ДЛЯ ПРОФЕССИОНАЛОВ

Rails 4
и Ruby 2.0

Rails 4

Гибкая разработка веб-приложений

Сэм Руби
Дэйв Томас
Дэвид Хэнссон

RAILS

ПИТЕР

Agile Web Development with Rails 4

Sam Ruby
Dave Thomas
David Heinemeier Hansson

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

ДЛЯ ПРОФЕССИОНАЛОВ

Rails 4

Гибкая разработка веб-приложений

Сэм Руби
Дэйв Томас
Дэвид Хэнссон

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.988.02-018
УДК 004.738.5
Р82

Руби С., Томас Д., Хэнссон Д.

Р82 Rails 4. Гибкая разработка веб-приложений. — СПб.: Питер, 2014. — 448 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-00898-3

Перед вами новое издание бестселлера «Agile web development with Rails», написанного Сэмом Руби — руководителем Apache Software Foundation и разработчиком формата Atom, Дэйвом Томасом — автором книги «Programming Ruby», и Дэвидом Хэнссоном — создателем технологии Rails.

Rails представляет собой среду, облегчающую разработку, развертывание и обслуживание веб-приложений. За время, прошедшее с момента ее первого релиза, Rails прошла путь от малоизвестной технологии до феномена мирового масштаба и стала именно той средой, которую выбирают, чтобы создавать так называемые «приложения Web 2.0».

Эта книга, уже давно ставшая настольной по изучению Ruby on Rails, предназначена для всех программистов, собирающихся создавать и развертывать современные веб-приложения. Из первой части книги вы получите начальное представление о языке Ruby и общие сведения о самой среде Rails. Далее на примере создания интернет-магазина вы изучите концепции, положенные в основу Rails. В третьей части рассматривается вся экосистема Rails: ее функции, возможности и дополнительные модули. Обновленное издание книги описывает работу с Rails поколения 4 и Ruby 1.9 и 2.0.

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1937785567 (англ.)
ISBN 978-5-496-00898-3

Оглавление

Благодарности

Rails постоянно развивается, и вместе с ней изменяется и данная книга. Части приложения Depot переписывались по нескольку раз, кроме этого обновлялись все комментарии к этому коду. Исключение ряда средств по мере того как они попадали в разряд нерекомендуемых, неоднократно приводило к изменению структуры книги, как только горячее становилось еле теплым.

Поэтому данная книга не появилась бы на свет без огромной помощи со стороны сообществ Ruby и Rails. Для начала перечислю очень полезных официальных рецензентов проектов данной книги:

Джереми Андерсон (Jeremy Anderson), Andrea Barisone (Андреа Барисоне), Кен Коар (Ken Coar), Джеф Коэн (Jeff Cohen), Джоел Клермонт (Joel Clermont), Джеф Дрейк (Geoff Drake), Jeremy Frens (Джереми Френс), Паван Горакави (Pavan Gorakavi), Майкл Юревич (Michael Jurewitz), Майкл Линдсаар (Mickel Lindsaar), Nigel Lowry (Нигель Лоури), Stephen Orr (Стефен Орр), Aaron Patterson (Аарон Паттерсон), Пол Рейнер (Paul Rayner), Мартин Реверс (Martin Reuvers), Даг Ротен (Doug Rhoten), Гари Шерман (Gary Sherman), Tibor Simic (Тибор Симик), Gianluigi Spagnuolo (Джанлуиджи Спаньюоло), Даванум Шринивас (Davanum Srinivas), Charley Stran (Чарли Стран), Federico Tomassetti (Федерико Томассетти), Стефан Туралски (Stefan Turalski) и Хосе Валим (José Valim).

Кроме этого, каждое издание данной книги выпускалось в бета-версии: каждая версия публиковалась в PDF-формате, и люди комментировали ее в Интернете. В результате оставленных ими комментариев поступило более 1000 предложений и сообщений об ошибках.

Значительная часть предложений в конечном итоге была принята, что сделало эту книгу намного полезнее. Хотя наша благодарность распространяется на всех, кто поддерживал программу выпуска бета-версии книги и внес в своих отзывах так много ценных предложений, среди них есть люди, которые при этом вносили свой вклад не по долгу службы:

Мануэль Э. Видаурре Аренас (Manuel E. Vidaurre Arenas), Сет Арнолд (Seth Arnold), Уилл Боулин (Will Bowlin), Энди Брайс (Andy Brice), Джейсон Катена (Jason Catena), Виктор Мариус Костан (Victor Marius Costan), Дэвид Хэдли (David Hadley), Джейсон Холловей (Jason Holloway), Дэвид Капп

(David Kapp), Trung LE, Кристиан Рибер Мандруп (Kristian Riiber Mandrup), mltsy, Стив Николсон (Steve Nicholson), Джим Палс (Jim Puls), Джонатан Ритци (Johnathan Ritzi), Leonel S, Ким Шриер (Kim Shrier), Дон Смит (Don Smith), Джо Страйтиф (Joe Straitiff) и Мартин Золлер (Martin Zoller).

И наконец, огромная помощь была оказана со стороны команды разработчиков ядра Rails, представители которой отвечали на вопросы, проверяли наши фрагменты кода и исправляли ошибки, причем применительно даже к той части работы по выпуску нового издания, которая относилась к проверке того, не становятся ли приводимые в данной книге примеры неработоспособными в новых выпусках Rails. Мы говорим огромное спасибо следующим специалистам:

Rafael França (Рафаэль Франца, rafaelfranca), Guillermo Iguaran (Джиллермо Игуаран, guilleiguaran), Джереми Кемпер (Jeremy Kemper, bitsweat), Иегуда Кац (Yehuda Katz, wycats), Майкл Козарски (Michael Koziarski), Santiago Pastorino (Сантьяго Пасторино, spastorino), Aaron Patterson (Аарон Паттерсон) и Хосе Валим (José Valim, josevalim).

Сэм Руби (Sam Ruby)
mailto: rubys@intertwingly.net

Введение

Ruby on Rails является средой, облегчающей разработку, развертывание и обслуживание веб-приложений. За время, прошедшее с ее начального выпуска, Rails прошла путь от малоизвестной технологии до феномена мирового масштаба и, что более важно, стала именно той средой, которую выбирают, чтобы создавать так называемые *приложения Web 2.0*.

Почему это произошло?

Просто Rails хорошо прижилась с самого начала. Большое количество разработчиков было недовольно теми технологиями, которые применялись ими для создания веб-приложений. И дело, наверное, не в том, что именно они использовали — Java, PHP или .NET, — у них накапливалось ощущение излишней трудоемкости их работы. А затем в один прекрасный момент пришла Rails, с которой работать стало намного проще.

Но сама по себе простота не означает упрощенность. Речь идет о профессиональных разработчиках, создающих по-настоящему востребованные во всем мире веб-сайты. Им хочется видеть созданные ими приложения выдержавшими испытание временем — спроектированными и разработанными с использованием современных, профессиональных технологий. Поэтому разработчики занялись Rails всерьез и обнаружили, что она является не только инструментом для разработки веб-сайтов.

К примеру, *все* Rails-приложения выполняются с использованием архитектуры Модель-Представление-Контроллер (Model-View-Controller, MVC). Привычная Java-разработчикам среда выполнения, к примеру Tapestry или Struts, тоже основана на MVC. Но Rails идет в использовании MVC еще дальше: при ведении разработки в Rails вы начинаете уже с работающего приложения, в котором есть место для каждой части кода, и все части вашего приложения стандартным образом взаимодействуют друг с другом.

Профессиональные программисты пишут тесты. И Rails опять вносит свою лепту. Все Rails-приложения имеют встроенное тестирование. По мере добавления к программному коду какой-либо функциональной возможности Rails автоматически создает программные заглушки тестов, предназначенные для ее тестирования. Эта среда облегчает тестирование своих приложений, подстегивая тем самым разработчиков к этому занятию.

Rails-приложения пишутся на Ruby — современном объектно-ориентированном языке сценариев. Лаконичность кода Ruby не влияет на его разборчивость — свои идеи на этом языке можно выражать вполне четко и естественно. В результате чего программы легко пишутся и (что не менее важно) по прошествии нескольких месяцев вполне легко читаются.

Rails использует все возможности Ruby, являясь его оригинальным расширением, облегчающим жизнь программистов. Программы становятся короче, читаются легче. Это также позволяет нам выполнять те задачи, которые иначе выполнялись бы в исходном коде внешних файлов конфигурации. Это облегчает понимание происходящего. Следующий код определяет для проекта модель класса. Сейчас не стоит вдаваться в детали этого кода — лучше просто подумать о том, как много информации было выражено в каких-то нескольких строках программы.

```
class Project < ActiveRecord::Base
    belongs_to :portfolio
    has_one :project_manager
    has_many :milestones
    has_many :deliverables, through: milestones
    validates :name, :description, presence: true
    validates :non_disclosure_agreement, acceptance: true
    validates :short_name, uniqueness: true
end
```

Укоротить код Rails и сделать его более читаемым позволяют две другие философские основы этой среды: DRY и превалирование соглашения над конфигурацией. DRY означает «don't repeat yourself», то есть *«никогда не повторяться»*: каждая частичка знаний в системе должна быть выражена только в одном месте. Чтобы воплотить все это в жизнь, Rails пользуется всей эффективностью языка Ruby. В Rails-приложениях можно увидеть лишь малую долю повторений, то, что нужно сказать, говорится только в одном месте, которое зачастую предлагается соглашениями о MVC-архитектуре, и далее об этом можно уже не беспокоиться. Для программистов, привыкших работать в других средах веб-разработки, где простое изменение может заставить их вносить в код программы полдюжины, а то и больше правок, это было открытием.

Превалирование соглашения над конфигурацией является не менее важным принципом. Он означает, что в Rails практически для каждого аспекта, связывающего в единое целое ваше приложение, имеются рациональные умолчания. Следуйте соглашениям, и тогда вы сможете написать Rails-приложение, используя меньше кода, чем в обычном веб-приложении, написанном на Java и использующем XML-конфигурацию. Если нужно переписать соглашения, Rails облегчает и эту задачу.

Разработчики, которые переходят на Rails, замечают еще одну особенность. Rails не играет в догонялки со ставшими де-факто новыми стандартами: напротив, она помогает их определять. К тому же Rails облегчает разработчикам интегрирование в их код таких функций, как интерфейсы AJAX и RESTful, поскольку их поддержка уже встроена в Rails. (Если вы не знакомы с интерфейсами AJAX и REST, не стоит беспокоиться, чуть позже мы объясним вам, что это такое.)

Разработчики озабочены также развертыванием своих продуктов. И тут оказывается, что с Rails можно распространять удачную версию своего приложения на

любое количество серверов всего лишь одной командой (и так же легко возвращать все назад, если версия окажется не вполне удачной).

Rails была выделена из реального коммерческого приложения. Оказалось, что лучшим способом создания среды является определение основных составляющих конкретного приложения, а затем занесение их в общий фонд кода. При разработке Rails-приложения в вашем распоряжении с самого начала уже имеется половина по-настоящему хорошего приложения.

Но у Rails еще есть кое-что такое, что трудно поддается описанию. Она каким-то непостижимым образом создает уверенность в правильном выборе. Разумеется, пока вы самостоятельно не напишете на Rails какие-нибудь приложения (что может произойти в ближайшие 45 минут), вам придется поверить нам на слово. Обо всем этом и будет рассказано в нашей книге.

Rails является средством гибкой разработки

Эта книга называется «Гибкая разработка веб-приложений в среде Rails 4». Возможно, вы удивитесь, обнаружив отсутствие четко обозначенных разделов, посвященных использованию гибких методов Rails-программирования.

Объясняется это простыми и в то же время довольно тонкими обстоятельствами. Дело в том, что гибкость разработки является качественной составляющей Rails.

Давайте взглянем на ценные положения, которые изложены в Манифесте по гибкой разработке (Agile Manifesto), как на набор из четырех предпочтений:[1]

- личностей и их взаимодействия над процессами и инструментами;
- работающего программного обеспечения над подробной документацией;
- сотрудничества с заказчиком над контрактными обязательствами;
- реакций на изменения над следованием плану.

В Rails предпочтение целиком отдается людям и их взаимодействию. В ней отсутствует громоздкий инструментарий, нет ни сложных конфигураций, ни детально проработанных процессов. Вполне достаточно небольших групп разработчиков с привычными текстовыми редакторами и фрагментами Ruby-кода. Этот подход ведет к такой прозрачности, при которой все, что делается разработчиками, немедленно отражается на том, что видит заказчик. По сути, это интерактивный процесс.

Rails не отвергает документацию. Создание в этой среде HTML-документации для всего программного кода сводится к весьма простой задаче. Но процесс разработки в Rails не регулируется какими-либо документами. Вы не найдете в основе Rails-проекта 500-страничной спецификации. Вместо нее будет группа пользователей и разработчиков, совместно изучающих свои потребности и возможные пути их удовлетворения. Вы найдете решения, изменяющиеся по мере того, как и разработчики, и пользователи набираются опыта, пытаясь решить

[1] http://agilemanifesto.org/. Дэйв Томас был одним из 17 авторов этого документа.

возникающие проблемы. Вы найдете среду, предоставляющую работоспособное программное обеспечение на ранней стадии цикла разработки. Это программное обеспечение может быть грубоватым, но зато оно даст пользователям возможность составить начальное представление о том, что они от вас получат. Таким образом, Rails поощряет сотрудничество с заказчиком. Когда заказчики видят, насколько быстро Rails-проект может реагировать на изменения, они начинают верить, что команда способна поставить им то, что требуется, а не только то, что было изначально запрошено. Конфронтации подменяются разговорами на тему: «А что, если...?».

Все это привязано к идее возможности реагировать на изменения. Одержимость, с которой в Rails придерживаются принципов DRY, означает, что изменения в Rails-приложениях оказывают намного меньше влияния на программный код, чем в других средах. А так как Rails-приложения пишутся на Ruby, где понятия могут выражаться четко и кратко, изменения чаще всего локализованы и легко вносятся в программный код. Существенный упор как на блочное, так и на функциональное тестирование, наряду с поддержкой в процессе тестирования испытательных стендов и программных заглушек, предоставляет разработчикам страховку, столь необходимую при внесении изменений. При хорошем наборе тестов изменения меньше действуют на нервы.

