
def authenticated?(remember_token)
  digest = self.send('remember_digest')
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(remember_token)
end
```

Воспользовавшись этим шаблоном, можно обобщить метод, добавив аргумент с именем дайджеста, а затем применить интерполяцию строк:

```
def authenticated?(attribute, token)
  digest = self.send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

(Здесь мы переименовали второй аргумент в token, чтобы подчеркнуть универсальность метода.) Так как мы находимся внутри модели User, можно опустить self, получив в результате идиоматически более корректную версию:

```
def authenticated?(attribute, token)
  digest = send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

Прежнее поведение authenticated? можно воспроизвести, вызвав его вот так:

```
user.authenticated?(:remember, remember_token)
```

Применив рассуждения выше к модели User, получим обобщенный метод authenticated? , показанный в листинге 10.24.

Листинг 10.24 ❖ Обобщенный метод authenticated? **КРАСНЫЙ** (app/models/user.rb)

```
class User <ActiveRecord::Base
  .
  .
  .
  # Возвращает true, если указанный токен соответствует дайджесту.
  def authenticated?(attribute, token)
    digest = send("#{attribute}_digest")
    return false if digest.nil?
    BCrypt::Password.new(digest).is_password?(token)
  end
  .
  .
  .
end
```

Заголовок листинга 10.24 сообщает, что набор тестов терпит неудачу:

Листинг 10.25 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Причина в том, что метод `current_user` (листинг 8.36) и проверка на пустой дайджест (листинг 8.43) используют старую версию `authenticated?`, которая принимает один аргумент вместо двух. Чтобы это исправить, подставим в обоих местах вызов обобщенного метода (листинги 10.26 и 10.27).

Листинг 10.26 ❖ Вызов обобщенного метода authenticated? в current_user (app/helpers/sessions_helper.rb)

```
module SessionsHelper
  .
  .
  .
  # Возвращает текущего вошедшего пользователя (если имеется).
  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
```

Листинг 10.27 ❖ Вызов обобщенного метода authenticated? в тестах User **ЗЕЛЕНЫЙ** (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
```

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 10.28 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

При рефакторинге такого кода легко ошибиться без надежного набора тестов, вот почему мы решились преодолеть множество трудностей, чтобы написать достойные тесты в разделах 8.4.2 и 8.4.6.

Теперь мы готовы приступить к методу `edit`, подтверждающему подлинность пользователя по адресу электронной почты в хэше `params`. Проверка будет выглядеть так:

```
if user && !user.activated? && user.authenticated?(:activation, params[:id])
```

Обратите внимание на выражение `!user.activated?`, это то самое дополнительное условие, упоминавшееся выше. Оно предотвращает повторную активацию уже активированной учетной записи, и это важно, так в процессе подтверждения осуществляется вход пользователя, а нам совсем не нужно, чтобы злоумышленник, завладевший активационной ссылкой, смог войти в приложение под видом пользователя.

Если условие выше выполняется, нужно активировать учетную запись и обновить метку времени `activated_at`[1]:

```
user.update_attribute(:activated,    true)
user.update_attribute(:activated_at, Time.zone.now)
```

В результате получаем реализацию метода `edit`, представленную в листинге 10.29. Обратите внимание, что здесь учитывается случай недопустимого токена активации; это будет происходить весьма редко, и в таком случае достаточно переадресовать пользователя на главную страницу.

Листинг 10.29 ❖ Метод edit для активации учетных записей (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
```

[1] Мы используем два вызова `update_attribute` вместо одного `updated_attributes`, так как (согласно разделу 6.1.5) второй вариант запускает проверки, которые в нашем случае не пройдут из-за отсутствия пароля.

```
      flash[:danger] = "Invalid activation link"
      redirect_to root_url
    end
  end
end
```

Теперь можно воспользоваться адресом URL из листинга 10.23, чтобы активировать соответствующую учетную запись. Например, в моей системе я ввел адрес URL

```
http://rails-tutorial-c9-mhartl.c9.io/account_activations/fFb_F94mgQtmlSvRFGsITw/edit?email=michael%40michaelhartl.com
```

и получил результат, представленный на рис. 10.5.

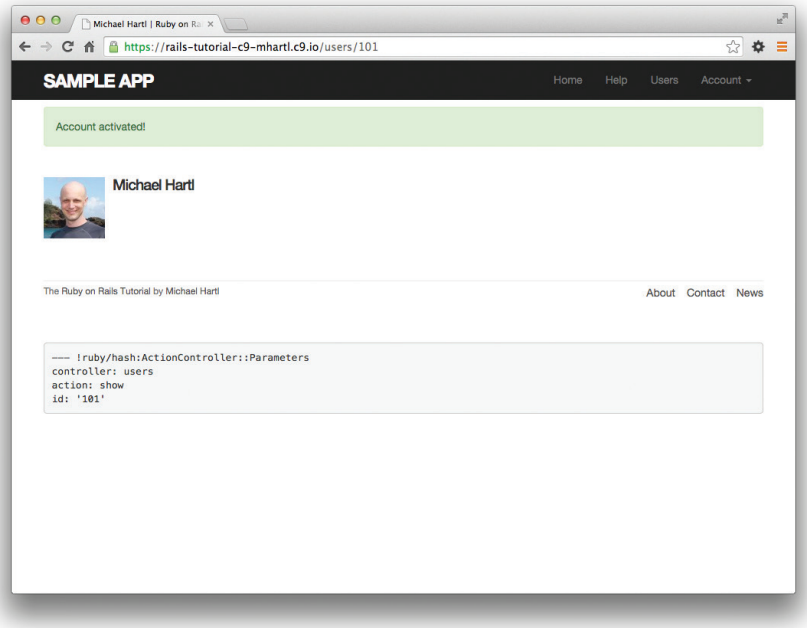


Рис. 10.5 ❖ Страница профиля после успешной активации

Конечно, сейчас активация на самом деле ничего не дает, так как мы не изменили механизма входа пользователей. Чтобы активация что-либо значила, необходимо разрешить вход пользователям только при условии успешной активации. Как показано в листинге 10.30, для этого вход должен происходить по обычной схеме, только если `user.activated?` возвращает `true`; в противном случае пользователь должен перенаправляться на корневой URL с предупреждающим сообщением (рис. 10.6).

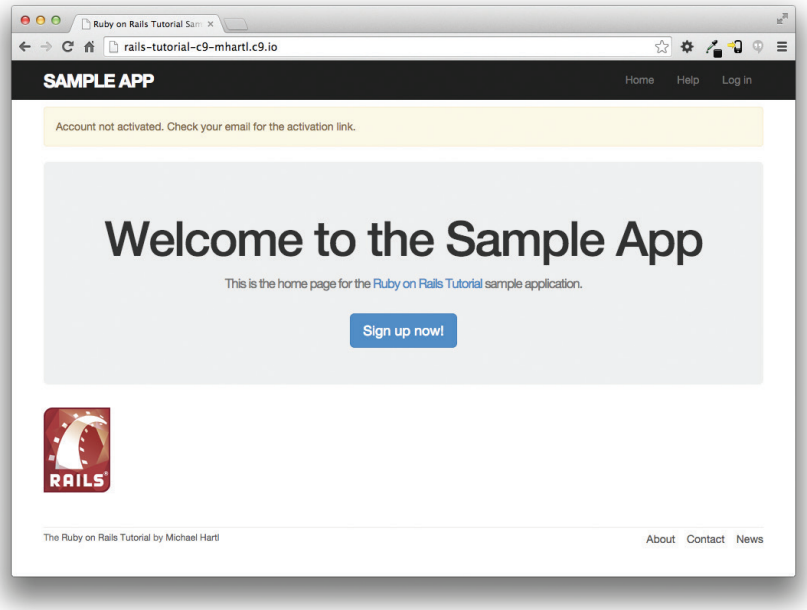


Рис. 10.6 ❖ Сообщение, предупреждающее о том, что учетная запись не активирована

Листинг 10.30 ❖ Предотвращение входа пользователей с неактивными учетными записями (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])
      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
```

На этом основная функциональность активации учетных записей завершена, кроме одной детали. (Нам нужно предотвратить отображение неактивированных учетных записей, но это мы оставим в качестве упражнения (раздел 10.5).) В разделе 10.1.4 мы закончим процесс, добавив немного тестов и проведя небольшой рефакторинг.

10.1.4. Тестирование активации и рефакторинг

В этом разделе мы добавим интеграционные тесты для проверки активации учетной записи. У нас уже есть тест для регистрации с допустимой информацией, поэтому мы просто добавим в него несколько этапов (тест из раздела 7.4.4 в листинге 7.26). Их довольно много, но они весьма простые (листинг 10.31).

Листинг 10.31 ❖ Тестирование активации учетной записи **ЗЕЛЕНЫЙ**
(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?
    # Попытаться выполнить вход до активации.
    log_in_as(user)
    assert_not is_logged_in?
    # Недопустимый токен активации
    get edit_account_activation_path("invalid token")
    assert_not is_logged_in?
    # Допустимый токен, неверный адрес электронной почты
    get edit_account_activation_path(user.activation_token, email: 'wrong')
    assert_not is_logged_in?
    # Допустимый токен
    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
```

Здесь довольно много кода, но новый находится только в строке

```
assert_equal 1, ActionMailer::Base.deliveries.size
```

Она проверяет, что было доставлено точно одно письмо. Так как массив `deliveries` – глобальный, нужно очистить его в методе `setup`, чтобы тесты не закончились неудачей, если почта отправляется еще на каком-то этапе тестирования (такой случай появится в разделе 10.2.5). Кроме того, здесь впервые применяется метод `assigns`; как рассказывалось в упражнении (раздел 8.6) к главе 8, метод `assigns` позволяет получить доступ к переменной экземпляра в соответствующем методе. Например, метод `create` контроллера Users определяет переменную `@user` (листинг 10.21), поэтому в тестах можно обратиться к ней посредством `assigns(:user)`. Наконец, обратите внимание, что снова включены в работу строки, закомментированные в листинге 10.22.

На данный момент набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 10.32 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Покончив с тестами, можно приступать к небольшому рефакторингу – вынесем некоторые манипуляции с пользователем из контроллера в модель. В частности, создадим метод `activate` для обновления атрибутов активации и `send_activation_email` для отправки письма со ссылкой на страницу активации. Эти дополнительные методы представлены в листинге 10.33, а переработанный код приложения – в листингах 10.34 и 10.35.

Листинг 10.33 ❖ Добавление методов активации в модель User (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  # Активирует учетную запись.
  def activate
    update_attribute(:activated,    true)
    update_attribute(:activated_at, Time.zone.now)
  end

  # Посылает письмо со ссылкой на страницу активации.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  private
    .
    .
    .
end
```

Листинг 10.34 ❖ Отправка электронного письма посредством объекта модели User (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
```

Листинг 10.35 ❖ Активация учетной записи через объект модели User (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
```

В листинге 10.33 мы избавились от переменной `user`., так как внутри модели User ее нет:

```
-user.update_attribute(:activated,    true)
-user.update_attribute(:activated_at, Time.zone.now)
+update_attribute(:activated,    true)
+update_attribute(:activated_at, Time.zone.now)
```

(Вместо `user` можно было бы использовать ссылку `self`, но, как рассказывалось в разделе 6.2.5, ссылку `self` можно не употреблять внутри модели.) Кроме того, мы заменили `@user` на `self` в вызове объекта рассылки:

```
-UserMailer.account_activation(@user).deliver_now
+UserMailer.account_activation(self).deliver_now
```

Это те самые детали, которые легко не заметить даже при таком простом рефакторинге, но ошибки в них не позволят пропустить хороший набор тестов. Кстати, он сейчас должен по-прежнему быть **ЗЕЛЕНЫМ**:

Листинг 10.36 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Реализация активации учетной записи завершена полностью, и этот момент достоин, чтобы отметить его отправкой кода в репозиторий:

```
$ git add -A
$ git commit -m "Add account activations"
```

10.2. Сброс пароля

После завершения реализации активации учетной записи (а значит, и подтверждения адреса электронной почты) самое время приступить к обработке весьма распространенной ситуации, когда пользователь забыл свой пароль. Многие этапы реализации похожи, поэтому у нас появится возможность применить знания, полученные в разделе 10.1. Тем не менее начало процесса отличается; в отличие от активации учетной записи, сброс пароля требует изменить одно из представлений и добавить две новые формы (для ввода адреса электронной почты и нового пароля).

Прежде чем перейти к коду, набросаем ожидаемую последовательность действий при сбросе пароля. Сначала в форму входа нужно добавить ссылку «Я забыл пароль» (рис. 10.7). Она будет ссылаться на страницу с формой ввода электронного адреса, куда будет отправлена ссылка для сброса пароля (рис. 10.8). Эта ссылка, в свою очередь, приведет на страницу с формой ввода нового пароля (с подтверждением, рис. 10.9).

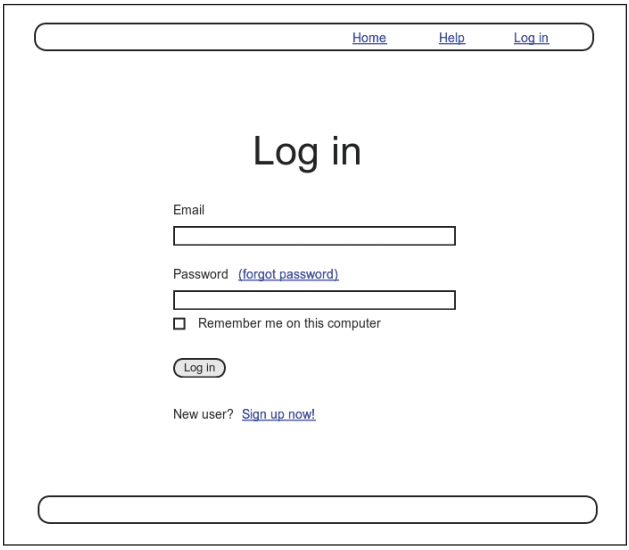


Рис. 10.7 ❖ Макет ссылки «Я забыл пароль»

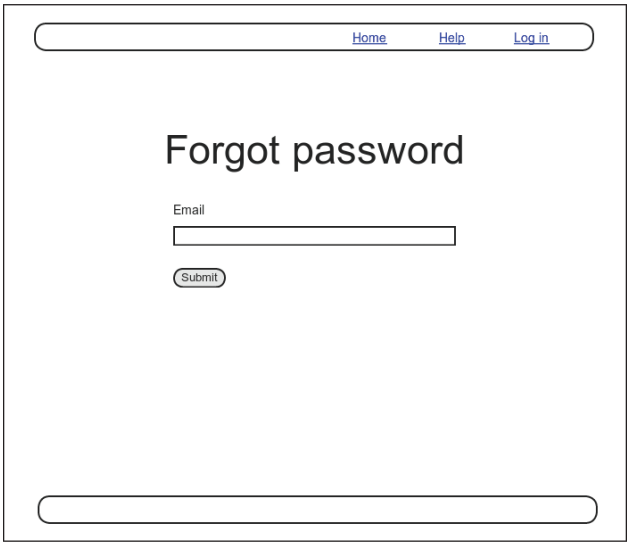


Рис. 10.8 ❖ Макет формы «Я забыл пароль»

Рис. 10.9 ❖ Макет формы ввода нового пароля

По аналогии с активацией учетной записи, наш генеральный план будет состоять в создании ресурса Password Resets, сбрасывающего пароль, состоящий из токена сброса и соответствующего дайджеста. Основная последовательность выглядит так:

1) когда пользователь запросит сброс пароля, нужно найти его по адресу электронной почты;
2) если адрес существует в базе данных, сгенерировать токен сброса и дайджест;
3) сохранить дайджест в базе данных и отправить пользователю электронное письмо со ссылкой, в которой находятся токен и электронный адрес;
4) когда пользователь перейдет по ссылке, найти его по электронному адресу и подтвердить подлинность токена, сравнив его с дайджестом;
5) если подлинность подтверждена, вернуть форму для смены пароля.

10.2.1. Ресурс для сброса пароля

Как и в случае с активацией учетной записи (раздел 10.1.1), сначала создадим контроллер для нового ресурса:

```
$ rails generate controller PasswordResets new edit --no-test-framework
```

Как и в разделе 10.1.1, мы указали флаг `--no-test-framework`, чтобы пропустить создание тестов, вместо них мы дополним интеграционные тесты из раздела 10.1.4.

Так как формы будут использоваться для сброса пароля (рис. 10.8) и изменения пароля в модели User (рис. 10.9), необходимо добавить маршруты для методов `new`, `create`, `edit` и `update`. Все это можно организовать в одной строке `resources` (листинг 10.37).

Листинг 10.37 ❖ Добавление ресурса для сброса пароля (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]
end
```

Код в листинге 10.37 создает маршруты RESTful, перечисленные в табл. 10.2. В частности, первый маршрут в этой таблице соответствует ссылке «Я забыл пароль»:

```
new_password_reset_path
```

которая представлена в листинге 10.38 и на рис. 10.10.

Таблица 10.2 ❖ Маршруты, поддерживаемые ресурсом Password Resets

HTTP-запрос	URL	Метод	Именованный маршрут
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)

Листинг 10.38 ❖ Добавление ссылки для сброса пароля (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>
```

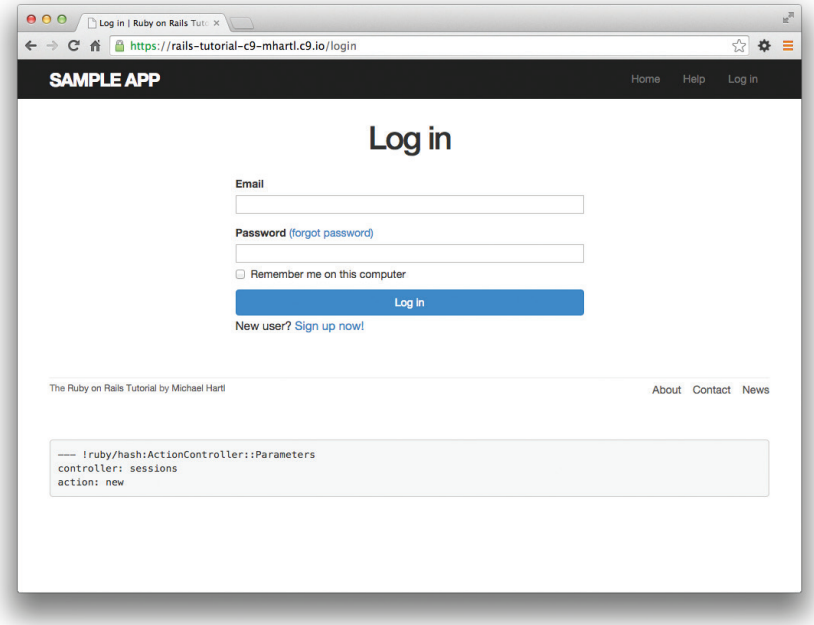


Рис. 10.10 ❖ Страница входа со ссылкой «Я забыл пароль»

Модель данных для сброса пароля похожа на ту, что использовалась для активации учетной записи (рис. 10.1). Следуя шаблону токенов запоминания (раздел 8.4) и токенов активации (раздел 10.1), сброс пароля подразумевает пару: виртуальный токен сброса для электронного письма и соответствующий дайджест для поиска пользователя. Если хранить токен в открытом виде, злоумышленник с доступом к базе данных сможет отправить запрос на сброс пароля по электронному адресу пользователя, а затем использовать токен и адрес электронной почты для перехода по ссылке на страницу ввода нового сброса пароля и таким образом получить контроль над учетной записью. Поэтому дайджест совершенно необходим для безопасного сброса пароля. В качестве дополнительной меры безопасности планируется также ограничить срок действия ссылки парой часов, а это требует сохранить время ее отправки. Получившиеся атрибуты `reset_digest` и `reset_sent_at` представлены на рис. 10.11.

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

Рис. 10.11 ❖ Модель User с дополнительными атрибутами для сброса пароля

Миграция для их добавления выглядит вот так:

```
$ rails generate migration add_reset_to_users reset_digest:string \
> reset_sent_at:datetime
```

(Как и раньше, значок > во второй строке обозначает «продолжение строки» и появляется автоматически, поэтому его не нужно набирать.) Затем, как всегда, проводим миграцию:

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

10.2.2. Контроллер и форма для сброса пароля

Чтобы создать представление для сброса пароля, будем действовать по аналогии с формой входа на сайт (листинг 8.2) для создания нового сеанса, повторно представленной в листинге 10.39 для ясности.

Листинг 10.39 ❖ Повторный обзор кода формы входа на сайт (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>
```

Форма сброса пароля имеет много общего с ней; самое важное отличие заключается в использовании другого ресурса и URL в вызове `form_for` и в отсутствии атрибута `password`. Результат показан в листинге 10.40 и на рис. 10.12.

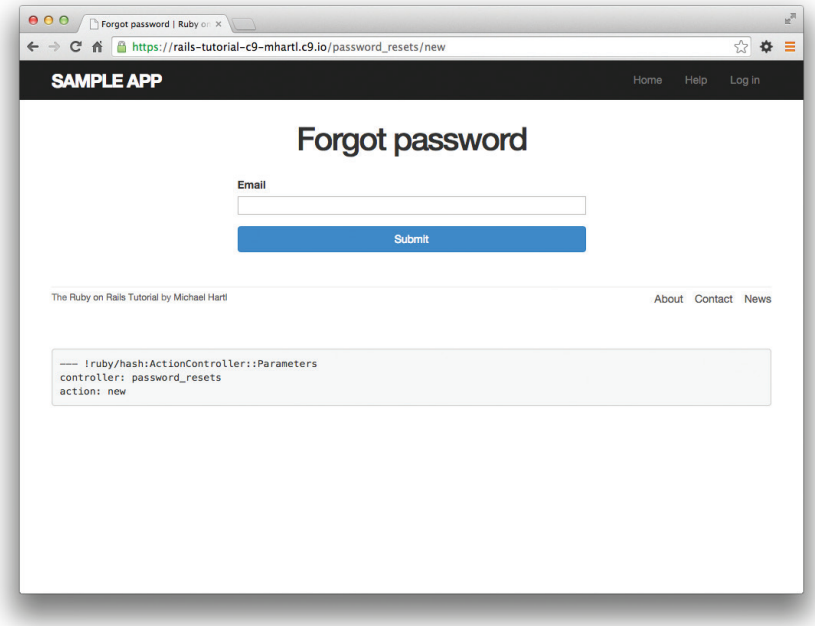


Рис. 10.12 ❖ Форма «Я забыл пароль»

Листинг 10.40 ❖ Представление для сброса пароля (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>
```

После отправки этой формы нужно найти пользователя по адресу электронной почты и обновить его атрибуты: токен сброса и время отправки. Затем переадресовать пользователя на корневой URL с информационным сообщением. Как и в случае со входом на сайт (листинг 8.9), при отправке недопустимой информации нужно снова отобразить страницу `new` с сообщением `flash.now`. Результат представлен в листинге 10.41.

Листинг 10.41 ❖ Метод create для сброса пароля (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
      @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
```

Код в модели User повторяет метод `create_activation_digest` в функции обратного вызова `before_create` (листинг 10.3), как показано в листинге 10.42.

Листинг 10.42 ❖ Добавление в модель User методов для сброса пароля (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
  .
  .
  .
  # Активирует учетную запись.
  def activate
```

```
    update_attribute(:activated,    true)
    update_attribute(:activated_at, Time.zone.now)
  end

  # Посылает письмо со ссылкой на страницу активации.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  # Устанавливает атрибуты для сброса пароля.
  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

  # Посылает письмо со ссылкой на форму ввода нового пароля.
  def send_password_reset_email
    UserMailer.password_reset(self).deliver_now
  end

  private

    # Преобразует адрес электронной почты в нижний регистр.
    def downcase_email
      self.email = email.downcase
    end

    # Создает и присваивает токен активации и его дайджест.
    def create_activation_digest
      self.activation_token = User.new_token
      self.activation_digest = User.digest(activation_token)
    end
end
```

Как показано на рис. 10.13, приложение уже правильно обрабатывает ввод неверного адреса электронной почты. Чтобы оно заработало так же хорошо в ответ на ввод верного адреса, необходимо определить метод объекта рассылки для сброса пароля.

10.2.3. Метод объекта рассылки для сброса пароля

Отправка электронного письма для сброса пароля в листинге 10.42 реализована так:

```
UserMailer.password_reset(self).deliver_now
```

Метод `password_reset` объекта рассылки очень похож на аналогичный метод для активации учетной записи, разработанный в разделе 10.1.2. Сначала напишем метод `password_reset` (листинг 10.43), а затем определим шаблоны представлений письма в текстовом (листинг 10.44) и HTML-формате (листинг 10.45).

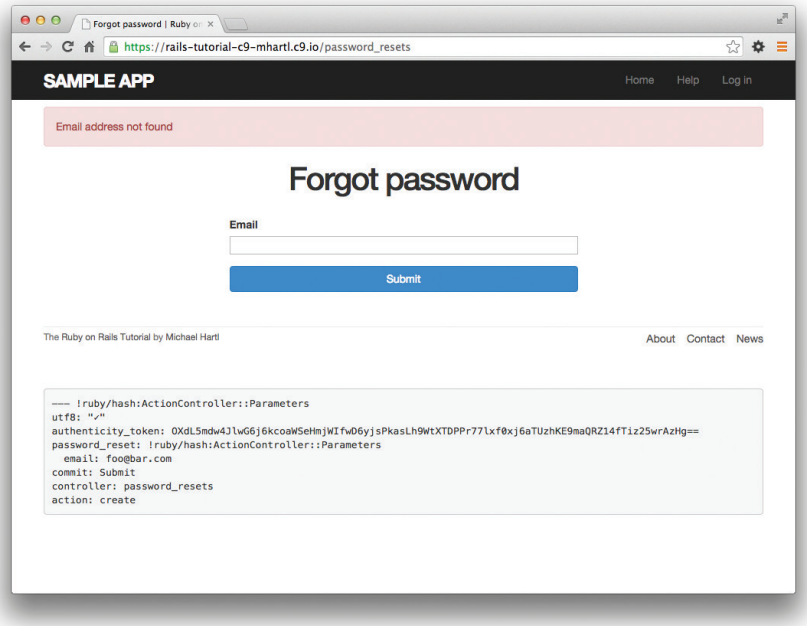


Рис. 10.13 ❖ Форма «Я забыл пароль» после ввода неверного адреса электронной почты

Листинг 10.43 ❖ Отправка ссылки на сброс пароля (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
```

Листинг 10.44 ❖ Текстовый шаблон электронного письма для сброса пароля (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 passwo
rd will stay as it is.
```

Листинг 10.45 ❖ HTML-шаблон электронного письма для сброса пароля (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 passwo
rd will stay as it is.
</p>
```

Так же как письма для активации учетной записи (раздел 10.1.2), эти письма можно просмотреть с помощью инструмента предварительного просмотра в Rails. Код в листинге 10.46 почти в точности повторяет код в листинге 10.16.

Листинг 10.46 ❖ Действующий метод предварительного просмотра для сброса пароля (test/mailers/previews/user_mailer_preview.rb)

```
# Предварительный просмотр всех писем по адресу:
# http://localhost:3000/rails/mailers/user_mailer
classUserMailerPreview<ActionMailer::Preview

  # Предварительный просмотр этого письма:
  # 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

  # Предварительный просмотр этого письма:
  # 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
```

Вид электронных писем в текстовом и HTML-формате показан на рис. 10.14 и 10.15.

По аналогии с тестом, проверяющим отправку письма для активации аккаунта (листинг 10.18), напишем короткий тест для сброса пароля, как показано в листинге 10.47. Обратите внимание на необходимость создания токена сброса для использования в представлениях; в отличие от токена активации, который создается в функции `before_create` (листинг 10.3) для каждого пользователя, токен сброса создается, только когда пользователь успешно отправит форму «Я забыл пароль».

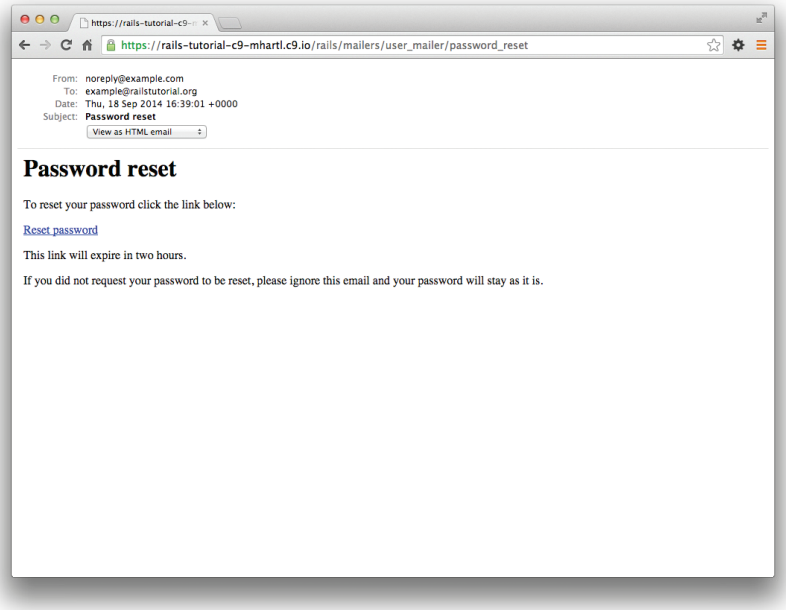


Рис. 10.14 ❖ HTML-версия электронного письма для сброса пароля

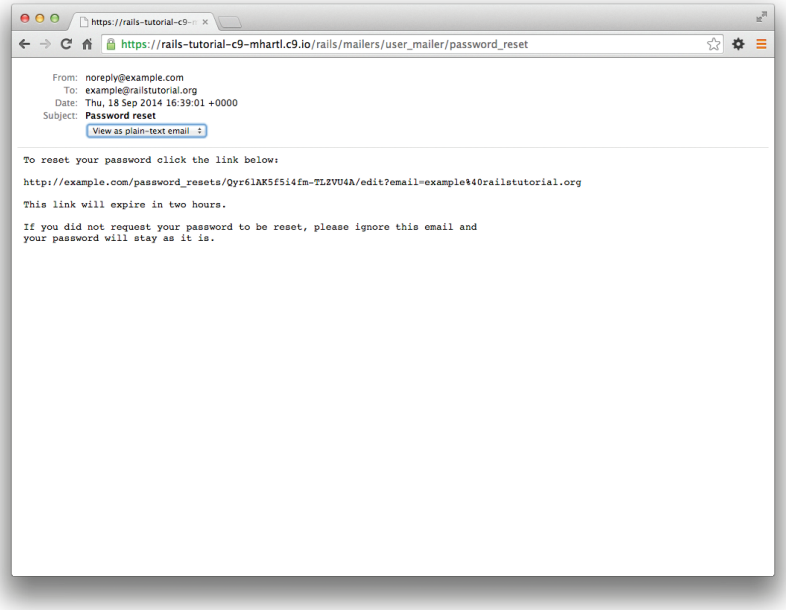


Рис. 10.15 ❖ Текстовая версия электронного письма для сброса пароля

В интеграционных тестах (листинг 10.54) это будет происходить само собой, но в данном случае необходимо создать его вручную.

Листинг 10.47 ❖ Добавление тестов, проверяющих отправку письма для сброса пароля **ЗЕЛЕНЫЙ** (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
```

В настоящий момент набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 10.48 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

Теперь в случае отправки верного адреса электронной почты пользователь получит страницу, изображенную на рис. 10.16. Соответствующее электронное письмо появится в журнале сервера, как показано в листинге 10.49.