Вместо постоянных попыток связать процессы, происходящие в Rails, с принципами гибкой разработки, мы решили позволить этой среде самой раскрыть свои возможности. По мере чтения глав руководства попытайтесь представить себе, что вы ведете разработку веб-приложения, работая рядом со своими заказчиками и совместно вырабатывая приоритеты и решения проблем. Затем, при переходе в части третьей к изучению самых передовых концепций, посмотрите, как структура, положенная в основу Rails, может позволить вам быстрее и с меньшей долей формализма удовлетворять потребности ваших заказчиков.

И еще одно, последнее замечание по поводу гибкой разработки и Rails: хотя упоминание об этом может выглядеть непрофессионально, но подумайте, насколько веселее станет сам процесс программирования!

Для кого предназначена эта книга

Эта книга предназначена для программистов, присматривающихся к созданию и развертыванию веб-приложений. К их числу относятся прикладные программисты, не работавшие ранее с Rails (и, возможно, даже незнакомые с Ruby), и программисты, знакомые с основами, но желающие прийти к более глубокому пониманию среды Rails.

Предполагаются некоторые познания в HTML, Cascading Style Sheets (CSS) и JavaScript, иными словами, речь идет о способности разбираться в исходном коде веб-страниц. Вам не нужно быть экспертами по данной тематике, самое сложное, что предстоит делать, — это переносить в файлы программный материал этой книги, весь объем которого можно загрузить.

Как нужно читать эту книгу

Первая часть этой книги является подготовительной. После ее прочтения вы получите начальное представление о языке Ruby и общее представление о самой среде Rails, у вас будут установлены Ruby и Rails, и вы проверите работоспособность этой установки на простом примере. В следующей части на более объемном примере (создание простого интернет-магазина) вы изучите концепции, положенные в основу Rails. Здесь не будет последовательного путешествия по каждому из компонентов Rails («вот глава о моделях, а вот о представлениях» и т. д.). Эти компоненты созданы для совместной работы, и каждая глава в этой части привязана к конкретному набору родственных задач, в решение которых вовлекается сразу несколько совместно работающих компонентов.

Многим, похоже, нравится создавать приложение по мере чтения этой книги. Если вам не хочется набирать код, можно схитрить, загрузив исходный код (сжатый tar-архив или zip-файл[1]). Эта загрузка содержит отдельные наборы исходного кода для Rails 3.0, Rails 3.1, Rails 3.2 и Rails 4.0. Поскольку вам понадобится Rails 4.0, нужные файлы будут находиться в каталоге rails40. Подробности изложены в файле README-FIRST.

Принимать решение о копировании файла непосредственно из загрузки нужно осмотрительно, поскольку если метки времени файла окажутся старыми, сервер не будет знать, что ему нужно выбрать именно эти изменения. В Mac OS X или в Linux метки времени можно обновить с помощью команды `touch` или же можно отредактировать файл с последующим его сохранением. Кроме того, можно перезапустить свой Rails-сервер.

В части третьей, «Углубленное изучение Rails», рассматривается вся экосистема Rails. Эта часть начинается с функций и возможностей Rails, с которыми вы уже ознакомились. Затем охватывается ряд ключевых зависимостей, используемых средой Rails, которые вносят непосредственный вклад в общую функциональность, предоставляемую этой средой. И наконец, в ней рассматривается ряд популярных дополнительных модулей, расширяющих среду Rails и превращающих ее из простой среды в открытую экосистему.

При чтении книги вам будут попадаться различные принятые нами способы оформления текста.

ИСПОЛНЯЕМЫЙ КОД

Большинство демонстрируемых фрагментов кода взяты из полноценных, работоспособных примеров, которые можно загрузить.

Если листинг кода можно найти в загружаемых примерах, чтобы облегчить задачу его поиска, перед ним будет стоять заголовок (такой же, как в следующем примере).

[1] Скачайте файлы по адресу http://pragprog.com/titles/rails4/source_code

rails40/work/demo1/app/controllers/say_controller.rb

```
  class SayController < ApplicationController
▶   def hello
▶   end

      def goodbye
      end
  end
```

В этом заголовке содержится путь к листингу в скачанной структуре исходного кода. А в некоторых случаях, где используются модификации существующих файлов и где трудно заметить измененные строки, слева от таких строк можно заметить значки в виде треугольников, помогающие их распознать. Две такие строки показаны в предыдущем примере кода.

ДЭВИД ГОВОРИТ...

Время от времени вам будут попадаться врезки **«Дэвид говорит...»**. В них Дэвид Хайнемайер Хэнсон (David Heinemeier Hansson) будет делиться с вами ценными сведениями о Rails — давать пояснения и рекомендации, показывать трюки и т. п. Поскольку он создатель Rails, эти врезки не стоит пропускать, если вы хотите подойти к изучению этой среды профессионально.

ДЖО СПРАШИВАЕТ...

В книге иногда появляется некий мифический разработчик по имени Джо, задающий вопросы по существу изучаемого материала, на которые мы отвечаем.

Эта книга не является справочным руководством по Rails. Наш опыт подсказывает, что справочные руководства для большинства людей учебниками служить не могут. Вместо этого мы показали большинство модулей и многие из их методов либо в примерах, либо в текстовых описаниях в контексте использования этих компонентов и их совместной работы.

Вы не найдете здесь и многих сотен страниц с листингами API-функций. Для этого есть серьезные основания — вы получаете всю соответствующую документацию при каждой установке Rails, и она, несомненно, будет новее того материала, который представлен в данной книге. Если вы устанавливаете Rails, используя RubyGems (что соответствует нашим рекомендациям), нужно просто запустить сервер gem-документации (воспользовавшись командой `gem_server`), и вы сможете получить доступ к описанию всех API-функций Rails, набрав в адресной строке браузера http://localhost:8808. Создание дополнительной документации и руководств будет рассмотрено в главе 18, в разделе «Место для документации».

Кроме этого, вы увидите, что сама Rails помогает вам, создавая ответы, четко идентифицирующие любую найденную ошибку, а также показывая пути, сообщающие вам не только о месте обнаружения ошибки, но и о том, как до него добраться. Пример можно увидеть на рис. 10.3. Если нужна дополнительная информация, загляните в раздел 10.2 «Шаг Д2: обработка ошибок», чтобы увидеть как можно вставить инструкции для ведения регистрационного журнала.

Если вы на чем-то застопоритесь, помощь можно будет найти в огромной массе интернет-ресурсов. Вдобавок к упомянутым ранее листингам, есть еще форум[1], где вы можете задавать вопросы и делиться опытом, перечень замеченных опечаток[2], где вы можете сообщить об ошибках, и wiki[3], где вы можете обсудить упражнения, которые приводятся по всей книге.

Это общедоступные ресурсы, и вы можете вполне свободно помещать на форуме и wiki не только вопросы и проблемы, но также и любые предложения, и ответы на вопросы, заданные другими людьми.

Итак, приступим! Первым нашим шагом будет установка Ruby и Rails и проверка работоспособности этой установки на простом примере.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете скачать по адресу http://pragprog.com/titles/rails4/source_code.

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

[1] http://forums.pragprog.com/forums/148
[2] http://www.pragprog.com/titles/rails4/errata
[3] http://www.pragprog.com/wikis/wiki/RailsPlayTime

I

Начало

1 Установка Rails

Основные темы:

- установка Ruby, RubyGems, SQLite3 и Rails;
- установка среды разработки и инструментария.

В первой части этой книги вам будут представлены язык Ruby и среда Rails. Но сначала нужно установить оба этих средства и убедиться в их работоспособности.

Чтобы среда Rails заработала на вашей системе, нужно располагать следующими программными средствами:

- интерпретатором Ruby. Система Rails написана на Ruby, и свои приложения вы также будете создавать на Ruby. В Rails 4.0 рекомендуется применять Ruby версии 2.0.0, но работать можно и с версией 1.9.3. На Ruby версии 1.8.7 или на Ruby 1.9.2 эта среда работать не будет;
- системой Ruby on Rails. Эта книга была написана с использованием Rails версии 4.0 (а именно Rails 4.0.0);
- интерпретатором JavaScript. Встроенные интерпретаторы этого языка имеются как в Microsoft Windows, так и в Mac OS X, и Rails будет использовать ту версию, которая уже установлена на вашей системе. В других операционных системах может понадобиться отдельная установка интерпретатора JavaScript;
- некоторыми библиотеками, в зависимости от операционной системы;
- базой данных. В данной книге используются как SQLite 3, так и MySQL 5.5.

Для машины, используемой в целях разработки программ, это практически все, что нужно (не считая редактора, но о редакторах мы поговорим отдельно). Если же вам захочется ввести приложение в эксплуатацию, дополнительно понадобится установить промышленный веб-сервер (как минимум), а также поддерживающее его программное обеспечение, без которого эффективная работа Rails невозможна. Этому посвящена целая глава, и здесь данный вопрос больше рассматриваться не будет.

Ну и как же все это устанавливать? А это зависит от вашей операционной системы...

1.1. Установка под Windows

Самым простым способом установки Rails под Windows является использование пакета RailsInstaller[1]. На момент написания данной книги самой последней версией RailsInstaller была 2.2.1, которая включала в себя Ruby 1.9.3 и Rails 3.2. А чтобы приступить к работе, пока не выпущена новая версия, поддерживающая Rails 4.0.0 или Ruby 2.0, можно использовать версию RailsInstaller с порядковым номером 2.1.

Основная установка происходит довольно просто. После загрузки щелкните на кнопке Run, затем на кнопке Next. Установите переключатель в положение «I accept the License» (разумеется, после внимательного прочтения самой лицензии, условия которой принимаются) и щелкните на кнопках Next ▸ Install ▸ Finish.

После этого будет открыто окно командной строки и выдано приглашение на ввод имени и адреса электронной почты. Это нужно только лишь для настройки версии системы управления Git. Генерируемый ssh-ключ для выполнения упражнений данной книги не понадобится.

Закройте это окно и откройте новое окно командной строки. При работе в Windows 8 на начальном экране (Start screen) с интерфейсом на основе плиток наберите команду `cmd` и нажмите клавишу Enter. В версиях Windows, предшествующих Windows 8, щелкните на кнопке Пуск (Start) и выберите пункт Выполнить... (Run...), после чего введите команду `cmd` и щелкните на кнопке OK.

Пользователям Windows 8 нужно выполнить дополнительные шаги по установке node.js[2]. Когда они будут выполнены, потребуется закрыть окно командной строки и открыть новое такое же окно, чтобы изменения, внесенные в системную переменную `%PATH%`, возымели эффект. Затем нужно будет проверить, что установка прошла нормально, введя команду `node -v`.

Если используемая вами версия RailsInstaller установила версию Ruby с номером 1.9.3 или выше, обновлять Ruby до более высокой версии нет необходимости.

А теперь, пожалуйста, перейдите к разделу 1.4, «Выбор версии Rails», чтобы убедиться, что установленная вами версия Rails соответствует той версии, которая рассматривается в данном издании книги. Там мы и встретимся.

1.2. Установка под Mac OS X

Поскольку Mac OS X поставляется с Ruby 1.8.7, придется загрузить более свежую версию Ruby, работающую с Rails 4.0. Проще всего это сделать с помощью

[1] http://railsinstaller.org/
[2] http://nodejs.org/download/

средства RailsInstaller, которое на момент написания данной книги устанавливало Ruby 1.9.3. Второй способ решения данной задачи заключается в использовании самой последней версии RVM для разработчиков, которой можно воспользоваться при установке Ruby 2.0.0. Версия Ruby 2.0 рекомендована основной командой разработчиков Rails и работает заметно быстрее версии Ruby 1.9.3, но с материалом данной книги можно задействовать обе эти версии. В книге дается описание обоих подходов, а выбор остается за вами.

Прежде чем начать, перейдите в папку Utilities (Утилиты) и перетащите на свой док приложение Terminal. Им вы будете пользоваться в ходе установки, а затем довольно часто применять, выступая в роли разработчика приложений в среде Rails.

Установка с помощью RailsInstaller

Сначала перейдите в RailsInstaller и щелкните на большой зеленой кнопке Download the Kit (Загрузить набор).

Как только загрузка завершится, дважды щелкните кнопкой мыши на файле, чтобы его распаковать. Прежне чем щелкнуть на создаваемом файле приложения, нужно удерживать клавишу Control. Выберите в меню пункт Open (Открыть). Открываемое таким способом приложение дает возможность установить программу от имени разработчика, который неизвестен магазину приложений (app store). Здесь придется ответить на ряд вопросов (таких как ваше имя, которое будет использовано для настройки git), и установка продолжится.

Теперь откройте приложение Terminal и введите в строке приглашения следующую команду:

```
$ ruby -v
```

Должен получиться следующий результат:

```
ruby 1.9.3p392 (2013-02-22 revision 39386) [x86_64-darwin11.4.0]
```

После этого обновите Rails до версии, описываемой в данной книге, используя следующую команду:

```
$ gem install rails --version 4.0.0 --no-ri --no-rdoc
```

Вот теперь все готово к дальнейшей работе! Присоединяйтесь к пользователям Windows и переходите к разделу 1.4, «Выбор версии Rails».

Установка с помощью RVM

Сначала загрузите и установите самую последнюю версию (январь 2013) инструментария командной строки Xcode (Command Line Tools for Xcode) для своей операционной системы (OS X Lion или OS X Mountain Lion), используя в XCode панель настроек Downloads (Загрузки).

Теперь откройте приложение Terminal и в поле ввода наберите следующую команду, предназначенную для установки разработочной версии RVM:

```
$ curl -L https://get.rvm.io | bash -s stable
```

Проследите за тем, как в выводимых этой командой данных будут появляться сведения обо всех обновлениях.

Выполнив данные инструкции, можно продолжить установку интерпретатора Ruby:

```
$ rvm install 2.0.0 --autolibs=enable
```

Предыдущее действие потребует времени на загрузку, конфигурацию и компиляцию необходимых исполняемых файлов. Как только работа будет завершена, воспользуйтесь этой средой окружения и установите Rails:

```
$ rvm use 2.0.0
$ gem install rails --version 4.0.0 --no-ri --no-rdoc
```

За исключением инструкции `rvm use`, каждая из приведенных выше инструкций должна быть выполнена только один раз. Инструкцию `rvm use` нужно повторять при каждом открытии окна оболочки. Ключевое слово `use` является необязательным, поэтому инструкцию можно сократить до вида `rvm 2.0.0`. Можно также для новых сеансов работы с терминалом сделать эту версию интерпретатора Ruby используемой по умолчанию, воспользовавшись следующей командой:

```
$ rvm --default 1.9.2
```

Убедиться в успехе установки можно с помощью следующей команды:

```
$ rails -v
```

Если возникнут проблемы, воспользуйтесь рекомендациями, перечисленными на веб-сайте rvm под заголовком «Troubleshooting Your Install»[1].

А теперь пользователи OS X могут перейти сразу к разделу 1.4, «Выбор версии Rails», чтобы присоединиться к пользователям Windows. Там мы и встретимся.

1.3. Установка под Linux

Начните с одной из систем управления пакетами, принадлежащей вашей платформе: apt-get, dpkg, portage, rpm, rug, synaptic, up2date или yum.

Сначала нужно установить необходимые взаимодействующие средства. Следующие инструкции предназначены для Ubuntu 13.04 (Raring Ringtail); при работе под другими операционными системами и отсутствии уверенности в имеющихся в них установках самыми важными устанавливаемыми на данном этапе пакетами являются git и curl, а все остальные пакеты можно будет установить по мере необходимости.

```
$ sudo apt-get install apache2 curl git libmysqlclient-dev mysql-server nodejs
```

[1] https://rvm.beginrescueend.com/rvm/install

У вас будет запрошен корневой пароль для вашего сервера MySQL. Если оставить его поле пустым, он будет запрошен еще несколько раз. Если будет указан пароль, его нужно будет использовать при создании базы данных, которое рассматривается в главе 16, в разделе «Использование MySQL для создания базы данных».

Хотя команда разработчиков Rails рекомендует использовать с Rails 4.0 версию Ruby 2.0, если вам захочется задействовать уже установленную в системе версию Ruby 1.9.3, это возможно и к тому же сократит объем подготовительных действий.

Начиная с Ubuntu 12.04 можно установить Ruby 1.9.3 и Rails 4.0 с помощью следующих команд:

```
$ sudo apt-get install ruby1.9.3
$ sudo gem install rails --version 4.0.0 --no-ri --no-rdoc
```

Если вам этого будет достаточно, значит подготовительные действия выполнены и можно продолжить чтение с раздела 1.4, «Выбор версии Rails».

Многие все же предпочитают специально установить Ruby на своей машине, предназначенной для поддержки разрабатываемых приложений, и поэтому выбирают загрузку и компоновку Ruby. Самый простой из обнаруженных нами способ предполагает использование RVM. Установка RVM рассмотрена на веб-сайте RVM[1]. Здесь же вам предлагается краткий обзор необходимых действий.

Сначала установите RVM с помощью команды:

```
$ curl -L https://get.rvm.io | bash -s stable
```

Затем в настройках профиля Gnome Terminal Profile Preference установите флажок Run command as login shell. За инструкциями обратитесь к странице Integrating RVM with gnome-terminal[2].

Выйдите из окна командной строки или приложения Terminal и откройте их новый экземпляр. Это приведет к перезагрузке с использованием вашего файла .bash_profile.

Выполните следующую команду, которая установит все, что заведомо будет необходимо именно вашей операционной системе:

```
$ rvm requirements --autolibs=enable
```

Как только все будет завершено, можете продолжить установку интерпретатора Ruby.

```
$ rvm install 2.0.0
```

Этот этап установки потребует времени на загрузку, конфигурирование и компиляцию необходимых исполняемых файлов. Как только он будет завершен, воспользуйтесь созданной средой и установите Rails:

```
$ rvm use 2.0.0
$ gem install rails --version 4.0.0 --no-ri --no-rdoc
```

---

[1] https://rvm.io/rvm/install

[2] https://rvm.io/integration/gnome-terminal/

За исключением инструкции `rvm use`, каждая из показанных выше инструкций должна быть выполнена только один раз. Инструкцию `rvm use` нужно повторять при каждом открытии окна оболочки. Ключевое слово `use` является необязательным, поэтому инструкцию можно сократить до вида `rvm 2.0.0`. Можно также для новых сеансов работы с терминалом сделать эту версию интерпретатора Ruby используемой по умолчанию, воспользовавшись следующей командой:

```
$ rvm --default 2.0.0
```

Убедиться в успехе установки можно с помощью следующей команды:

```
$ rails -v
```

Если возникнут проблемы, воспользуйтесь рекомендациями, перечисленными на веб-сайте rvm под заголовком «Troubleshooting Your Install»[1].

Теперь, когда были рассмотрены инструкции для Windows, Mac OS X и Linux, все последующие инструкции будут общими для всех трех операционных систем.

1.4. Выбор версии Rails

Предыдущие инструкции помогли вам установить версию Rails, используемую в примерах данной книги. Но порой именно эта версия может и не понадобиться. Например, это может быть более свежая версия Rails с исправлениями или новыми функциями. Или, возможно, разработка ведется на одной машине, а эксплуатация предполагается на другой, содержащей неподконтрольную вам версию Rails.

Попав в какую-либо из подобных ситуаций, нужно быть в курсе нескольких вещей. Сначала нужно с помощью команды `gem` найти все установленные версии Rails:

```
$ gem list --local rails
```

Также нужно с помощью команды `rails --version` проверить, какая версия Rails запускается по умолчанию. Эта команда должна вернуть индекс версии 4.0.0.

Если результат будет иным, вставьте перед первым параметром любой команды `rails` номер версии Rails, окруженный символами подчеркивания. Например:

```
$ rails _4.0.0_ --version
```

Это особенно важно при создании нового приложения, поскольку если приложение создано с использованием конкретной версии Rails, оно будет и дальше продолжать использовать эту версию, даже если на системе установлены более свежие версии, до тех пор пока не будет принято решение обновить приложение. Для обновления нужно будет просто указать номер новой версии в файле Gemfile, который находится в корневом каталоге вашего приложения, и запустить команду

[1] https://rvm.beginrescueend.com/rvm/install

`bundle install`. Более подробно эта команда будет рассмотрена в главе 24, в разделе «Управление компонентами, от которых зависит работа приложения с помощью системы Bundler».

1.5. Настройка среды разработки

Повседневная работа по созданию Rails-программ не отличается особым разнообразием, но каждый работает по-своему. Мы это делаем следующим образом.

Командная строка

Основную работу мы проделываем в командной строке. Хотя появляются все новые и новые средства графического пользовательского интерфейса (GUI), помогающие генерировать Rails-приложения и управлять ими, мы считаем, что командная строка по-прежнему является наиболее эффективным способом разработки. Поэтому вам стоит потратить немного времени на ознакомление с командной строкой вашей операционной системы. Нужно выяснить, как она используется для редактирования вводимых команд, как можно найти и отредактировать предыдущие команды и как осуществлять автозавершение вводимых имен файлов и команд.

Для оболочек Unix `Bash` и `zsh` стандартным является автозавершение с помощью клавиши табуляции (так называемое tab-завершение). Оно позволяет набирать первые несколько символов имени файла и нажатием клавиши Tab заставлять оболочку провести поиск и завершить имя, используя соответствующие имена файлов.

Управление версиями

Вся наша работа ведется в системе управления версиями (на данный момент в Git). При создании нового Rails-проекта в Git заносится предмет проверки, и после прохождения тестов туда передаются все изменения. Обычно передача данных в репозиторий ведется по нескольку раз в час.

А ГДЕ ЖЕ IDE?

Если вы занялись Ruby и Rails после того, как поработали с такими языками, как C# и Java, у вас может появиться вопрос насчет IDE (интегрированной среды разработки). Ведь всем известно, что невозможно создать код современного приложения без хотя бы 100 Мбайт IDE, поддерживающего каждое нажатие клавиши. Всем, кто считает себя слишком грамотным, мы предлагаем прямо сейчас усесться поудобнее и обложиться со всех сторон грудами справочников по среде разработки и книгами по 1000 страниц на тему «как все просто можно сделать».

Для Ruby или Rails пока не существует совершенных IDE (хотя некоторые среды разработки уже на подходе). Вместо них большинство Rails-разработчиков используют старые добрые текстовые редакторы. Оказывается, не стоит драматизировать ситуацию. Используя другие,

менее выразительные языки, программисты полагаются на IDE для того, чтобы она делала за них большую часть рутинной работы: генерировала код, помогала осуществлять навигацию по файловой системе и проводила постоянную компиляцию, моментально выдавая предупреждения об ошибках.

При работе с Ruby такая мощная поддержка просто ни к чему. Текстовые редакторы, наподобие TextMate, предоставят вам 90% всего, что вы получали от IDE, выступая при этом в более легком весе.

Пожалуй, единственно полезной из утраченных функций IDE будет поддержка мгновенной проверки кода.

Если работа над Rails-проектом ведется совместно с другими людьми, нужно подумать об установке распределенной системы интеграции — continuous integration system (CI). Когда кто-нибудь зарегистрирует изменения, CI-система проверит новую копию приложения и выполнит все тесты. Это самый простой способ немедленного обнаружения случайных повреждений. Можно также настроить свою CI-систему на ее использование клиентами для апробации самой последней версии вашего приложения. Такая прозрачность является отличной гарантией того, что ваш проект не уйдет в сторону от запросов потребителей.

Редакторы

Мы создаем свои Rails-программы, используя редакторы для программистов. За годы работы мы поняли, что различные редакторы хороши для работы с различными языками и средами. Например, Дэйв сначала написал эту главу, используя редактор Emacs, поскольку его режим Filladapt он считает непревзойденным, когда дело касается форматирования вводимого XML-кода. Сэм внес поправки в эту главу, используя редактор Vim. Но многие считают, что ни Emacs, ни Vim не являются идеальными средствами для Rails-разработки. Выбор редактора, конечно, дело сугубо личное, но в редакторе для Rails стоит все же поискать следующие свойства.

- Поддержку выделения синтаксиса Ruby и HTML. Она идеально подойдет для файлов формата .erb (формат файлов Rails, включающих в себя вставки фрагментов Ruby в HTML-код).
- Поддержку автоматических отступов и обратных отступов в исходном коде Ruby. Это свойство не только улучшает эстетическое восприятие; создание редактором отступов по мере ввода программы — это наилучший способ отслеживания неправильных структурных вложений в программном коде. Способность делать обратные отступы важна при пересмотре кода и перемещении материала. (Очень удобной представляется способность редактора TextMate обрабатывать обратные отступы после вставки кода из буфера.)
- Поддержку вставок обычных логических структур Ruby и Rails. Вам придется вводить множество коротких методов, и, если IDE создает структуру метода путем нажатия одной или двух клавишных комбинаций, вы сможете сконцентрироваться на главном материале, находящемся внутри этой структуры.

- Хорошую навигацию по файловой системе. Мы еще увидим, что Rails-приложения состоят из множества файлов: только что созданное приложение попадает в мир, состоящий из сорока шести файлов, разбросанных по тридцати четырем каталогам, — и это еще до создания конкретного наполнения этого приложения.

  Вам нужна среда, помогающая осуществлять быстрые переходы: вы будете добавлять строчку к контроллеру для загрузки значения, переключаться на представление и добавлять строчку, чтобы вывести это значение на экран, а затем переключаться на тестирование, чтобы убедиться, что все сделано правильно. Такие средства, как Блокнот, в которых для выбора каждого редактируемого файла приходится работать с диалоговым окном File ▸ Open, вряд ли будут соответствовать этому требованию. Мы предпочитаем иметь сочетание изображения дерева файлов на боковой панели, небольшого набора клавишных комбинаций, позволяющих найти по имени в дереве каталогов файл (или файлы), и некоторых встроенных механизмов, «знающих», как осуществлять переходы, скажем, между действием контроллера и соответствующим ему представлением.
- Автозавершение имен. В Rails часто встречаются довольно длинные имена. Хороший редактор позволяет вводить первые несколько символов, а затем предлагает одним нажатием клавиши ввести возможное продолжение.

Мы не решаемся рекомендовать какие-то конкретные редакторы, поскольку сами по-настоящему пользовались всего лишь несколькими из них и ненароком можем оставить чей-нибудь любимый редактор вне этого списка. Тем не менее, чтобы помочь вам начать работу с чем-нибудь посерьезнее, чем Блокнот, мы смеем предложить следующие редакторы.

- TextMate[1], ставший фактически стандартным редактором для Ruby on Rails на Mac OS X.
- Sublime Text[2], являющийся кросс-платформенным редактором, в котором некоторые видят фактического преемника TextMate.
- Aptana Studio 3[3]. Интегрированная среда разработки Rails-приложений на основе Eclipse. Она работает под управлением Windows, Mac OS X и Linux. Первоначально известное как RadRails, это средство в 2006 году завоевало премию как лучшее открытое средство разработки на основе Eclipse, а в 2007 году этот проект стала курировать компания Aptana.
- jEdit[4]. Полнофункциональный редактор с поддержкой Ruby. Имеет встроенную поддержку внешних модулей.

[1] http://macromates.com/
[2] http://www.sublimetext.com/
[3] http://www.aptana.com/products/studio3
[4] http://www.jedit.org/

- Komodo[1]. IDE, разработанная компанией ActiveState для динамичных языков, включая Ruby.
- RubyMine[2]. Коммерческая IDE-среда для Ruby, бесплатно доступная для специальных образовательных проектов и проектов с открытым кодом. Работает под Windows, Mac OS X и Linux.
- Дополнительный модуль NetBeans Ruby and Rails[3], который является модулем с открытым кодом для популярной среды NetBeans IDE.

Спросите у опытных разработчиков, пользующихся такой же операционной системой, с каким редактором они работают. Перед тем как сделать окончательный выбор, попробуйте поработать около недели в различных редакторах.

Рабочий стол

Мы не собираемся указывать вам, как следует оформлять Рабочий стол при работе с Rails, но расскажем, как мы делаем это.