Листинг 10.49 ❖ **Пример** записи об отправке электронного письма для сброса пароля в журнале сервера

```
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: <5407babbee139_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=mi
chael%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 passwo
rd will stay as it is.
----==_mimepart_5407babbe3505_8722b257d045617
Content-Type: text/html;
 charset=UTF-8
Content-Transfer-Encoding: 7bit
```

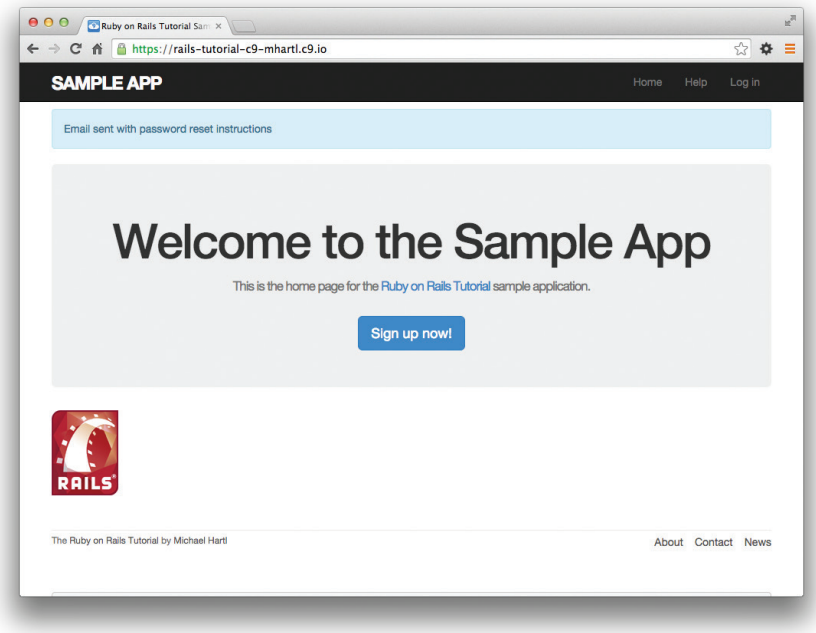


Рис. 10.16 ❖ Результат отправки верного адреса электронной почты

```
<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 passwo
rd will stay as it is.
</p>
----==_mimepart_5407babbe3505_8722b257d045617--
```

10.2.4. Смена пароля

Чтобы заставить работать ссылку на форму

```
http://example.com/password_resets/3BdBrXeQZSWqFIDRN8cxHA/edit?email=foo%40bar.com
```

необходимо создать форму для смены пароля. Задача похожа на изменение данных пользователя через представление редактирования (листинг 9.2), но теперь нам нужны только поля для ввода пароля и подтверждения. Хотя есть еще одно условие: нам потребуется найти пользователя по адресу электронной почты, значит, этот адрес потребуется в методах `edit` и `update`. В методе `edit` он будет доступен автоматически, так как присутствует в ссылке выше, но после отправки формы он будет потерян. Решение заключается в использовании скрытых полей для размещения (но не отображения) адреса электронной почты на странице с последующей отправкой его вместе со всей остальной формой. Результат представлен в листинге 10.50.

Листинг 10.50 ❖ Форма для смены пароля (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>
```

Здесь вместо

```
f.hidden_field :email, @user.email
```

использован вспомогательный метод, генерирующий теги элементов формы

```
hidden_field_tag :email, @user.email
```

потому что ссылка на страницу сброса помещает адрес электронной почты в `params[:email]`, в то время как в первом варианте она будет находиться в `params[:user][:email]`.

Чтобы эта форма заработала, нужно определить переменную `@user` в методе `edit` контроллера PasswordResets. Так же как при активации учетной записи (листинг 10.29), это подразумевает поиск пользователя по адресу электронной почты из `params[:email]`. По окончании поиска нужно проверить его допустимость, то есть: пользователь существует, учетная запись активирована, подлинность подтверждена по токену сброса из `params[:id]` (с помощью обобщенного метода `authenticated?` в листинге 10.24). Так как допустимость `@user` является обязательным условием в обоих методах – `edit` и `update`, – мы поместим поиск и проверку в пару предварительных фильтров, как показано в листинге 10.51.

Листинг 10.51 ❖ Метод edit для сброса пароля (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

    # Подтверждает допустимость пользователя.
    def valid_user
      unless (@user && @user.activated? &&
              @user.authenticated?(:reset, params[:id]))
        redirect_to root_url
      end
    end
end
```

Сравните строку

```
authenticated?(:reset, params[:id])
```

в листинге 10.51 со строкой

```
authenticated?(:remember, cookies[:remember_token])
```

в листинге 10.26 и со строкой

```
authenticated?(:activation, params[:id])
```

в листинге 10.29. Все вместе эти три случая завершают методы подтверждения подлинности из табл. 10.1.

Теперь переход по ссылке из листинга 10.49 должен привести в форму смены пароля, изображенную на рис. 10.17.

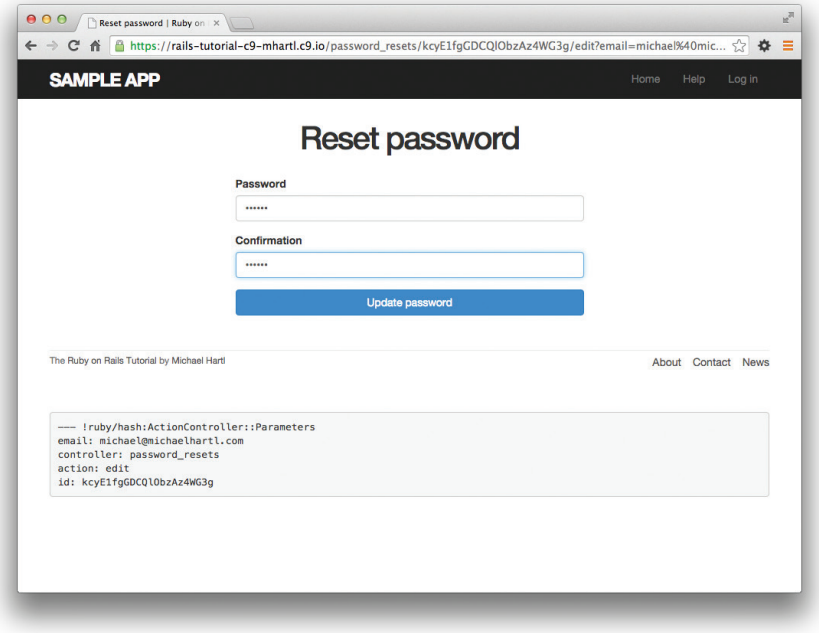


Рис. 10.17 ❖ Форма смены пароля

Чтобы реализовать метод `update`, соответствующий методу `edit` из листинга 10.51, рассмотрим четыре возможные ситуации: окончание срока действия ссылки для сброса пароля, успешное изменение пароля, неудачное изменение из-за его недопустимости и неудачное изменение (которое изначально выглядит успешным) из-за пустого пароля и подтверждения. Первый случай применим к обоим методам – `edit` и `update`, – поэтому логически должен быть реализован в предварительном фильтре (листинг 10.52). Следующие два случая соответствуют двум основным веткам основной инструкции `if` из листинга 10.52. Так как форма редактирования изменяет объект модели Active Record (то есть пользователя), для отображения сообщений об ошибках можно полностью положиться на

общедоступный шаблон из листинга 10.50. Единственным исключением является случай с пустым паролем, так как сейчас в нашей модели User (листинг 9.10) это допустимо, данный случай должен проверяться и обрабатываться явно[1]. Для этого добавим ошибку прямо в сообщение для объекта `@user`:

Листинг 10.52 ❖ Метод update для смены пароля (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 password_blank?
      flash.now[:danger] = "Password can't be blank"
      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
```

[1] Нам нужно разобраться только с пустым паролем, потому что если пустым будет лишь подтверждение, проверка не пройдет из-за несовпадения с паролем (эту проверку мы пропускаем при пустом пароле), и будет показано соответствующее сообщение об ошибке.

```
    def user_params
      params.require(:user).permit(:password, :password_confirmation)
    end

    # Возвращает true, если пароль пустой.
    def password_blank?
      params[:user][:password].blank?
    end

    # Предварительные фильтры

    def get_user
      @user = User.find_by(email: params[:email])
    end

    # Подтверждает допустимость пользователя.
    def valid_user
      unless (@user && @user.activated? &&
              @user.authenticated?(:reset, params[:id]))
      redirect_to root_url
    end
  end

    # Проверяет срок действия токена.
    def check_expiration
      if @user.password_reset_expired?
        flash[:danger] = "Password reset has expired."
        redirect_to new_password_reset_url
      end
    end
end
```

Эта реализация делегирует проверку срока действия ссылки для сброса пароля модели User:

```
@user.password_reset_expired?
```

Чтобы это сработало, нужно определить в модели метод `password_reset_expired?`. Как отмечено в шаблонах электронных писем из раздела 10.2.3, срок действия ссылки истекает через два часа после отправки, с помощью Ruby это можно выразить так:

```
reset_sent_at < 2.hours.ago
```

Этот код может показаться странным, если символ < прочитать как «менее, чем», потому что тогда выражение будет звучать так: «Ссылка на сброс пароля отправлена менее двух часов назад», а это совершенно противоположно тому, что нам нужно. В данном случае знак < следует читать «ранее, чем», тогда получится: «Ссылка на сброс пароля отправлена раньше, чем два часа назад». Это именно то, что нужно. В результате мы получаем метод `password_reset_expired?`, как показано в листинге 10.53. (Формальная демонстрация корректности сравнения приводится в разделе 10.6.)

Листинг 10.53 ❖ Добавление методов для сброса пароля к модели User (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  # Возвращает true, если время для сброса пароля истекло.
  def password_reset_expired?
    reset_sent_at < 2.hours.ago
  end

  private
  .
  .
  .
end
```

Теперь вызов `update` из листинга 10.52 должен заработать. Результат отправки допустимого и недопустимого паролей и подтверждения представлен на рис. 10.18 и 10.19 соответственно. (Нам не хватит терпения ждать два часа, поэтому мы покроем тестами третью ветку, это оставлено в качестве упражнения (раздел 10.5).)

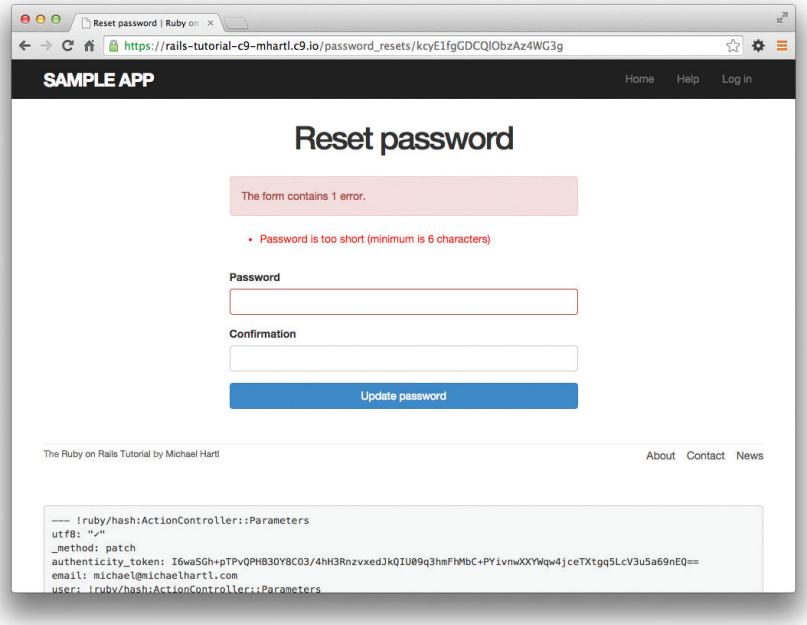


Рис. 10.18 ❖ Неудачная попытка смены пароля

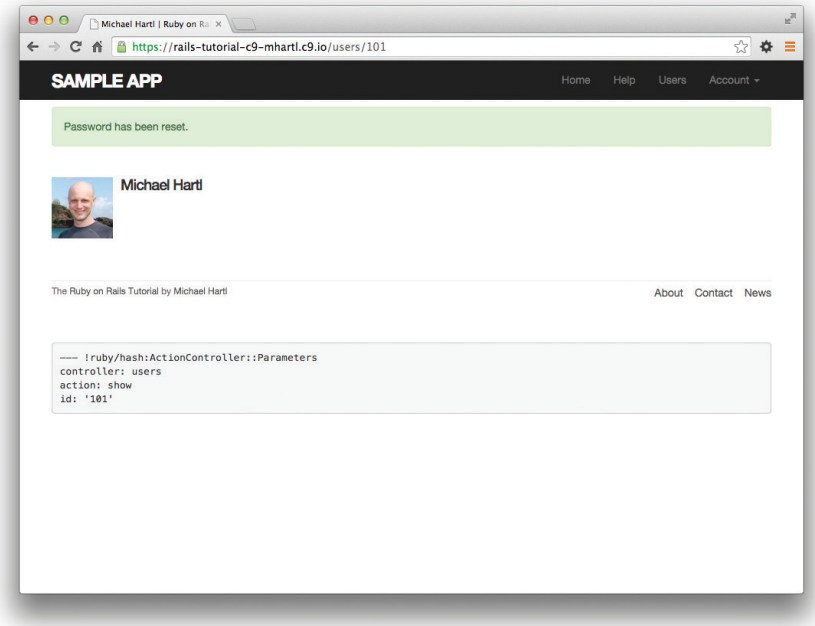


Рис. 10.19 ❖ Успешная попытка смены пароля

10.2.5. Тестирование сброса пароля

В этом разделе мы напишем интеграционные тесты, покрывающие два из трех случаев в листинге 10.52: отправку верной и неверной информации. (Как было сказано выше, третий случай оставлен в качестве упражнения.) Начнем с создания файла тестов для сброса пароля:

```
$ rails generate integration_test password_resets
      invoke  test_unit
      create    test/integration/password_resets_test.rb
```

Этапы тестирования сброса пароля очень близки к тестам активации учетной записи в листинге 10.31, хотя есть отличие в самом начале: сначала вызовем форму «Я забыл пароль», отправим неверный, затем верный адрес электронной почты, в последнем случае будет создан токен сброса и отправлено электронное письмо со ссылкой для сброса пароля. Затем выполняется переход по ссылке из письма, и снова отправляется неверная и верная информация, чтобы проверить корректность поведения в каждом случае. Получившийся тест, представленный в листинге 10.54, является превосходным упражнением в чтении кода.

Листинг 10.54 ❖ Интеграционные тесты для проверки сброса пароля (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'
    # Неверный адрес электронной почты
    post password_resets_path, password_reset: { email: "" }
    assert_not flash.empty?
    assert_template 'password_resets/new'
    # Верный адрес электронной почты
    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
    # Форма сброса пароля
    user = assigns(:user)
    # Неверный адрес электронной почты
    get edit_password_reset_path(user.reset_token, email: "")
    assert_redirected_to root_url
    # Неактивированная учетная запись
    user.toggle!(:activated)
    get edit_password_reset_path(user.reset_token, email: user.email)
    assert_redirected_to root_url
    user.toggle!(:activated)
    # Верный адрес электронной почты, неверный токен
    get edit_password_reset_path('wrong token', email: user.email)
    assert_redirected_to root_url
    # Верный адрес электронной почты, верный токен
    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
    # Недопустимый пароль и подтверждение
    patch password_reset_path(user.reset_token),
          email: user.email,
          user: { password:              "foobaz",
                  password_confirmation: "barquux" }
    assert_select 'div#error_explanation'
    # Пустой пароль
    patch password_reset_path(user.reset_token),
```

```
          email: user.email,
          user: { password:              "  ",
                  password_confirmation: "foobar" }
    assert_not flash.empty?
    assert_template 'password_resets/edit'
    # Допустимый пароль и подтвержение
    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
```

Большинство идей в листинге 10.54 уже высказывалось выше; действительно новым элементом является только тест тега `input`:

```
assert_select "input[name=email][type=hidden][value=?]", user.email
```

Здесь проверяется наличие тега `input` с правильными именем, типом (`hidden`) и адресом электронной почты:

```
<input id="email" name="email" type="hidden" value="michael@example.com" />
```

Сейчас набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 10.55 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

10.3. Отправка электронных писем из эксплуатационного окружения

Главной нашей задачей в этом разделе является настройка приложения для отправки электронных писем из эксплуатационного окружения. Сначала разберемся с настройками бесплатной службы отправки писем, а затем сконфигурируем и развернем приложение.

Для отправки электронных писем из эксплуатационного окружения воспользуемся службой SendGridTo. Она доступна в виде дополнения для Heroku и может использоваться в подтвержденных учетных записях. (Это потребует добавить в вашу учетную запись Heroku информацию о кредитной карте, но само подтверждение выполняется бесплатно.) Лучше всего для наших целей подходит уровень «новичок» («starter», на момент написания этих строк он ограничен четырьмя сотнями писем в день, зато услуга предоставляется совершенно бесплатно). Добавим ее в приложение:

```
$ heroku addons:create sendgrid:starter
```

(Это может не сработать в системах со старой версией интерфейса командной строки Heroku. В таком случае обновите ее или попробуйте старый синтаксис `heroku addons:add sendgrid:starter`.)

Чтобы настроить взаимодействие со службой SendGrid, нужно заполнить настройки SMTP для эксплуатационного окружения. Также нужно определить переменную `host` – адрес вашего сайта, как показано в листинге 10.56.

Листинг 10.56 ❖ Настройка Rails для использования SendGrid в эксплуатационном окружении (config/environments/production.rb)

```
Rails.application.configure do
  .
  .
  .
  config.action_mailer.raise_delivery_errors = true
  config.action_mailer.delivery_method = :smtp
  host = '<имя heroku-приложения>.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
```

Настройка включает в себя определение параметров `user_name` и `password` для учетной записи SendGrid, но обратите внимание, что это делается через переменную окружения `ENV` вместо прямого определения значений. Это лучшая практика для приложений, действующих в эксплуатационной среде, которые из соображений безопасности никогда не должны содержать в исходном коде конфиденциальную информацию, такую как незашифрованные пароли. В данном случае эти переменные определяются автоматически дополнением SendGrid, но в разделе 11.4.4 мы увидим пример, как определить их самостоятельно. Если вам любопытно, заглянуть в переменные окружения можно, как показано ниже:

```
$ heroku config:get SENDGRID_USERNAME
$ heroku config:get SENDGRID_PASSWORD
```

Сейчас следует объединить рабочую и основную ветви:

```
$ bundle exec rake test
$ git add -A
```

```
$ git commit -m "Add password resets & email configuration"
$ git checkout master
$ git merge account-activation-password-reset
```

отправить все в удаленный репозиторий и развернуть в Heroku:

```
$ bundle exec rake test
$ git push
$ git push heroku
$ heroku run rake db:migrate
```

По окончании развертывания попробуйте зарегистрироваться в учебном приложении, действующем в эксплуатационном окружении, используя принадлежащий вам адрес электронной почты. Вы должны получить письмо со ссылкой для активации, реализованное в разделе 10.1.1 (рис. 10.20). Если вы забудете (или сделаете вид, что забыли) свой пароль, вы сможете сбросить его, воспользовавшись функцией, реализованной в разделе 10.2 (рис. 10.21).

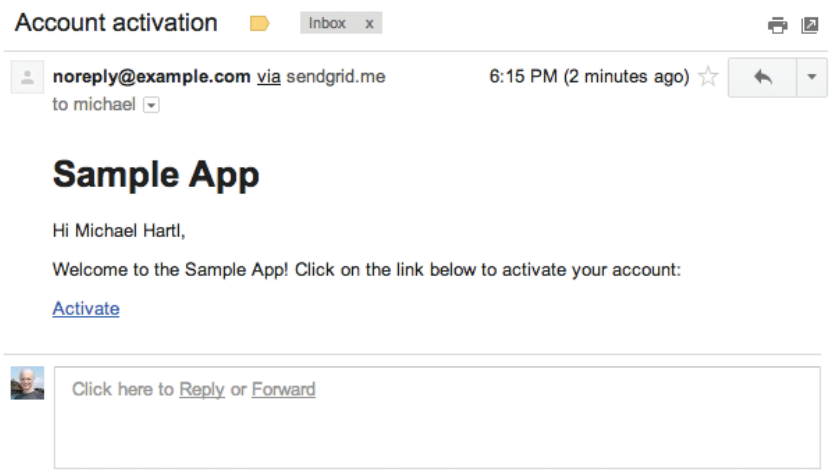


Рис. 10.20 ❖ Электронное письмо со ссылкой для активации учетной записи, отправленное приложением в эксплуатационном окружении

10.4. Заключение

Механизм регистрации и входа/выхода в нашем приложении после добавления активации аккаунта и сброса пароля окончательно завершен и реализован на профессиональном уровне. Остальная часть данной книги будет опираться на этот фундамент для создания сайта с Twitter-подобными микросообщениями (глава 11) и лентой сообщений от читаемых пользователей (глава 12). Попутно мы познакомимся с некоторыми мощными возможностями Rails, в том числе с загрузкой сообщений, выполнением довольно сложных запросов к базе данных и продвинутым моделированием данных посредством `has_many` и `has_many :through`.

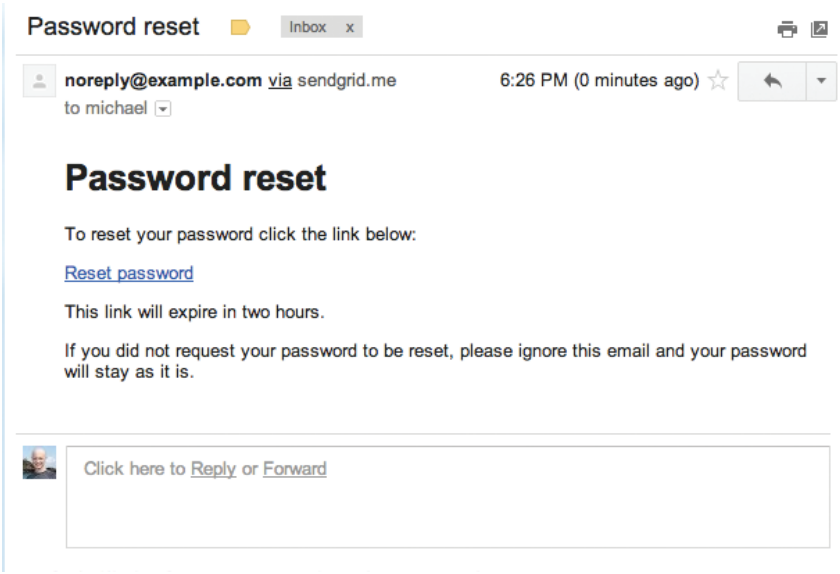


Рис. 10.21 ❖ Электронное письмо для сброса пароля, отправленное приложением в эксплуатационном окружении

10.4.1. Что мы узнали в этой главе

- Подобно сеансам, активацию учетной записи можно смоделировать в виде ресурса, не являющегося при этом объектом Active Record.
- Rails может создавать методы и представления Active Mailer для отправки электронных писем.
- Action Mailer поддерживает электронные письма в текстовом формате и в формате HTML.
- Так же как в случае с обычными методами и представлениями, переменные экземпляра, определяемые в методах Action Mailer, доступны в его представлениях.
- Так же как сеансы и активацию учетной записи, сброс пароля можно смоделировать в виде ресурса, не являющегося при этом объектом Active Record.
- Для активации учетной записи и сброса пароля используется сгенерированный токен, чтобы создать уникальный URL, с помощью которого происходят активация учетной записи и смена пароля соответственно.
- Для проверки поведения механизма рассылки электронной почты большую пользу приносят тесты самого механизма рассылки и интеграционные тесты.
- Организовать отправку электронных писем в эксплуатационном окружении можно с применением службы SendGrid.

10.5. Упражнения

Примечание. *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Напишите интеграционный тест для ветви из листинга 10.52, отклоняющей сброс пароля в случае перехода по ссылке с истекшим сроком годности, заполнив шаблон в листинге 10.57. (Здесь используется `response.body`, возвращающий разметку HTML тела страницы.) Есть много способов протестировать проверку срока действия ссылки, но метод, предложенный в листинге 10.57, направлен на (нечувствительную к регистру) проверку наличия слова «expired» в теле ответа.
2. Сейчас *все* пользователи отображаются на странице /users со списком и доступны для просмотра через URL вида `/users/:id`, хотя имеет смысл показывать только активированных пользователей. Позаботьтесь о таком поведении, заполнив шаблон в листинге 10.58[1]. (В нем используется метод `where` механизма Active Record, о котором подробнее рассказывается в разделе 11.3.3.) *Дополнительно*: напишите интеграционные тесты для адресов /users и /users/:id.
3. В листинге 10.42 оба метода – `activate` и `create_reset_digest` – выполняют два вызова `update_attribute`, каждый из которых требует отдельного обращения к базе данных. Заполните шаблон в листинге 10.59, заменив каждую пару вызовов `update_attribute` одним вызовом `update_columns`, чтобы обращаться к базе данных только один раз. После внесения изменений убедитесь, что набор тестов по-прежнему **ЗЕЛЕНЫЙ**.

Листинг 10.57 ❖ Тест проверки срока годности ссылки на сброс пароля **ЗЕЛЕНЫЙ** (test/integration/password_resets_test.rb)

```
require 'test_helper'

class PasswordResetsTest < ActionDispatch::IntegrationTest

  def setup
    ActionMailer::Base.deliveries.clear
    @user = users(:michael)
  end
```

[1] Обратите внимание на использование оператора `and` вместо `&&`. Они практически идентичны, но второй имеет более высокий приоритет, за счет чего он будет слишком тесно связан с `root_url` в данном случае. Мы могли бы исправить эту проблему, заключив `root_url` в скобки, но идиоматически правильнее использовать оператор `and`.

```
  .
  .
  .
  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
```

Листинг 10.58 ❖ Заготовка реализации отображения только активированных пользователей (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
```

Листинг 10.59 ❖ Заготовка метода 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
  .
  .
  .
  # Активирует учетную запись.
  def activate
    update_columns(activated: FILL_IN, activated_at: FILL_IN)
```

```
  end

  # Посылает письмо со ссылкой на страницу активации.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  # Устанавливает атрибуты для сброса пароля.
  def create_reset_digest
    self.reset_token = User.new_token
    update_columns(reset_digest: FILL_IN,
                   reset_sent_at: FILL_IN)
  end

  # Посылает письмо со ссылкой на форму ввода нового пароля.
  def send_password_reset_email
    UserMailer.password_reset(self).deliver_now
  end

  private

    # Преобразует адрес электронной почты в нижний регистр.
    def downcase_email
      self.email = email.downcase
    end

    # Создает и присваивает токен активации и его дайджест.
    def create_activation_digest
      self.activation_token = User.new_token
      self.activation_digest = User.digest(activation_token)
    end
end
```

10.6. Доказательство сравнения срока годности ссылки

В этом разделе мы докажем, что сравнение срока годности ссылки в реализации сброса пароля (раздел 10.2.4) действует правильно. Начнем с определения двух временных интервалов. Пусть Δt_r – интервал с момента отправки ссылки, Δt_e – срок годности (то есть 2 часа). Срок годности ссылки истек, если временной интервал с момента отправки больше этого ограничения:

$$\Delta t_r > \Delta t_e. \tag{10.1}$$

Если записать текущее время как t_N, время отправки ссылки как t_r и время годности как t_e (то есть 2 часа назад), получим:

$$\Delta t_r = t_N - t_r \tag{10.2}$$

и

$$\Delta t_e = t_N - t_e. \tag{10.3}$$

Подставив (10.2) и (10.3) в (10.1), получим:

$$\Delta t_r > \Delta t_e$$
$$t_N - t_r > t_N - t_e$$
$$-t_r > -t_e,$$

что при умножении на –1 дает

$$t_r < t_e. \tag{10.4}$$

Превратив (10.4) в код со значением t_e = 2 часа тому назад, получим метод `password_reset_expired?` в листинге 10.53:

```
def password_reset_expired?
  reset_sent_at < 2.hours.ago
end
```

Как отмечалось в разделе 10.2.4, если читать знак < как «ранее, чем» вместо «менее, чем», этот код будет иметь смысл: «Ссылка на сброс пароля выслана ранее, чем два часа назад».

Глава 11

Микросообщения пользователей

В процессе разработки ядра учебного приложения мы столкнулись с четырьмя ресурсами – пользователи, сеансы, активация учетных записей и сброс пароля, но только первый из них основан на модели Active Record с таблицей в базе данных. Теперь пришло время добавить второй такой ресурс: *микросообщения пользователей* – короткие сообщения, связанные с конкретными пользователями[1]. Зачатки микросообщений мы видели в главе 2, а в этой главе реализуем полноценную версию поддержки микросообщений из раздела 2.3: создадим модель данных Micropost, свяжем ее с моделью User при помощи методов `has_many` и `belongs_to`, а затем напишем формы и частичные шаблоны, необходимые для обработки и отображения результатов (включая загруженные изображения). В главе 12 мы завершим создание нашего крохотного клона Твиттера, добавив понятие *следования* за пользователями для получения *потока* их микросообщений.

11.1. Модель Micropost

Работу с ресурсом Microposts начнем с создания модели Micropost, которая содержит основные характеристики микросообщений. За основу мы возьмем опыт, полученный в разделе 2.3, – новая модель Micropost все так же будет связана с моделью User и содержать проверки данных. В отличие от прежней модели, она будет полностью протестирована, кроме того, в ней автоматически будет происходить *упорядочивание*, а также *уничтожение* при удалении «родительского» пользователя.

Если вы используете Git для управления версиями, я рекомендую сейчас создать рабочую ветку:

```
$ git checkout master
$ git checkout -b user-microposts
```

---

[1] Такое название подсказано распространенным описанием Твиттера как *микроблога*; в блоге есть сообщения, значит, в микроблоге должны быть микросообщения.

11.1.1. Базовая модель

В модели Micropost необходимы лишь два атрибута: `content` для хранения текста и `user_id` для связи с конкретным пользователем. Получившаяся структура модели показана на рис. 11.1.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime

Рис. 11.1 ❖ Модель данных Micropost

Стоит отметить, что для содержимого микросообщений здесь использован тип данных `text` (вместо `string`), способный хранить текст произвольного объема. Несмотря на то что текст будет ограничен 140 символами (раздел 11.1.2), который без труда уместится в 255-символьный тип `string`, использование типа `text` лучше отражает сущность микросообщений, которые естественнее рассматривать как блоки текста. В самом деле, в разделе 11.3.2 мы воспользуемся текстовой областью вместо однострочного поля для отправки микросообщений. Кроме того, текст даст нам в будущем большую гибкость, если мы захотим изменить ограничение длины (например, для нужд интернационализации). Наконец, тип `text` не ухудшает производительности в эксплуатационном окружении[1], поэтому нам ничего не будет стоить его использование.

Так же как модель User (листинг 6.1), создадим модель Micropost командой `generate model`:

```
$ rails generate model Micropost content:text user:references
```

Команда `generate` производит миграцию для создания таблицы `microposts` в базе данных (листинг 11.1); сравните с аналогичной миграцией для таблицы `users` из листинга 6.2. Самое большое отличие – в использовании `references` (ссылки)), которая автоматически добавит столбец `user_id` (наряду с индексом и внешним ключом)[2] для использования в связи user/micropost. Так же как для модели User, миграция для Micropost автоматически включает строку `t.timestamps`, которая (как упоминалось в разделе 6.1.1) автоматически добавляет столбцы `created_at` и `updated_at`, которые показаны на рис. 11.1. (Столбец `created_at` будет задействован в разделах 11.1.4 и 11.2.1.)

---

[1] http://www.postgresql.org/docs/9.1/static/datatype-character.html.

[2] Внешний ключ – это ограничение на уровне базы данных, указывающее, что столбец `user_id` в таблице `microposts` ссылается на столбец `id` в таблице `users`. Эта деталь не важна в данной книге, ограничение внешнего ключа поддерживается даже не всеми базами данных. (PostgreSQL, используемая нами в эксплуатационном окружении, поддерживает, а драйвер базы данных SQLite в окружении разработки – нет.) Больше о внешних ключах мы узнаем в разделе 12.1.2.

Листинг 11.1 ❖ Миграция Micropost с дополнительным индексом (db/migrate/[timestamp]_create_microposts.rb)

```
class CreateMicroposts < ActiveRecord::Migration
  def change
    create_table :microposts do |t|
      t.text :content
      t.references :user, index: true

      t.timestamps null: false
    end
    add_foreign_key :microposts, :users
    add_index :microposts, [:user_id, :created_at]
  end
end
```

Так как мы собираемся получать все микросообщения, связанные с заданным `user_id`, в порядке, обратном их созданию, добавим также индекс (блок 6.2) для столбцов `user_id` и `created_at`:

```
add_index :microposts, [:user_id, :created_at]
```

Указав столбцы `user_id` и `created_at` в виде массива, мы сообщаем Rails о необходимости создания *составного индекса*, то есть Active Record будет одновременно использовать *оба* столбца.

Теперь, как обычно, обновим базу данных:

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

11.1.2. Проверка микросообщений

После создания основной модели добавим немного проверок, реализующих необходимые ограничения. Одним из важных аспектов модели Micropost является наличие идентификатора пользователя, определяющего принадлежность микросообщения. Идиоматически верным способом для этого являются *ассоциации* Active Record, которые мы реализуем в разделе 11.1.3, а пока мы будем работать с моделью Micropost напрямую.

Первоначальные тесты микросообщений аналогичны таковым для модели User (листинг 6.7). На этапе подготовки создается микросообщение, связанное с действительным пользователем из тестовых данных, а затем проверяется правильность результата. Так как каждое микросообщение должно включать идентификатор пользователя, добавим проверку наличия `user_id`. Собрав все вместе, получим тесты, представленные в листинге 11.2.

Листинг 11.2 ❖ Тесты нового микросообщения **КРАСНЫЙ** (test/models/micropost_test.rb)

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
```

```
    # Этот код идиоматически не корректен.
    @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
```

Как отмечено в комментарии, код создания микросообщения идиоматически неправильный, мы исправим этот недостаток в разделе 11.1.3.