Чаще всего мы занимаемся написанием кода, запуском тестов и исследованием работы своего приложения в браузере. Поэтому на нашем главном Рабочем столе разработчика находятся в постоянно открытом состоянии окна редактора и браузера. Мы также хотим отслеживать регистрационные записи, генерируемые приложением, поэтому держим открытым еще и окно терминала. В нем для прокрутки содержимого регистрационного файла по мере его обновления мы пользуемся командой `tail -f`. Обычно это окно настроено на отображение информации очень мелким шрифтом, чтобы оно занимало меньше места, но как только будет замечено появление чего-нибудь интересного, этот шрифт увеличивается, чтобы все можно было как следует рассмотреть.

Нам также нужен доступ к документации Rails API, которая просматривается в браузере. Во введении шла речь об использовании команды `gem_server` для запуска локального веб-сервера, содержащего Rails-документацию. Это, конечно, удобно, но, к сожалению, Rails-документация разбросана по нескольким отдельным справочным каталогам. Если у вас есть постоянное подключение к Интернету, вы можете воспользоваться ресурсом http://api.rubyonrails.org, чтобы увидеть всю Rails-документацию в одном месте.

1.6. Rails и базы данных

В данной книге при создании примеров использовалась база данных SQLite 3 (версии 3.7.4 или ближайших к ней версий). Если вы собираетесь придерживаться нашего кода, то, наверное, проще всего будет также воспользоваться SQLite 3. Если же будет принято решение об использовании какого-нибудь другого средства,

[1] http://www.activestate.com/komodo-ide
[2] http://www.jetbrains.com/ruby/features/index.html
[3] http://plugins.netbeans.org/plugin/38549

это также не составит проблем. Возможно, для этого придется внести небольшие поправки в какой-нибудь явно выраженный SQL-фрагмент вашего кода, но Rails в своих приложениях старается максимально избавиться от специфичного для баз данных SQL-кода.

Если нужно будет подключиться к базе данных, отличной от SQLite 3, Rails также работает с DB2, MySQL, Oracle, Postgres, Firebird и SQL Server. Для всех баз данных, кроме SQLite 3, потребуется установить драйвер базы данных, библиотеку, которую Rails сможет использовать для подключения и использования вашего процессора базы данных. В этом разделе содержатся ссылки на инструкции, позволяющие справиться с этой задачей.

Все драйверы баз данных написаны на C и распространяются в основном в виде исходного кода. Если вам не хочется возиться с созданием драйвера из исходного кода, присмотритесь повнимательнее к веб-сайту драйвера. В большинстве случаев обнаружится, что автор распространяет также и двоичные версии.

СОЗДАНИЕ СВОЕЙ СОБСТВЕННОЙ ДОКУМЕТАЦИИ RAILS API

Вы можете создать собственную локальную версию сводной документации Rails API. Для этого нужно ввести в командную строку следующий набор команд:

```
rails_apps> rails new dummy_app
rails_apps> cd dummy_app
dummy_app> rake doc:rails
```

На последней стадии следует немного подождать. Когда все закончится, у вас будет документация Rails API в дереве каталогов, начинающемся с doc/api. Я советую переместить эту папку на Рабочий стол, а затем удалить дерево dummy_app.

Для просмотра документации Rails API откройте в своем браузере страницу doc/api/index.html.

Если вы не сможете найти двоичную версию или если вы все равно хотите создать драйвер из исходного кода, вам для его создания понадобится среда разработки, установленная на вашей машине. При работе под Windows это означает, что у вас должна быть копия Visual C++. При работе под Linux вам понадобится gcc и сопутствующие компоненты (которые, скорее всего, уже установлены).

При работе под OS X вам понадобится установить инструментарий разработчика (который поставляется вместе с операционной системой, но по умолчанию не устанавливается). Вам также понадобится установить драйвер вашей базы данных в подходящую версию Ruby. Если вы установили свою собственную копию Ruby в обход встроенной копии, важно помнить, что при создании и установке драйвера базы данных эта версия Ruby должна быть указана в пути поиска первой. Чтобы убедиться, что Ruby не запускается из каталога /usr/bin, можно воспользоваться командой `which ruby`.

В следующем списке перечислены все доступные адаптеры баз данных и приведены ссылки на веб-страницы, где они могут быть найдены.

DB2	http://raa.ruby-lang.org/project/ruby-db2 или http://rubyforge.org/projects/rubyibm
Firebird	http://rubyforge.org/projects/fireruby/

MySQL	http://www.tmtm.org/en/mysql/ruby/
Oracle	http://rubyforge.org/projects/ruby-oci8
Postgres	https://bitbucket.org/ged/ruby-pg/wiki/Home
Server	https://github.com/rails-sqlserver SQL
SQLite	https://github.com/luislavena/sqlite3-ruby

Адаптеры MySQL и SQLite также можно загрузить в виде RubyGems (соответственно, mysql2 и sqlite3).

Наши достижения

- ☑ Установили (или обновили) интерпретатор языка Ruby.
- ☑ Установили (или обновили) среду Rails.
- ☑ Установили (или обновили) базы данных SQLite3 и MySQL.
- ☑ Выбрали редактор.

Теперь, имея установленную среду Rails, давайте перейдем к ее использованию. Настало время приступить к чтению следующей главы, где будет создано наше первое приложение.

2 Немедленное использование

Основные темы:

- создание нового приложения;
- запуск сервера;
- обращение к серверу из браузера;
- генерирование динамического контента;
- добавление гипертекстовых ссылок;
- передача данных из контроллера в представление.

Давайте напишем простое приложение, чтобы проверить, удачно ли среда Rails была установлена на наших машинах, а также получить представление о работе Rails.

2.1. Создание нового приложения

При установке среды Rails вы также получаете новый инструмент командной строки, `rails`, который используется для конструирования каждого нового создаваемого вами Rails-приложения.

А зачем для этого требуется какой-то инструмент? Почему бы для этого просто не приспособить наш любимый текстовый редактор и не создать исходный код для приложения с нуля? Можно было бы, конечно, заняться и этой нудной работой. В конце концов, Rails-приложение — это всего лишь исходный код на языке Ruby. Но Rails, кроме всего прочего, творит множество скрытых чудес, заставляя наши приложения работать при минимуме явных настроек. Чтобы заставить Rails

творить чудеса, ей нужно дать возможность отыскать все разнообразие компонентов вашего приложения. Далее (в разделе 18.1 «Где что искать») станет ясно, что для этого нужно создать конкретную структуру каталогов, разложив создаваемый нами код по надлежащим местам. Команда `rails` просто создает для нас эту структуру каталогов и заполняет ее стандартным Rails-кодом.

Для создания нашего первого Rails-приложения откройте окно командной оболочки и перейдите в то место файловой системы, в котором вы хотите создать структуру каталогов для своих приложений. В нашем примере мы будем создавать приложения в каталоге под названием work. Находясь в этом каталоге, введите команду `rails` для создания приложения с названием demo. Имейте в виду: если у вас уже существует каталог с таким названием, вам будет задан вопрос, не хотите ли вы перезаписать все существующие в нем файлы.

ПРИМЕЧАНИЕ

Если, как описывалось в разделе 1.4 «Выбор версии Rails», нужно указать, какую из версий Rails использовать, то это следует сделать именно здесь.

```
rubys> cd work
work> rails new demo
create
create README.rdoc
create Rakefile
create config.ru
  :         :     :
create vendor/assets/stylesheets
create vendor/assets/stylesheets/.keep
    run bundle install
Fetching gem metadata from https://rubygems.org/...........
  :         :     :
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is installed.
work>
```

Команда создала каталог по имени demo. Зайдите в него и выведите его содержимое (используя `ls` в окне Unix или `dir` при работе под Windows).

Вы должны увидеть набор файлов и подкаталогов:

```
work> cd demo
demo> dir /w
[.]            [..]            [app]          [bin]          [config]
config.ru       [db]           Gemfile        Gemfile.lock   [lib]
[log]          [public]        Rakefile.rdoc  README         [test]
[tmp]          [vendor]
```

Поначалу все эти каталоги (и файлы, которые в них содержатся) могут отпугнуть, но на данный момент на существование большинства из них можно просто не обращать внимания. В данной главе непосредственно будет использоваться только один из них: каталог app, в котором будет создаваться наше приложение.

Включенные туда файлы представляют собой все, что нужно для запуска автономного веб-сервера, способного выполнять наше только что созданное Rails-приложение. Давайте без лишних проволочек запустим наше приложение под названием demo:

```
demo> rails server
=> Booting WEBrick
=> Rails 4.0.0 application starting on http://0.0.0.0:3000
=> Run 'rails server -h' for more startup options
=> Ctrl-C to shutdown server
[2013-04-18 20:22:16] INFO WEBrick 1.3.1
[2013-04-18 20:22:16] INFO ruby 2.0.0 (2013-02-24) [x86_64-linux]
[2013-04-18 20:22:16] INFO WEBrick::HTTPServer#start: pid=25170 port=3000
```

Какой веб-сервер запущен, зависит от того, какой сервер установлен. WEBrick — веб-сервер, специально предназначенный для Ruby, который распространяется вместе с Ruby и поэтому является гарантированно доступным. Но если на вашей системе установлен другой веб-сервер (и Rails может его найти), команда `rails server` может предпочесть его серверу WEBrick. Rails можно заставить воспользоваться WEBrick, предоставив команде `rails` соответствующий ключ:

```
demo> rails server webrick
```

В последней строчке трассировка запуска показывает, что мы только что запустили веб-сервер с использованием порта 3000. Часть адреса 0.0.0.0 означает, что WEBrick допустит подключения ко всем интерфейсам. На установленной у Дэйва системе OS X это означает доступ к обоим локальным интерфейсам (127.0.0.1 и ::1) и его подключению к локальной сети. Мы можем получить доступ к приложению, указав веб-браузеру URL-адрес http://localhost:3000. Результат показан на рис. 2.1.

Если смотреть в окно, из которого был запущен сервер, вы увидите трассировку, показывающую, что вы запустили приложение. Мы собираемся оставить сервер работать в этом консольном окне. Позже, после того как вы напишете код приложения и оно будет запущено в нашем браузере, у нас будет возможность воспользоваться этим консольным окном для отслеживания входящих запросов. Когда наступит время выключить ваше приложение, можно будет, находясь в этом окне, нажать Ctrl+C, чтобы остановить WEBrick (не делайте этого сейчас, через минуту мы именно этим приложением и воспользуемся).

Итак, у нас есть запущенное новое приложение, но в нем вообще нет нашего кода. Давайте исправим эту ситуацию.

2.2. Привет, Rails!

Ничего не поделаешь, для испытания новой системы придется написать программу *Hello, World!* Начнем с создания простейшего приложения, отправляющего на браузер наше радостное приветствие. Как только оно заработает, мы украсим его показом текущего времени и ссылками.

В главе 3 «Архитектура Rails-приложений» мы выясним, что Rails относится к среде Модель—Представление—Контроллер (Model—View—Controller). Rails получает входящие запросы от браузера, декодирует запрос для поиска контроллера и вызывает в контроллере метод, который называется действием. Затем контроллер вызывает соответствующее представление, отображающее результаты на экране пользователя. Для нас Rails хороша тем, что берет на себя управление большей частью внутренних каналов, связывающих все эти действия. Чтобы создать простейшее приложение *Hello, World!*, нам нужен код для контроллера и представления и нужен связывающий их маршрут. Для модели нам код не нужен, поскольку мы не имеем дела с данными. Начнем с контроллера.

Затем контроллер вызовет конкретное представление, чтобы пользователь мог увидеть результаты.

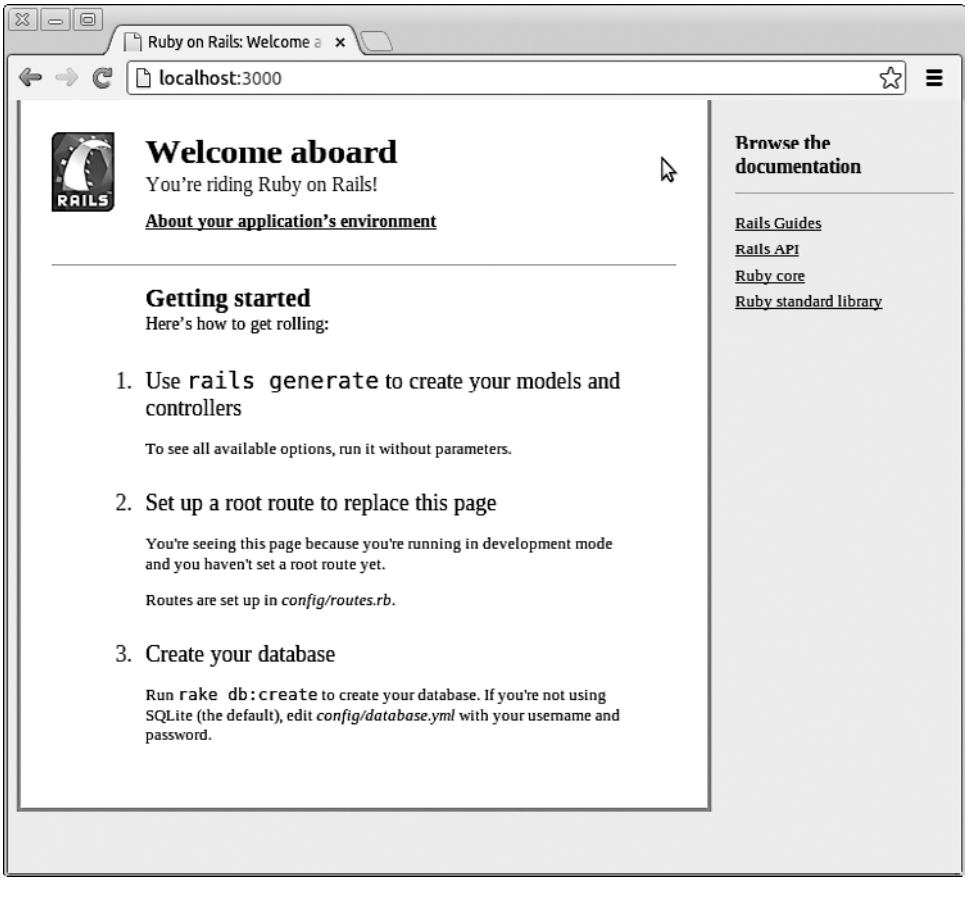


Рис. 2.1. Только что созданное Rails-приложение

Аналогично тому, как для создания нового Rails-приложения мы воспользовались командой `rails`, для создания нового контроллера для нашего проекта мы

можем также воспользоваться генерирующим сценарием. Он вызывается командой `rails generate`. Итак, для создания контроллера под названием `say` нужно убедиться, что мы находимся в каталоге demo, и запустить команду, передав ей имя создаваемого контроллера и имена действий, предназначаемых для поддержки этого контроллера:

```
demo> rails generate controller Say hello goodbye
create app/controllers/say_controller.rb
  route get "say/goodbye"
  route get "say/hello"
invoke erb
create   app/views/say
create   app/views/say/hello.html.erb
create   app/views/say/goodbye.html.erb
invoke test_unit
create   test/controllers/say_controller_test.rb
invoke helper
create   app/helpers/say_helper.rb
invoke   test_unit
create     test/unit/helpers/say_helper_test.rb
invoke assets
invoke   coffee
create     app/assets/javascripts/say.js.coffee
invoke   scss
create     app/assets/stylesheets/say.css.scss
```

Команда `rails generate` ведет протокол проверяемых файлов и каталогов, отмечая добавление новых Ruby-сценариев или каталогов к вашему приложению. На данный момент нас интересует один из этих сценариев, а еще через минуту заинтересуют и файлы .html.erb.

Сначала нужно найти исходный файл контроллера. Он находится в файле app/controllers/say_controller.rb. Посмотрим на его содержимое:

rails40/demo1/app/controllers/say_controller.rb

```
class SayController < ApplicationController
    def hello
   end

    def goodbye
    end
end
```

Довольно лаконично, не правда ли? `SayController` — это класс, наследуемый из `ApplicationController`, поэтому он автоматически перенимает все исходное поведение контроллера. Что этот код должен делать? Пока он не делает ничего, у нас просто есть пустые методы действия по имени `hello()` и `goodbye()`. Чтобы понять, почему они так названы, нужно посмотреть, как Rails обрабатывает запросы.

Rails и запросы URL-адресов

В представлении своих пользователей Rails-приложение, как, собственно, и любое другое веб-приложение, связано с URL-адресом. Когда браузеру указывается этот URL-адрес, происходит обращение к коду приложения, который генерирует для вас ответ.

Давайте попробуем прямо сейчас. Перейдите в окне браузера на URL-адрес http://localhost:3000/say/hello. Вы увидите на экране окно, похожее на то, которое показано на рис. 2.2.

Рис. 2.2. Шаблон, подготовленный для нашего заполнения

Наше первое действие

На данный момент мы можем увидеть не только ответ на вопрос, подключен ли URL-адрес к нашему контроллеру, но также и то, что Rails указывает путь к нашему следующему шагу, а именно — сообщению Rails о том, что отобразить на экране. Здесь в игру вступают представления. Вспомните: когда мы запустили сценарий для создания нового контроллера, эта команда добавила к нашему приложению шесть файлов и новый каталог. В этом каталоге содержатся файлы шаблонов для представлений контроллера. В нашем случае мы создали контроллер, названный `say`, следовательно, представления будут находиться в каталоге app/views/say.

По умолчанию Rails ищет шаблоны в файле с тем же именем, что и у действия, занимающегося его обработкой. В нашем случае это означает, что нам нужно заменить файл по имени hello.html.erb в каталоге app/views/say. (Почему .html.erb? Совсем скоро мы все объясним.) А теперь давайте просто поместим в файл элементарный код HTML:

rails40/demo1/app/views/say/hello.html.erb

```
<h1>Привет от Rails!</h1>
```

Сохраните файл hello.html.erb и обновите окно браузера. Вы увидите в нем наше приветствие (рис. 2.3).

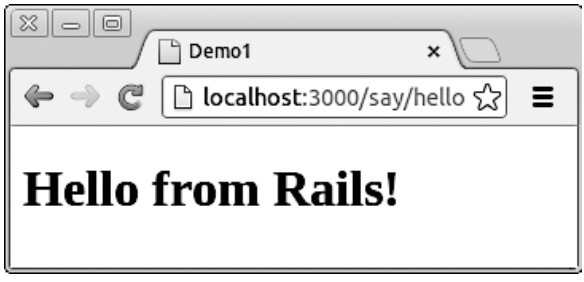


Рис. 2.3. Наше приветствие в окне браузера

Всего мы просмотрели два файла в нашем дереве Rails-приложения. Мы смотрели на контроллер, и мы изменили шаблон отображения страницы в браузере.

Эти файлы находятся в стандартных местах иерархической структуры Rails: контроллеры размещаются в каталоге app/controllers, а представления — в подкаталогах, принадлежащих каталогу app/views. Стандартные места для контроллеров и представлений показаны на рис. 2.4.

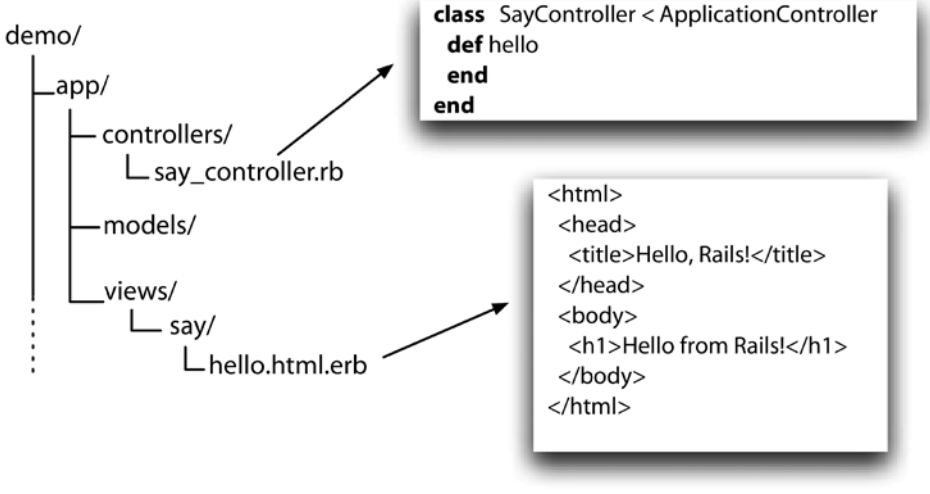


Рис. 2.4. Стандартные места для контроллеров и представлений

Придание динамичности

Пока у нашего Rails-приложения довольно скучный вид — оно просто отображает статичную страницу. Для придания ему динамичности давайте заставим его показывать текущее время при каждом отображении страницы.

Для этого необходимо внести изменения в файл шаблона в представлении: теперь в него нужно включить время в виде строки. При этом возникает два вопроса. Во-первых, как добавляется динамическое содержимое к шаблону? И во-вторых, откуда мы возьмем это время?

Динамическое содержимое

В Rails есть множество способов создания динамических шаблонов. Наиболее распространенный из них, которым мы здесь воспользуемся, заключается во вставке кода Ruby непосредственно в шаблон. Именно поэтому наш файл шаблона называется hello.html.erb — суффикс .html.erb предписывает Rails расширить содержимое файла с помощью системы, которая называется ERB.

ERB — это фильтр, устанавливаемый как часть Rails, который берет файл .erb и выдает преобразованную версию. Выходной файл в Rails чаще всего имеет формат HTML, но вообще-то может иметь какой угодно формат. Обычно содержимое пропускается через фильтр без изменений. А содержимое, находящееся между группами символов <%= и %>, интерпретируется как Ruby-код, который выполняется. В результате этого выполнения содержимое превращается в строку, значение которой подставляется в файл вместо последовательности <%=…%>. Например, внесите в hello.html.erb изменения, позволяющие вывести текущее время:

rails40/demo2/app/views/say/hello.html.erb

```
<h1>Привет от Rails!</h1>
<p>
    Сейчас <%= Time.now %>
</p>
```

После обновления окна нашего браузера мы увидим время, выведенное на экран в стандартном для Ruby формате (рис. 2.5).

Рис. 2.5. Время в стандартном для Ruby формате

Обратите внимание: при щелчке на кнопке браузера Обновить (Refresh) время обновляется при каждом выводе страницы. Это похоже на настоящую генерацию динамического содержимого.

Добавление времени

Исходная задача заключалась в демонстрации времени пользователям нашего приложения. Теперь мы знаем, как заставить приложение выводить динамические данные. Но нужно решить еще одну задачу: откуда брать время?

Мы показали, что подход со вставкой вызова Ruby-метода `Time.now()` в нашем шаблоне hello.html.erb работает. При каждом обращении к этой странице пользователь будет видеть текущее время, вставленное в тело ответа. Для нашего простейшего приложения этого может быть вполне достаточно. Но нам все-таки хочется сделать все немного по-другому. Мы переместим определение отображаемого значения текущего времени в контроллер, а представлению оставим простую работу по его отображению. Для этого мы изменим метод действия в контроллере, чтобы присвоить значение времени переменной экземпляра под названием `@time`:

rails40/demo3/app/controllers/say_controller.rb

```
class SayController < ApplicationController
    def hello
▶       @time = Time.now
    end

    def goodbye
    end
end
```

В шаблоне .html.erb мы воспользуемся этой переменной экземпляра, чтобы подставить время в выводимые данные:

rails40/demo3/app/views/say/hello.html.erb

```
<h1>Привет от Rails!</h1>
<p>
▶   Сейчас <%= @time %>
</p>
```

После обновления окна браузера мы опять увидим текущее время, что свидетельствует об успешной связи между контроллером и представлением.

УПРОЩЕНИЕ РАЗРАБОТКИ

В процессе той разработки, которой мы только что занимались, уже можно было кое-что заметить. Когда к уже работающему приложению добавлялся какой-нибудь код, его не приходилось перезапускать. Все необходимое делалось без нашего участия в фоновом режиме. И каждое вносимое нами изменение становилось доступным, как только мы обращались к приложению через браузер. Благодаря чему все это происходило?

Оказывается, благодаря достаточно разумному поведению диспетчера Rails. В режиме разработки (в отличие от режимов тестирования или эксплуатации) при поступлении нового запроса диспетчер автоматически перезагружает исходные файлы приложения. Таким образом, при редактировании приложения диспетчер обеспечивает запуск самой последней версии. Для режима разработки это очень хорошее качество.

Тем не менее за такую гибкость поведения приходится платить — она приводит к небольшой паузе между вводом URL-адреса и откликом приложения, которая вызвана тем, что

диспетчер осуществляет перезагрузку. Для разработки это вполне разумная цена, но при эксплуатации готового приложения такой режим будет неприемлем. Поэтому перед развертыванием приложения этот режим отключается (см. главу 16 «Задача Л: развертывание и эксплуатация»).

Зачем нам понадобились дополнительные хлопоты по установке отображаемого времени в контроллере с последующим использованием этого времени в представлении? Хороший вопрос. В данном приложении нет практически никакой разницы, но, помещая логику работы приложения из представления в контроллер, мы получаем некоторые преимущества. Например, в будущем может потребоваться усовершенствовать наше приложение, чтобы оно поддерживало работу пользователей во многих странах. В таком случае понадобится локализовать отображение времени, выбрав время, соответствующее часовому поясу пользователей. Для этого может понадобиться достаточно большой объем кода на уровне приложения, и, возможно, его вставка на уровне представления окажется неприемлемой. Устанавливая значение отображаемого времени в контроллере, мы придаем своему приложению дополнительную гибкость — теперь мы можем изменять часовой пояс в контроллере, и тогда уже не придется обновлять каждое представление, в котором используется этот объект времени. Время — это *данные*, которыми контроллер должен снабжать представление. При согласовании моделей вам будет представлено намного больше подобных примеров.

Итак, чего мы добились

Рассмотрим вкратце работу созданного приложения.

1. Пользователь переходит к работе с приложением, воспользовавшись в нашем случае локальным URL-адресом http://localhost:3000/say/hello.
2. Затем Rails проводит сравнение со схемой маршрута, которая предварительно разбивается на две части.

   Фрагмент say воспринимается как имя контроллера, поэтому Rails создает новый экземпляр Ruby-класса `SayController` (который находит в файле app/controllers/say_controller.rb).
3. Следующий фрагмент URL, hello, рассматривается как идентификатор действия. Rails вызывает в контроллере метод с таким же названием. Этот метод, определяющий действие, создает новый объект `Time`, содержащий текущее время, и убирает его содержимое в переменную экземпляра `@time`.
4. Rails ищет шаблон, чтобы отобразить результат. Для этого просматривается каталог app/views, в котором идет поиск подкаталога с именем, аналогичным имени контроллера (say), а в этом подкаталоге ищется файл, названный по имени действия (hello.html.erb).
5. Rails обрабатывает этот файл с использованием системы работы с шаблонами ERB, исполняя все Ruby-вставки и подставляя значения, установленные контроллером.
6. Результат возвращается браузеру, и Rails завершает обработку данного запроса.

Но это еще не все. Rails предоставляет вам массу возможностей для внесения изменений в основную последовательность выполняемых действий (и мы отчасти этим воспользуемся). В данных условиях наша история является типичным примером использования соглашения по конфигурации, одной из основ философии Rails. Благодаря тому, что Rails-приложениям предоставляются удобные исходные настройки, а при их разработке применяются определенные соглашения по составлению URL-адреса или по файлу, в который помещается определение контроллера, а также по имени используемого класса и именам методов, Rails-приложения создаются, как правило, с незначительным использованием или вовсе без использования внешних конфигурационных настроек, что позволяет связывать их в единое целое вполне естественным образом.

2.3. Соединение страниц

Веб-приложения, состоящие из одной страницы, встречаются довольно редко. Давайте посмотрим, как к нашему приложению с приветствием можно добавить еще один впечатляющий пример веб-дизайна.

Обычно каждой странице приложения соответствует отдельное представление. В нашем случае для обработки страницы также будет использован новый метод действия (хотя далее в данной книге будет показано, что так бывает не всегда). Для обоих действий будет использован один и тот же контроллер. Вообще-то, так делать необязательно, но каких-то причин, вынуждающих использовать новый контроллер, пока нет.