Тест микросообщения проходит успешно, но тест наличия идентификатора пользователя терпит неудачу, так как в модели Micropost пока нет никаких проверок:

Листинг 11.3 ❖ КРАСНЫЙ

```
$ bundle exec rake test:models
```

Чтобы это исправить, нужно просто добавить проверку наличия идентификатора пользователя, как показано в листинге 11.4. (Обратите внимание на строку `belongs_to`, она создана автоматически миграцией из листинга 11.1. Более подробно влияние этой строки мы обсудим в разделе 11.1.3.)

Листинг 11.4 ❖ Проверка user_id в микросообщениях **ЗЕЛЕНЫЙ** (app/models/micropost.rb)

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :user_id, presence: true
end
```

Теперь тест модели должен стать **ЗЕЛЕНЫМ**:

Листинг 11.5 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test:models
```

Затем добавим проверку атрибута `content` микросообщений (следуя примеру из раздела 2.3.2). Так же, как `user_id`, атрибут `content` должен быть в наличии, кроме того, он должен быть не длиннее 140 символов, что делает его действительно *микросообщением*. Сначала напишем несколько простых тестов, которые в основном повторяют тесты для модели User в листинге 6.2, как показано в листинге 11.6.

Листинг 11.6 ❖ Тесты для проверок в модели Micropost **КРАСНЫЙ** (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
```

Так же как в разделе 6.2, мы воспользовались оператором повторения строки для тестирования проверки длины микросообщения:

```
$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

Соответствующий прикладной код практически идентичен проверке имени пользователя (листинг 6.16), как показано в листинге 11.7.

Листинг 11.7 ❖ Проверки для модели Micropost **ЗЕЛЕНЫЙ** (app/models/micropost.rb)

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

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 11.8 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

11.1.3. Связь User/Micropost

При построении модели данных для веб-приложений важно иметь возможность создавать *ассоциации* (связи) между отдельными моделями. В данном случае каждое микросообщение связано с одним пользователем, а каждый пользователь связан с (возможно) множеством микросообщений – это взаимоотношение вкратце было рассмотрено в разделе 2.3.3 и схематично показано на рис. 11.2 и 11.3. В рамках реализации этих связей напишем тест для модели Micropost и добавим пару тестов для модели User.

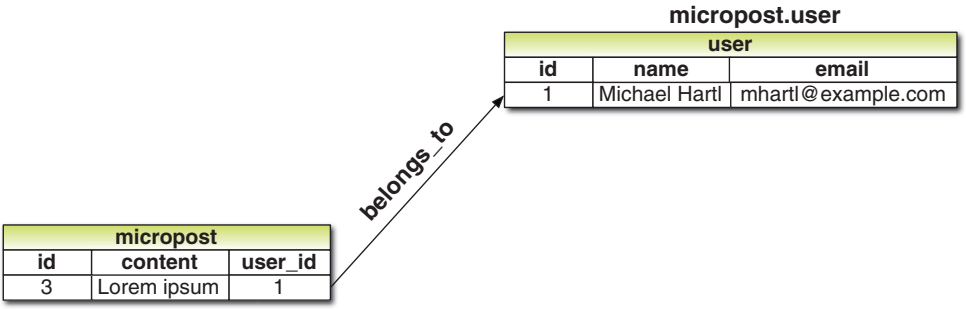


Рис. 11.2 ❖ Связь belongs_to между микросообщением и пользователем, его написавшим

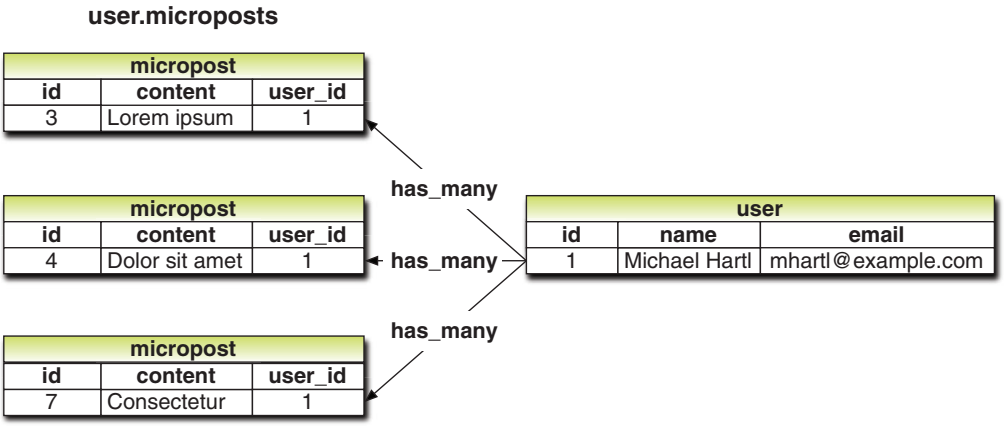


Рис. 11.3 ❖ Связь has_many между пользователем и его микросообщениями

При помощи связей `belongs_to`/`has_many`, определенных в этом разделе, Rails создает методы, показанные в табл. 11.1. Обратите внимание, вместо

```
Micropost.create
Micropost.create!
Micropost.new
```

у нас имеются

```
user.microposts.create
user.microposts.create!
user.microposts.build
```

Эти последние методы представляют собой идиоматически верный способ создания микросообщения, а именно через его связь с пользователем. При создании микросообщения таким способом атрибут `user_id` автоматически принимает правильное значение. В частности, код из листинга 11.2

```
@user = users(:michael)
# Этот код идиоматически не корректен.
@micropost = Micropost.new(content: "Lorem ipsum",
user_id: @user.id)
```

можно заменить на:

```
@user = users(:michael)
@micropost = @user.microposts.build(content: "Lorem ipsum")
```

(Так же как `new`, метод `build` возвращает объект в памяти, но не изменяет содержимого в базе данных.) После определения надлежащих связей атрибут `user_id` переменной `@micropost` автоматически будет получать значение идентификатора связанного с ней пользователя.

Чтобы заставить работать код, такой как `@user.microposts.build`, нужно добавить код, связывающий модели User и Micropost. Часть его автоматически уже была добавлена миграцией из листинга 11.1 через `belongs_to :user`, как показано в листинге 11.9. Вторую половину связи, `has_many :microposts`, нужно добавить вручную (листинг 11.10).

Таблица 11.1 ❖ Краткое описание методов, полученных в результате создания связи пользователь/микросообщение

Метод	Назначение
`micropost.user`	Возвращает объект User, связанный с микросообщением
`user.microposts`	Возвращает массив микросообщений пользователя
`user.microposts.create(arg)`	Создает микросообщение, связанное с пользователем
`user.microposts.create!(arg)`	Создает микросообщение, связанное с пользователем (выдает исключение при неудачной попытке)
`user.microposts.build(arg)`	Возвращает новый объект Micropost, связанный с пользователем
`user.microposts.find_by(id: 1)`	Ищет микросообщение с идентификатором 1 и `user_id`, равным `user.id`

Листинг 11.9 ❖ Микросообщения belongs_to (принадлежат) пользователю **ЗЕЛЕНЫЙ** (app/models/micropost.rb)

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

Листинг 11.10 ❖ У пользователя has_many (много) микрособщений **ЗЕЛЕНЫЙ** (app/models/user.rb)

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

После создания связей можно обновить метод `setup` в листинге 11.2 и создать новое микрособщение идиоматически верным способом, как показано в листинге 11.11.

Листинг 11.11 ❖ Идиоматически верный способ создания микроособщения **ЗЕЛЕНЫЙ** (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
```

После такого незначительного рефакторинга набор тестов должен быть по-прежнему **ЗЕЛЕНЫМ**:

Листинг 11.12 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

11.1.4. Усовершенствование микрособщений

В этом разделе мы добавим парочку улучшений к связи пользователь/микрособщение. В частности, организуем получение микрособщений пользователя в определенном порядке, а также сделаем микрособщения зависимыми от пользователя, чтобы происходило их автоматическое уничтожение при удалении связанного с ними пользователя.

Область видимости по умолчанию

По умолчанию метод `user.microposts` не дает никаких гарантий о порядке следования возвращаемых им сообщений, но мы бы хотели (следуя обыкновению блогов), чтобы микросообщения возвращались в обратном хронологическом порядке, то есть последнее созданное сообщение должно быть первым в списке[1]. Мы добьемся этого с помощью *области видимости по умолчанию*.

Внедрение именно таких особенностей легко может приводить к ложным результатам во время тестирования (то есть тесты будут успешно выполняться, даже если прикладной код будет работать неверно), поэтому дальнейшую разработку мы будем вести через тестирование, чтобы убедиться в корректности тестов. В частности, напишем тест, проверяющий соответствие первого микросообщения в базе данных `most_recent` (последнему) микросообщению в тестовых данных, как показано в листинге 11.13.

Листинг 11.13 ❖ Тестирование порядка микросообщений **КРАСНЫЙ** (test/models/micropost_test.rb)

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase
  .
  .
  .
  test "order should be most recent first" do
    assert_equal Micropost.first, microposts(:most_recent)
  end
end
```

Код в листинге 11.13 опирается на некоторые тестовые данные, которые определены в листинге 11.14.

Листинг 11.14 ❖ Тестовые данные для создания микросообщений (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 %>
```

[1] С подобной проблемой мы уже сталкивались в разделе 9.5, в контексте списка пользователей.

Обратите внимание на явное определение значения атрибута `created_at` с использованием встроенного Ruby. Так как эти столбцы автоматически обновляются фреймворком Rails, установка их вручную обычно невозможна, но это позволительно делать в тестовых данных. На практике в этом нет необходимости, и фактически во многих системах тестовые сообщения создаются по порядку. В данном случае заключительное сообщение в файле создается последним (и, следовательно, является самым свежим), но было бы глупо полагаться на такое непредсказуемое и зависящее от конкретной системы поведение.

Сейчас набор тестов должен быть **КРАСНЫМ**:

Листинг 11.15 ❖ КРАСНЫЙ

```
$ bundle exec rake test TEST=test/models/micropost_test.rb \
>                       TESTOPTS="--name test_order_should_be_most_recent_first"
```

Мы обеспечим успешное прохождение тестов, использовав Rails-метод `default_scope`, – кроме всего прочего, его можно применять для определения порядка по умолчанию извлечения элементов из базы данных. Чтобы гарантировать верную последовательность, включим аргумент `order` в `default_scope`, он позволит упорядочить записи по столбцу `created_at`:

```
order(:created_at)
```

К сожалению, этот способ упорядочивает записи в порядке *возрастания*, от меньшего к большему, а это значит, что наиболее старые микросообщения появятся первыми. Чтобы извлечь их в обратном порядке, придется еще немного углубиться и прибегнуть к помощи языка SQL:

```
order('created_at DESC')
```

`DESC` в языке SQL означает «descending» (по убыванию), то есть нисходящий порядок от самой новой записи к самой старой[1]. В более старых версиях Rails использование SQL было единственной возможностью добиться желаемого поведения, но начиная с Rails 4.0 можно также использовать более привычный Ruby-синтаксис:

```
order(created_at: :desc)
```

Добавив это выражение в область видимости по умолчанию для модели Micropost, получим результат, представленный в листинге 11.16.

Листинг 11.16 ❖ Упорядочивание микросообщений с помощью default_scope ЗЕЛЕНЫЙ (app/models/micropost.rb)

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  default_scope -> { order(created_at: :desc) }
```

[1] Язык SQL нечувствителен к регистру, но ключевые слова (такие как `DESC`) принято писать заглавными буквами.

```
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

Этот листинг знакомит нас с лямбда-синтаксисом объектов, называемых *объектами-процедурами*, или *лямбда-выражениями*, – это *анонимные функции* (функции, не имеющие имен). Оператор -> принимает блок (раздел 4.3.2) и возвращает объект-процедуру, которую затем можно вызвать с помощью метода `call`. Давайте посмотрим, как работают такие объекты:

```
>> -> { puts "foo" }
=> #<Proc:0x007fab938d0108@(irb):1 (lambda)>
>> -> { puts "foo" }.call
foo
=> nil
```

(Это весьма сложная тема, поэтому не переживайте, если сразу не смогли уловить смысл в написанном выше.)

Теперь набор тестов должен стать **ЗЕЛЕНЫМ**:

Листинг 11.17 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Dependent: :destroy (зависимое удаление)

Помимо упорядочивания, в реализацию микросообщений можно добавить еще одно усовершенствование. Вспомним, как рассказывалось в разделе 9.4, что администраторы сайта имеют полномочия *удаления* учетных записей пользователей. Разумеется, при удалении учетной записи следует удалить и все микросообщения соответствующего пользователя.

Для этого нам достаточно лишь передать параметр в метод `has_many`, как показано в листинге 11.18.

Листинг 11.18 ❖ Включение каскадного удаления микросообщений вместе с учетной записью их автора (app/models/user.rb)

```
class User < ActiveRecord::Base
  has_many :microposts, dependent: :destroy
  .
  .
  .
end
```

Параметр `dependent: :destroy` приговаривает связанные микросообщения (то есть принадлежащие данному пользователю) к удалению в случае удаления самого пользователя. Это предотвращает захламление базы данных никому не принадлежащими микросообщениями.

Проверить работу этого механизма можно, протестировав модель User. Для этого нужно лишь сохранить пользователя (чтобы он получил идентификатор) и создать связанное с ним микросообщение, а затем проверить – приводит ли уда-

ление учетной записи пользователя к уменьшению количества микросообщений на 1. Результат представлен в листинге 11.19. (Сравните с интеграционным тестом удаления ссылок в листинге 9.57.)

Листинг 11.19 ❖ Тест dependent: :destroy **ЗЕЛЕНЫЙ** (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
```

Если код в листинге 11.18 работает верно, набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 11.20 ❖ **ЗЕЛЕНЫЙ**

```
$ bundle exec rake test
```

11.2. Вывод микросообщений

У нас пока не реализована возможность создания микросообщений с помощью веб-браузера – она появится в разделе 11.3.2, но это не мешает нам реализовать их отображение (и ее тестирование). Следуя по стопам Twitter, мы будем отображать микросообщения не на отдельной странице с их списком, а непосредственно на странице профиля пользователя, как показано на рис. 11.4. Начнем с довольно простых ERb-шаблонов добавления микросообщений в профиль пользователя, а затем добавим сами микросообщения в набор тестовых данных из раздела 9.3.2, чтобы у нас появился материал для отображения.

11.2.1. Отображение микросообщений

Наша цель – показать микросообщения каждого пользователя на странице его профиля (`show.html.erb`) вместе с их общим количеством. Как вскоре станет видно, многие из этих идей имеют много схожего с идеями, лежащими в основе отображения списка пользователей (раздел 9.3).

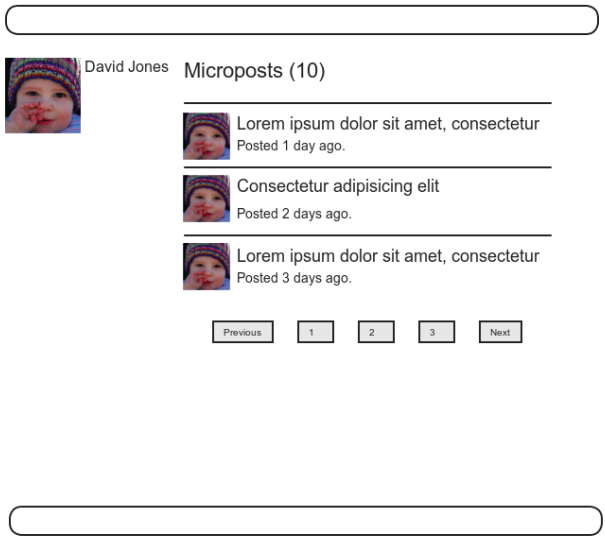


Рис. 11.4 ❖ Макет страницы профиля с микросообщениями

Контроллер Microposts не понадобится нам вплоть до раздела 11.3, но каталог с представлениями нужен уже прямо сейчас, поэтому давайте сгенерируем контроллер:

```
$ rails generate controller Microposts
```

Основной целью этого раздела является отображение всех микросообщений для каждого пользователя. Мы видели в разделе 9.3.5, что код

```
<ul class="users">
  <%= render @users %>
</ul>
```

автоматически отображает информацию о пользователе из переменной `@users` с применением частичного шаблона `_user.html.erb`. Теперь мы определим аналогичный шаблон `_micropost.html.erb` для вывода коллекции микросообщений, использовав тот же прием:

```
<ol class="microposts">
  <%= render @microposts %>
</ol>
```

Обратите внимание: здесь используется тег *упорядоченного списка* `ol` (в противоположность неупорядоченному списку `ul`), потому что микросообщения выводятся в определенном порядке (обратном хронологическому). Соответствующий шаблон представлен в листинге 11.21.

Листинг 11.21 ❖ Частичный шаблон для отображения одного микросообщения (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>
```

Здесь используется интересный вспомогательный метод `time_ago_in_words`, смысл которого вполне очевиден, результат его работы мы увидим в разделе 11.2.2. Кроме того, мы добавили CSS-атрибут `id` в каждое микросообщение:

```
<li id="micropost-<%= micropost.id %>">
```

Это хорошая практика, так как дает возможность манипулировать отдельными микросообщениями в будущем (применив JavaScript, например).

Следующий шаг – решить проблему с отображением потенциально большого количества микросообщений. Мы поступим так же, как со списком пользователей в разделе 9.3.3, а именно – воспользуемся постраничным выводом. Как и раньше, это будет метод `will_paginate`:

```
<%= will_paginate @microposts %>
```

Если сравнить этот код с аналогичной строкой, реализующей постраничный вывод списка пользователей в листинге 9.41, можно увидеть, что раньше нам было достаточно

```
<%= will_paginate %>
```

потому что в случае с контроллером Users метод `will_paginate` предполагает наличие переменной экземпляра `@users` (которая, как мы видели в разделе 9.3.3, должна принадлежать к классу `ActiveRecord::Relation`). В данном случае, поскольку мы все еще в контроллере Users, но нам нужно обеспечить постраничный вывод *микросообщений*, а не пользователей, мы явно передаем переменную `@microposts` в `will_paginate`. Конечно, это означает, что нам придется определить такую переменную в методе `show` пользователя (листинг 11.22).

Листинг 11.22 ❖ Добавление переменной экземпляра @microposts в метод show (app/controllers/users_controller.rb)

```
class UsersController < ApplicationController
  .
  .
  .
  def show
    @user = User.find(params[:id])
    @microposts = @user.microposts.paginate(page: params[:page])
  end
```

```
  .
  .
  .
end
```

Обратите внимание, насколько умен метод `paginate` – он работает даже *через* связь микросообщений с пользователем, добираясь таким образом до таблицы `microposts` и извлекая необходимую страничку микросообщений.

Последняя наша задача – показать количество микросообщений для каждого пользователя. Это можно сделать с помощью метода `count`:

```
user.microposts.count
```

Как и `paginate`, метод `count` поддерживает ассоциации. В частности, `count` *не* извлекает все микросообщения из базы данных, чтобы затем получить размер полученного массива, так как эффективность такого подхода будет падать по мере роста количества микросообщений. Вместо этого он выполняет подсчет прямо в базе данных, запрашивая у нее количество микросообщений с заданным значением атрибута `user_id`. (В маловероятном случае, когда подсчет количества все же окажется узким местом в приложении, можно сделать его еще быстрее с помощью *counter cache* (http://railscasts.com/episodes/23-counter-cache-column).)

Собрав все части воедино, можно добавить микросообщения на страничку профиля (листинг 11.23). Обратите внимание на конструкцию `if @user.microposts.any?` (мы видели ее раньше в листинге 7.19), которая гарантирует, что пустой список не будет отображаться, если у пользователя нет сообщений.

Листинг 11.23 ❖ Добавление микросообщений на страницу профиля пользователя (app/views/users/show.html.erb)

```
<% provide(:title, @user.name) %>
<div class="row">
  <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>
```

Теперь можно взглянуть на обновленную страницу профиля пользователя (рис. 11.5). Она выглядит весьма... бледно. Конечно, ведь там пока нет ни одного микросообщения. Пришло время изменить эту ситуацию.

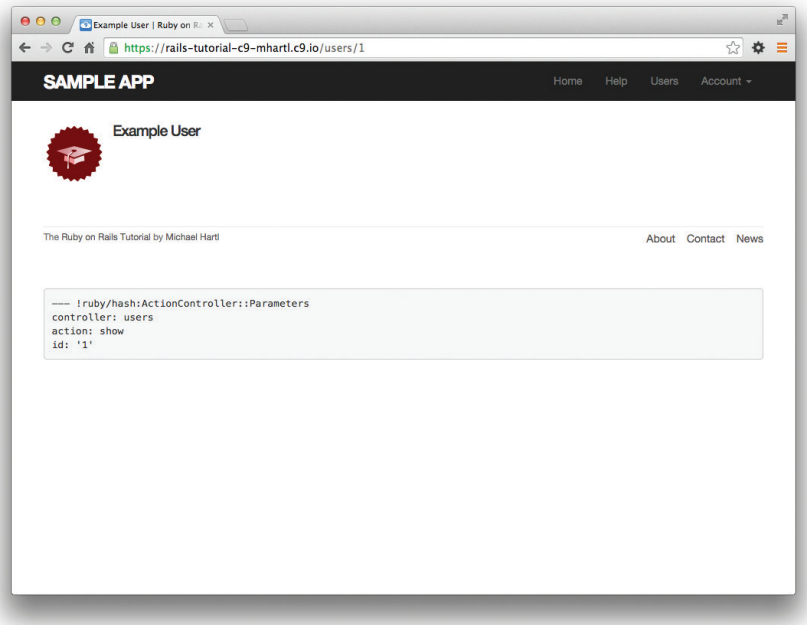


Рис. 11.5 ❖ Страница профиля пользователя с кодом для отображения микросообщений, но без микросообщений

11.2.2. Образцы микросообщений

Несмотря на всю работу по созданию шаблонов для микросообщений, проделанную в разделе 11.2.1, окончание было весьма разочаровывающим. Мы можем исправить ситуацию, добавив микросообщения в тестовые данные, которые были определены в разделе 9.3.2.

Добавление образцов микросообщений для *всех* пользователей займет довольно много времени, поэтому сначала выберем только первые шесть из них (то есть пять пользователей с собственными аватарами и один с аватаром по умолчанию) с помощью метода `take`:

```
User.order(:created_at).take(6)
```

Вызов `order` вернет именно тех шестерых пользователей, которых мы создали первыми.

Создадим по 50 микросообщений для каждого выбранного пользователя (достаточно для переполнения лимита механизма постраничного просмотра, равного 30). Чтобы сгенерировать содержимое для каждого из этих микросообщений,

воспользуемся удобным методом `Lorem.sentence` (http://www.rubydoc.info/gems/faker/1.3.0/Faker/Lorem) из гема Faker[1]. Получившийся результат показан в листинге 11.24. (Причина именно такой последовательности циклов в том, чтобы перемешать микросообщения для использования в ленте (раздел 12.3). Если сначала перебрать пользователей в цикле, мы получим ленту с большим количеством сообщений от одного пользователя, что визуально выглядит не очень привлекательно.)

Листинг 11.24 ❖ Добавление микросообщений в тестовые данные (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
```

Теперь мы можем повторно заполнить базу данных в окружении разработки:

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

Также необходимо перезапустить Rails-сервер.

Теперь можно насладиться плодами наших трудов, приложенных в разделе 11.2.1, и увидеть наконец все микросообщения[2]. Предварительные результаты показаны на рис. 11.6.

Микросообщения на этой странице пока не имеют специального оформления, поэтому добавим его (листинг 11.25) и посмотрим на результат[3].

Листинг 11.25 ❖ Стили CSS для микросообщений, а также весь код CSS для этой главы (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* микросообщения */

.microposts {
  list-style: none;
  padding: 0;
  li {
```

[1] `Faker::Lorem.sentence` возвращает текст *lorem ipsum*; о котором рассказывалось в главе 6. Этот текст имеет увлекательную предысторию.

[2] Гем Faker генерирует случайный текст **loremipsum**, поэтому содержимое микросообщений будет отличаться.

[3] Для удобства в листинге 11.25 на самом деле находится весь код CSS, необходимый в этой главе.

```
    padding: 10px 0;
    border-top: 1px solid #e8e8e8;
  }
  .user {
    margin-top: 5em;
    padding-top: 0;
  }
  .content {
    display: block;
    margin-left: 60px;
    img {
      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;
  }
}
```

На рис. 11.7 показана страница профиля первого пользователя, а на рис. 11.8 – второго. Наконец, на рис. 11.9 представлена вторая страница микросообщений первого пользователя, а также ссылки на страницы внизу. Во всех трех случаях видно, что для каждого микросообщения отображается время его создания (например, «Posted 1 minute ago» (Отправлено 1 минуту назад)); именно так работает метод `time_ago_in_words` из листинга 11.21. Если вы подождете пару минут и перезагрузите страницу, то увидите, как текст автоматически обновится в соответствии с новым временем.

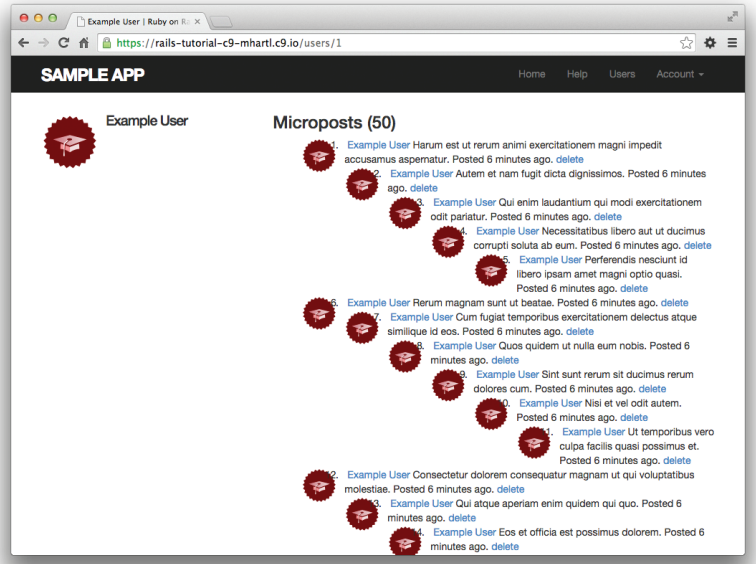


Рис. 11.6 ❖ Профиль пользователя с микросообщениями без применения стилей

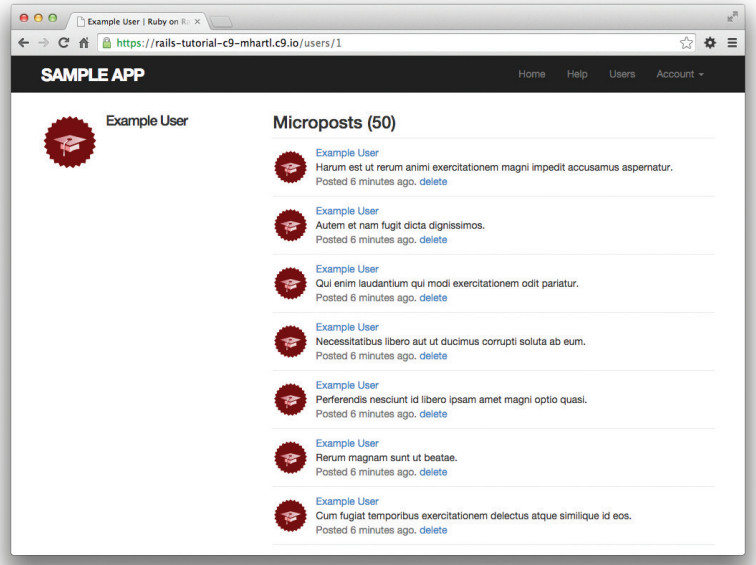


Рис. 11.7 ❖ Страница профиля пользователя с микросообщениями (/users/1)

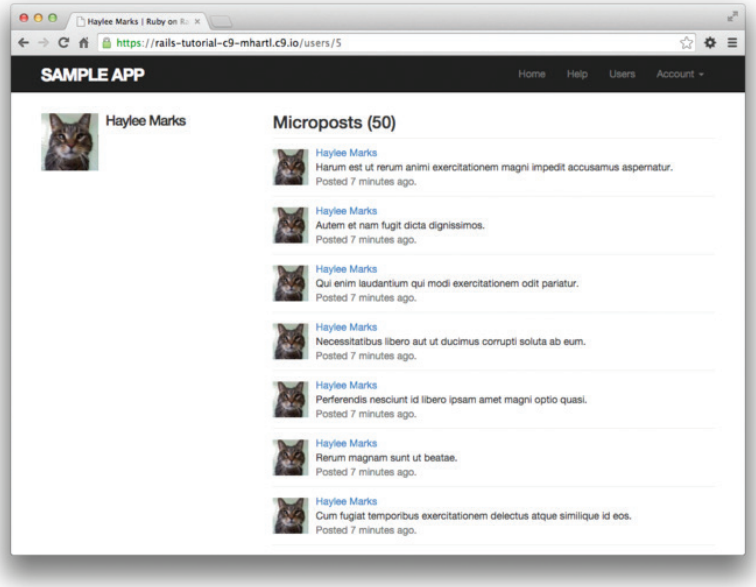


Рис. 11.8 ❖ Профиль второго пользователя, также с микросообщениями (/users/5)

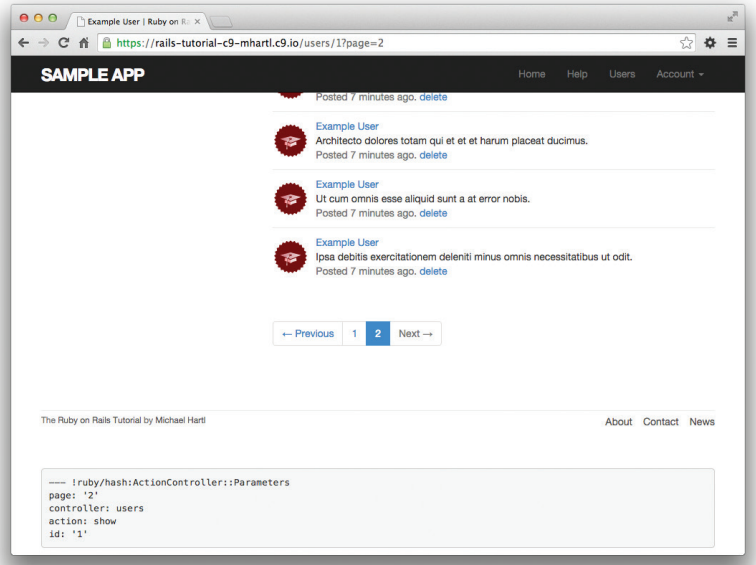


Рис. 11.9 ❖ Ссылки на страницы в списке микросообщений (/users/1?page=2)

11.2.3. Тесты профиля с микросообщениями

Так как после активации пользователи перенаправляются на страницу своего профиля, у нас уже есть тесты отображения этой страницы (листинг 10.31). В этом разделе мы напишем короткий интеграционный тест для еще нескольких элементов страницы профиля, исходя из проделанной работы. Начнем с создания интеграционного теста профилей наших пользователей:

```
$ rails generate integration_test users_profile
      invoke  test_unit
      create    test/integration/users_profile_test.rb
```

Чтобы протестировать правильность отображения микросообщений, нужно связать тестовые микросообщения с пользователем. В Rails для этого имеется очень удобный способ:

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

Добавив параметр `user: michael`, мы сообщаем фреймворку Rails, что это микросообщение нужно связать с соответствующим пользователем в тестовых данных:

```
michael:
  name: Michael Example
  email: michael@example.com
  .
  .
  .
```

Для проверки постраничного вывода микросообщений создадим дополнительные микросообщения, используя тот же прием, которой применялся для создания дополнительных пользователей в листинге 9.43:

```
<% 30.times do |n| %>
micropost_<%= n %>:
  content: <%= Faker::Lorem.sentence(5) %>
  created_at: <%= 42.days.ago %>
  user: michael
<% end %>
```

Собрав все вместе, получим обновленные тестовые данные с микросообщениями (листинг 11.26).

Листинг 11.26 ❖ Тестовые микросообщения с привязкой к пользователям (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 %>
```

При наличии готовых тестовых данных само тестирование выглядит довольно просто: открываем страницу профиля и проверяем ее заголовок, имя пользователя, аватар, количество микросообщений и ссылки механизма постраничного вывода. Результат представлен в листинге 11.27. Обратите внимание на вспомогательный метод `full_title` из листинга 4.2, проверяющий заголовок страницы, доступ к которому мы получаем подключением модуля Application Helper[1].