Для этого контроллера уже было определено действие goodbye, поэтому остается только создать новый шаблон в каталоге app/views/say. На этот раз он называется goodbye.html.erb, поскольку по умолчанию шаблоны получают имена тех действий, которые с ними связаны.

rails40/demo4/app/views/say/goodbye.html.erb

```
<h1>До свидания!</h1>
<p>
    Очень приятно, что вы нас навестили.
</p>
```

Обратимся еще раз к нашему уже испытанному браузеру, но теперь уже укажем на новое действие, воспользовавшись URL http://localhost:3000/say/goodbye. Должна появиться картинка, похожая на рис. 2.6.

Теперь нам нужно связать эти два экрана. Мы поместим на страницу hello ссылку, которая приведет нас на страницу goodbye, и наоборот. В настоящем приложении для этого, скорее всего, понадобится создать соответствующие кнопки, но сейчас мы воспользуемся простыми гиперссылками.

Нам уже известно, что в Rails используется соглашение о синтаксическом разборе URL-адреса с целью получения целевого контроллера и имеющиеся внутри него действия. Поэтому проще всего будет воспользоваться этим URL-соглашением для наших ссылок.

Рис. 2.6. Наше второе действие

В файле hello.html.erb будет содержаться следующий код:

```
...
<p>
    Сказать <a href="/say/goodbye">до свиданья</a>!
</p>
...
```

А в файле goodbye.html.erb будет указатель на обратный путь:

```
...
<p>
    Сказать <a href="/say/hello">привет</a>!
</p>
...
```

Конечно, все это будет работать, но сам по себе этот код слишком несовершенен. Если придется перемещать наше приложение в другое место на веб-сервере, URL-адреса уже не будут указывать на правильное место. К тому же в наш код также закладываются предположения об используемом в Rails формате URL-адреса, который в будущих версиях Rails может измениться.

К счастью, мы можем обойтись без всякого риска. Rails поставляется с целым пакетом *вспомогательных методов*, которые могут использоваться в шаблонах представлений. В данном случае мы воспользуемся вспомогательным методом `link_to()`, создающим гиперссылку на действие. (Метод `link_to()` способен на большее, но пока ограничимся только этой возможностью.) При использовании `link_to()` hello.html.erb приобретает следующий вид:

rails40/demo5/app/views/say/hello.html.erb

```
<h1>Привет от Rails!</h1>
<p>
    Сейчас <%= @time %>
</p>
<p>
    Пора сказать
    <%= link_to "до свидания", say_goodbye_path %>!
</p>
```

Здесь мы видим вызов метода `link_to()` внутри ERB-последовательности `<%=…%>`. За счет него создается ссылка на URL-адрес, который вызовет действие `goodbye()`. Первым аргументом вызова `link_to()` является текст, отображаемый в гиперссылке, а второй аргумент заставляет Rails сгенерировать ссылку на действие `goodbye()`.

Давайте уделим пару минут рассмотрению порядка генерации ссылки. Мы написали следующий код:

```
link_to "до свидания", say_goodbye_path
```

Начнем с того, что `link_to()` — это вызов метода. (В Rails методы, облегчающие написание шаблонов, называются *помощниками*.) Если вам приходилось работать с такими языками, как Java, отсутствие обязательных круглых скобок вокруг аргументов метода может вызвать удивление. Но если они вам нравятся, их всегда можно добавить.

Фрагмент `say_goodbye_path` является заранее вычисленным значением, которое Rails делает доступным для представлений приложения. Этот фрагмент превращается в путь /say/goodbye. Со временем вы увидите, что Rails предоставляет возможность дать имена всем маршрутам, которые будут использоваться в вашем приложении.

А теперь вернемся к приложению. Если направить браузер на открытие нашей страницы hello, то на ней будет ссылка на страницу goodbye (рис. 2.7).

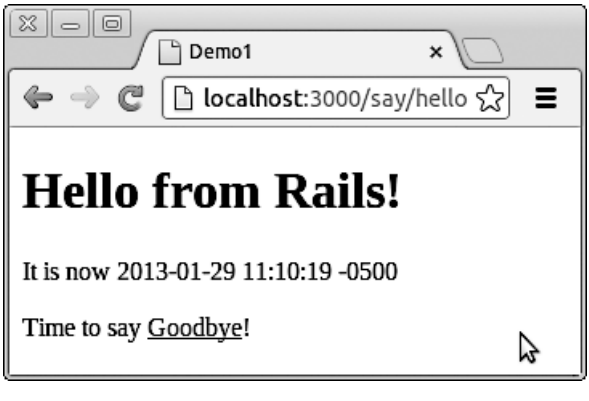


Рис. 2.7. Страница приветствия со ссылкой на страницу прощания

Соответствующие изменения могут быть сделаны и в файле goodbye.html.erb, чтобы появилась ссылка на начальную страницу hello:

rails40/demo5/app/views/say/goodbye.html.erb

```
<h1>До свиданья!</h1>
<p>
    Очень приятно, что вы нас навестили.
</p>
<p>
    Сказать <%= link_to "привет", say_hello_path %> еще раз.
</p>
```

Вот мы и завершили разработку нашего пробного приложения, проверив тем самым работоспособность нашей установки Rails. После краткого резюме наступит время перехода к созданию настоящего приложения.

Наши достижения

Мы создали пробное приложение, увидев при этом:

- ☑ как создается новое Rails-приложение и как создается новый контроллер этого приложения;
- ☑ как в контроллере создается динамическое содержимое и как оно отображается через шаблон представления;
- ☑ как страницы соединяются друг с другом.

Это послужит неплохой стартовой площадкой, для создания которой не пришлось тратить слишком много времени или усилий. Этот опыт будет наращиваться при переходе к следующей главе и создании нового, более крупного приложения.

Чем заняться на досуге

Поэкспериментируйте со следующими выражениями:

- ❍ Сложение: `<%= 1+2 %>`
- ❍ Объединение: `<%= "cow" + "boy" %>`
- ❍ Время через час: `<%= 1.hour.from_now %>`

  Вызов следующего Ruby-метода возвращает список всех файлов в текущем каталоге:

  ```
  @files = Dir.glob('*')
  ```

  Воспользуйтесь им для задания значения переменной экземпляра в действии контроллера, а затем напишите соответствующий шаблон, выводящий в браузере список имен файлов.

  Подсказка: просканировать коллекцию можно с помощью следующего кода:

  ```
  <% for file in @files %>
  file name is: <%= file %>
  <% end %>
  ```

  Для формирования списка можно воспользоваться тегом `<ul>`.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

Ликвидация последствий

Если вы, следуя за повествованием, набирали код, то, возможно, приложение все еще работает на вашем компьютере. Когда в главе 6, «Задача A: создание приложения», мы приступим к программированию нашего следующего приложения, при его первом запуске возникнет конфликт, поскольку для общения с браузером оно также попытается воспользоваться компьютерным портом 3000. Сейчас как раз настало время остановить текущее приложение, нажав комбинацию клавиш Ctrl+C в том окне, которое использовалось для его запуска. Пользователям Microsoft Windows вместо этого может понадобиться нажать комбинацию клавиш Ctrl+Pause/Break.

А теперь перейдем к обзору Rails.

3 Архитектура Rails-приложений

Основные темы:

- модели;
- представления;
- контроллеры.

Одной из существенных особенностей Rails является то, что она накладывает весьма серьезные ограничения на структурирование веб-приложений. Как ни странно, эти ограничения упрощают создание приложений, причем существенно. Посмотрим, почему так происходит.

3.1. Модели, представления и контроллеры

В далеком 1979 году Трюгве Реенскауг (Trygve Reenskaug) придумал новую архитектуру для разработки интерактивных приложений. По его замыслу приложения разбиваются на компоненты трех типов: модели, представления и контроллеры.

Модель отвечает за поддержку состояния приложения. Иногда это состояние является кратковременным, продолжающимся только на время нескольких взаимодействий с пользователем. А иногда состояние является постоянным и сохраняется вне приложения, чаще всего в базе данных.

Модель — это не просто данные; в ней прописаны все бизнес-правила, применяемые к этим данным. К примеру, если скидка не должна применяться к заказам стоимостью менее 20 долларов, моделью будут предписаны соответствующие ограничения. И в этом есть свой определенный смысл. Помещая реализацию бизнес-правил в модель, мы гарантируем, что ничто иное в приложении

не может сделать наши данные некорректными. Модель работает и сторожем, и хранителем данных.

Представление отвечает за формирование пользовательского интерфейса, который обычно основан на данных модели. Например, интернет-магазин должен иметь перечень товаров для отображения на экране каталога. Этот перечень доступен через модель, но именно представление будет его форматировать для показа конечному пользователю. Хотя представление может предлагать пользователю различные способы ввода данных, оно никогда не занимается их непосредственной обработкой. Работа представления завершается, как только данные будут отображены на экране. Доступ к одним и тем же данным модели, зачастую с разными целями, может иметь множество представлений. В интернет-магазине может быть представление, отображающее информацию о товаре на странице каталога, и другой набор представлений, используемых администраторами для добавления товаров или редактирования сведений о них.

Контроллеры организуют работу приложения. Они воспринимают события внешнего мира (обычно ввод данных пользователем), взаимодействуют с моделью и отображают соответствующее представление для пользователя.

Этот триумвират — модель, представление и контроллер — формирует архитектуру, известную как MVC (Model—View—Controller — Модель—Представление-Контроллер). Взаимодействие этих трех элементов концепции показано на рис. 3.1.

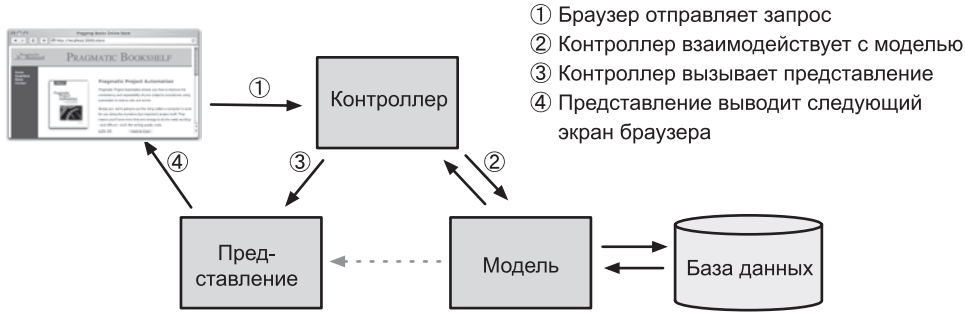


Рис. 3.1. Архитектура Модель—Представление—Контроллер

Сначала архитектура MVC предназначалась для обычных GUI-приложений, при создании которых разработчики поняли, что распределение ролей приводит к существенному уменьшению взаимных увязок, что, в свою очередь, облегчает написание и сопровождение программного кода. Каждое понятие или действие четко определялось только в одном, хорошо известном месте. Использование MVC напоминало строительство небоскреба при наличии уже готового каркаса: когда есть готовая структура, навесить на нее все остальные элементы намного проще. В процессе разработки нашего приложения мы часто будем пользоваться возможностями Rails по генерации для нашего приложения *временных платформ* (scaffold).

Ruby on Rails также относится к среде MVC. Rails навязывает структуру для вашего приложения — вы разрабатываете модели, представления и контроллеры как отдельные функциональные блоки, а Rails при выполнении вашей программы связывает их вместе. Изюминкой Rails является то, что процесс увязки базируется на использовании разумных умолчаний, которые, как правило, избавляют вас от написания каких-либо внешних конфигурационных метаданных, обеспечивающих взаимную работу. Приоритет соглашения над конфигурацией является примером философии Rails.

В Rails-приложении входящий запрос сначала посылается маршрутизатору, который решает, в какое место приложения должен быть отправлен запрос и как должен быть произведен синтаксический разбор этого запроса. В конечном итоге на данном этапе где-то в коде контроллера идентифицируется конкретный метод (называемый в Rails *действием*). Действие может искать запрошенные данные, может взаимодействовать с моделью и может вызвать другое действие. В результате выполнения действие подготавливает информацию для представления, которое создает изображение для пользователя.

Rails обрабатывает входящие запросы по схеме, показанной на рис. 3.2. В данном примере приложение уже вывело на экран страницу каталога товаров и пользователь только что щелкнул на кнопке Добавить в корзину, расположенной рядом с одним из товаров. Щелчок на этой кнопке приводит к отправке сообщения по адресу http://localhost:3000/line_items?product_id=2, где `line_items` — это ресурс в нашем приложении, а `2` — это внутренний идентификатор выбранного товара.

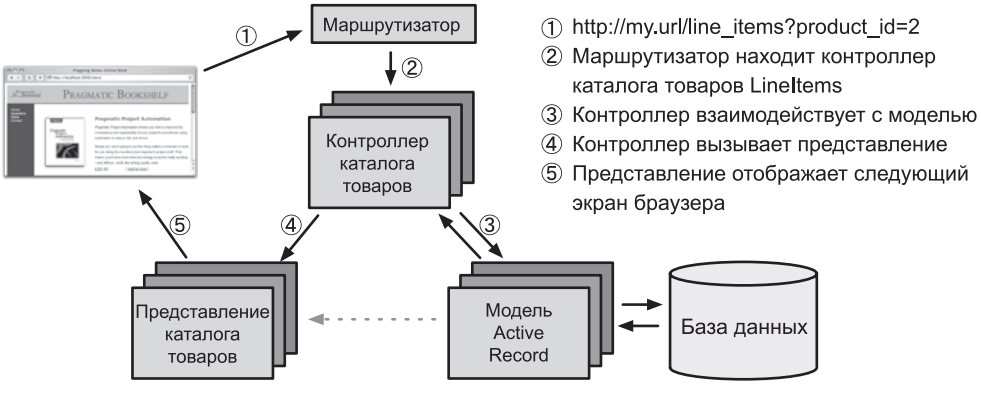


Рис. 3.2. Rails и MVC

Компонент, занимающийся маршрутизацией, получает входящий запрос и тут же разбивает его на части. В запросе содержится путь (/line_items?product_id=2) и метод (данная кнопка инициирует операцию POST; другими обычными методами являются GET, PUT, PATCH и DELETE). В нашем простейшем примере Rails рассматривает первую часть пути, line_items, в качестве имени контроллера, а product_id — в качестве идентификатора товара. По соглашению POST-методы

связаны с действием `create()`. В результате этого анализа маршрутизатор знает, что нужно вызвать метод `create()` в классе контроллера `LineItemsController` (соглашение об именах будет рассмотрено в разделе 18.2).

Метод `create()` занимается обработкой пользовательских запросов. В данном случае он ищет корзину покупок текущего пользователя (которая является объектом, управляемым моделью). Он также обращается к модели за поиском информации для товара 2. Затем он заставляет корзину добавить в нее этот товар. (Видите, как модель используется для отслеживания всех бизнес-данных? Контроллер говорит ей, *что* делать, а модель знает, *как* это сделать.)

Теперь, когда в корзине новый товар, ее можно показать пользователю. Контроллер вызывает код представления, но перед этим он делает так, чтобы представление имело доступ к объекту корзины `cart` из модели. В Rails этот вызов зачастую просто подразумевается, здесь опять соглашения помогают связать конкретное представление с данным действием.

Это все, что можно сказать в отношении веб-приложения, использующего архитектуру MVC. Следуя набору соглашений и соответствующим образом разграничивая выполняемые функции, вы увидите, что с вашим кодом становится проще работать, а ваше приложение легче поддается расширению и поддержке. Все это напоминает хорошо организованную коммерческую деятельность.

Если концепция MVC сводится всего лишь к вопросу конкретного способа разделения вашего программного кода, зачем тогда нужна такая среда, как Ruby on Rails? Ответ очевиден: Rails берет на себя всю низкоуровневую работу, всю эту мелочную возню, которая отнимает у вас так много времени, и позволяет вам сконцентрироваться на основных функциях вашего приложения. Посмотрим, как это делается.

3.2. Поддержка модели Rails

Как правило, от нашего веб-приложения требуется, чтобы его информация хранилась в реляционной базе данных. В системах учета поступающих заказов все заказы, товарные позиции и данные клиентов хранятся в таблицах базы данных. Даже такие приложения, которые обычно работают с неструктурированным текстом, например блоги и сайты новостей, зачастую используют базы данных в своих серверных хранилищах.

Хотя из названия языка SQL[1], который используется для доступа к базам данных, этого сразу и не понять, но все реляционные базы данных фактически построены на основе математической теории множеств. Может быть, концептуально это и неплохо, но данное обстоятельство сильно затрудняет сочетаемость реляционных баз данных с объектно-ориентированными языками программирования. Объекты описываются данными и операциями, а базы данных — наборами значений. Операции, которые легко выражаются в реляционных понятиях, иногда

[1] SQL, который некоторые называют структурированным языком запросов — Structured Query Language, является языком, используемым для запросов к реляционным базам данных и для обновления их содержимого.

довольно трудно поддаются программированию в объектно-ориентированных системах. Справедливо также и обратное утверждение.

Со временем специалисты разработали ряд способов согласования реляционных и объектно-ориентированных взглядов на свои корпоративные данные. Давайте посмотрим на тот способ отображения реляционных данных на объекты, который был выбран в Rails.

Объектно-реляционное отображение

Отображением таблиц баз данных на классы занимаются библиотеки ORM. Если в базе данных есть таблица под названием `orders` (заказы), у нашей программы будет класс по имени `Order`. Сроки в этой таблице соответствуют объектам класса — конкретный заказ представлен объектом класса `Order`. Для получения и установки значений отдельных столбцов используются свойства этого объекта. У нашего объекта `Order` есть методы получения и установки количества товара, размера налога с продаж и т. д.

Кроме этого Rails-классы, поглотившие таблицы нашей базы данных, предоставляют набор методов на уровне класса, которые выполняют операции на уровне всей таблицы. Например, может потребоваться найти заказ с конкретным идентификатором. Этот поиск реализуется в виде метода класса, возвращающего соответствующий объект `Order`. В коде Ruby это может иметь следующий вид:

```
order = Order.find(1)
puts "Клиент #{order.customer_id}, количество=$#{order.amount}"
```

Иногда такие методы на уровне класса возвращают коллекцию объектов:

```
Order.where(name: 'dave').each do |order|
    puts order.amount
end
```

И, наконец, объекты, соответствующие отдельным строкам таблицы, имеют методы, работающие со строкой. Наверное, наиболее широко используемым является метод `save()`, выполняющий операцию сохранения строки в базе данных:

```
Order.where(name: 'dave').each do |order|
    order.pay_type = "Заказ товара"
    order.save
end
```

Итак, на уровне ORM таблицы отображаются на классы, строки — на объекты, а столбцы — на свойства этих объектов. Методы класса используются для выполнения операций на уровне таблиц, а методы экземпляров выполняют операции над отдельными строками.

Для определения порядка отображения между примитивами базы данных и примитивами программы обычной ORM-библиотеке предоставляются конфигурационные данные. Программисты, использующие такие ORM-инструменты, знают, что им придется создавать и поддерживать полный пакет конфигурационных файлов в формате XML.

Active Record

Active Record — это ORM-вставка, предоставляемая Rails. Она строго следует стандартам ORM-модели: таблицы отображаются на классы, строки — на объекты, а столбцы — на свойства объекта. От большинства других ORM-библиотек она отличается способом конфигурирования. Основываясь на соглашении и приступая к работе с оптимальными настройками по умолчанию, Active Record сводит к минимуму объем настроек, выполняемых разработчиками.

Чтобы проиллюстрировать эту особенность, рассмотрим программу, использующую Active Record для поглощения нашей таблицы заказов:

```
require 'active_record'

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.pay_type = "Заказ товара"
order.save
```

В этом коде для извлечения ордера с идентификатором, равным 1, и изменения свойства `pay_type` используется новый класс `Order`. (Здесь не показан код, создающий соответствующее подключение к базе данных.) Active Record избавляет нас от возни с исходной базой данных, позволяя спокойно работать над бизнес-логикой.

Но Active Record занимается не только этим. В главе 5, при разработке приложения, обслуживающего корзину покупок, будет показано, что Active Record легко интегрируется со всей остальной средой Rails. Если веб-форма отправляет данные приложения, связанные с бизнес-объектом, Active Record может извлечь их в нашу модель. Active Record поддерживает современную проверку приемлемости данных модели, и если данные формы не проходят проверку, представление Rails может извлечь и отформатировать ошибки.

Active Record является надежным фундаментом модели MVC-архитектуры, используемой в Rails.

3.3. Action Pack: представление и контроллер

Если призадуматься, части MVC — представление и контроллер — тесно связаны друг с другом. Контроллер снабжает представление данными, и он же воспринимает события от страниц, сгенерированных представлениями. Из-за такого тесного взаимодействия поддержка представлений и контроллеров в Rails объединена в единый компонент — Action Pack.

Но не стоит думать, что в вашем приложении код представления и код контроллера будут перемешаны только потому, что Action Pack является единым

компонентом. Все как раз наоборот, Rails предоставляет вам разделение, необходимое для создания приложений с четко разграниченным кодом для логики управления и представления.

Поддержка представления

Представление в Rails отвечает за создание полного или частичного ответа, отображаемого в браузере, обработанного приложением или посланного в виде электронной почты. В простейшем виде представление является фрагментом HTML-кода, отображающего какой-нибудь неизменный текст. Но чаще всего вам потребуется включить динамическое содержимое, созданное методом действия в контроллере.

Динамическое содержимое в Rails генерируется шаблонами трех видов. В самой распространенной схеме создания шаблонов, которая называется встроенным Ruby — Embedded Ruby (ERb), фрагменты кода Ruby вставляются в представляемый документ, что во многом похоже на способ, применяемый в других веб-средах, например в PHP или в JSP. При всей гибкости данного подхода некоторые специалисты озабочены тем, что он нарушает сам смысл MVC. Вставляя код в представление, мы рискуем заложить в него логику, которая должна быть в модели или в контроллере. Как и во всем остальном, разумная умеренность будет полезной, а злоупотребление может превратиться в серьезную проблему. Соблюдение четкого разделения интересов является частью труда разработчика. (HTML-шаблоны будут рассмотрены в разделе 25.2 «Генерация HTML с ERb».)

ERb можно также использовать для конструирования на сервере JavaScript-фрагментов, выполняемых в браузере. Эта технология отлично подходит для создания динамичных AJAX-интерфейсов. К этому вопросу мы еще вернемся в разделе 11.2.

В Rails также имеется такой инструмент, как XML Builder, позволяющий конструировать XML-документы, использующие код Ruby, — структура генерируемого XML будет автоматически следовать за структурой кода. Шаблоны `xml.builder` будут рассмотрены, начиная с раздела 24.1.

И наконец, контроллер!

Контроллер Rails является логическим центром вашего приложения. Он координирует взаимодействие между пользователем, представлениями и моделью. Но с большинством этих взаимодействий Rails справляется без вашего участия, а создаваемый вами код концентрируется на функциональности прикладного уровня. Это существенно упрощает разработку и поддержку кода Rails-контроллера.

Контроллер также является местом для нескольких важных вспомогательных служб.

- Он отвечает за перенаправление внешних запросов внутренним действиям. Он отлично справляется с удобными для человеческого восприятия URL-адресами.

- Он управляет кэшированием, ускоряющим работу приложения в несколько раз.
- Он управляет вспомогательными модулями, расширяющими возможности шаблонов представлений без увеличения объема их кода.
- Он управляет сессиями, дающими пользователям ощущение непрерывного взаимодействия с нашими приложениями.

Мы уже работали с контроллером в разделе 2.2, а при разработке учебного приложения нам еще придется поработать с несколькими контроллерами, начиная с контроллера товаров в разделе 8.1 «Шаг B1: создание каталога товаров».

О Rails еще многое можно сказать. Но перед тем, как продолжить, давайте немного освежим свои знания Ruby, а для некоторых из вас предоставим краткое введение в этот язык.

4

Введение в Ruby

Основные темы:

- объекты: имена и методы;
- данные: строки, массивы, хэши и регулярные выражения;
- элементы управления: if, while, блоки, итераторы и исключения;
- строительные блоки: классы и модули;
- YAML и маршализация;
- общие идиомы, которые вы увидите в данной книге.

Для многих, кто только начинает знакомиться с Rails, также в новинку и Ruby. При знакомстве с такими языками, как Java, JavaScript, PHP, Perl или Python, освоить Ruby будет нетрудно.

Эта глава не является полноценным введением в Ruby. В ней не рассматриваются такие темы, как порядок выполнения действий (как и в большинстве других языков программирования, в Ruby 1+2*3==7). Она предназначена для объяснения Ruby только в том объеме, который позволяет сделать приводимые в книге примеры более понятными.

Эта глава во многом повторяет материал книги «Programming Ruby»[1]. Если у вас есть потребность в более основательном изучении языка Ruby, то мы, рискуя быть обвиненными в преследовании личных интересов, все же хотели бы предложить лучший способ изучения Ruby и лучший справочник по классам, модулям и библиотекам этого языка — книгу «Programming Ruby» (известную также как «PickAxe»). Добро пожаловать в сообщество Ruby!

---

[1] *David Thomas, Chad Fowler, and Andrew Hun*t. Programming Ruby: The Pragmatic Programmer's Guide. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Third, 2008.

4.1. Ruby — объектно-ориентированный язык

Все, с чем вы работаете в Ruby, является объектом, а результаты этой работы также являются объектами.

При написании объектно-ориентированного кода обычно обращаются к понятиям модели из реального мира. Как правило, в ходе этого процесса моделирования изучаются категории тех вещей, которые должны быть представлены. В интернет-магазине такой категорией может стать понятие отдельной товарной позиции. Для представления каждой из таких категорий в Ruby следует определить отдельный класс. Затем этот класс используется в качестве фабрики, создающей объекты — экземпляры этого класса. Объект является комбинацией состояния (например, количество и идентификатор товара) и методов, использующих это состояние (возможно, метода для вычисления общей стоимости товарной позиции). Создание классов будет показано в разделе 4.4.

Объекты создаются путем вызова *конструктора*, который является специальным методом, связанным с классом. Стандартный конструктор называется `new()`.

При наличии класса под названием `LineItem` объекты товарных позиций можно создавать следующим образом:

```
line_item_one = LineItem.new
line_item_one.quantity = 1
line_item_one.sku = "AUTO_B_00"
```

Методы вызываются путем отправки объекту сообщения. Это сообщение содержит имя метода наряду с любыми необходимыми методу аргументами. Когда объект получает сообщение, он ищет соответствующий метод внутри своего собственного класса. Рассмотрим несколько вызовов метода:

```
"dave".length
line_item_one.quantity()
cart.add_line_item(next_purchase)
submit_tag "Добавить в корзину"
```

Круглые скобки при вызове метода являются, как правило, необязательными. В приложениях Rails можно заметить, что в длинных выражениях в большинстве вызовов метода круглые скобки используются, а там, где эти вызовы больше похожи на команду или объявление, скобки обычно не ставятся.

У методов, как и у многих других конструкций Ruby, имеются имена, которые подчиняются определенным правилам. Если вы занялись Ruby после работы с другими языками, возможно, эти правила вам еще не попадались.

Имена, используемые в Ruby

Все имена локальных переменных, аргументов методов и самих методов должны начинаться с буквы в нижнем регистре или со знака подчеркивания: `order`,

`line_item` и `xr2000` являются допустимыми именами. Имена переменных экземпляра начинаются с символа «at» (`@`), например `@quantity` и `@product_id`. По принятому в Ruby соглашению знак подчеркивания используется для разделения слов в составном имени метода или переменной (то есть `line_item` использовать предпочтительнее, чем `lineItem`).

Имена классов модулей и констант должны начинаться с буквы в верхнем регистре. По соглашению для выделения начала слов внутри их имен используются не знаки подчеркивания, а заглавные буквы. Имена классов имеют вид `Object`, `PurchaseOrder` и `LineItem`.

В целях идентификации в Ruby используются *обозначения*. В частности, они используются в качестве ключей при названии аргументов методов и при поисках в хэшах. Например:

```
redirect_to :action => "edit", :id => params[:id]
```

Как видите, обозначения похожи на имена переменных, только в качестве префикса в них используется двоеточие. В качестве примеров обозначений можно привести `:action`, `:line_items` и `:id`. Обозначения можно считать строковыми литералами, магическим образом превращающимися в константы. Смысл двоеточия также можно рассматривать как «что-то по имени...», соответственно, `:id` обозначает «что-то по имени id».