Листинг 11.27 ❖ Тест профиля пользователя **ЗЕЛЕНЫЙ** (test/integration/users_profile_test.rb)

```
require 'test_helper'

class UsersProfileTest < 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'
```

[1] Если у вас появится желание провести рефакторинг остальных тестов и задействовать в них метод `full_title` (как в листинге 3.38), подключите модуль ApplicationHelper в файле `test_helper.rb`.

```
    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
```

В утверждении, проверяющем количество микросообщений, используется метод `response.body`, который мы уже видели в упражнениях (раздел 10.5) к главе 10. Несмотря на его название, `response.body` содержит весь исходный HTML страницы (а не только ее тело). То есть если вас заботит только присутствие количества микросообщений *где-то* на странице, проверить это можно так:

```
assert_match @user.microposts.count.to_s, response.body
```

Это менее специфичное утверждение, чем `assert_select`; в частности в отличие от `assert_select`, в данном случае утверждение `assert_match`, не требует указывать конкретный HTML-тег.

Кроме того, здесь демонстрируется вложенный синтаксис для `assert_select`:

```
assert_select 'h1>img.gravatar'
```

Это утверждение проверяет наличие тега `img` с классом `gravatar` *внутри* тега заголовка верхнего уровня (`h1`).

Так как код приложения работает безупречно, набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 11.28 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

11.3. Манипулирование микросообщениями

Мы закончили работу над моделью данных и шаблонами отображения микросообщений, теперь обратим наше внимание на веб-интерфейс для их создания. В этом разделе мы также увидим первый намек на *поток* (ленту) сообщений – полностью эта идея будет реализована в главе 12. Наконец, так же как с пользователями, мы сделаем возможным удаление микросообщений через веб-интерфейс.

Существует один разрыв с предыдущими соглашениями, который стоит отметить: интерфейс ресурса Microposts работает в основном за счет главной страницы и страницы профиля, поэтому нам не понадобятся методы `new` и `edit` в контроллере Microposts – только `create` и `destroy`. В результате получаем список маршрутов для ресурса Microposts, представленный в листинге 11.29, и список маршрутов RESTful в табл. 11.2, который является сокращенным вариантом полного набора маршрутов из табл. 2.3. Конечно, эта простота – показатель *большей* сложности, а не меньшей; мы прошли долгий путь от опоры на механизм скаффолдинга в главе 2 и больше не нуждаемся в его сложностях.

Листинг 11.29 ❖ Маршруты ресурса 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
```

Таблица 11.2 ❖ Маршруты RESTful, поддерживаемые ресурсом Microposts в листинге 11.29

HTTP-запрос	URL	Действие	Предназначение
POST	/microposts	create	Создает новое микросообщение
DELETE	/microposts/1	destroy	Удаляет микросообщение с идентификатором 1

11.3.1. Управление доступом к микросообщениям

Начнем разработку ресурса Microposts с управления доступом на уровне контроллера Microposts. В частности, так как доступ к микросообщениям осуществляется через связанных с ними пользователей, методы `create` и `destroy` должны быть доступны только вошедшим пользователям.

Тесты, проверяющие факт входа, повторяют аналогичные тесты для контроллера Users (листинги 9.17 и 9.56). Мы просто отправляем запрос в каждый метод и убеждаемся, что количество микросообщений не изменилось и произошла переадресация на страницу входа (листинг 11.30).

Листинг 11.30 ❖ Тесты авторизации для контроллера Microposts **КРАСНЫЙ** (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
```

Прежде чем писать прикладной код, который обеспечит успешное выполнение этих тестов, проведем небольшой рефакторинг. Как вы помните, в разделе 9.2.1 требование авторизации обеспечивает предварительный фильтр с методом `logged_in_user` (листинг 9.12). Тогда он был нужен только в контроллере Users, но теперь он понадобился и в контроллере Microposts, поэтому переместим его в контроллер Application – базовый класс для всех контроллеров (раздел 4.4.4). Результат показан в листинге 11.31.

Листинг 11.31 ❖ Перемещение метода `logged_in_user` в контроллер Application (app/controllers/application_controller.rb)

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper

  private

    # Подтверждает вход пользователя.
    def logged_in_user
      unless logged_in?
        store_location
        flash[:danger] = "Please log in."
        redirect_to login_url
      end
    end
end
```

Чтобы избежать повторения кода, удалим `logged_in_user` из контроллера Users.

Теперь метод `logged_in_user` доступен в контроллере Microposts, а значит, мы можем добавить методы `create` и `destroy` и ограничить доступ к ним через предварительный фильтр, как показано в листинге 11.32.

Листинг 11.32 ❖ Добавление требования авторизации в методы контроллера Microposts **ЗЕЛЕНЫЙ** (app/controllers/microposts_controller.rb)

```
class MicropostsController < ApplicationController
  before_action :logged_in_user, only: [:create, :destroy]

  def create
  end

  def destroy
  end
end
```

Теперь тесты должны выполняться успешно:

Листинг 11.33 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

11.3.2. Создание микросообщений

В главе 7 мы реализовали регистрацию пользователя, создав форму HTML, которая посылала HTTP-запрос `POST` в метод `create` контроллера Users. Создание микросообщения реализуется аналогично, с той лишь разницей, что форма будет находиться на главной, а не на отдельной странице с адресом /microposts/new, как показано на рис. 11.10.

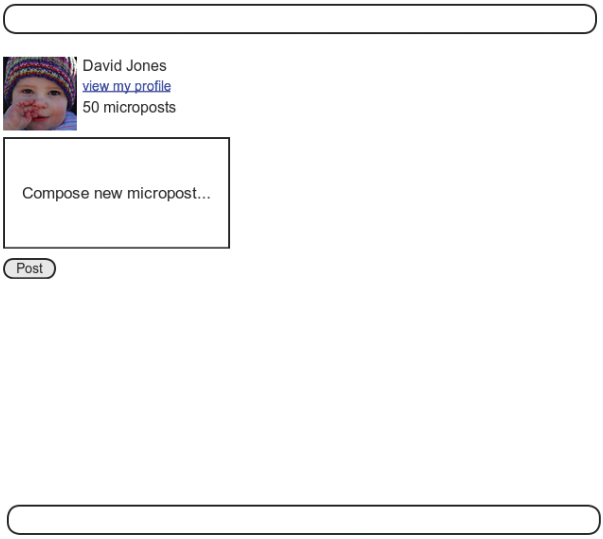


Рис. 11.10 ❖ Макет главной страницы с формой для создания микросообщений

Когда мы в последний раз видели главную страницу, она выглядела, как на рис. 5.6, то есть на ней была кнопка **Sign up now!** посередине. Так как форма для создания микросообщения имеет смысл только при наличии авторизованного пользователя, одной из целей данного раздела будет формирование разных версий главной страницы, в зависимости от статуса посетителя. Мы осуществим это в листинге 11.35.

Начнем с метода `create` для микросообщений, который очень похож на свой аналог для пользователей (листинг 7.23); принципиальное отличие заключается в использовании связи пользователь/микросообщение для создания нового микросообщения (листинг 11.34). Обратите внимание на использование строгих параметров в методе `micropost_params`, разрешающих редактировать через веб только атрибут `content` микросообщения.

Листинг 11.34 ❖ Метод create контроллера 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
```

Для реализации формы создания микросообщений воспользуемся кодом из листинга 11.35, который возвращает разную разметку HTML, в зависимости от статуса пользователя.

Листинг 11.35 ❖ Добавление формы для создания микросообщений на главную страницу (/) (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 %>
```

(Такое большой объем кода в каждой ветке условного оператора `if-else` придает представлению весьма неряшливый вид; мы приведем его в порядок с помощью частичных шаблонов в разделе 11.6 с упражнениями.)

Чтобы эта страница заработала, нужно создать и заполнить пару частичных шаблонов. Первый – новая боковая панель (листинг 11.36).

Листинг 11.36 ❖ Шаблон боковой панели с информацией о пользователе (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>
```

Так же как в боковой панели на странице профиля (листинг 11.23), здесь отображается общее количество микросообщений. Хотя есть небольшое отличие: текст «Microposts» – это метка, поэтому написание «Microposts (1)» («Сообщений (1)») вполне допустимо. Но выражение «1 microposts» («1 сообщений») является безграмотным, поэтому мы выводим «1 micropost» («1 сообщение»), но «2 microposts» («2 сообщения»), задействовав метод `pluralize`, который мы видели в разделе 7.3.3.

Далее определим форму для создания микросообщений (листинг 11.37), аналогичную форме регистрации из листинга 7.13.

Листинг 11.37 ❖ Частичный шаблон формы для создания микросообщений (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 %>
```

Нам понадобится внести два изменения, прежде чем эта форма заработает. Во-первых, определим переменную экземпляра `@micropost` через ассоциацию:

```
@micropost = current_user.microposts.build
```

Результат показан в листинге 11.38.

Листинг 11.38 ❖ Добавление переменной экземпляра @micropost в метод 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

  def contact
  end
end
```

Конечно, current_user существует, только когда пользователь авторизовался в системе, поэтому переменная @micropost должна определяться лишь в этом случае.

Во-вторых, необходимо изменить шаблон сообщения об ошибках, чтобы заработал код из листинга 11.37:

```
<%= render 'shared/error_messages', object: f.object %>
```

Возможно, вы помните, что шаблон сообщения об ошибке (листинг 7.18) явно ссылается на переменную @user, но в данном случае ее роль играет переменная @micropost. Чтобы унифицировать эти случаи, можно передать в шаблон переменную формы f и получить доступ к соответствующему объекту через f.object, поэтому в

```
form_for(@user) do |f|
```

f.object – это @user, а в

```
form_for(@micropost) do |f|
```

f.object – это @micropost.

Для передачи объекта в шаблон используется хэш с объектом в виде значения и именем переменной в шаблоне в виде ключа. Именно этим занимается вторая строка в листинге 11.37. Другими словами, object: f.object создает переменную object в шаблоне error_messages, и мы можем воспользоваться ею для создания настраиваемого сообщения об ошибке, как показано в листинге 11.39.

Листинг 11.39 ❖ Сообщение об ошибке для работы с любыми объектами **КРАСНЫЙ** (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 %>
```

В настоящий момент набор тестов должен быть **КРАСНЫМ**:

Листинг 11.40 ❖ КРАСНЫЙ

```
$ bundle exec rake test
```

Это намек на необходимость обновить ссылки на шаблон сообщения об ошибках в других местах, а именно в форме регистрации пользователей (листинг 7.18), сброса паролей (листинг 10.50) и редактирования профилей пользователей (листинг 9.2). Обновленные версии представлены в листингах 11.41, 11.43 и 11.42.

Листинг 11.41 ❖ Отображение сообщения об ошибках при регистрации пользователя (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>
```

Листинг 11.42 ❖ Отображение сообщения об ошибках при редактировании пользователя (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>
```

Листинг 11.43 ❖ Отображение сообщения об ошибках при сбросе пароля (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>
```

Теперь набор тестов должен быть **ЗЕЛЕНЫМ**:

```
$ bundle exec rake test
```

Кроме того, вся разметка HTML в этом разделе должна отображаться верно. Пустая форма должна выглядеть, как на рис. 11.11, а форма с сообщением об ошибке – как на рис. 11.12.

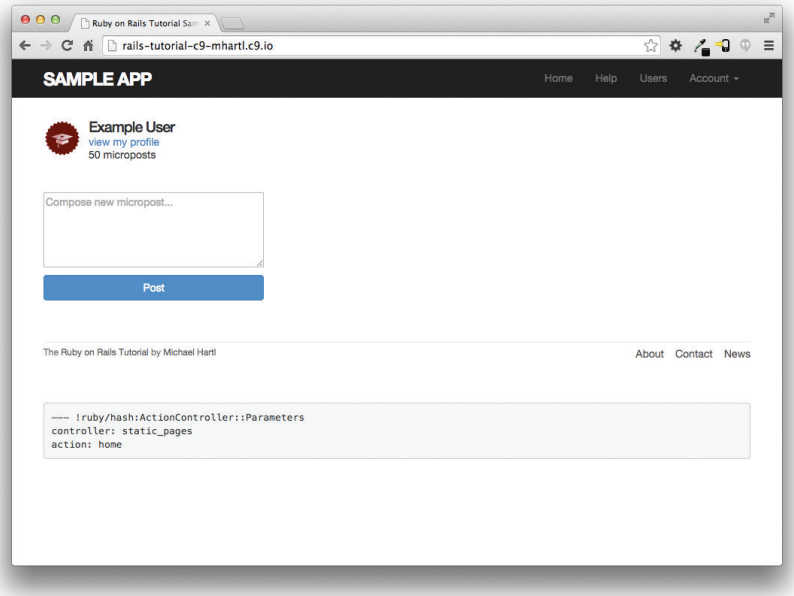


Рис. 11.11 ❖ Главная страница
с формой для ввода нового микросообщения

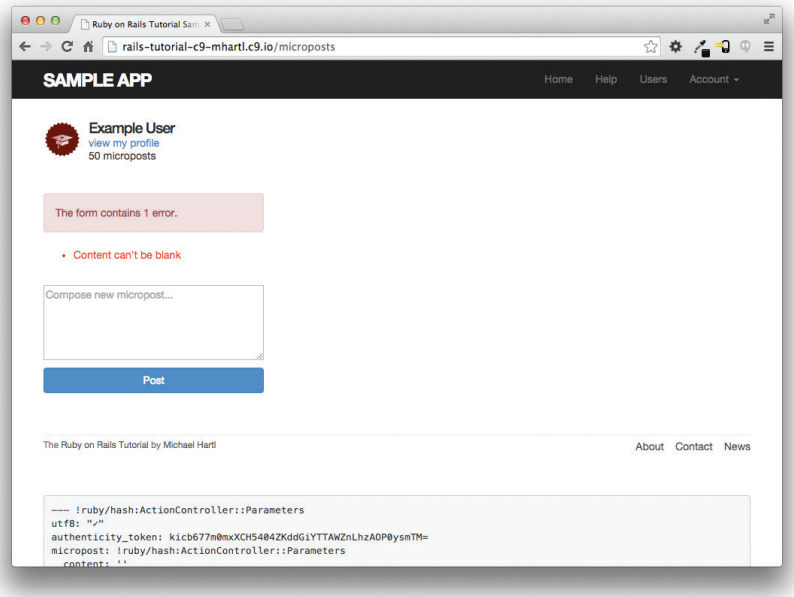


Рис. 11.12 ❖ Главная страница с сообщением об ошибке

11.3.3. Прото-лента сообщений

Форма создания микросообщений сейчас действительно работает, однако пользователи не могут немедленно видеть результаты ее успешной отправки, так как текущий вариант главной страницы не выводит никаких микросообщений. Если хотите, можете проверить – введите в форму сообщение, отправьте ее, а затем перейдите на страницу профиля, чтобы увидеть сообщение, но это довольно долго и неудобно. Было бы лучше иметь *ленту* с собственными сообщениями пользователя, как показано на рис. 11.13. (В главе 12 мы сделаем ее более глобальной, добавив микросообщения тех пользователей, за которыми следует текущий пользователь.)

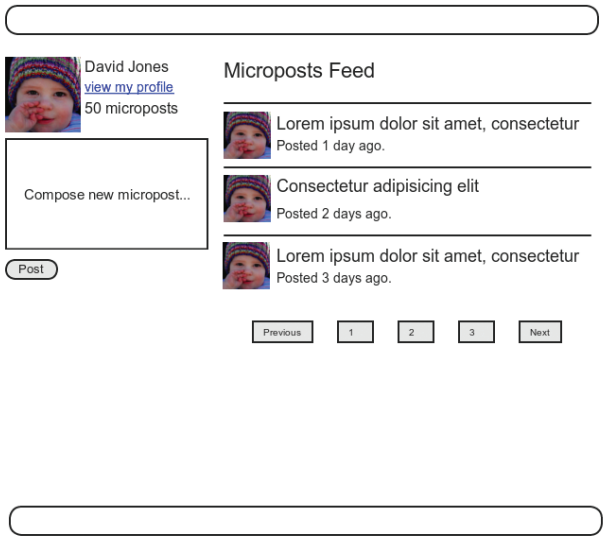


Рис. 11.13 ❖ Макет главной страницы с прото-лентой

Так как лента сообщений должна быть у каждого пользователя, мы естественным образом подошли к методу `feed` в модели User, который пока будет просто выбирать все микросообщения, принадлежащие текущему пользователю. Мы реализуем его с помощью метода `where` модели Micropost (раньше он встречался в разделе 10.5), как показано в листинге 11.44[1].

Листинг 11.44 ❖ Предварительная реализация ленты микросообщений (app/models/user.rb)

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

[1] Более подробную информацию вы найдете в разделе «Active Record Query Interface» руководства «RailsGuide» (http://guides.rubyonrails.org/active_record_querying.html).

```
  .
  .
  # Определяет прото-ленту.
  # Полная реализация приводится в разделе "Следование за пользователями".
  def feed
    Micropost.where("user_id = ?", id)
  end

    private
    .
    .
    .
end
```

Знак вопроса в инструкции

```
Micropost.where("user_id = ?", id)
```

гарантирует корректное *экранирование* значения `id` перед включением в SQL-запрос; это позволит избежать серьезной дыры в системе безопасности, называемой *инъекцией SQL-кода*. Атрибут `id` в данном случае – просто целое число (то есть `self.id` – уникальный числовой идентификатор пользователя), поэтому в данном случае нет никакой опасности, но обязательное экранирование переменных, вводимых в SQL-выражения, – это хорошая привычка, которую стоит развивать.

Внимательные читатели могли заметить, что код в листинге 11.44, по сути, эквивалентен следующему определению:

```
def feed
  microposts
end
```

Однако код в листинге 11.44 более естественно превращается в полный поток сообщений, как будет показано в главе 12.

Чтобы задействовать ленту сообщений, добавим переменную экземпляра `@feed_items` для хранения ленты текущего пользователя (с постраничным выводом, листинг 11.45), а затем – шаблон ленты (листинг 11.46) в главную страницу (листинг 11.47). Обратите внимание, что теперь, когда пользователь войдет на сайт, должны выполниться две строки, поэтому инструкцию в листинге 11.38

```
@micropost = current_user.microposts.build if logged_in?
```

в листинге 11.45 заменила конструкция:

```
if logged_in?
  @micropost  = current_user.microposts.build
  @feed_items = current_user.feed.paginate(page: params[:page])
end
```

полученная преобразованием оператора `if` в конце выражения в инструкцию `if-end`.

Листинг 11.45 ❖ Добавление переменной экземпляра @feed_items в метод 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
```

Листинг 11.46 ❖ Шаблон ленты сообщений (app/views/shared/_feed.html.erb)

```
<% if @feed_items.any? %>
  <ol class="microposts">
    <%= render @feed_items %>
  </ol>
  <%= will_paginate @feed_items %>
<% end %>
```

Этот шаблон перекладывает задачу отображения на шаблон микросообщений в листинге 11.21:

```
<%= render @feed_items %>
```

Rails знает, что нужно вызвать шаблон микросообщений, так как каждый элемент `@feed_items` принадлежит к классу `Micropost`. Это заставляет Rails искать шаблон с соответствующим названием в каталоге представлений данного ресурса:

```
app/views/microposts/_micropost.html.erb
```

Чтобы добавить ленту сообщений в главную страницу, достаточно включить в нее шаблон, как обычно (листинг 11.47). Результат показан на рис. 11.14.

Листинг 11.47 ❖ Добавление ленты сообщений в главную страницу (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">
      <h3>Micropost Feed</h3>
      <%= render 'shared/feed' %>
    </div>
  </div>
<% else %>
  .
  .
  .
<% end %>
```

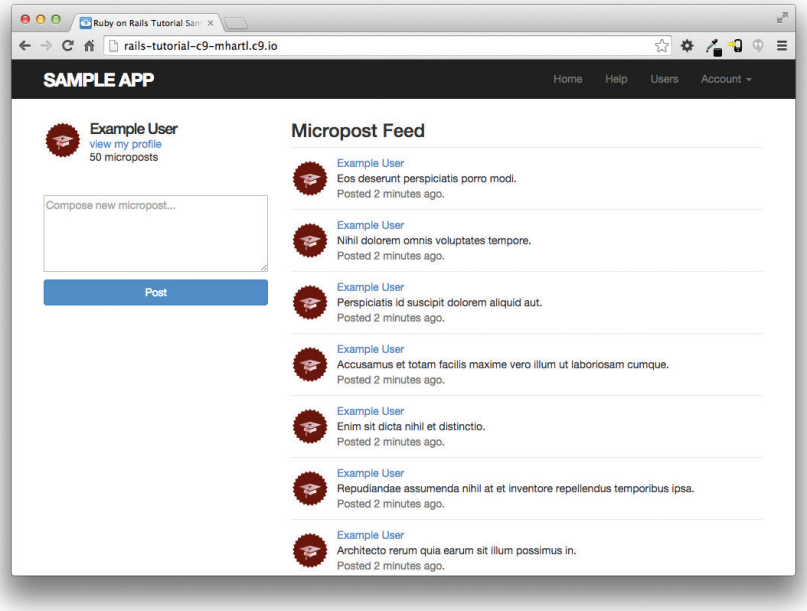


Рис. 11.14 ❖ Главная страница с прото-лентой

Теперь создание новых микросообщений работает как надо (рис. 11.15). Однако есть одна тонкость: при *неудачной* отправке микросообщения главная страница ожидает получить переменную экземпляра `@feed_items`, поэтому неудачная попытка в данный момент обрабатывается неверно. Проще всего исправить проблему можно полным подавлением ленты сообщений путем присваивания пустого массива, как показано в листинге 11.48. (Увы, в данном случае механизм постраничного вывода работать не будет. Попробуйте исправить проблему, как пред-

ложено, и щелкните на ссылке перехода к другой странице, чтобы понять, почему так происходит.)

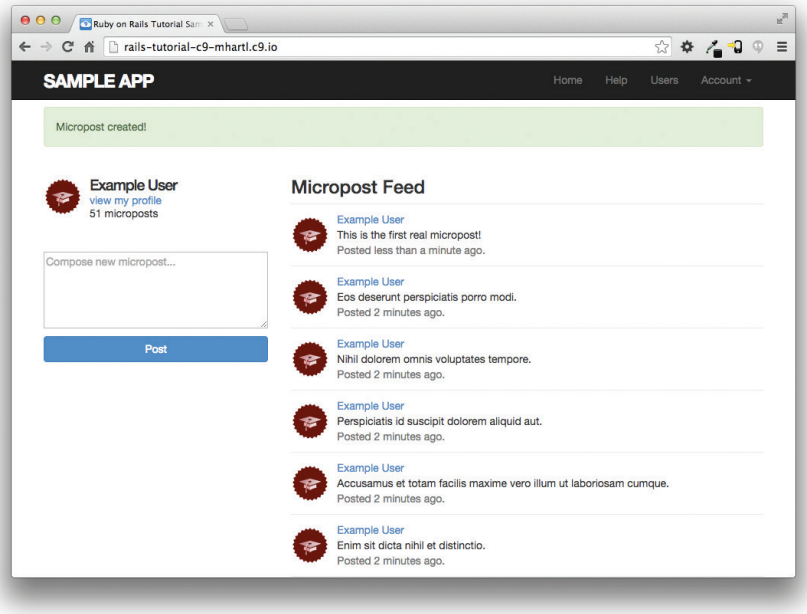


Рис. 11.15 ❖ Главная страница после создания нового микросообщения

Листинг 11.48 ❖ Добавление (пустой) переменной экземпляра *@feed_items* в метод 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. Удаление микросообщений

Последнее, что осталось добавить в ресурс Microposts, – возможность удаления сообщений. Так же как в случае с удалением учетных записей (раздел 9.4.2), мы реализуем это добавлением ссылок «delete» (удалить), как показано на рис. 11.16. Но, в отличие от учетных записей, удаление которых было доступно только администраторам, микросообщения могут удаляться лишь их авторами.

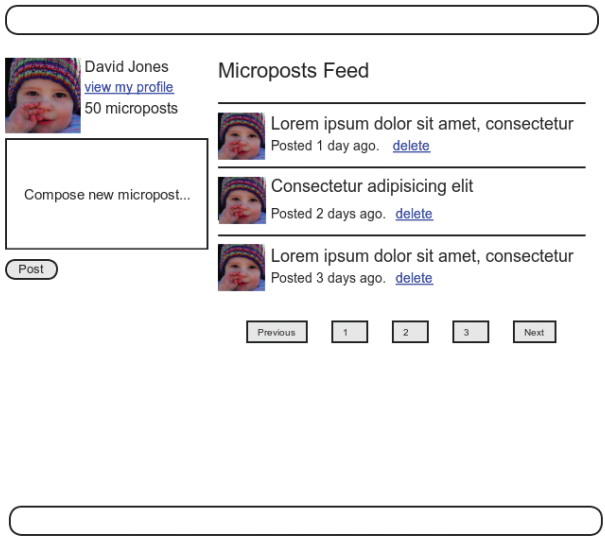


Рис. 11.16 ❖ Макет прото-ленты со ссылками для удаления микросообщений

Для начала добавим ссылку для удаления в шаблон микросообщений (листинг 11.49).

Листинг 11.49 ❖ Добавление ссылки для удаления в шаблон микросообщений (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>
```

Теперь определим метод `destroy` в контроллере Microposts, аналогичный одноименному методу для пользователей в листинге 9.54. Основное отличие – не в использовании переменной `@user` и предварительного фильтра `admin_user`, а в поиске микросообщений через их связь, что автоматически не позволит пользователю удалить чужое микросообщение. Поместим получившийся поиск внутрь предварительного фильтра `correct_user`, проверяющего наличие у текущего пользователя микросообщения с указанным `id`. Результат представлен в листинге 11.50.

Листинг 11.50 ❖ Метод *destroy* в контроллере 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
```

Метод `destroy` производит переадресацию на URL:

```
request.referrer || root_url
```

Здесь используется метод `request.referrer`[1], тесно связанный с переменной `request.url`, которая используется в дружелюбной переадресации (раздел 9.2.3)

[1] Соответствует отправке заголовка `HTTP_REFERER`, как определено в документации к HTTP. Обратите внимание: в названии заголовка `HTTP_REFERER` нет опечатки, слово `REFERER` действительно пишется с ошибкой. Rails исправляет эту ошибку, используя написание имени метода `referrer`.

и означает всего лишь предыдущий URL (в данном случае URL главной страницы)[1]. Это удобно, так как микросообщения есть и на главной странице, и на странице профиля пользователя, поэтому за счет `request.referrer` мы организуем переадресацию назад, на страницу, откуда был отправлен запрос на удаление. На тот случай, если ссылающийся URL имеет значение `nil` (как, например, внутри тестов), в листинге 11.50 используется вариант по умолчанию `root_url`, реализованный при помощи оператора `||`. (Сравните с параметрами по умолчанию, определенными в листинге 8.50.)

Результат удаления предпоследнего сообщения показан на рис. 11.17.

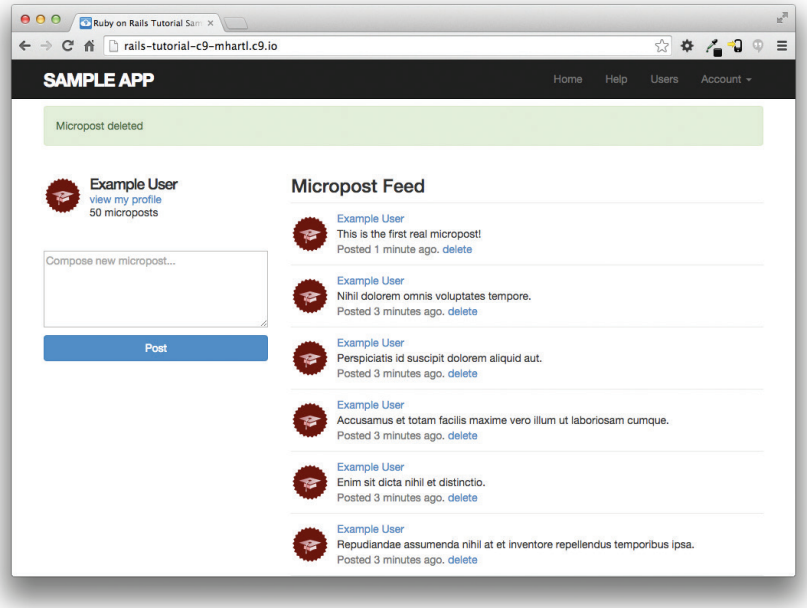


Рис. 11.17 ❖ Главная страница после удаления предпоследнего микросообщения

11.3.5. Тесты микросообщений

После реализации удаления микросообщений в разделе 11.3.4 модель Micropost и интерфейс к ней можно считать завершенными. Нам осталось лишь написать короткий тест контроллера Microposts для проверки авторизации и интеграционный тест микросообщений, чтобы связать все вместе.

---

[1] Я не смог сразу вспомнить, как получить этот URL внутри Rails-приложения, поэтому я погуглил по фразе «rails request previous» (запрос предыдущего URL) и нашел вопрос на Stack Overflow с ответом (http://stackoverflow.com/questions/4652084/ruby-on-rails-how-do-you-get-the-previous-url).

Начнем с добавления нескольких микросообщений от разных пользователей в тестовые данные, как показано в листинге 11.51. (Сейчас нам достаточно одного сообщения, но мы добавим сразу несколько, на будущее.)

Листинг 11.51 ❖ Добавление микросообщений от разных пользователей (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
```

Теперь напишем короткий тест, чтобы убедиться в том, что пользователи не могут удалять чужие сообщения, и заодно проверим корректность переадресации (листинг 11.52).

Листинг 11.52 ❖ Тестирование удаления микросообщений несоответствующим пользователем **ЗЕЛЕНЫЙ** (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
```

Наконец, напишем интеграционный тест. Этот тест будет осуществлять вход, проверять постраничный вывод микросообщений, отправлять сначала недопустимую, а затем допустимую информацию, удалять сообщение, затем открывать страницу другого пользователя и проверять отсутствие в ней ссылок для удаления. Начнем, как обычно, с создания файла тестов:

```
$ rails generate integration_test microposts_interface
      invoke  test_unit
      create    test/integration/microposts_interface_test.rb
```

Исходный код теста приводится в листинге 11.53. Сможете ли вы самостоятельно определить в нем этапы, перечисленные выше? (Здесь, в преддверии тестов загрузки изображения в упражнениях (листинг 11.69), используются методы `post` и `follow_redirect!` вместо равнозначного `post_via_redirect`.)

Листинг 11.53 ❖ Интеграционный тест интерфейса микросообщений **ЗЕЛЕНЫЙ** (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'
    # Недопустимая информация в форме.
    assert_no_difference 'Micropost.count' do
      post microposts_path, micropost: { content: "" }
    end
    assert_select 'div#error_explanation'
    # Допустимая информация в форме.
```

```
    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
    # Удаление сообщения.
    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
    # Переход в профиль другого пользователя.
    get  user_path(users(:archer))
    assert_select 'a', text: 'delete', count: 0
  end
end
```

Так как сначала мы написали работающий прикладной код, набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 11.54 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

11.4. Изображения в микросообщениях

Теперь, когда у нас реализованы все необходимые операции с микросообщениями, в этом разделе мы реализуем добавление изображений в микросообщения. Начнем с простой версии, достаточной для окружения разработки, а затем дополним ее рядом усовершенствований, чтобы сделать возможной выгрузку изображений в эксплуатационном окружении.

Это нововведение содержит два основных видимых элемента: поле формы для загрузки изображений и собственно изображения в микросообщениях. Макет страницы с кнопкой **Upload image** (Выгрузить картинку) и с картинками в микросообщениях показан на рис. 11.18[1].

11.4.1. Выгрузка изображений

Чтобы обработать выгруженное изображение и связать его с моделью Micropost, воспользуемся инструментом выгрузки картинок CarrierWave (https://github.com/carrierwaveuploader/carrierwave). Для начала добавим гем `carrierwave` в `Gemfile` (листинг 11.55). Для полноты в этот листинг также добавлены гемы `mini_magick` и `fog`, необходимые для изменения размеров изображений (раздел 11.4.3) и выгрузки изображений в эксплуатационном окружении (раздел 11.4.4).

[1] Фото пляжа взято по адресу: https://www.flickr.com/photos/grungepunk/14026922186.

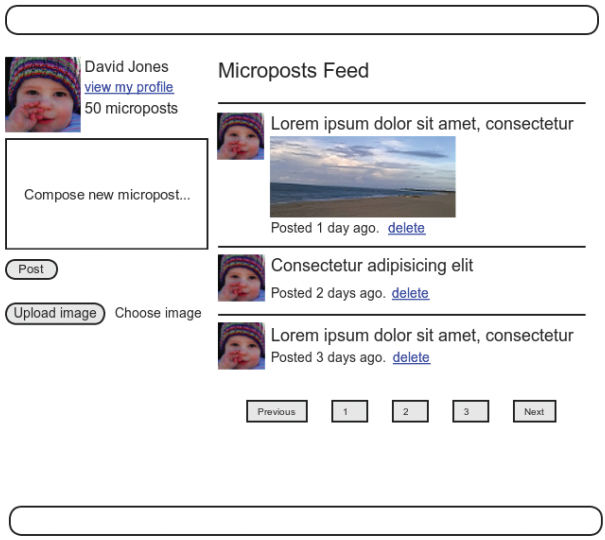


Рис. 11.18 ❖ Макет страницы с кнопкой выгрузки изображения

Листинг 11.55 ❖ Добавление CarrierWave в 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'
gem 'mini_magick',             '3.8.0'
gem 'fog',                     '1.36.0'
gem 'will_paginate',           '3.0.7'
gem 'bootstrap-will_paginate', '0.0.10'
.
.
.
```

Затем выполняем установку, как обычно:

```
$ bundle install
```

Гем CarrierWave добавляет Rails-генератор для создания загрузчиков изображений, один из которых мы и создадим, назвав его `picture`[1]:

[1] Первоначально я назвал новый атрибут `image`, но это настолько часто употребляемое слово, что в конце концов оно стало создавать путаницу.

microposts	
id	integer
content	text
user_id	integer
created_at	datetime
updated_at	datetime
picture	string

Рис. 11.19 ❖ Модель данных Micropost с атрибутом picture

Картинки, выгруженные с помощью CarrierWave, должны быть связаны с соответствующим атрибутом модели Active Record, который просто содержит имя файла изображения в строковом поле. Получившаяся дополненная модель данных для микросообщений показана на рис. 11.19.

Чтобы добавить требуемый атрибут `picture` в модель Micropost, сгенерируем миграцию базы данных и выполним ее:

```
$ rails generate migration add_picture_to_microposts picture:string
$ bundle exec rake db:migrate
```

Чтобы с помощью CarrierWave связать изображение с моделью, нужно воспользоваться методом `mount_uploader`, который принимает символ, представляющий атрибут, и имя класса созданного загрузчика:

```
mount_uploader :picture, PictureUploader
```

(`PictureUploader` определен в файле `picture_uploader.rb`, который мы начнем править в разделе 11.4.2, но в данный момент нас вполне устроит то, что было создано по умолчанию.) Добавление загрузчика в модель Micropost показано в листинге 11.56.

Листинг 11.56 ❖ Добавление загрузчика изображений в модель 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
```

В некоторых системах может потребоваться перезагрузить Rails-сервер, чтобы обеспечить успешное выполнение тестов. (Если вы используете Guard, как описано в разделе 3.7.3, ему тоже может потребоваться перезагрузка, а может быть, даже придется закрыть терминал и снова запустить Guard в новом.)

Чтобы добавить загрузчик на главную страницу, как показано на рис. 11.18, нужно добавить тег `file_field` в форму микросообщений (листинг 11.57).

Листинг 11.57 ❖ Добавление выгрузки изображений в форму создания микросообщения (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 %>
```

Обратите внимание на

```
html: { multipart: true }
```

в аргументах `form_for`, это необходимо для выгрузки файла.

Наконец, нужно добавить `picture` в список атрибутов, разрешенных для изменения через Веб. Для этого отредактируем метод `micropost_params` (листинг 11.58).

Листинг 11.58 ❖ Добавление picture в список разрешенных атрибутов (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
```

После выгрузки картинку можно показать вызовом вспомогательного метода `image_tag` в шаблоне микросообщений, как показано в листинге 11.59. Обратите внимание на логический метод `picture?`, который предотвращает появление тега изображения в отсутствие последнего. Этот метод автоматически создает CarrierWave, опираясь на название атрибута изображения. Результат успешной отправки показан на рис. 11.20. Реализация автоматизированных тестов выгрузки изображений оставлена в качестве упражнения (раздел 11.6).

Листинг 11.59 ❖ Добавление изображения в микросообщение (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>
```

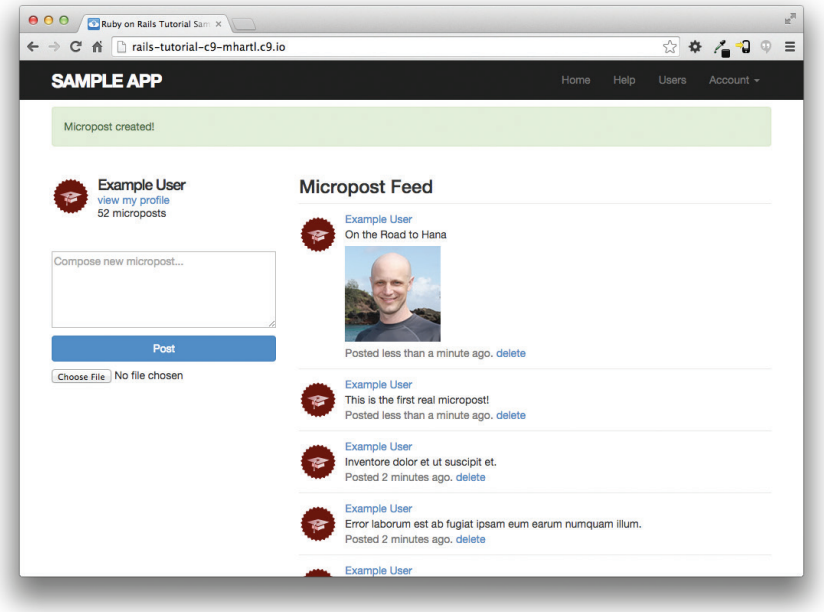


Рис. 11.20 ❖ Результат отправки микросообщения с изображением

11.4.2. Проверка изображений

Загрузчик из раздела 11.4.1 – неплохое начало, но имеет множество недостатков. В частности, он не позволяет ввести какие-либо ограничения для загружаемых файлов, а это может вызвать проблемы, если пользователь попытается выгрузить

файл большого размера или неподдерживаемого типа. Чтобы устранить этот недочет, добавим проверку размера и формата с обеих сторон – на сервере и на клиенте (то есть в браузере).

Первая проверка изображения, ограничивающая возможность выгрузки определенным набором типов файлов, уже есть в самом загрузчике CarrierWave. Этот код (он был закомментирован в сгенерированном загрузчике) проверяет расширение в имени файла (PNG, GIF и оба варианта JPEG), как показано в листинге 11.60.

Листинг 11.60 ❖ Проверка формата картинки (app/uploaders/picture_uploader.rb)

```
class PictureUploader < CarrierWave::Uploader::Base
  storage :file

  # Переопределяет каталог для выгруженных файлов.
  # Есть смысл оставить значение по умолчанию, чтобы не приходилось настраивать загрузчики
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Белый список поддерживаемых расширений имен файлов.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

Вторая проверка, контролирующая размер изображения, определяется непосредственно в модели Micropost. В отличие от предыдущих проверок, для нее отсутствует встроенный Rails-валидатор. Поэтому придется определить собственный, который мы назовем `picture_size`, как показано в листинге 11.61. Обратите внимание, что для вызова нестандартных валидаторов используется метод `validate` (а не `validates`).

Листинг 11.61 ❖ Добавление проверки размера изображения (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

    # Проверяет размер выгруженного изображения.
    def picture_size
      if picture.size > 5.megabytes
        errors.add(:picture, "should be less than 5MB")
      end
    end
end
```

Эта проверка вызывает метод, соответствующий заданному символу (:picture_size). В самом методе picture_size мы добавляем собственное сообщение об ошибке в коллекцию errors (мы видели ее раньше в разделе 6.2.2), в данном случае при превышении лимита в 5 Мбайт (такой синтаксис мы использовали раньше в блоке 8.2).

Добавим также две проверки изображений на стороне клиента. Сначала реализуем проверку формата, воспользовавшись параметром accept в теге ввода file_field:

```
<%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
```

Допустимые форматы определяются списком MIME-типов, пропускаемых проверкой в листинге 11.60.

Затем добавим немного кода на JavaScript (точнее, jQuery[1]), чтобы вывести предупреждение, если пользователь попытается выгрузить изображение слишком большого размера (это предотвратит длительные по времени попытки выгрузки и снизит нагрузку на сервер):

```
$('#micropost_picture').bind('change', function() {
  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.');
  }
});
```

Хотя jQuery не является предметом обсуждения в этой книге, вы можете заметить, что этот код проверяет элементы страницы с атрибутом id=micropost_picture (на это указывает значок решетки #), что соответствует атрибуту id формы ввода микросообщений из листинга 11.57. (Чтобы убедиться в этом, щелкните правой кнопкой мыши в форме и выберите пункт контекстного меню, отвечающий за вывод исходного кода элемента в браузере.) Когда элемент с этим атрибутом изменяется, вызывается функция jQuery, которая запускает метод alert, если размер файла слишком велик[2].

Результат добавления этих проверок представлен в листинге 11.62.

Листинг 11.62 ❖ Проверка размера файла при помощи 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">
```

[1] https://ru.wikipedia.org/wiki/JQuery.

[2] Чтобы узнать, как это делается, поищите в Интернете по фразе «javascript maximum file size» (максимальный размер файла), пока не найдете что-нибудь на Stack Overflow.

```
    <%= f.file_field :picture, accept: 'image/jpeg,image/gif,image/png' %>
  </span>
<% end %>

<script type="text/javascript">
  $('#micropost_picture').bind('change', function() {
    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>
```

Важно понять, что на самом деле такой код не может предотвратить выгрузку файла слишком большого размера. Даже если ваш код не позволяет отправку через веб, всегда можно отредактировать JavaScript через веб-инспектор или выполнить прямой запрос `POST` с помощью, например, `curl`. Вот почему необходимо выполнять проверки на стороне сервера, как в листинге 11.61.

11.4.3. Изменение размеров изображений

Проверка размеров изображений в разделе 11.4.2 – это здорово, но она все еще позволяет загружать изображения слишком большие, чтобы исказить верстку нашего сайта, результат может получиться просто ужасным (рис. 11.21). Поэтому было бы неплохо изменять их размеры перед отображением на странице[1].

Мы будем изменять размер изображений с помощью программы ImageMagick (http://www.imagemagick.org/)[2], поэтому ее нужно установить в окружении разработки. (Как мы увидим в разделе 11.4.4, если развертывание выполняется в среде Heroku, программа ImageMagick уже предустановлена в ней.) В облачной IDE это можно сделать следующим образом[3]:

```
$ sudo apt-get update
$ sudo apt-get install imagemagick --fix-missing
```

Теперь добавим MiniMagick-интерфейс (https://github.com/minimagick/minimagick) вместе с командой для изменения размера. В документации к MiniMagick описано несколько вариантов таких команд, но нам понадобится только одна – `resize_to_limit: [400, 400]`, она уменьшает большие изображения так, чтобы

---

1 *Видимые размеры* изображений можно ограничить средствами CSS, но размеры файлов изображений при этом не изменятся. Как известно, большие изображения требуют больше времени на загрузку. (Вероятно, вы посещали сайты, где всегда загружаются такие «маленькие» картинки. Именно поэтому мы так поступать не будем.)

2 https://ru.wikipedia.org/wiki/ImageMagick.

3 Я взял этот пример из официальной документации Ubuntu. Если вы не работаете в облачной IDE или эквивалентной Linux-системе, поищите в Интернете по фразе: «imagemagick <ваша платформа>». В OS X используйте команду `brew install imagemagick`, если у вас установлен диспетчер пакетов Homebrew.

при любом исходном размере они умещались в квадрат 400×400 пикселей, оставляя маленькие изображения нетронутыми. (В MiniMagick имеются также команды, *растягивающие* изображения, если они слишком маленькие, но нам этого не требуется.) После добавления кода из листинга 11.63 большие изображения будут выглядеть более привлекательно (рис. 11.22).

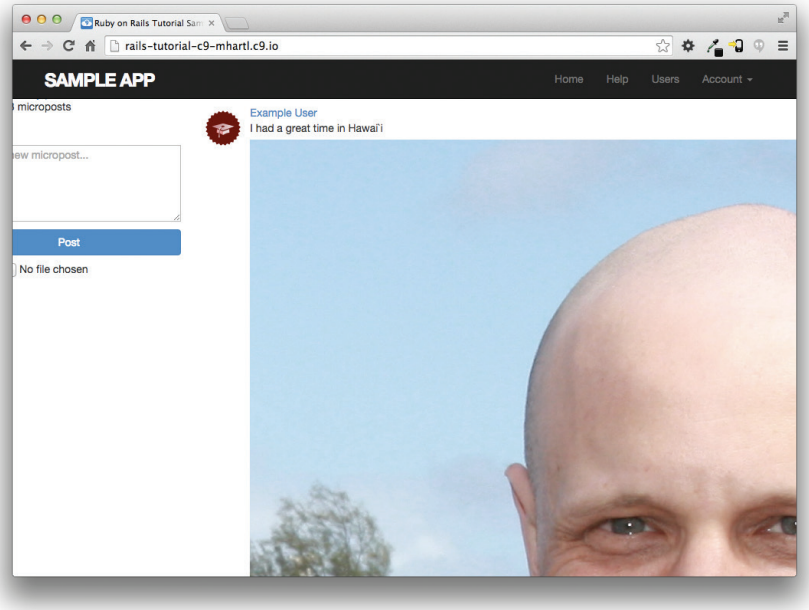


Рис. 11.21 ❖ Слишком большое изображение

Листинг 11.63 ❖ Настройка изменения размеров изображений (app/uploaders/picture_uploader.rb)

```
class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
  process resize_to_limit: [400, 400]

  storage :file

  # Переопределяет каталог для выгруженных файлов.
  # Есть смысл оставить значение по умолчанию, чтобы не приходилось настраивать загрузчики
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Белый список поддерживаемых расширений имен файлов.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

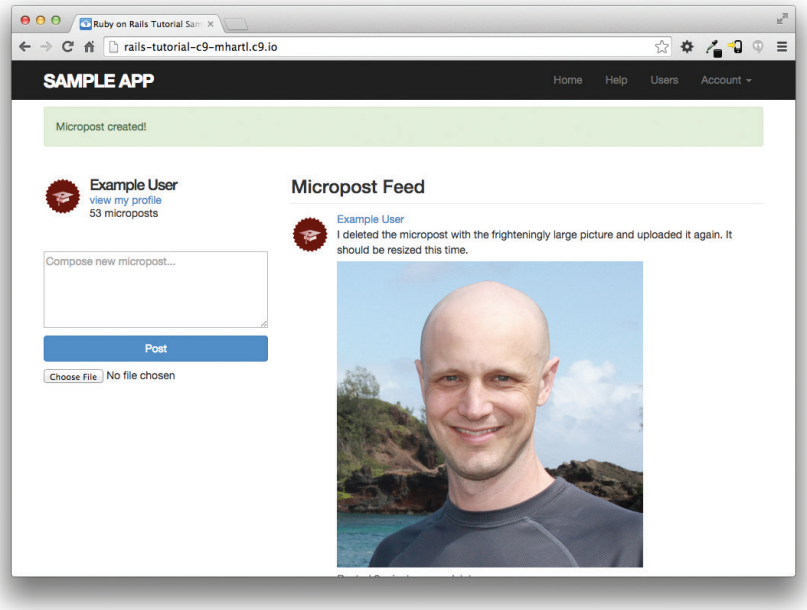


Рис. 11.22 ❖ Изображение с отрегулированными размерами

11.4.4. Выгрузка изображений в эксплуатационном окружении

Загрузчик изображений из раздела 11.4.3 хорошо подходит для среды разработки, но (как видно в строке `storage :file` в листинге 11.63) для хранения картин окон использует локальную файловую систему, а это не самое удачное решение для эксплуатационного окружения[1]. Поэтому для хранения изображений воспользуемся облачной службой[2].

Настроить использование облачного хранилища в приложении нам поможет гем `fog` (листинг 11.64).

Листинг 11.64 ❖ Настройка загрузчика изображений для эксплуатационного окружения (app/uploaders/picture_uploader.rb)

```
class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick
  process resize_to_limit: [400, 400]

  if Rails.env.production?
```

[1] Кроме всего прочего, хранилище файлов на Heroku временное, поэтому загруженные картинки будут удаляться при каждом развертывании.

[2] В этом разделе обсуждается довольно сложная тема, но его можно пропустить, не опасаясь упустить что-то важное.

```
    storage :fog
  else
    storage :file
  end

  # Переопределяет каталог для выгруженных файлов.
  # Есть смысл оставить значение по умолчанию, чтобы не приходилось настраивать загрузчики
  defstore_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Белый список поддерживаемых расширений имен файлов.
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

Здесь для выбора места хранения в зависимости от типа окружения использован логический метод `production?` из блока 7.1:

```
if Rails.env.production?
  storage :fog
else
  storage :file
end
```

Есть множество вариантов облачных хранилищ, но мы воспользуемся одним из самых популярных, с хорошей поддержкой – Simple Storage Service (S3) от Amazon.com[1]. Ниже перечислены основные этапы установки его поддержки:

1. Зарегистрируйтесь на сайте Amazon Web Services (http://aws.amazon.com/).
2. Создайте пользователя через *службу управления идентификацией и доступом* (AWS Identity and Access Management, IAM, http://aws.amazon.com/iam/) и сохраните ключ доступа и секретный ключ.
3. Создайте корзину S3 (выбрав название) в AWS-консоли (http://console.aws.amazon.com/s3), затем предоставьте право на чтение и запись пользователю, созданному на предыдущем этапе.

Дополнительные детали можно уточнить в документации к S3[2] (а также в Google или на сайте Stack Overflow).

После создания и настройки учетной записи S3 создайте и заполните файл конфигурации для CarrierWave, как показано в листинге 11.65. *Примечание*: если ваши настройки не работают, возможно, проблема кроется в вашем местонахождении. Некоторым пользователям понадобится добавить параметр `:region => ENV['S3_REGION']` в `fog_credentials` с последующим вызовом команды `heroku config:set S3_REGION=<регион корзины>` в командной строке, где вместо `<регион корзи-`

[1] S3 – это платная служба, но хранилище, необходимое для установки и тестирования учебного приложения Rails Tutorial, стоит меньше 1 цента в месяц.

[2] http://aws.amazon.com/documentation/s3/.

ны> должно быть что-то вроде 'eu-central-1', в зависимости от вашего местонахождения. Чтобы определить верный регион, загляните в список регионов на Amazon (http://docs.aws.amazon.com/general/latest/gr/rande.html).

Листинг 11.65 ❖ Настройка CarrierWave для использования S3 (config/initializers/carrier_wave.rb)

```
if Rails.env.production?
  CarrierWave.configure do |config|
    config.fog_credentials = {
      # Настройки для 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
```

По аналогии с настройкой рассылки электронных писем (листинг 10.56), мы задействовали здесь переменные ENV, чтобы избежать прямого упоминания конфиденциальной информации. В разделе 10.3 эти переменные определялись автоматически, с помощью дополнения SendGrid, но сейчас нужно определить их явно, это делается с помощью команды heroku config:set:

```
$ heroku config:set S3_ACCESS_KEY=<ключ доступа>
$ heroku config:set S3_SECRET_KEY=<секретный ключ>
$ heroku config:set S3_BUCKET=<имя корзины>
```

Теперь можно зафиксировать изменения и перейти к развертыванию. Я рекомендую обновить файл .gitignore, как показано в листинге 11.66, чтобы пропустить каталог загрузки изображений.

Листинг 11.66 ❖ Добавление каталога загрузки изображений в файл .gitignore

```
# Подробнее о пропуске файлов:
# https://help.github.com/articles/ignoring-files.
#
# Если окажется, что вы пропускаете файлы, создаваемые
# текстовым редактором или операционной системой, вероятно,
# будет лучше добавить глобальное игнорирование:
# git config --global core.excludesfile '~/.gitignore_global'

# Пропуск настроек bundler.
/.bundle

# Пропуск базы данных SQLite по умолчанию.
/db/*.sqlite3
/db/*.sqlite3-journal

# Пропуск всех журналов и временных файлов.
/log/*.log
```

```
/tmp

# Пропуск файлов Spring.
/spring/*.pid

# Пропуск загруженных тестовых изображений.
/public/uploads
```

Теперь можно послать изменения в рабочую ветвь и слить ее с основной ветвью:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

а затем развернуть приложение, очистить базу данных, выполнить миграцию и снова заполнить ее тестовыми данными:

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

Так как программа ImageMagick уже установлена в Heroku, мы сразу получаем действующую выгрузку изображений и изменение их размера (рис. 11.23).

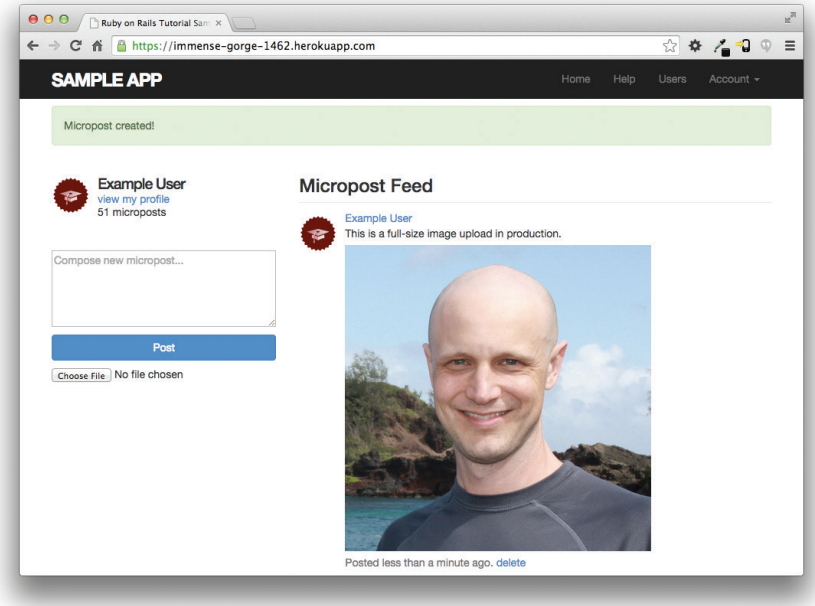


Рис. 11.23 ❖ Выгрузка изображений в эксплуатационном окружении

11.5. Заключение

Добавив ресурс Microposts, мы практически подобрались к завершению учебного приложения. Осталось лишь придать ему черты социальных сетей и позволить пользователям следовать друг за другом. Мы узнаем, как смоделировать такие отношения между пользователями, и увидим всю значимость ленты сообщений в главе 12.

Если вы пропустили раздел 11.4.4, выгрузите все изменения в репозиторий и объедините их с основной ветвью:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user microposts"
$ git checkout master
$ git merge user-microposts
$ git push
```

Затем разверните приложение в эксплуатационном окружении:

```
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

Стоит отметить, что в этой главе был установлен последний необходимый гем. Для сравнения: окончательный `Gemfile` преставлен в листинге 11.67.

Листинг 11.67 ❖ Окончательный Gemfile учебного приложения

```
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',                     '1.36.0'
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. Что мы узнали в этой главе

- Микросообщения, как и учетные записи, моделируются в виде ресурсов, поддерживаемых моделью Active Record.
- Rails поддерживает составные индексы.
- Взаимосвязь между пользователем и множеством его микросообщений моделируется с помощью методов `has_many` и `belongs_to` в моделях User и Micropost, соответственно.
- Пара `has_many/belongs_to` позволяет другим методам обнаруживать и использовать связи в моделях.
- Вызов `user.microposts.build(...)` возвращает новый объект Micropost, автоматически связанный с данным пользователем.
- Rails поддерживает упорядочивание по умолчанию посредством `default_scope`.
- Области видимости принимают анонимные функции в качестве аргумента.
- Параметр `dependent: :destroy` одновременно уничтожает указанный объект со связанными с ним объектами.
- Постраничный вывод и подсчет объектов поддерживают связи между моделями, что автоматически улучшает эффективность кода.
- Связи можно определять в тестовых данных.
- Переменные можно передавать в частичные шаблоны Rails.
- Для выборки объектов Active Record, отвечающих определенным требованиям, можно использовать метод `where`.
- Создание и удаление зависимых объектов через их связи обеспечит безопасность операций.
- Выгружать изображения и изменять их размер можно с помощью CarrierWave.

11.6. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Проведите рефакторинг главной страницы, чтобы использовать отдельные шаблоны для каждой ветки в инструкции `if-else`.
2. Добавьте тесты для проверки счетчика микросообщений в боковой панели (в том числе для проверки правильности выбора множественного числа). Возьмите за основу код в листинге 11.68.
3. Следуя шаблону из листинга 11.69, напишите тест для загрузчика изображений из раздела 11.4. Предварительно вам понадобится добавить изображение в каталог с тестовыми данными (например, командой `cp app/assets/images/rails.png test/fixtures/`). Чтобы избежать сбивающей с толку ошибки, вам также понадобится настроить пропуск изменения размера изображений в тестах CarrierWave, создав файл инициализатора, как показано в листинге 11.70. Дополнительные утверждения в листинге 11.69 проверяют поле загрузки файла на главной странице и допустимость типа изображения. Обратите внимание на специальный метод `fixture_file_upload`, загружающий файлы с тестовыми данными внутри тестов[1]. *Подсказка*: для проверки допустимости атрибута `picture` воспользуйтесь методом `assigns`, упомянутым в разделе 10.1.4.

Листинг 11.68 ❖ Заготовка теста счетчика микросообщений в боковой панели (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
    # У пользователя нет сообщений
    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
```

[1] Пользователи Windows могут добавить параметр `:binary` в вызов `:fixture_file_upload(file, type, :binary)`.

Листинг 11.69 ❖ Заготовка для теста, проверяющего выгруженные изображения (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]'
    # Недопустимая информация в форме
    post microposts_path, micropost: { content: "" }
    assert_select 'div#error_explanation'
    # Допустимая информация в форме
    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
    # Удаление сообщения.
    assert_select 'a', 'delete'
    first_micropost = @user.microposts.paginate(page: 1).first
    assert_difference 'Micropost.count', -1 do
      delete micropost_path(first_micropost)
    end
    # Переход в профиль другого пользователя.
    get user_path(users(:archer))
    assert_select 'a', { text: 'delete', count: 0 }
  end
  .
  .
  .
end
```

Листинг 11.70 ❖ Инициализатор, позволяющий пропустить изменение размера изображения в тестах (config/initializers/skip_image_resizing.rb)

```
if Rails.env.test?
  CarrierWave.configure do |config|
    config.enable_processing = false
  end
end
```

Глава 12

Следование за пользователями

В этой главе мы завершим учебное приложение, добавив социальный слой, что позволит пользователям подписываться (и отменять подписку) на сообщения других пользователей[1]; в результате на главной странице каждого пользователя появится лента микросообщений читаемых им пользователей. Для начала, в разделе 12.1, выясним, как смоделировать взаимоотношения между пользователями, а соответствующий интерфейс создадим в разделе 12.2 (включая введение в Ajax). В конце, в разделе 12.3, мы разработаем полнофункциональную ленту сообщений.

Эта заключительная глава содержит самый сложный материал во всем учебнике, в том числе несколько хитростей, связанных с применением Ruby/SQL для создания ленты сообщений. На этих примерах вы увидите, как Rails обрабатывает даже самые замысловатые модели данных, что будет весьма полезно для вас, когда начнете разработку собственных приложений с их собственными требованиями. Чтобы помочь с переходом от обучения к самостоятельной разработке, в разделе 12.4 предложены ссылки на дополнительные источники информации.