После того как мы уже использовали несколько методов, давайте перейдем к вопросу их определения.

Методы

Давайте напишем метод, возвращающий теплое персональное пожелание. А затем мы этот метод пару раз вызовем:

```
def say_goodnight(name)
    result = 'Спокойной ночи, ' + name
    return result
end

# Пора спать...
puts say_goodnight('Мэри-Эллен') # => 'Спокойной ночи, Мэри-Эллен'
puts say_goodnight('крошка Джон') # => 'Спокойной ночи, крошка Джон'
```

После определения мы дважды вызываем этот метод. В обоих случаях результат передается методу `puts()`, который выводит на консоль свои аргументы, завершая вывод символом новой строки (перемещением на новую строку вывода).

При помещении каждой инструкции в отдельной строке точка с запятой в конце инструкции не требуется. Комментарии в Ruby начинаются с символа `#` и продолжаются до конца строки. Отступы не играют никакой роли (но де-факто отступ в две символьных позиции является в Ruby стандартом).

Для ограничения тел составных инструкций и определений (например, методов и классов) фигурные скобки в Ruby не используются. Вместо этого тело просто завершается ключевым словом `end`. Ключевое слово `return`

является необязательным, и если оно отсутствует, возвращается результат вычисления последнего выражения.

4.2. Типы данных

Наряду с тем, что все в Ruby является объектом, некоторые типы данных имеют специальную синтаксическую поддержку, в частности для определения литеральных значений. В данных примерах использован ряд простых строк и даже конкатенация строк.

Строки

В предыдущем примере был, кроме всего прочего, показан ряд строковых объектов. Один из способов создания строкового объекта заключается в использовании *строковых литералов*, представляющих собой последовательности символов внутри одинарных или двойных кавычек. Разница между двумя формами состоит в объеме той обработки, которой подвергается строка со стороны Ruby при создании литерала. В случае использования одинарных кавычек Ruby практически ничего не делает. За некоторыми исключениями все, что набрано в строковом литерале, заключенном в одинарные кавычки, становится значением строки.

В случае использования двойных кавычек Ruby проводит более серьезную работу. Во-первых, он ищет *подстановки* — последовательности, начинающиеся с символа обратного слэша, — и заменяет их неким двоичным значением. Самой распространенной подстановкой является пара символов `\n`, которая заменяется кодом символа новой строки. При записи строки, содержащей эту подстановку на консоль, `\n` приводит к переносу строки.

Во-вторых, в строках, заключенных в двойные кавычки, Ruby выполняет *вставку результата вычисления выражения*. Имеющаяся в строке последовательность `#{` *выражение* `}` заменяется значением *выражения*. Этим можно воспользоваться, чтобы переписать наш предыдущий метод:

```
def say_goodnight(name)
    "Спокойной ночи, #{name.capitalize}"
end
puts say_goodnight('pa')
```

Когда Ruby создает этот строковый объект, он смотрит на текущее значение `name` и подставляет его в строку. В конструкции `#{…}` допускаются выражения произвольной сложности. В данном случае вызывается метод `capitalize()`, определенный для всех строк, чтобы выдать наш аргумент с первой буквой в верхнем регистре.

Строки являются довольно простым типом данных, содержащим упорядоченную совокупность байтов или символов. Ruby также предоставляет средства для определения коллекций произвольных объектов с помощью массивов и хэшей.

Массивы и хэши

Массивы и хэши в Ruby являются индексированными коллекциями. В них хранятся совокупности объектов, доступных при использовании ключа. В массивах ключ является целым числом, а кэши в качестве ключа поддерживают любой объект. Массивы и хэши разрастаются по мере необходимости содержания новых элементов. Доступ к элементам массива более эффективен, но хэши предоставляют более высокую гибкость. Каждый конкретный массив или хэш может содержать объекты разных типов: к примеру, можно создать массив, содержащий целые числа, строки и числа с плавающей точкой.

Создать и инициализировать новый объект массива можно с помощью *литерала массива* — набора элементов между квадратными скобками. Как показано в следующем примере, получение объекта массива дает возможность обратиться к отдельным элементам путем предоставления индекса, заключенного в квадратные скобки. В Ruby индексация массивов начинается с нуля.

```
a = [ 1, 'cat', 3.14 ]  # массив из трех элементов
a[0]                    # обращение к первому элементу (1)
a[2] = nil              # установка значения третьего элемента
                        # теперь массив имеет следующий вид: [ 1, 'cat', nil ]
```

Можно было заметить, что в этом примере использовалось специальное значение `nil`. Во многих языках понятие *nil* (или *null*) означает «объект отсутствует». Но в Ruby все по-другому: `nil` является таким же объектом, как и все остальные, и предназначен для представления отсутствия.

С массивами часто используется метод `<<()`. Он добавляет значение к своему получателю:

```
ages = []
for person in @people
    ages << person.age
end
```

Для создания массива слов в Ruby есть сокращенная форма записи:

```
a = [ 'муравей' , 'пчела' , 'кот' , 'собака' , 'лось' ]
# эта запись аналогична предыдущей:
a = %w{ муравей пчела кот собака лось }
```

Хэши в Ruby похожи на массивы. В хэш-литералах вместо квадратных скобок используются фигурные. Каждая запись литерала должна быть представлена двумя объектами: одним — в качестве ключа, и другим — в качестве значения. Например, может потребоваться отобразить музыкальные инструменты по оркестровым группам.

```
inst_section = {
    :виолончель  => 'струнные инструменты' ,
    :кларнет     => 'деревянные духовые инструменты' ,
    :барабан     => 'ударные инструменты' ,
    :гобой       => 'деревянные духовые инструменты' ,
```

```
    :труба       => 'медные духовые инструменты' ,
    :скрипка     => 'струнные инструменты'
}
```

Все, что находится левее группы символов `=>`, является ключом, а все, что справа, — соответствующим ему значением. Ключи в каждом хэше должны быть уникальны — не может быть двух записей с ключом `:барабан`. В качестве ключей и значений хэша могут использоваться объекты любого типа — могут быть хэши, значения которых представлены массивами, другими хэшами и т. д. Как правило, в Rails в качестве ключей для хэшей используются обозначения. Многие хэши в Rails подверглись незначительной модификации, поэтому в качестве ключей при вставке и просмотре значений в равной степени могут использоваться как строки, так и обозначения.

Использование обозначений в качестве ключей хэшей стало настолько привычным, что, начиная с Ruby 1.9, для этого используется специальный синтаксис, более простой для набора и более наглядный:

```
inst_section = {
    виолончель:   'струнные инструменты' ,
    кларнет:      'деревянные духовые инструменты' ,
    барабан:      'ударные инструменты' ,
    гобой:        'деревянные духовые инструменты' ,
    труба:        'медные духовые инструменты' ,
    скрипка:      'струнные инструменты'
}
```

Не правда ли, смотрится намного лучше?

Можно использовать любой понравившийся вам синтаксис. Можно даже смешивать в одном выражении разные варианты. Разумеется, если в качестве ключей используются *не* обозначения, следует придерживаться синтаксиса, применяющего «стрелки».

При индексном доступе к элементам хэша используются те же квадратные скобки, что и в массивах:

```
inst_section[:гобой]        #=> 'деревянные духовые инструменты'
inst_section[:виолончель]   #=> 'струнные инструменты'
inst_section[:фагот]        #=> nil
```

Как показано в предыдущем примере, при индексном доступе с ключом, не содержащимся в хэше, возвращается `nil`. Такое свойство представляется удобным, поскольку при использовании в условных выражениях `nil` означает `false`.

Хэши можно передавать в качестве аргументов при вызове методов. Ruby позволяет не ставить скобки, но только в том случае, когда в списке аргументов вызова хэш идет последним. В Rails эта возможность используется довольно широко. В следующем фрагменте кода показан двухэлементный хэш, передаваемый методу `redirect_to`. О том, что это хэш, можно просто забыть, и представить, что в Ruby есть именованные аргументы:

```
redirect_to :action => 'show' , :id => product.id
```

Есть еще один тип данных, вполне заслуживающий внимания, — это регулярные выражения.

Регулярные выражения

Регулярные выражения позволяют указать *шаблон* символов для его сравнения со строкой. В Ruby регулярное выражение обычно создается в формате `/`*шаблон*`/` или `%r{`*шаблон*`}`.

Например, используя выражение `/Perl|Python/`, можно создать шаблон, соответствующий строке, в которой содержится текст «Perl» или текст «Python».

Шаблон ограничивается знаками прямых слэшей и содержит два сравниваемых фрагмента, разделенные вертикальной линией (`|`). Символ вертикальной черты означает «либо то, что слева, либо то, что справа», в данном случае — либо Perl, либо Python. В шаблонах, как и в арифметических выражениях, можно использовать скобки, поэтому данный шаблон можно написать следующим образом: `/P(erl|ython)/`. В программах для сравнения строк с регулярными выражениями используется оператор сравнения `=~`:

```
if line =~ /P(erl|ython)/
    puts "Похоже, здесь используется другой язык сценариев"
end
```

В шаблонах можно задавать повторения. Выражение `/ab+c/` соответствует строке, содержащей символ `a`, за которым следует один или несколько символов `b`, за которыми, в свою очередь, следует символ `c`. Если заменить знак «плюс» звездочкой, получится регулярное выражение `/ab*c/`, которое соответствует одному символу `a`, нулевому или большему количеству символов `b` и одному символу `c`.

Специальные последовательности начинаются с обратного слэша; наиболее заметна последовательность `\d`, соответствующая одной цифре, последовательность `\s`, соответствующая любому пробельному символу, и последовательность `\w`, соответствующая любым буквенно-цифровым («словарным») символам.

Регулярные выражения Ruby — довольно глубокая и сложная тема, которая в данном разделе рассмотрена весьма поверхностно. Полноценную информацию о них можно получить в книге «PickAxe», а в данной книге регулярные выражения будут применяться лишь эпизодически.

После краткого представления данных, давайте перейдем к логике.

4.3. Логика

Вызовы методов являются инструкциями. В Ruby также предоставляется ряд способов принятия решения, влияющих на повторения и на тот порядок, в котором вызываются методы.

Управляющие структуры

В Ruby имеются все стандартные управляющие структуры, например инструкции `if` и циклы `while`. Программисты, работавшие на Java, C и Perl, могут заметить,

что вокруг тел этих операторов отсутствуют фигурные скобки. Вместо них для обозначения окончания тела в Ruby используется ключевое слово `end`:

```
if count > 10
    puts "Попробуйте еще раз"
elsif tries == 3
    puts "Вы проиграли"
else
    puts "Введите число"
end
```

Операторы `while` также заканчиваются ключевым словом `end`:

```
while weight < 100 and num_pallets <= 30
pallet = next_pallet()
weight += pallet.weight
num_pallets += 1
end
```

В Ruby имеются также варианты этих инструкций: `unless` похожа на `if`, но она проверяет условие на недостоверность. Аналогичным образом инструкция `until` похожа на инструкцию `while`, за исключением того, что цикл продолжается до тех пор, пока вычисляемое условие не вернет `true`.

Если тело инструкций `if` или `while` укладывается в одно выражение, в Ruby может использоваться удобное сокращение, называемое *модификатором инструкции*. Нужно просто написать выражение, сопровождаемое ключевым словом модификатора и условием:

```
puts "Уилл Робинсон, Вас подстерегает опасность" if radiation > 3000
distance = distance * 1.2 while distance < 100
```

Инструкции `if` встречаются в Ruby-приложениях довольно часто, но тех, кто начинает осваивать язык Ruby, зачастую удивляет редкое использование конструкций циклов. Вместо них часто используются блоки и итераторы.

Блоки и итераторы

Блоки кода — это фрагменты программного кода, заключенные в фигурные скобки или помещенные между ключевыми словами `do...end`. По общепринятому соглашению, фигурные скобки используются для однострочных блоков, а конструкция `do/end` — для многострочных:

```
{ puts "Привет" }        # это блок

do                       ###
    club.enroll(person)  # и это тоже блок
    person.socialize     #
end                      ###
```

Чтобы передать блок методу, его нужно поместить после аргументов метода (если таковые имеются). Иными словами, начало блока нужно поместить в конец

исходной строки, содержащей вызов метода. Например, в следующем коде блок, содержащий `puts "Привет"`, связан с вызовом метода `greet()`:

```
greet { puts "Привет" }
```

Если у вызова метода имеются аргументы, они ставятся перед блоком:

```
verbose_greet("Дэйв", "постоянный клиент") { puts "Привет" }
```

Используя Ruby-инструкцию `yield`, метод может вызывать связанный с ним блок один или несколько раз. Инструкцию `yield` можно считать своеобразным вызовом метода, который вызывает блок, связанный с тем методом, в котором содержится `yield`. Предоставляя аргументы инструкции `yield`, блоку можно передавать значения. Внутри блока перечень имен аргументов, получающих эти значения, помещается между вертикальными линиями (`|`).

Блоки кода встречаются в Ruby-приложениях повсеместно. Зачастую они используются вместе с итераторами — методами, которые возвращают последовательные элементы из какой-нибудь коллекции, например из массива:

```
animals = %w{ муравей пчела кот собака лось } # создание массива
animals.each {|animal| puts animal }       # последовательный перебор содержимого
```

Каждое целое число *N* вызывает метод `times()`, который, в свою очередь, вызывает связанный с ним блок *N* раз:

```
3.times { print "Эй! " } #=> Эй! Эй! Эй!
```

Префиксный оператор **&** позволит методу захватить переданный блок в качестве поименованного аргумента:

```
def wrap &b
print "Санта сказал: "
    3.times(&b)
    print "\n"
end
wrap { print "Эй! " }
```

Внутри блока или метода управление инструкциям передается последовательно, если только там не будет исключения.

Исключения

Исключения являются объектами (класса `Exception` или его подклассов). Для выдачи исключения используется метод `raise`. Тем самым прерывается нормальная последовательность выполнения программного кода, и Ruby просматривает в обратном порядке стек вызовов в поиске кода, который сообщит о возможности обработки этого исключения.

Соответствующие классы исключений перехватываются с помощью выражений `rescue` как методами, так и блоками кода, помещенными между ключевыми словами `begin` и `end`.

```
begin
    content = load_blog_data(file