Так как материал этой главы отличается высокой сложностью, прежде чем писать код, мы сделаем небольшой обзор интерфейса. Как и в предыдущих главах, на этом раннем этапе мы будем представлять страницы при помощи макетов[2]. Вся последовательность страниц работает следующим образом: пользователь (Джон Кальвин) открывает страницу своего профиля (рис. 12.1) и переходит на страни-

---

[1] Позволю себе небольшой комментарий к переводу этой главы. В оригинале очень часто употребляются различные формы слова follow (following, follower, followeds и т. п.), что в дословном переводе означает «следовать» («преследуемый», «последователь», «преследуемые»). Однако Twitter (клоном которого является данное учебное приложение) вместо этого использует термины «читаемые» и «читатели», и лично мне такой вариант нравится гораздо больше. Я буду стараться придерживаться именно его в оставшейся части книги, несмотря на несколько казусов, связанных с этим. При этом общее название этой функции приложения переведено как «следование за пользователями». *P.S.* Надеюсь, данный комментарий не запутал вас окончательно :) – *Прим. перев.*

[2] Фотографии для макетов взяты по адресам: http://www.flickr.com/photos/john_lustig/2518452221/ и https://www.flickr.com/photos/renemensen/9187111340.

цу со списком пользователей (рис. 12.2), чтобы выбрать пользователя, сообщения которого он будет читать. Затем Джон переходит на страницу профиля второго пользователя, Томаса Хоббса (рис. 12.3), щелкает на кнопке **Follow** (Подписаться), чтобы подписаться на его сообщения. В результате надпись «Follow» изменяется на «Unfollow» (Отписаться) и увеличивает количество «followers» (читателей) Хоббса на единицу (рис. 12.4). Вернувшись на свою главную страницу, Кальвин теперь видит увеличившееся количество «following» (читаемых) и обнаруживает микросообщения Хоббса в своей ленте сообщений (рис. 12.5). Остальная часть этой главы посвящена реализации данной последовательности.

12.1. Модель Relationship

Первый шаг к реализации следования за пользователями – построение модели данных, которая не так проста, как думается. На первый взгляд кажется, что достаточно добавить связь `has_many`: пользователи имеют много (`has_many`) читаемых и много (`has_many`) читателей. Но, как мы увидим, это решение имеет проблему, которую можно исправить с помощью `has_many :through`.

Как обычно, если вы используете репозиторий Git, создайте новую рабочую ветку:

```
$ git checkout master
$ git checkout -b following-users
```

12.1.1. Проблема модели данных (и ее решение)

В качестве первого шага на пути построения модели данных для следования за пользователями обсудим наиболее типичный случай. Рассмотрим пользователя, который читает сообщения второго пользователя: например, можно сказать, что Кальвин *читает* сообщения Хоббса, или сообщения Хоббса *читаются* Кальвином, то есть Кальвин является *читателем*, а Хоббс – *читаемым*. Согласно стандартному способу образования множественного числа в Rails, множество всех пользователей, читающих данного, называется *followers* (читателями), и `hobbes.followers` будет массивом таких пользователей. К сожалению, это правило не работает в обратном направлении: по умолчанию множество всех *читаемых* пользователей можно было бы назвать *followeds*, что в английском языке весьма безграмотно и неуклюже. Последовав обычаю Twitter, мы назовем их *following* (читаемые) (как в надписи «50 following, 75 followers» («50 читаемых, 75 читателей»)), с соответствующим массивом `calvin.following`.

Это обсуждение предполагает создание модели читаемых пользователей по образцу на рис. 12.6, с помощью таблицы `following` (читаемые) и связи `has_many`. Так как массив `user.following` – это коллекция пользователей, каждая запись в таблице `following` должна представлять пользователя, идентифицируемого по `followed_id`, совместно с `follower_id`, чтобы установить связь[1]. Кроме того, так как

[1] Для простоты на рис. 12.6 в таблице `following` опущен столбец `id`.

каждая строка представляет пользователя, мы должны были бы включить другие его атрибуты, такие как имя, адрес электронный почты, пароль и т. д.

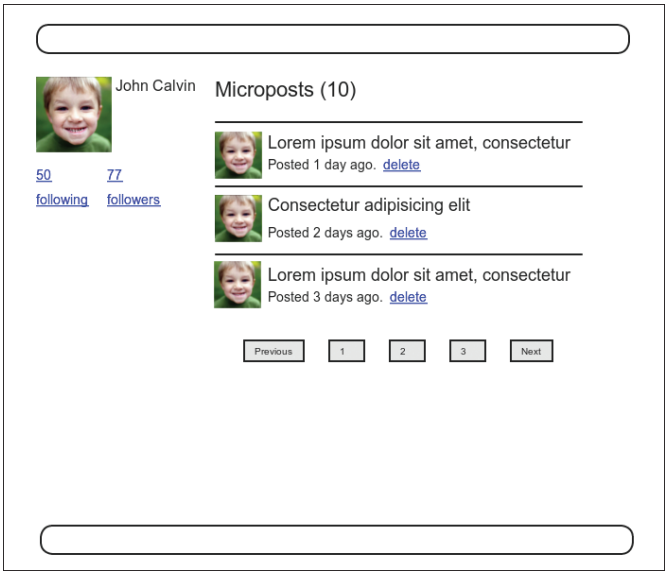


Рис. 12.1 ❖ Профиль текущего пользователя

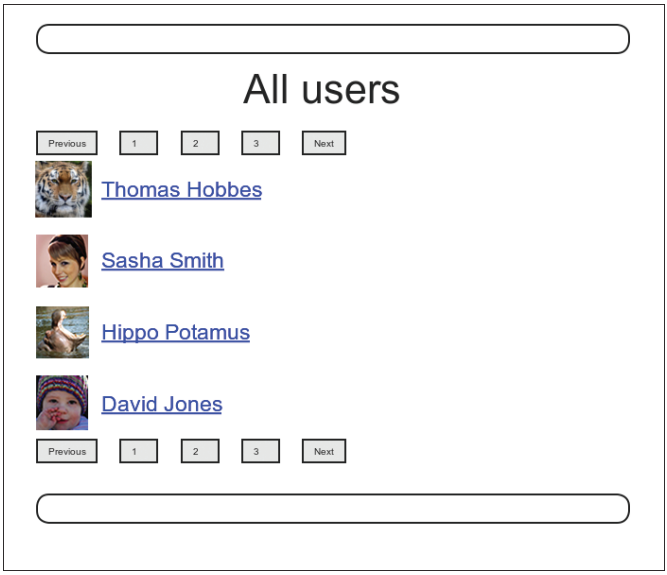


Рис. 12.2 ❖ Поиск пользователя, сообщения которого будут интересны для чтения

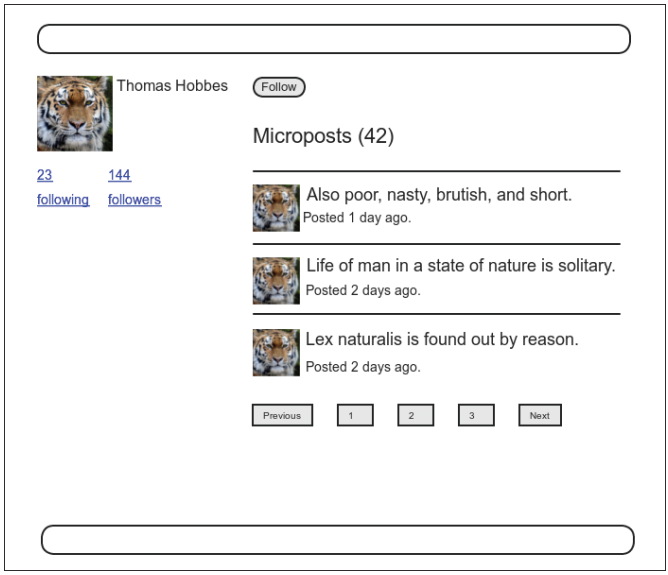


Рис. 12.3 ❖ Профиль этого пользователя с кнопкой **Follow** (Подписаться)

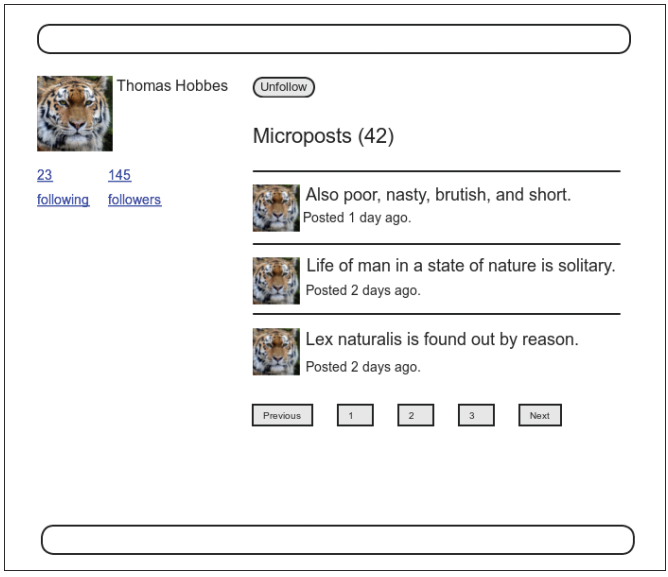


Рис. 12.4 ❖ Этот же профиль с кнопкой **Unfollow** (Отписаться) и увеличившимся количеством читателей

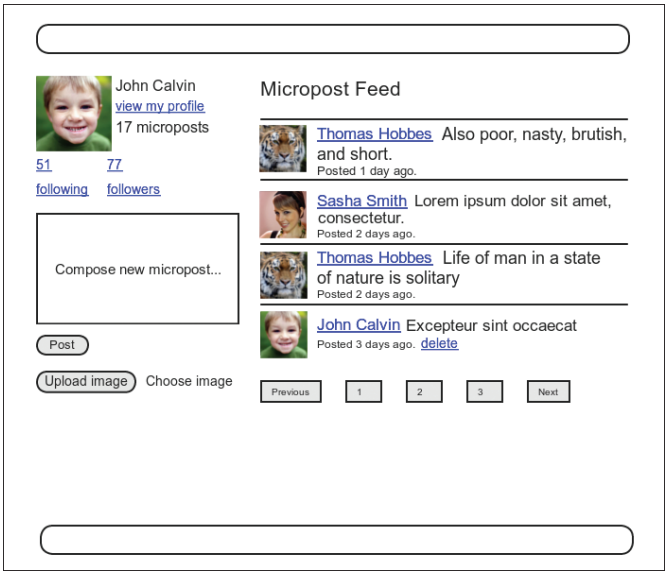


Рис. 12.5 ❖ Главная страница с лентой сообщений и увеличившимся количеством читаемых

Проблема модели данных на рис. 12.6 – в том, что она ужасно избыточна: строка содержит не только идентификатор каждого читаемого пользователя, но и всю остальную информацию, которая уже есть в таблице `users`. Хуже того, для моделирования читателей понадобится отдельная, настолько же избыточная таблица `followers`. Наконец, эта модель данных кошмарно неудобна в эксплуатации, так как каждый раз при изменении пользователем (скажем) своего имени нам пришлось бы обновлять запись не только в таблице `users`, но и каждую строку с этим пользователем в обеих таблицах `following` и `followers`.

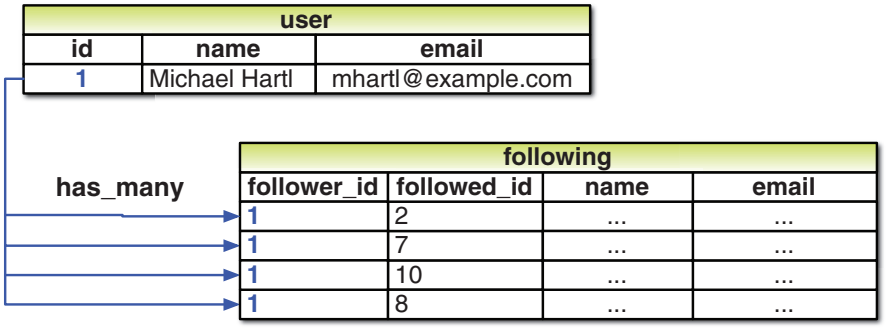


Рис. 12.6 ❖ Наивная реализация следования за пользователями

Сложность в том, что нам не хватает абстракции, которая легла бы в основу. Чтобы прийти к более удачной модели, обдумаем реализацию *следования* за поль-

зователями в веб-приложении. Вспомните, как рассказывалось в разделе 7.1.2, что REST-архитектура предполагает использование *ресурсов*, которые создаются и уничтожаются. Это приводит нас к двум вопросам. Во-первых, что создается, когда пользователь начинает читать сообщения другого пользователя? Во-вторых, что уничтожается, когда пользователь перестает это делать? Поразмыслив, мы видим, что в этих случаях приложение должно создать либо удалить *взаимоотношение* между двумя пользователями. То есть пользователь связан множеством взаимоотношений с *читаемыми* (или *читателями*) *через* эти взаимоотношения.

Модель данных нашего приложения имеет еще одну дополнительную деталь, о которой нельзя не упомянуть: в отличие от симметричной дружбы в стиле Facebook (или ВКонтакте), которая всегда взаимна (по крайней мере, на уровне модели данных), взаимоотношения в стиле Twitter потенциально *асимметричны* – Кальвин может читать сообщения Хоббса, при этом Хоббс может не читать сообщений Кальвина. Чтобы определить различие между двумя этими случаями, примем терминологию *активных* и *пассивных* взаимоотношений: если Кальвин читает сообщения Хоббса, но не наоборот, у Кальвина с Хоббсом *активные* взаимоотношения, а у Хоббса с Кальвином – *пассивные*[1].

Сейчас мы сосредоточимся на активных взаимоотношениях и реализуем список читаемых пользователей, а пассивные взаимоотношения рассмотрим в разделе 12.1.5. Модель на рис. 12.6 подсказывает, как это реализовать: так как каждый читаемый пользователь однозначно определяется через `followed_id`, можно переименовать таблицу `following` в `active_relationships`, опустив все подробности о пользователе и использовав `followed_id` для получения читаемого пользователя из таблицы `users`. Получившаяся модель данных представлена на рис. 12.7.

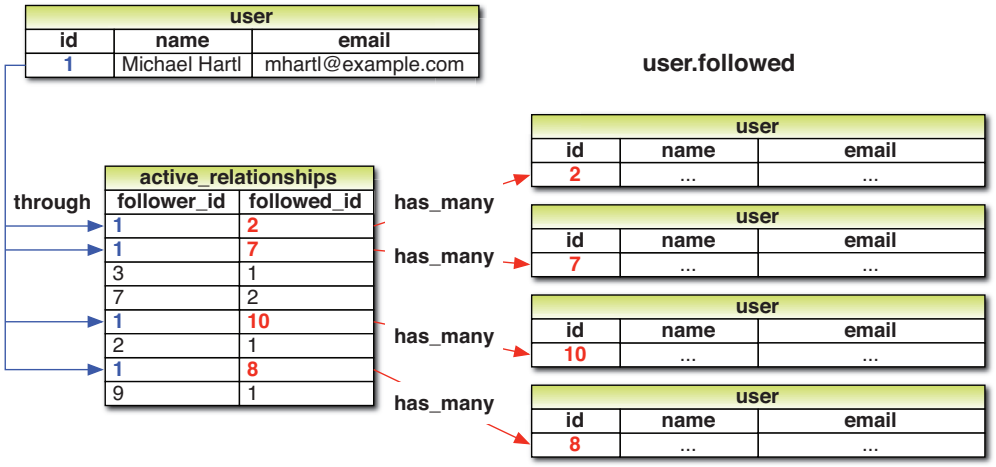


Рис. 12.7 ❖ Модель читаемых пользователей через активные взаимоотношения

[1] Спасибо читателю Паулю Фиораванти (Paul Fioravanti) за предложенную терминологию.

Так как в конечном итоге таблицы базы данных для активных и пассивных взаимоотношений не будут отличаться, воспользуемся общим термином *relationship* для названия таблицы, с соответствующей моделью Relationship, которая показана на рис. 12.8. В разделе 12.1.4 мы узнаем, как использовать ее для имитации моделей ActiveRelationship и PassiveRelationship.

relationships	
id	integer
follower_id	integer
followed_id	integer
created_at	datetime
updated_at	datetime

Рис. 12.8 ❖ Модель данных Relationship

Чтобы приступить к реализации, сгенерируем соответствующую миграцию:

```
$ rails generate model Relationship follower_id:integer followed_id:integer
```

Так как мы будем искать взаимоотношения по `follower_id` и `followed_id`, добавим индексы к обоим столбцам для эффективности (листинг 12.1).

Листинг 12.1 ❖ Добавление индексов в таблицу 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
```

Здесь также добавлен составной индекс, отвечающий за уникальность пары (`follower_id`/`followed_id`), чтобы один пользователь мог последовать за другим только один раз. (Сравните с индексом уникальность адреса электронной почты в листинге 6.28 и составным индексом в листинге 11.1.) В разделе 12.1.4 мы увидим, что этого не допустит интерфейс пользователя, но добавление уникальности поможет предотвратить такую попытку, если пользователь попробует создать дубликат взаимоотношений каким-то другим способом (например, с помощью инструмента командной строки, такого как `curl`).

Как обычно, выполним миграцию базы данных, чтобы создать таблицу `relationships`:

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

12.1.2. Связь пользователь/взаимоотношения

Прежде чем приступить к реализации читателей и читаемых, необходимо установить связь между пользователями и взаимоотношениями. Пользователь has_many (имеет много) взаимоотношений, и – так как взаимоотношения подразумевают двух пользователей – взаимоотношения belongs_to (принадлежат) и читателям, и читаемым.

По аналогии с микросообщениями в разделе 11.1.3, мы будем создавать новые взаимоотношения через их связь с пользователем, как показано ниже:

```
user.active_relationships.build(followed_id: ...)
```

Вероятно, вы ожидали увидеть такой же код, как в разделе 11.1.3, и он действительно похож, но при этом имеет два важных отличия.

Во-первых, связь пользователь/микросообщение мы смогли определить как:

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

И это правильно, так как по умолчанию Rails ищет модель Micropost, которая соответствует символу :microposts[1]. Однако в данном случае мы должны использовать определение:

```
has_many :active_relationships
```

несмотря на то что модель, лежащая в основе, называется Relationship. Поэтому нам придется сообщить фреймворку Rails, какое имя класса модели искать.

Во-вторых, до этого в модели Micropost мы писали:

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

потому что в таблице microposts имеется атрибут user_id для идентификации пользователя (раздел 11.1.1). Идентификатор, используемый для согласования двух таблиц в базе данных, называется *внешним ключом*, и если внешним ключом для объекта модели User является user_id, то Rails автоматически устанавливает ассоциацию: по умолчанию Rails ищет внешний ключ с именем в формате <class>_id, где <class> – имя класса в нижнем регистре[2]. В нашем случае, хотя мы все еще

---

[1] Технически Rails преобразует аргумент has_many в имя класса с помощью метода classify, превращающего "foo_bars" в "FooBar".

[2] Технически для преобразования имени класса в идентификатор Rails применяет метод underscore. Например, "FooBar".underscore вернет "foo_bar", поэтому внешний ключ для объекта FooBar должен иметь имя foo_bar_id.

имеем дело с пользователями, читающий чьи-то сообщения пользователь теперь идентифицируется по внешнему ключу follower_id, и об этом необходимо сообщить фреймворку.

В результате обсуждения выше получаем связь пользователь/взаимоотношение, как показано в листингах 12.2 и 12.3.

Листинг 12.2 ❖ Реализация связи has_many для активных взаимоотношений (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
  .
  .
  .
end
```

(Так как удаление учетной записи пользователя должно так же удалить все его взаимоотношения, мы добавили dependent: :destroy в определение связи.)

Листинг 12.3 ❖ Добавление связи belongs_to в модель Relationship (app/models/relationship.rb)

```
class Relationship < ActiveRecord::Base
  belongs_to :follower, class_name: "User"
  belongs_to :followed, class_name: "User"
end
```

Связь followed на самом деле не понадобится до раздела 12.1.4, но сходство структуры моделей читателей/читаемых более понятно при одновременной реализации.

После добавления связей появляются методы, подобные тем, что мы видели в табл. 11.1, они перечислены в табл. 12.1.

Таблица 12.1 ❖ Список методов, реализующих связь пользователь/активное_взаимоотношение

Метод	Назначение
active_relationship.follower	Возвращает читателя
active_relationship.followed	Возвращает читаемого
user.active_relationships.create(followed_id: user.id)	Создает активное взаимоотношение, связанное с пользователем
user.active_relationships.create!(followed_id:user.id)	Создает активное взаимоотношение, связанное с пользователем (возбуждает исключение в случае ошибки)
user.active_relationships.build(followed_id: user.id)	Возвращает новый объект Relationship, связанный с пользователем

12.1.3. Проверка взаимоотношений

Прежде чем продолжить, добавим в модель Relationship пару проверок для полноты картины. Тесты (листинг 12.4) и прикладной код (листинг 12.5) достаточно просты. Тестовые данные с пользователями (листинг 6.29) и взаимоотношениями нарушают ограничение уникальности, наложенное соответствующей миграцией (листинг 12.1). Решение (удаление содержимого, как в листинге 6.30) осталось прежним (листинг 12.6).

Листинг 12.4 ❖ Тестирование проверок в модели 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
```

Листинг 12.5 ❖ Добавление проверок в модель 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
```

Листинг 12.6 ❖ Удаление содержимого тестовых данных взаимоотношений (test/fixtures/relationships.yml)

```
# ничего нет
```

Сейчас набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 12.7 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

12.1.4. Читаемые пользователи

Теперь мы подходим к самой сути ассоциаций Relationship: *читаемые* и *читатели*. Мы впервые воспользуемся отношением `has_many :through`: пользователь может *иметь много* (`has_many`) читаемых *через* (`through`) взаимоотношения, как показано на рис. 12.7. По умолчанию, при наличии связи `has_many :through`, Rails ищет внешний ключ, соответствующий единственному числу этой связи. Другими словами, если написать такой код:

```
has_many :followeds, through: :active_relationships
```

Rails увидит «followeds» (читаемые) и будет пользоваться его единственным числом «followed» для выборки коллекции по полю `followed_id` в таблице `relationships`. Но, как отмечалось в разделе 12.1.1, имя `user.followeds` выглядит неестественно, поэтому будем использовать имя `user.following`. Разумеется, Rails позволяет переопределять умолчания, в данном случае с помощью параметра `source` (листинг 12.8), явно указывающего, что источником массива `following` служит множество идентификаторов `followed`.

Листинг 12.8 ❖ Добавление к модели User ассоциации following (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
```

Эта ассоциация определяет мощное сочетание поведения Active Record и массивов. Например, благодаря ей с помощью метода `include?` (раздел 4.3.1) можно проверить, включает ли коллекция читаемых пользователей другого пользователя, или найти объект через ассоциацию:

```
user.following.include?(other_user)
user.following.find(other_user)
```

Во многих случаях с ассоциацией `following` можно эффективно работать как с массивом, однако Rails добавляет внутреннюю интеллектуальную обработку данных. Например, на первый взгляд кажется, что следующая строка

```
following.include?(other_user)
```

извлекает массив всех читаемых пользователей из базы данных, чтобы применить метод include?, но на самом деле Rails организует сравнение прямо в базе данных, тем самым увеличивая производительность. (Сравните с кодом из раздела 11.2.1, где строка

```
user.microposts.count
```

выполняет подсчет прямо в базе данных.)

Для управления взаимоотношениями с читаемым (пользователями) введем вспомогательные методы follow и unfollow, чтобы можно было написать, например, user.follow(other_user). Кроме того, добавим логический метод following?, чтобы с его помощью проверять, читает ли пользователь сообщения другого пользователя[1].

Это как раз та ситуация, когда я предпочту сначала написать тесты. Причина в том, что мы еще довольно далеки от работающего веб-интерфейса, реализующего следование за пользователями, а продолжать работу сложно без своеобразного *клиента* для разрабатываемого кода. Итак, напишем короткий тест для модели User, в котором используем метод following?, чтобы проверить, читает ли один пользователь сообщения другого пользователя. В тесте будет вызываться метод follow, чтобы образовать такую связь, following? – чтобы проверить успешность операции, и, наконец, unfollow вместе с проверкой его работы. Результат представлен в листинге 12.9.

Листинг 12.9 ❖ Тесты вспомогательных методов follow/unfollow/following? **КРАСНЫЙ** (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
```

[1] Если у вас есть опыт моделирования в конкретной области, вы, скорее всего, сами догадаетесь написать такие вспомогательные методы, и даже если не догадаетесь, то осознаете их необходимость, чтобы сделать тесты более прозрачными. Однако если вам такая идея не пришла в голову, ничего страшного. Разработка программного обеспечения обычно протекает итеративно – вы пишете код, пока он не станет трудным для чтения, а затем проводите рефакторинг. Но для краткости изложение в книге немного упрощено.

Используя как справочник табл. 12.1, напишем методы `follow`, `unfollow` и `following?` с использованием связи `following`, как показано в листинге 12.10. (Обратите внимание, ссылка на пользователя `self` опущена везде, где только можно.)

Листинг 12.10 ❖ Вспомогательные методы follow/unfollow/following? **ЗЕЛЕНЫЙ** (app/models/user.rb)

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

  # Выполняет подписку на сообщения пользователя.
  def follow(other_user)
    active_relationships.create(followed_id: other_user.id)
  end

  # Отменяет подписку на сообщения пользователя.
  def unfollow(other_user)
    active_relationships.find_by(followed_id: other_user.id).destroy
  end

  # Возвращает true, если текущий пользователь читает
  #другого пользователя.
  def following?(other_user)
    following.include?(other_user)
  end

  private
    .
    .
    .
end
```

Теперь набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 12.11 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

12.1.5. Читатели

Последним недостающим элементом в мозаике взаимоотношений является метод `user.followers`, сопутствующий методу `user.following`. Глядя на рис. 12.7, можно заметить, что все сведения, необходимые для извлечения массива читателей, уже присутствуют в таблице `relationships` (которую в листинге 12.2 мы обрабатываем как таблицу `active_relationships`). Действительно, техника та же, что и для чи-

таемых пользователей, но только со сменой ролей идентификаторов follower_id и followed_id и подстановкой passive_relationships вместо active_relationships. Модель данных показана на рис. 12.9.

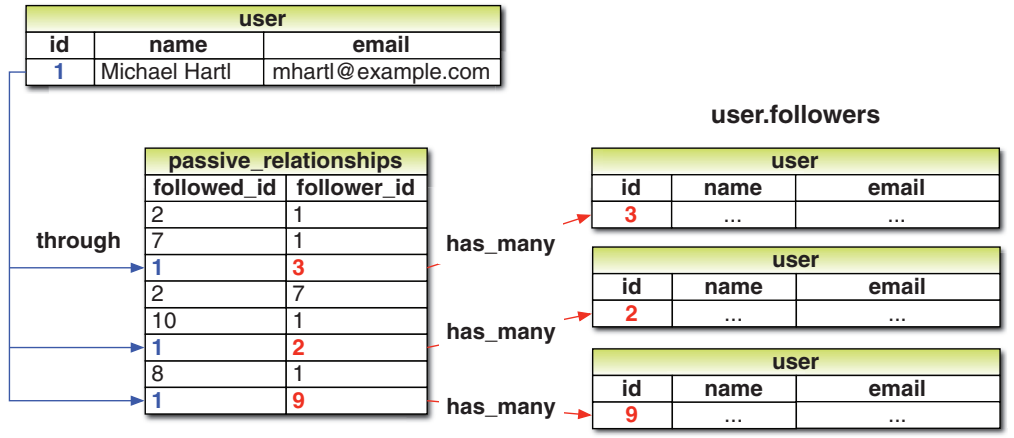


Рис. 12.9 ❖ Модель читателей с использованием пассивных взаимоотношений

Реализация модели данных в листинге 12.12 аналогична модели в листинге 12.8.

Листинг 12.12 ❖ Реализация user.followers с использованием пассивного взаимоотношения (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
```

Стоить отметить, что ключ :source в листинге 12.12 можно было бы опустить для followers:

```
has_many :followers, through: :passive_relationships
```

Это связано с тем, что в данном случае Rails переведет имя атрибута :followers в единственное число и автоматически найдет внешний ключ follower_id. Мы сохранили ключ :source, чтобы подчеркнуть аналогию структуры со связью has_many :following.

Вышеописанную модель данных легко протестировать с помощью метода followers.include? , как показано в листинге 12.13. (Мы могли бы использовать метод followed_by? в дополнение к following?, но он не понадобится нам в приложении.)

Листинг 12.13 ❖ Тест-метод ffollowers **ЗЕЛЕНЫЙ** (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
```

В листинг 12.13 мы добавили всего одну строку в тест из листинга 12.9, но как много механизмов должно работать безукоризненно, чтобы пройти ее, значит, мы создали весьма чувствительный тест.

Сейчас весь набор тестов должен быть **ЗЕЛЕНЫМ**:

```
$ bundle exec rake test
```

12.2. Веб-интерфейс следования за пользователями

В разделе 12.1 были предъявлены довольно высокие требования к нашим навыкам моделирования данных, и ничего страшного, если вам понадобится какое-то время, чтобы все это осознать. По сути, лучший способ разобраться в связях – реализовать их использование в веб-интерфейсе.

Во введении к этой главе был дан предварительный обзор последовательности переходов по страницам, чтобы оформить подписку на получение сообщений пользователя. В этом разделе мы реализуем базовый интерфейс и функциональность, описанные в тех макетах. Мы также создадим отдельные страницы для демонстрации массивов читателей и читаемых. В разделе 12.3 мы завершим учебное приложение, добавив ленту сообщений пользователя.

12.2.1. Образцы данных

Так же как в предыдущих главах, используем Rake-задачу для заполнения базы данных образцами взаимоотношений. Это позволит нам заняться в первую оче-

редь внешним видом страниц, временно отложив серверную часть функциональности.

Код, заполняющий базу данных взаимоотношениями, показан в листинге 12.14. Если описать его упрощенно: первый пользователь подписывается на получение сообщений пользователей с 3 по 51, а пользователи с 4 по 41 подписываются на получение сообщений первого пользователя, – образовавшихся в результате взаимоотношений будет достаточно для разработки интерфейса приложения.

Листинг 12.14 ❖ Добавление взаимоотношений читаемый/читатель в тестовые данные (db/seeds.rb)

```
# Пользователи
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

# Микросообщения
users = User.order(:created_at).take(6)
50.times do
  content = Faker::Lorem.sentence(5)
  users.each { |user| user.microposts.create!(content: content) }
end

# Взаимоотношения следования
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) }
```

Чтобы выполнить этот код, перезапустим заполнение базы данных, как обычно:

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

12.2.2. Статистика и форма для оформления следования

Теперь, после образования связей между читателями и читаемыми, отразим их на главной странице и в профиле. Начнем с создания шаблона отображения статистики. Затем добавим форму «подписаться/отписаться» и создадим отдельные страницы для вывода читаемых и читателей.

Как отмечалось в разделе 12.1.1, мы последуем за примером Twitter и используем слово «following» (читаемые) в качестве метки для пользователей, чьи сообщения читает данный пользователь, например «50 following» (50 читаемых), как показано на макетах, начиная с рис. 12.1, она же показана крупным планом на рис. 12.10.

Рис. 12.10 ❖ Макет частичного шаблона для вывода статистики

В число отображаемых статистик входит количество читаемых и читателей. Каждое из этих чисел должно быть оформлено в виде ссылки на соответствующую страницу со списком. В главе 5 мы временно «заглушали» подобные ссылки знаком `'#'`, но это было до того, как мы набрались достаточного опыта работы с маршрутами. Сейчас, несмотря на то что сами страницы будут отложены до раздела 12.2.3, мы определим маршруты, как показано в листинге 12.15. В нем впервые появляется метод `:member` внутри *блока* `resources`. Проверьте себя – сможете ли вы понять, что он делает?

Листинг 12.15 ❖ Добавление методов following и followers в контроллер Users (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
```

Можно догадаться, что адреса URL читаемых и читателей будут иметь вид: «/users/1/following» и «/users/1/followers». Именно такие адреса определяет код в листинге 12.15. Поскольку обе страницы будут *отображать* данные, соответствующим HTTP-глаголом для них будет запрос `GET`, поэтому применим метод `get` в реализации реакции на запрос. Собственно, реализация ответа на HTTP-запрос к адресу URL с идентификатором пользователя находится в методе `member`. Другой метод, `collection`, не требует идентификатора, то есть

```
resources :users do
  collection do
    get :tigers
  end
end
```

будет отвечать на URL вида /users/tigers (по-видимому, чтобы вывести список всех тигров)[1]. Таблица маршрутов, сгенерированная кодом в листинге 12.15, представлена в табл. 12.2. Обратите внимание на эти именованные маршруты – мы очень скоро ими воспользуемся.

Таблица 12.2 ❖ RESTful-маршруты, созданные правилами для ресурса в листинге 12.15

HTTP-запрос	URL	Метод	Именованный маршрут
`GET`	/users/1/following	`following`	`following_user_path(1)`
`GET`	/users/1/followers	`followers`	`followers_user_path(1)`

Покончив с маршрутами, определим шаблон отображения статистики с парой ссылок внутри элемента `div` (листинг 12.16).

Листинг 12.16 ❖ Шаблон для отображения статистики (app/views/shared/_stats.html.erb)

```
<% @user ||= current_user %>
<div class="stats">
  <a href="<%= following_user_path(@user) %>">
    <strong id="following" class="stat">
      <%= @user.following.count %>
    </strong>
    following
  </a>
  <a href="<%= followers_user_path(@user) %>">
    <strong id="followers" class="stat">
      <%= @user.followers.count %>
    </strong>
    followers
  </a>
</div>
```

[1] Дополнительную информацию о параметрах маршрутизации можно найти в статье «Rails Routing from the Outside In» (http://guides.rubyonrails.org/routing.html).

Поскольку статистика будет включена в страницу профиля пользователя и в главную страницу, первая строка выбирает верное значение с помощью

```
<% @user ||= current_user %>
```

Как отмечалось в блоке 8.1, этот код ничего не делает, если `@user` содержит значение, отличное от `nil` (как в профиле), в противном случае (как в главной странице) этот код присваивает `@user` текущего пользователя. Обратите внимание, что количество читаемых/читателей подсчитывается через связи с помощью

```
@user.following.count
```

и

```
@user.followers.count
```

Сравните это с подсчетом количества микросообщений в листинге 11.23:

```
@user.microposts.count
```

Для большей эффективности Rails подсчитывает количество прямо в базе данных.

И последняя деталь, которую стоит отметить, – наличие атрибута `id` у некоторых элементов, например:

```
<strong id="following" class="stat">
...
</strong>
```

Это сделано в угоду Ajax-реализации из раздела 12.2.5, которая получает доступ к элементам страницы по уникальным значениям их атрибутов `id`.

С готовым шаблоном добавить статистику в главную страницу проще простого (листинг 12.17).

Листинг 12.17 ❖ Вывод статистики на главной странице (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 %>
```

Для оформления блока со статистиками добавим несколько правил SCSS (листинг 12.18, он содержит все правила, необходимые в этой главе). Получившаяся главная страница изображена на рис. 12.11.

Листинг 12.18 ❖ Правила SCSS для боковой панели на главной странице (app/assets/stylesheets/custom.css.scss)

```
.
.
.
/* боковая панель */
.
.
.
.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;
}

/* формы */
.
.
.
```

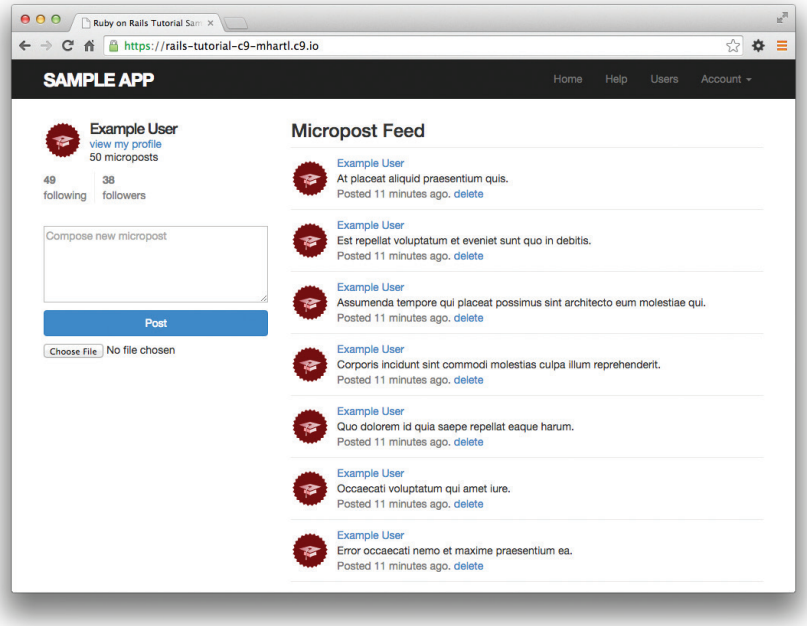


Рис. 12.11 ❖ Главная страница со статистикой

Мы подключим статистику к странице профиля чуть ниже, но сначала давайте напишем шаблон для кнопки «follow/unfollow» (подписаться/отписаться), как показано в листинге 12.19.

Листинг 12.19 ❖ Шаблон формы «подписаться/отписаться» (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 %>
```

Он ничего не делает, перекладывая фактическую работу на шаблоны `follow` и `unfollow`, для которых нам понадобится определить новые маршруты к ресурсу Relationships, подобные аналогичным маршрутам к ресурсу Microposts (листинг 11.29), как показано в листинге 12.20.

Листинг 12.20 ❖ Новые маршруты для взаимоотношений пользователей (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
```

Сами шаблоны `follow` и `unfollow` представлены в листингах 12.21 и 12.22.

Листинг 12.21 ❖ Форма оформления подписки на получение сообщений пользователя (app/views/users/_follow.html.erb)

```
<%= form_for(current_user.active_relationships.build) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
```

```
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

Листинг 12.22 ❖ Форма отмены подписки на получение сообщений пользователя (app/views/users/_unfollow.html.erb)

```
<%= form_for(current_user.active_relationships.find_by(followed_id: @user.id),
             html: { method: :delete }) do |f| %>
  <%= f.submit "Unfollow", class: "btn" %>
<% end %>
```

В обеих формах использован метод `form_for` для работы с объектом модели Relationship; основное отличие между ними в том, что форма в листинге 12.21 создает (`build`) новое взаимоотношение, а форма в листинге 12.22 ищет существующее. Соответственно, первый из них посылает в контроллер Relationships запрос `POST`, чтобы создать взаимоотношение, а второй – запрос `DELETE`, чтобы уничтожить его. (Сами эти методы описываются в разделе 12.2.4.) Наконец, обратите внимание, что форма подписки не содержит ничего, кроме самой кнопки, тем не менее она должна посылать `followed_id` в контроллер. Для этого нам понадобился метод `hidden_field_tag` в листинге 12.21, результатом работы которого является разметка HTML:

```
<input id="followed_id" name="followed_id" type="hidden" value="3" />
```

Как мы уже видели в разделе 10.2.4 (листинг 10.50), скрытый тег `input` добавляет соответствующую информацию на страницу, не отображая ее в браузере.

Теперь добавим форму подписки и блок статистики на страницу профиля пользователя, просто передав шаблоны в представление, как показано в листинге 12.23. Профиль с кнопками **Follow** и **Unfollow** показан на рис. 12.12 и рис. 12.13 соответственно.

Листинг 12.23 ❖ Добавление формы подписки и блока статистики на страницу профиля (app/views/users/show.html.erb)

```
<% provide(:title, @user.name) %>
<div class="row">
  <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? %>
```

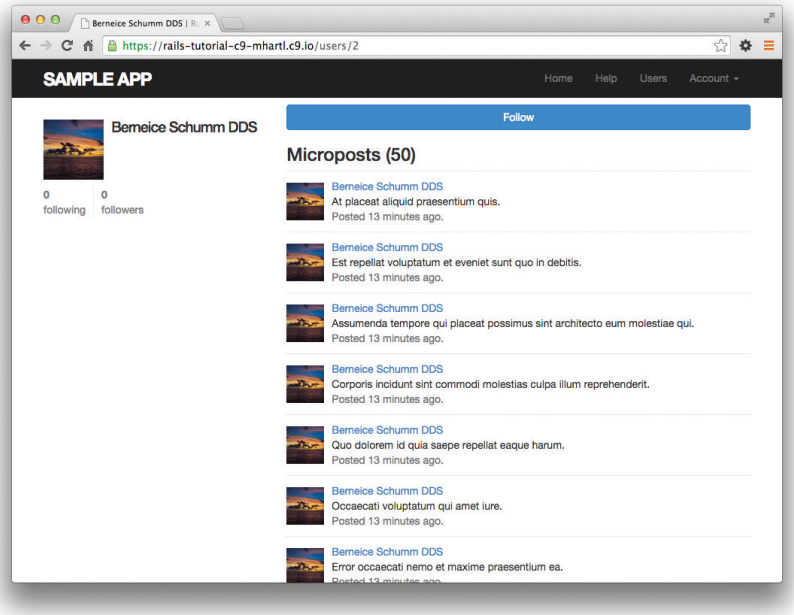


Рис. 12.12 ❖ Профиль пользователя с кнопкой **Follow** (/users/2)

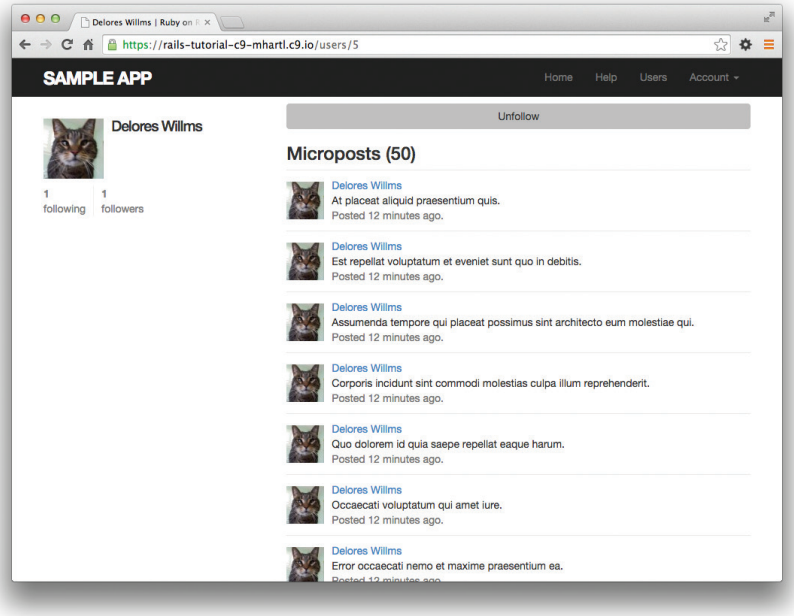


Рис. 12.13 ❖ Страница профиля с кнопкой **Unfollow** (/users/5)

```
      <h3>Microposts (<%= @user.microposts.count %>)</h3>
      <ol class="microposts">
        <%= render @microposts %>
      </ol>
      <%= will_paginate @microposts %>
    <% end %>
  </div>
</div>
```

Очень скоро мы заставим работать эти кнопки, причем сделаем это двумя способами: обычным (раздел 12.2.4) и используя технологию Ajax (раздел 12.2.5). Но сначала закончим HTML-интерфейс, создав страницы со списками читаемых и читателей.

12.2.3. Страницы читаемых и читателей

Страницы со списками читаемых и читателей будут напоминать гибрид страницы профиля пользователя и страницы со списком пользователей (раздел 9.3.1) – на них будут присутствовать боковая панель с информацией о пользователе (включая количество читаемых и читающих пользователей) и список всех пользователей. Кроме того, в боковую панель на этих страницах мы добавим блок маленьких изображений из профилей пользователей, ведущих к их домашним страницам. Макет, отвечающий всем этим требованиям, показан на рис. 12.14 (читаемые) и рис. 12.15 (читатели).

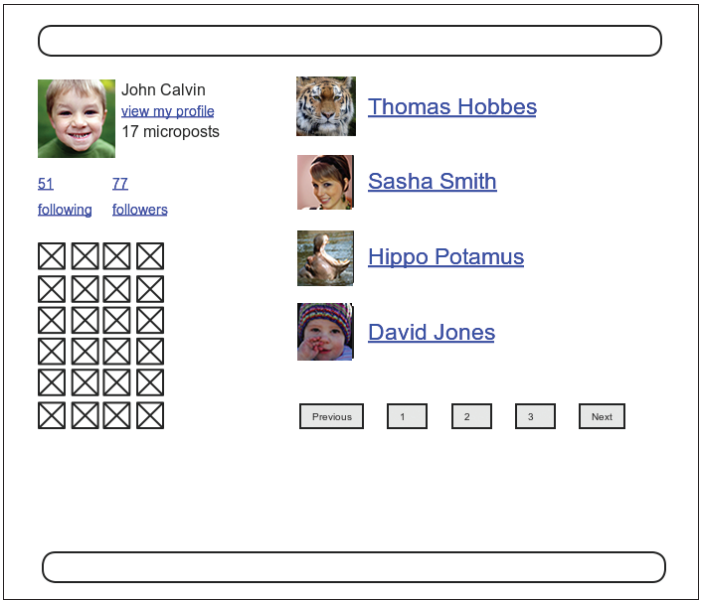


Рис. 12.14 ❖ Макет страницы со списком читаемых пользователей

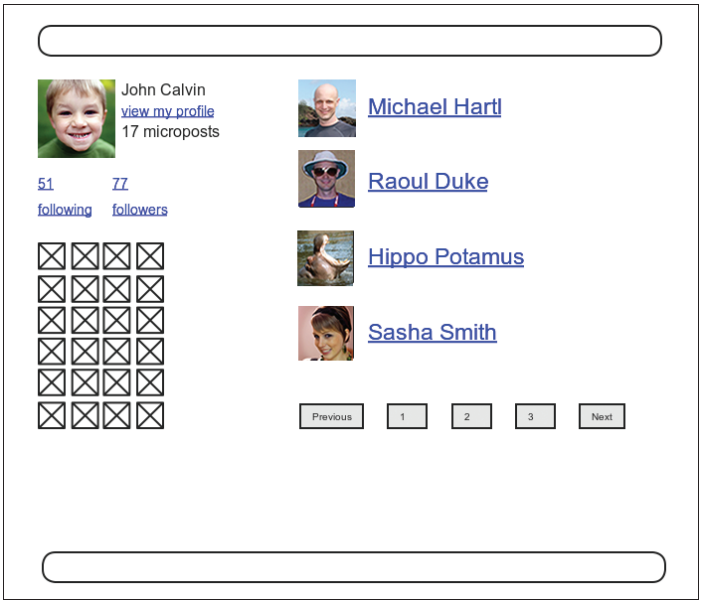


Рис. 12.15 ❖ Макет страницы со списком читающих пользователей

Для начала реализуем ссылки на профили читаемых и читающих пользователей. Последуем примеру Twitter и потребуем авторизации пользователя для доступа к обеим этим страницам. Как и в большинстве предыдущих примеров, сначала напишем тесты (листинг 12.24).

Листинг 12.24 ❖ Тесты авторизации для доступа к страницам со списками читаемых и читателей **КРАСНЫЙ** (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
```

Единственная сложность – необходимость создания двух новых методов в контроллере Users. Опираясь на маршруты из листинга 12.15, назовем их `following` и `followers`. Каждый метод должен определять заголовок страницы, находить пользователя, извлекать из базы `@user.following` или `@user.followers` (постранично) и затем отображать страницу. Результат представлен в листинге 12.25.

Листинг 12.25 ❖ Действия following и 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
```

Как мы неоднократно видели, Rails обычно неявно отображает шаблон, соответствующий методу, например `show.html.erb` в конце метода `show`. Напротив, оба этих метода явно вызывают `render`, в данном случае для отображения представления `show_follow`, которое предстоит создать. Идея реализации общего представления выбрана потому, что код ERb в этих двух случаях практически идентичен, и листинг 12.26 охватывает их оба.

Листинг 12.26 ❖ Представление show_follow для отображения списков читаемых и читателей (app/views/users/show_follow.html.erb)

```
<% provide(:title, @title) %>
<div class="row">
  <aside class="col-md-4">
```

```
    <section class="user_info">
      <%= gravatar_for @user %>
      <h1><%= @user.name %></h1>
      <span><%= link_to "view my profile", @user %></span>
      <span><b>Microposts:</b> <%= @user.microposts.count %></span>
    </section>
    <section class="stats">
      <%= render 'shared/stats' %>
      <% if @users.any? %>
        <div class="user_avatars">
          <% @users.each do |user| %>
            <%= link_to gravatar_for(user, size: 30), user %>
          <% end %>
        </div>
      <% end %>
    </section>
  </aside>
  <div class="col-md-8">
    <h3><%= @title %></h3>
    <% if @users.any? %>
      <ul class="users follow">
        <%= render @users %>
      </ul>
      <%= will_paginate %>
    <% end %>
  </div>
</div>
```

Методы в листинге 12.25 отображают представление из листинга 12.26 в двух контекстах, «читаемые» и «читатели» (рис. 12.16 и 12.17). Обратите внимание, что в коде нигде не используются ссылки на текущего пользователя, поэтому они будут работать для любых других пользователей, как показано на рис. 12.18.

Теперь, когда у нас есть действующие страницы со списками читаемых и читателей, самое время написать пару коротких интеграционных тестов, чтобы проверить их поведение. Тесты просто проверяют работоспособность страниц и не осуществляют всестороннего их исследования. Действительно, как отмечалось в разделе 5.3.4, обстоятельные тесты, например структуры разметки HTML, могут оказаться весьма хрупкими, а потому контрпродуктивными. Мы проверим отображение на страницах корректных значений счетчиков и ссылок с верными адресами URL.

Для начала, как обычно, создадим файл интеграционных тестов:

```
$ rails generate integration_test following
      invoke  test_unit
      create    test/integration/following_test.rb
```

Далее нам понадобится немного тестовых данных, для этого добавим несколько взаимоотношений, чтобы создать связи читаемые/читатели. Как рассказывалось в разделе 11.2.3, это можно сделать так:

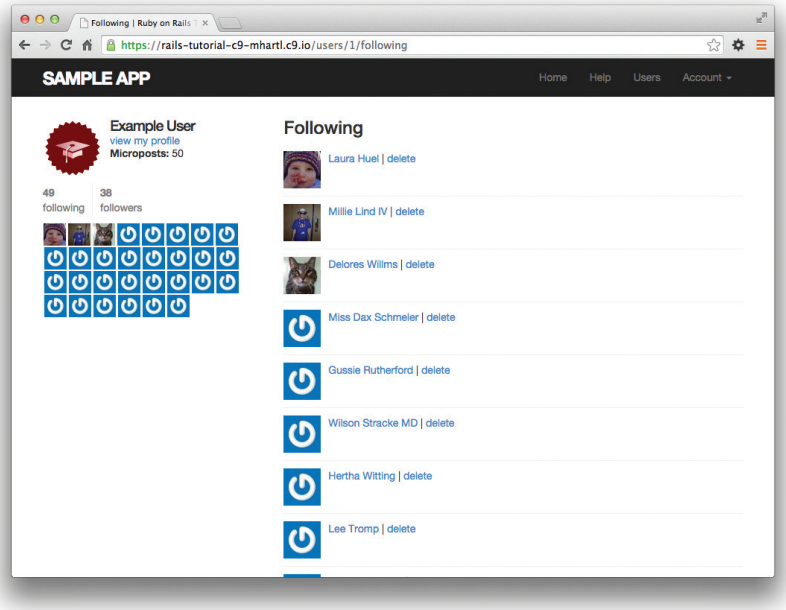


Рис. 12.16 ❖ Отображение списка читаемых пользователей

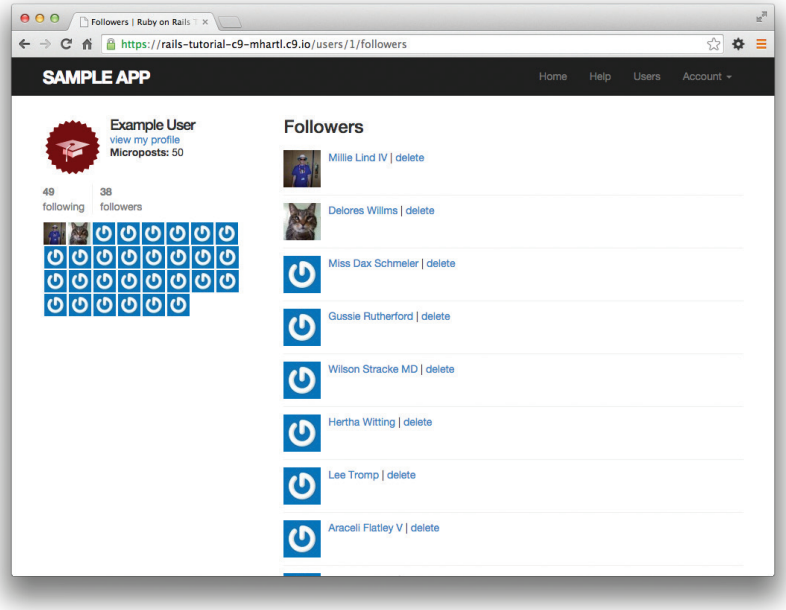


Рис. 12.17 ❖ Отображение списка читателей данного пользователя

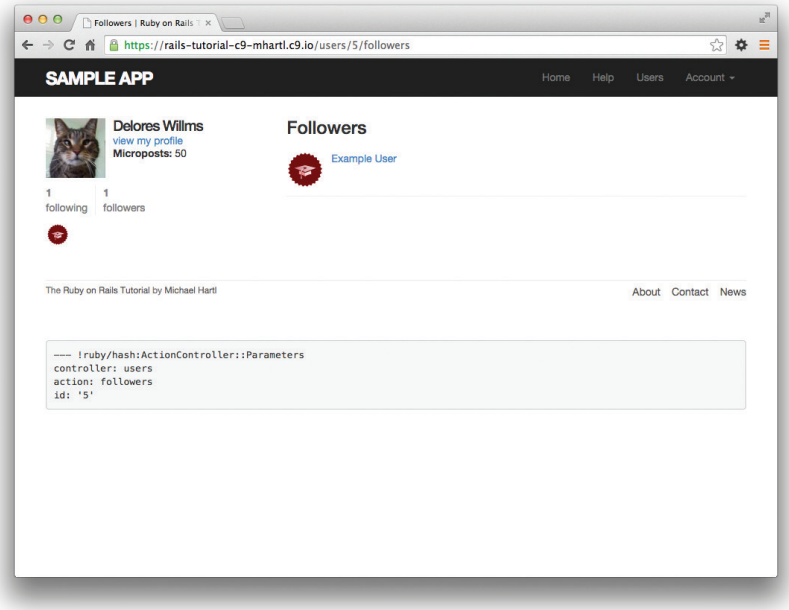


Рис. 12.18 ❖ Отображение списка читателей другого пользователя

```
orange:
  content: "Ijustateanorange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

и таким способом связать микросообщения с конкретным пользователем. Здесь можно писать

```
user: michael
```

вместо

```
user_id: 1
```

Применив эту идею к тестовым данным, описывающим взаимоотношения, получим определения, как показано в листинге 12.27.

Листинг 12.27 ❖ Данные о взаимоотношениях для использования в тестах «читаемые/читатели» (test/fixtures/relationships.yml)

```
one:
  follower: michael
  followed: lana

two:
  follower: michael
  followed: mallory
```

```
three:
  follower: lana
  followed: michael

four:
  follower: archer
  followed: michael
```

Здесь мы указали следующее: Майкл (michael) читает сообщения Ланы (lana) и Мэлори (mallory), а Лана (lana) и Арчер (archer) читают сообщения Майкла (michael). Для проверки значений счетчиков можно взять тот же метод `assert_match`, что использован в листинге 11.27 для проверки количества микросообщений на странице профиля пользователя. Добавим утверждения о корректности ссылок и получим тесты, представленные в листинге 12.28.

Листинг 12.28 ❖ Тесты для страниц «читаемые/читатели» **ЗЕЛЕНЫЙ** (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

  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
```

Обратите внимание на утверждение

```
assert_not @user.following.empty?
```

которое включено, чтобы гарантировать, что

```
@user.following.each do |user|
  assert_select "a[href=?]", user_path(user)
end
```

не является бессодержательной истиной (аналогичное утверждение используется в тесте для читателей).

Теперь набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 12.29 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

12.2.4. Стандартная реализация кнопки «Подписаться»

Теперь, когда представления приведены в порядок, пришло время реализовать кнопки **Follow**/**Unfollow** (Подписаться/Отписаться). Так как подписка и отказ от подписки на сообщения другого пользователя подразумевают создание и уничтожение взаимоотношений, нам понадобится контроллер Relationships, который мы сгенерируем, как обычно:

```
$ rails generate controller Relationships
```

Как будет показано в листинге 12.31, контроль доступа на уровне методов контроллера Relationships не имеет особого значения, но мы по-прежнему стараемся придерживаться практики воплощения модели безопасности на самых ранних этапах. В частности, проверим, что при попытке получить доступ к методам контроллера Relationships неавторизованный пользователь переадресуется на страницу входа, а количество взаимоотношений при этом не изменяется (листинг 12.30).

Листинг 12.30 ❖ Тесты механизма контроля доступа для взаимоотношений **КРАСНЫЙ** (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
```

Для успешного выполнения этих тестов достаточно добавить предварительный фильтр `logged_in_user` (листинг 12.31).

Листинг 12.31 ❖ Контроль доступа для взаимоотношений **ЗЕЛЕНЫЙ** (app/controllers/relationships_controller.rb)

```
class RelationshipsController < ApplicationController
  before_action :logged_in_user

  def create
  end

  def destroy
  end
end
```

Чтобы кнопки **Follow** (Подписаться) и **Unfollow** (Отписаться) заработали, достаточно найти пользователя, связанного с `followed_id` в соответствующей форме (листинг 12.21 или 12.22), а затем вызвать метод `follow` или `unfollow` из листинга 12.10. Полная реализация представлена в листинге 12.32.

Листинг 12.32 ❖ Контроллер 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
```

Теперь понятно, почему упомянутый вопрос безопасности не имеет особого значения: если неавторизованный пользователь попробует напрямую обратиться к любому из этих методов (например, с помощью утилиты `curl`), переменная `current_user` будет иметь значение `nil`, и вторая строка метода вызовет исключение, которое приведет к ошибке, но не причинит никакого вреда приложению или его данным. Тем не менее лучше не полагаться на это, поэтому мы добавили дополнительный уровень безопасности.

Теперь базовая функциональность подписки завершена, и любой пользователь сможет подписаться или отписаться от сообщений любого другого пользователя; вы можете проверить это, щелкая на соответствующих кнопках в браузере. (Интеграционные тесты проверки этого поведения мы напишем в разделе 12.2.6.) Результат щелчка на кнопке **Follow** (Подписаться) для подписки на сообщения пользователя #2 показан на рис. 12.19 и 12.20.

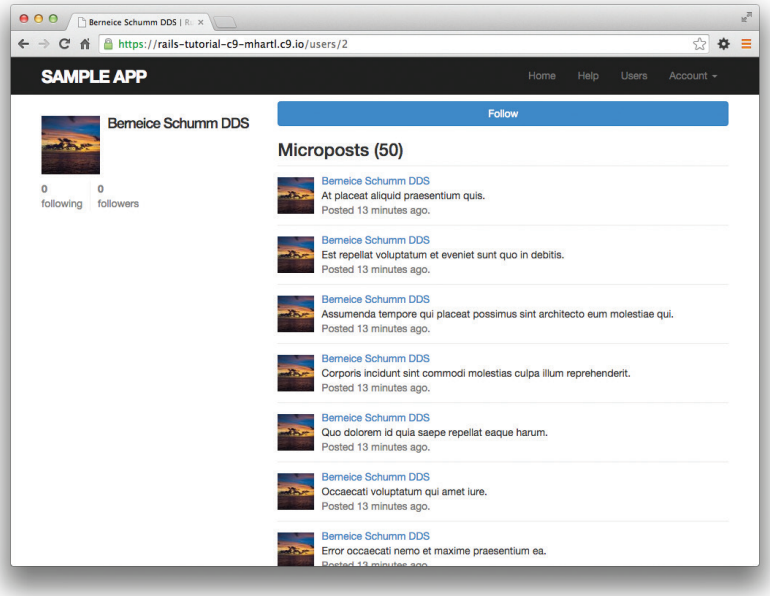


Рис. 12.19 ❖ Нечитаемый пользователь

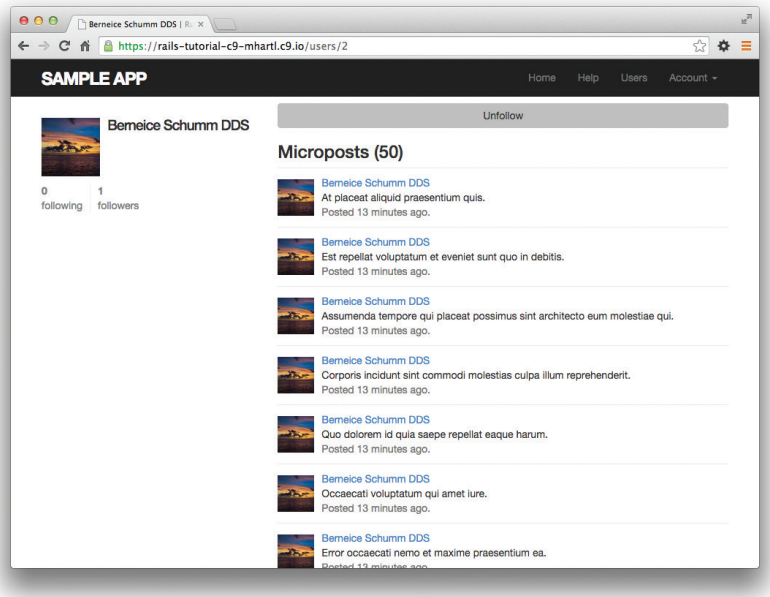


Рис. 12.20 ❖ Результат щелчка на кнопке **Follow** (Подписаться) в профиле нечитаемого пользователя

12.2.5. Реализация кнопки «Подписаться» с применением Ajax

Уже сейчас реализация подписки на сообщения пользователя имеет вполне законченный вид, нам осталось совсем немного подправить ее, прежде чем заняться лентой сообщений. В разделе 12.2.4 вы могли заметить, что методы `create` и `destroy` контроллера Relationships просто выполняют переадресацию назад, к исходному профилю. Другими словами, пользователь переходит на страницу профиля другого пользователя, щелкает на кнопке **Follow** (Подписаться) и немедленно переадресуется на исходную страницу. Возникает резонный вопрос: почему он вообще должен ее покидать?

Именно эту проблему решает технология *Ajax*[1], позволяя отправлять асинхронные запросы на сервер, не покидая страницы[2]. Поскольку практика добавления Ajax в веб-формы является довольно распространенной, фреймворк Rails пошел по пути упрощения этой задачи. Действительно, обновление частичных шаблонов в форме оформления или отказа от подписки реализуется тривиально просто: достаточно заменить

```
form_for
```

на

```
form_for ..., remote: true
```

и Rails автомагически (http://авторслова.рф/word/avtomagicheski) использует Ajax. Обновленные шаблоны показаны в листингах 12.33 и 12.34.

Листинг 12.33 ❖ Форма подписки на сообщения с использованием Ajax (app/views/users/_follow.html.erb)

```
<%= form_for(current_user.active_relationships.
                          build(followed_id: @user.id),
                          remote: true) do |f| %>
  <div><%= hidden_field_tag :followed_id, @user.id %></div>
  <%= f.submit "Follow", class: "btn btn-primary" %>
<% end %>
```

Листинг 12.34 ❖ Форма отписки от сообщений с использованием 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 %>
```

---

[1] https://ru.wikipedia.org/wiki/AJAX. – *Прим. перев.*

[2] Так как номинально Ajax – это аббревиатура от Asynchronous JavaScript and XML (асинхронный JavaScript и XML), Ajax часто пишется неверно, как «AJAX», несмотря на то что в оригинальной статье на всем ее протяжении используется название «Ajax».

Фактическая разметка HTML, сгенерированная этим кодом на ERb, не особенно относится к делу, но если вам интересно, взгляните на нее (детали могут отличаться):

```
<form action="/relationships/117" class="edit_relationship" data-remote="true"
      id="edit_relationship_117" method="post">
  .
  .
  .
</form>
```

Внутри тега `form` определяется переменная `data-remote="true"`, указывающая фреймворку Rails, что форма обрабатывается сценарием на языке JavaScript. Используя простое свойство HTML вместо вставки полного JavaScript-кода (как в предыдущих версиях Rails), Rails следует философии *ненавязчивого JavaScript*[1].

После обновления формы нужно «уговорить» контроллер Relationships отвечать на Ajax-запросы. В этом нам поможет метод `respond_to`, реагирующий в зависимости от типа запроса. Общий шаблон его использования выглядит вот так:

```
respond_to do |format|
  format.html { redirect_to user }
  format.js
end
```

Синтаксис может показаться странным, поэтому важно понять, что здесь выполняется только *одна* строка. (В этом смысле `respond_to` ближе к инструкции `if-else`, чем к последовательности строк.) Подготовка контроллера Relationships к реакции на запросы Ajax подразумевает добавление вызова `respond_to` в методы `create` и `destroy` (листинг 12.32). Получившийся результат представлен в листинге 12.35. Обратите внимание на замену локальной переменной `user` на переменную экземпляра `@user`; она не нужна была в листинге 12.32, но теперь стала необходима для форм в листингах 12.33 и 12.34.

Листинг 12.35 ❖ Обработка Ajax-запросов в контроллере 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
```

[1] https://ru.wikipedia.org/wiki/Ненавязчивый_JavaScript. – *Прим. перев.*

```
  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
```

Эти методы могут служить примером *постепенной деградации* (https://htmlacademy.ru/blog/6-graceful-degradation). Это значит, что они будут прекрасно работать, даже если в браузере отключить поддержку JavaScript (хотя для этого понадобится еще немного настроек, листинг 12.36).

Листинг 12.36 ❖ Настройки для поддержки постепенной деградации при отправке формы (config/application.rb)

```
require File.expand_path('../boot', __FILE__)
.
.
.
module SampleApp
  class Application < Rails::Application
    .
    .
    .
    # Включать токен аутентификации в удаленные формы.
    config.action_view.embed_authenticity_token_in_remote_forms = true
  end
end
```

С другой стороны, форма обязана должным образом реагировать при включенной поддержке JavaScript. Чтобы выполнить Ajax-запрос, Rails автоматически вызывает файл с кодом на *JavaScript и встроенном Ruby* (`.js.erb`) и именем, совпадающим с именем метода, то есть `create.js.erb` или `destroy.js.erb`. Как нетрудно догадаться, эти файлы могут содержать смешанный код на JavaScript и встроенном Ruby, выполняющий операции с текущей страницей. Именно эти файлы нужно создать и заполнить, чтобы обеспечить обновление страницы профиля пользователя, когда кто-то подписывается на его сообщения или отписывается от них.

Внутри файла JS-ERb фреймворк Rails автоматически предоставляет поддержку jQuery (http://jquery.com/) для работы со страницей. Библиотека jQuery (мы познакомились с ней в разделе 11.4.2) предоставляет большое количество методов для выполнения манипуляций с DOM, но здесь нам понадобятся только два из них. Во-первых, нам потребуется использовать синтаксис «знак доллара» для доступа к DOM-элементу на основе уникального значения его атрибута `id`. Например, если обращение к элементу `follow_form` мы запишем так:

```
$("#follow_form")
```

(В листинге 12.19 этому значению атрибута id соответствует элемент div, в который упакована форма, а не сама форма.) Этот синтаксис был придуман по аналогии с использованием символа # в CSS, где он соответствует атрибуту id. Как можно догадаться, библиотека jQuery, так же как CSS, использует точку . для работы с CSS-классами.

Второй необходимый метод – html, который обновляет разметку HTML внутри соответствующего элемента содержимым своего аргумента. Например, полную замену формы подписки строкой "foobar" можно выполнить так:

```
$("#follow_form").html("foobar")
```

В отличие от простых файлов JavaScript, файлы JS-ERb позволяют также использовать встроенный Ruby, чем мы и воспользуемся в файле create.js.erb для замены формы подписки на шаблон unfollow (он должен быть отображен после успешной подписки) и обновления счетчика читателей. Результат показан в листинге 12.37. Здесь использован метод escape_javascript, выполняющий экранирование результата при вставке разметки HTML в файл JavaScript.

Листинг 12.37 ❖ Код на JavaScript и встроенном Ruby для создания взаимоотношения (app/views/relationships/create.js.erb)

```
$("#follow_form").html("<%= escape_javascript(render('users/unfollow')) %>");
$("#followers").html('<%= @user.followers.count %>');
```

Обратите внимание на точки с запятой в конце строк, это характерная черта синтаксиса языков, произошедших от ALGOL.

Файл destroy.js.erb содержит похожий код (листинг 12.38).

Листинг 12.38 ❖ Ruby JavaScript (RJS) для уничтожения взаимоотношения (app/views/relationships/destroy.js.erb)

```
$("#follow_form").html("<%= escape_javascript(render('users/follow')) %>");
$("#followers").html('<%= @user.followers.count %>');
```

Теперь перейдите на страницу профиля пользователя и проверьте возможность подписаться на сообщения другого пользователя и отписаться от них без обновления страницы.

12.2.6. Тестирование подписки

Теперь, когда мы получили действующие кнопки **Follow** (Подписаться) и **Unfollow** (Отписаться), напишем несколько простых тестов для предотвращения регрессий. Чтобы протестировать подписку на сообщения пользователя, нужно отправить запрос POST по маршруту relationships_path и проверить, увеличилось ли количество читаемых пользователей на 1:

```
assert_difference '@user.following.count', 1 do
  post relationships_path, followed_id: @other.id
end
```

Так тестируется стандартная реализация. Тестирование Ajax-версии выполняется почти так же, только вместо обычного метода `post` вызывается `xhr :post`:

```
assert_difference '@user.following.count', 1 do
  xhr :post, relationships_path, followed_id: @other.id
end
```

Метод `xhr` (от XmlHttpRequest) выполняет Ajax-запрос, в результате чего блок `respond_to` в листинге 12.35 вызывает верный метод JavaScript.

Аналогично выглядит реализация уничтожения взаимоотношений в методе `delete`, только вызов `post` заменяется на вызов `delete`. Мы проверяем уменьшение количества читаемых пользователей на 1 при удалении взаимоотношения по идентификатору читаемого пользователя:

```
assert_difference '@user.following.count', -1 do
  delete relationship_path(relationship),
         relationship: relationship.id
end
```

и

```
assert_difference '@user.following.count', -1 do
  xhr :delete, relationship_path(relationship),
               relationship: relationship.id
end
```

Собрав вместе оба случая, получаем тесты, представленные в листинге 12.39.

Листинг 12.39 ❖ Тесты для кнопок **Follow** (Подписаться) и **Unfollow** (Отписаться) **ЗЕЛЕНЫЙ** (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
```

Набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 12.40 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

12.3. Лента сообщений

Мы подходим к кульминации учебного приложения – ленте сообщений. Соответственно, этот раздел содержит наиболее продвинутый материал во всей книге. Полноценная лента сообщений базируется на ее прототипе из раздела 11.3.3 и собирает массив сообщений читаемых пользователей и собственных сообщений текущего пользователя. На протяжении этого раздела мы рассмотрим ряд реализаций ленты сообщений с постепенным увеличением их сложности. Чтобы совершить этот подвиг, нам понадобятся довольно сложные приемы программирования в Rails, Ruby и даже SQL.

Так как нам предстоит тяжелая работа, особенно важно понимать, куда мы будем двигаться. Окончательный макет ленты сообщений, представленный на рис. 12.5, еще раз демонстрируется на рис. 12.21.

12.3.1. Мотивация и стратегия

Основная идея ленты сообщений проста. На рис. 12.22 приводятся пример таблицы `microposts` в базе данных и полученная в результате лента. Основная цель ленты состоит в том, чтобы извлечь из базы данных сообщения, принадлежащие пользователям, которых читает текущий пользователь (и самого текущего пользователя), как указано стрелками на схеме.

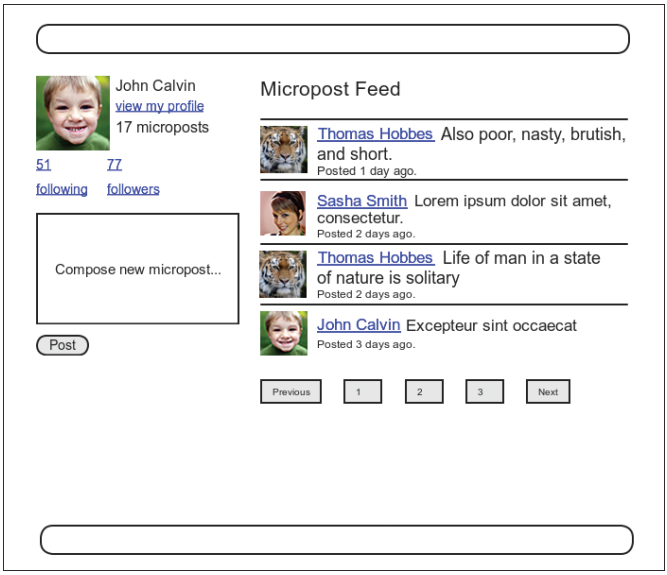


Рис. 12.21 ❖ Макет главной страницы пользователя с лентой сообщений

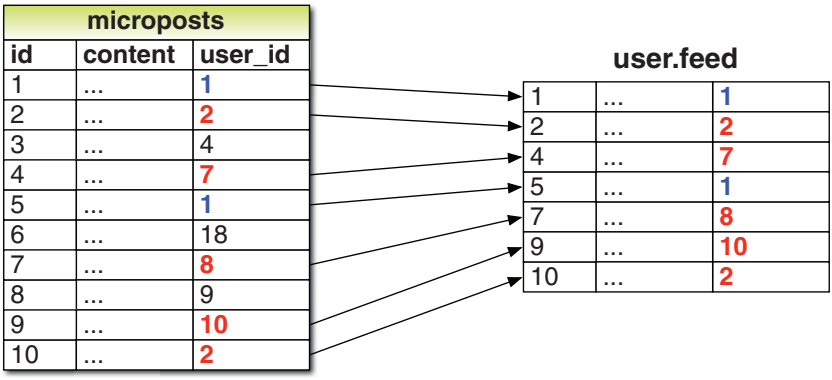


Рис. 12.22 ❖ Лента сообщений пользователя (с id = 1), читающего сообщения пользователей с id = 2, 7, 8 и 10

Мы пока не знаем, как реализовать ленту, но уже можем написать простые тесты. Поэтому (следуя рекомендациям из блока 3.3) сначала напишем именно их. Ключевым моментом является проверка всех трех требований к ленте: она должна включать сообщения читаемых пользователей и текущего пользователя, но в то же время туда не должны попадать сообщения нечитаемых пользователей. Если взять за основу тестовые данные из листингов 9.43 и 11.51, это будет означать, что Майкл должен видеть сообщения Ланы и свои собственные, но не должен видеть сообщения Арчера. Преобразовав эти требования в утверждения и вспомнив, что

лента находится в модели User (листинг 11.44), получаем обновленные тесты модели User (листинг 12.41).

Листинг 12.41 ❖ Тесты ленты сообщений **КРАСНЫЙ** (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)
    # Сообщения читаемого пользователя
    lana.microposts.each do |post_following|
      assert michael.feed.include?(post_following)
    end
    # Собственные сообщения
    michael.microposts.each do |post_self|
      assert michael.feed.include?(post_self)
    end
    # Сообщения нечитаемого пользователя
    archer.microposts.each do |post_unfollowed|
      assert_not michael.feed.include?(post_unfollowed)
    end
  end
end
```

Пока у нас имеется лишь прототип ленты, поэтому набор тестов сейчас **КРАСНЫЙ**.

12.3.2. Первая реализация ленты сообщений

После того как требования к проекту ленты сообщений были зафиксированы в тестах, можно приступать к ее реализации. Поскольку конечная реализация довольно сложная, мы будем строить ее поэтапно, добавляя по одному фрагменту за раз. В первую очередь следует обдумать запрос к базе данных. Мы должны выбрать из таблицы все микросообщения, поле `user_id` которых соответствует идентификаторам пользователей, читаемых текущим пользователем (а также идентификатору самого текущего пользователя). Схематично это можно записать так:

```
SELECT * FROM microposts
WHERE user_id IN (<list of ids>) OR user_id = <user id>
```

Здесь мы предположили, что SQL поддерживает ключевое слово `IN`, позволяющее проверить множественное включение. (К счастью, это так.)

Как вы наверняка помните, в предварительной реализации ленты (раздел 11.3.3) механизм Active Record использует метод `where` для осуществления выборки та-

кого вида, как показано в листинге 11.44. Тогда выборка была очень простой; мы просто выбрали все сообщения со значением в поле `user_id`, соответствующим текущему пользователю:

```
Micropost.where("user_id = ?", id)
```

Теперь предполагается, что выборка будет выполняться сложнее, например:

```
Micropost.where("user_id IN (?) OR user_id = ?",
                following_ids, id)
```

Из этих условий видно, что нам понадобится массив идентификаторов читаемых пользователей. Получить такой массив можно с помощью Ruby-метода `map`, имеющегося у любого «перечислимого» объекта, то есть у любого объекта (такого как массив или хэш), который состоит из коллекции элементов[1]. Пример применения этого метода был показан в разделе 4.3.2; как еще один пример применим `map` для преобразования массива чисел в массив строк:

```
$ rails console
>> [1, 2, 3, 4].map { |i| i.to_s }
=> ["1", "2", "3", "4"]
```

Ситуации, когда один и тот же метод вызывается для каждого элемента коллекции, настолько часто встречаются в практике, что для них существует сокращенная форма записи (мы уже видели ее в разделе 4.3.2), использующая *амперсанд* `&` и символ, соответствующий методу:

```
>> [1, 2, 3, 4].map(&:to_s)
=> ["1", "2", "3", "4"]
```

С помощью метода `join` (раздел 4.3.1) можно создать строку, состоящую из идентификаторов, перечисленных через запятую-пробел:

```
>> [1, 2, 3, 4].map(&:to_s).join(', ')
=> "1, 2, 3, 4"
```

Мы можем использовать этот метод для построения массива идентификаторов читаемых пользователей, вызвав метод `id` каждого элемента в `user.following`. Например, для первого пользователя в базе данных этот массив выглядит так:

```
>> 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]
```

На самом деле из-за высокой востребованности такой конструкции механизм Active Record поддерживает ее по умолчанию:

---

[1] Основное требование – перечислимый объект должен поддерживать метод `each` для обхода коллекции.

```
>> 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]
```

Метод following_ids синтезирован библиотекой Active Record на основании связи has_many :following (листинг 12.8); в результате достаточно лишь присоединить _ids к имени связи, и мы получим идентификаторы, соответствующие коллекции user.following. Строка из идентификаторов читаемых пользователей будет выглядеть так:

```
>> 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"
```

Впрочем, эту строку не требуется вставлять в SQL-запрос; оператор интерполяции ? позаботится об этом за вас (и попутно устранит некоторые несовместимости, связанные с базой данных). Это значит, что мы можем просто вызвать метод following_ids.

В результате первоначальное предположение

```
Micropost.where("user_idIN (?) OR user_id = ?", following_ids, id)
```

действительно верно! Результат представлен в листинге 12.43.

Листинг 12.43 ❖ Первый вариант действующей ленты сообщений **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  # Возвращает true, если время для сброса пароля истекло.
  def password_reset_expired?
    reset_sent_at < 2.hours.ago
  end

  # Возвращает ленту сообщений.
  def feed
    Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
  end

  # Выполняет подписку на сообщения пользователя.
  def follow(other_user)
    active_relationships.create(followed_id: other_user.id)
  end
  .
  .
  .
end
```

Набор тестов должен быть **ЗЕЛЕНЫМ**:

Листинг 12.44 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

В некоторых приложениях этой начальной реализации было бы достаточно, но в листинге 12.43 представлен не окончательный результат; догадываетесь, почему? (*Подсказка*: представьте, что пользователь подписался на сообщения 5000 других пользователей.)

12.3.3. Подзапросы

Как отмечалось в предыдущем разделе, текущая реализация ленты сообщений плохо масштабируется при большом количестве сообщений, что легко может произойти, если пользователь подпишется на сообщения, скажем, 5000 других пользователей. В этом разделе мы повторно реализуем ленту так, чтобы улучшить масштабируемость при увеличении количества читаемых пользователей.

Проблема версии из раздела 12.3.2 в том, что `following_ids` извлекает идентификаторы *всех* читаемых пользователей и создает полный массив в памяти. Поскольку условие в листинге 12.43 лишь проверяет включение во множество, должен иметься более эффективный способ, и язык SQL действительно включает оптимизации для таких множественных операций. Решение заключается в перенаправлении поиска идентификаторов читаемых пользователей в саму базу данных с помощью *подзапроса*.

Начнем с рефакторинга ленты и немного изменим код, как показано в листинге 12.45.

Листинг 12.45 ❖ Использование пар ключ/значение в методе where ленты сообщений **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  # Возвращает ленту сообщений.
  def feed
    Micropost.where("user_id IN (:following_ids) OR user_id = :user_id",
                    following_ids: following_ids, user_id: id)
  end
  .
  .
  .
end
```

В качестве подготовки к следующему шагу мы заменили

```
Micropost.where("user_id IN (?) OR user_id = ?", following_ids, id)
```

эквивалентом

```
Micropost.where("user_id IN (:following_ids) OR user_id = :user_id",
                following_ids: following_ids, user_id: id)
```

Синтаксис со знаком вопроса хорош, но когда *ту же* переменную нужно вставить более чем в одном месте, второй вариант выглядит удобнее.

Обсуждение выше подразумевает, что мы добавим *второе* вхождение `user_id` в SQL-запрос. В частности, мы можем заменить Ruby-код

```
following_ids
```

фрагментом SQL

```
following_ids = "SELECT followed_id FROM relationships
                 WHERE follower_id = :user_id"
```

Этот код содержит подзапрос, а весь запрос целиком для пользователя 1 будет выглядеть примерно так:

```
SELECT * FROM microposts
WHERE user_id IN (SELECT followed_id FROM relationships
                  WHERE follower_id = 1)
      OR user_id = 1
```

Этот подзапрос переносит всю логику работы со множествами в базу данных, где она реализована более эффективно.

С таким фундаментом можно построить более эффективную реализацию ленты сообщений (листинг 12.46). Так как теперь это чистый код на SQL, строка `following_ids` *интерполируется* без экранирования.

Листинг 12.46 ❖ Окончательная реализация ленты сообщений **ЗЕЛЕНЫЙ** (app/models/user.rb)

```
class User < ActiveRecord::Base
  .
  .
  .
  # Возвращает ленту сообщений.
  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
```

Получилась внушительная комбинация Rails, Ruby и SQL, но она делает свое дело, и вполне успешно:

Листинг 12.47 ❖ ЗЕЛЕНЫЙ

```
$ bundle exec rake test
```

Конечно, даже подзапрос не может масштабироваться бесконечно. Для больших сайтов почти наверняка потребуется генерировать ленту асинхронно, запуская для этого фоновый процесс, но такие приемы масштабирования выходят за рамки данного руководства.

Теперь, наконец-то, лента сообщений завершена. Вспомните раздел 11.3.3: лента сообщений уже присутствует на главной странице. Как напоминание в листинге 12.48 еще раз приводится реализация метода `home`. В главе 11 мы создали лишь прототип ленты (рис. 11.14), а теперь можем видеть ее полноценный вариант (рис. 12.23).

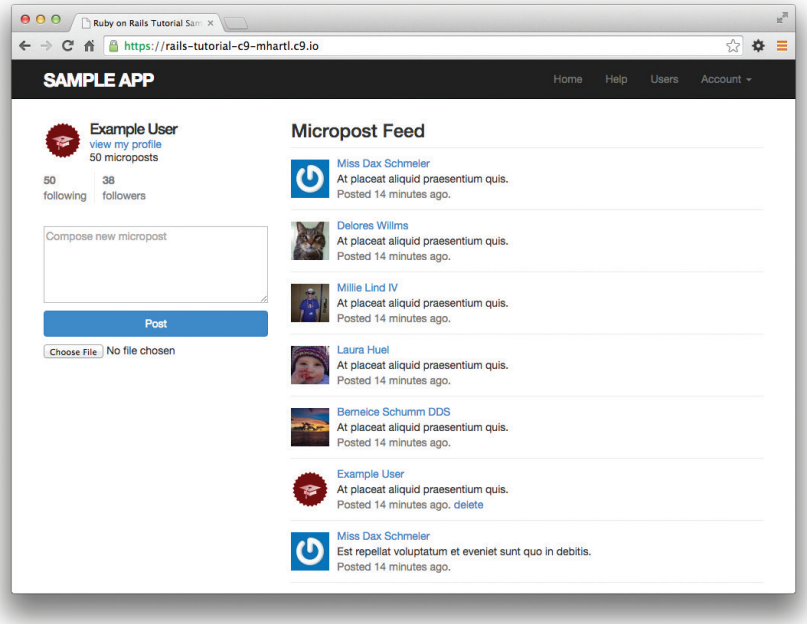


Рис. 12.23 ❖ Главная страница с лентой сообщений

Листинг 12.48 ❖ Метод home с постраничным выводом потока сообщений (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
```

Теперь можно объединить изменения с основной веткой:

```
$ bundle exec rake test
$ git add -A
$ git commit -m "Add user following"
$ git checkout master
$ git merge following-users
```

Отправить код в удаленный репозиторий и развернуть приложение в эксплуатационном окружении:

```
$ git push
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

В результате мы получили действующую ленту сообщений в Интернете (рис. 12.24).

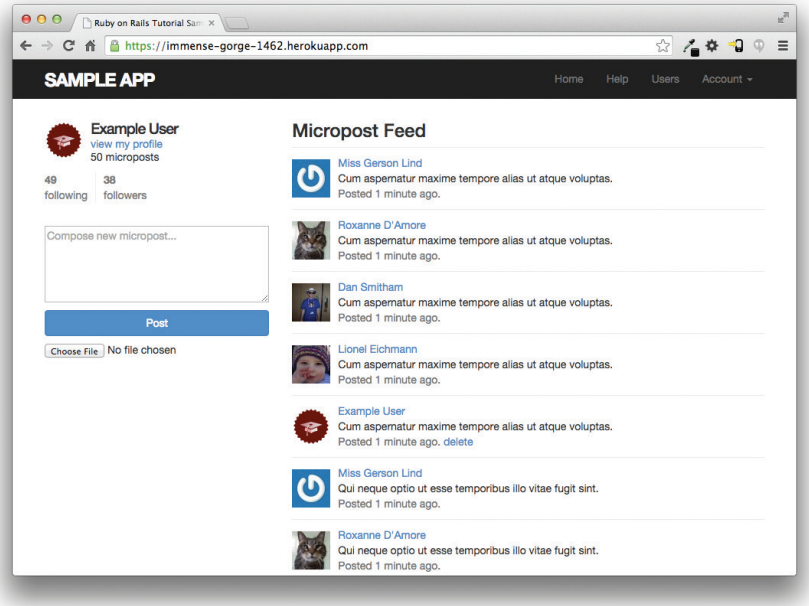


Рис. 12.24 ❖ Действующая лента сообщений в Интернете

12.4. Заключение

Добавив ленту сообщений, мы завершили разработку учебного приложения. Оно содержит примеры использования всех основных возможностей Rails, в том числе моделей, представлений, контроллеров, шаблонов, фильтров, валидаторов, функций обратного вызова, связей `has_many`/`belongs_to` и `has_many :through`, а также поддержку безопасности, тестирования и развертывания.

Несмотря на этот внушительный список, вам предстоит еще многое узнать о веб-разработке. В качестве первого шага на этом пути этот раздел содержит несколько подсказок, куда двигаться дальше.

12.4.1. Рекомендации по дополнительным ресурсам

Источников информации о Rails огромное количество – так много, что их богатство может ошеломить. Но коль скоро вы добрались до этого места, значит, вы готовы встретиться практически со всем, чем угодно. Чтобы помочь вам выбрать верный путь, ниже перечисляется несколько ресурсов, которые я рекомендовал бы исследовать в первую очередь:

- **Видеоуроки Ruby on Rails Tutorial** (https://www.railstutorial.org/#html_ebooks_and_screencast_videos): я подготовил полноценный видеокурс, основанный на этой книге. Видеоролики не только охватывают материал из этой книги, но также содержат советы, рекомендации и наглядные демонстрации, которые сложно зафиксировать в печатном варианте. Их можно приобрести через сайт «Ruby on Rails Tutorial» (https://www.railstutorial.org/).
- **Launch School** (https://launchschool.com/): в последнее время появилось множество образовательных площадок для разработчиков, и я рекомендую поискать такую недалеко от вас, но Launch School доступна в Интернете, а значит, может быть использована откуда угодно. Выбор Launch School особенно хорош, если вы ищете обратную связь с преподавателем и структурированный учебный план.
- **Turing School of Software & Design** (Школа программирования и проектирования имени Тьюринга): 27-недельная учебная программа Ruby/Rails/JavaScript с полной занятостью в Денвере, Колорадо. Большинство студентов начинает с весьма ограниченным опытом в программировании, но с достаточной решимостью и энергией, чтобы быстро его набраться. Школа Тьюринга гарантирует, что после выпуска студенты найдут работу, или им будет возвращена стоимость учебы.
- **Bloc** (https://www.bloc.io/): образовательная площадка в Интернете, структурированный учебный план, персональный подход, обучение на конкретных проектах. Читатели этой книги могут получить скидку $500 при зачислении на курс, если воспользуются кодом BLOCLOVESHARTL.
- **Firehose Project** (http://www.thefirehoseproject.com/?tid=HARTL-RAILS-TUT-EB2&pid=HARTL-RAILS-TUT-EB2): образовательная онлайн-площадка, сфокусированная на получении практических навыков программи-

рования (разработка через тестирование, алгоритмы, создание продвинутого веб-приложения в команде). Бесплатный двухнедельный вводный курс.

- **Thinkful** (https://www.thinkful.com/a/railstutorial): онлайн-обучение в паре с профессиональным инженером, работа по учебному плану, основанному на конкретном проекте. В число тем обучения входят: Ruby on Rails, верстка, веб-дизайн и работа с данными.
- **PragmaticStudio** (https://pragmaticstudio.com/refs/railstutorial): онлайн-курсы Ruby и Rails от Майка (Mike) и Николь Кларк (Nicole Clark).
- **RailsApps** (https://tutorials.railsapps.org/hartl): поучительные примеры Rails-приложений.
- **Code School** (https://my.getambassador.com/): большое разнообразие интерактивных курсов программирования.
- **Bala Paranj, разработка через тестирование на языке Ruby** (https://www.udemy.com/learn-test-driven-development-in-ruby/couponCode=hartl): продвинутый онлайн-курс, нацеленный на разработку через тестирование на языке Ruby.
- **RailsCasts** (http://railscasts.com/): посетите архив видеоуроков RailsCasts и выберите любую тему, которая бросится в глаза. (Увы, многие ролики уже устарели, но многие все еще могут пригодиться.)
- **Книги по Ruby и Rails**: для дальнейшего изучения Ruby я рекомендую следующие книги: «Beginning Ruby» Питера Купера (Peter Cooper); «The Well-Grounded Rubyist» Дэвида Блэка (David A. Black); «Eloquent Ruby» Расса Ольсена (Russ Olsen) и «The Ruby Way»[1] Хэла Фултона (Hal Fulton). Для дальнейшего изучения Rails я рекомендую «Agile Web Development with Rails»[2] Сэма Руби (Sam Ruby), Дэйва Томаса (Dave Thomas) и Дэвида Хайнмейера Ханссона (David Heinemeier Hansson); «The Rails 4 Way»[3] Оби Фернандеса (Obie Fernandez) и Кевина Фаустино (Kevin Faustino) и «Rails 4 in Action» Райана Бигга (Ryan Bigg) и Йехуды Катца (Yehuda Katz).

12.4.2. Что мы узнали в этой главе

- Метод `has_many :through` позволяет моделировать сложные взаимоотношения между данными.
- Метод `has_many` принимает несколько дополнительных аргументов, в том числе название класса объекта и внешний ключ.
- Используя методы `has_many` и `has_many :through` с правильно подобранными именами классов и внешними ключами, можно смоделировать и активные, и пассивные взаимоотношения.
- Механизм маршрутизации в Rails поддерживает вложенные маршруты.

---

[1] Фултон Х. Путь Ruby. 3-е изд. М.: ДМК-Пресс, 2015. ISBN: 978-5-97060-320-8.

[2] Руби С. Rails 4. Гибкая разработка веб-приложений. СПб.: Питер, 2014. ISBN: 978-5-496-00898-3.

[3] Оби Фернандес. Путь Rails. Подробное руководство по созданию приложений в среде Ruby on Rails. СПб.: Символ-Плюс, 2008. ISBN: 5-93286-137-1.

- Метод `where` – гибкий и мощный инструмент создания запросов к базе данных.
- Rails поддерживает вызов низкоуровневых SQL-запросов.
- Собрав вместе все, что мы узнали в этой книге, мы успешно реализовали следование за пользователями вместе с лентой сообщений от читаемых пользователей.

12.5. Упражнения

> **Примечание.** *Руководство по решению упражнений бесплатно прилагается к любой покупке на* www.railstutorial.org.

Предложения, помогающие избежать конфликтов между упражнениями и кодом основных примеров в книге, вы найдете в примечании об отдельных ветках для выполнения упражнений, в разделе 3.6.

1. Напишите тесты для статистик на главной странице и странице профиля. *Подсказка*: добавьте их к тестам в листинге 11.27. (Почему не следует тестировать статистику на главной странице отдельно?)
2. Напишите тест, проверяющий появление первой страницы ленты на главной странице, как это и требуется. Начальная реализация представлена в листинге 12.49. Обратите внимание на экранирование разметки HTML вызовом `CGI.escapeHTML` (попробуйте сами понять, зачем это необходимо, – уберите маскировку и внимательно просмотрите исходный код содержимого сообщения, чтобы увидеть различия.)

Листинг 12.49 ❖ Тестирование разметки HTML ленты сообщений **ЗЕЛЕНЫЙ** (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
```

Предметный указатель

Ф

Ч

Э

A

S

T

U

V

W

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Майкл Хартл

Ruby on Rails для начинающих

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Разуваев А. П.*
Научный редактор *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 53,62. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф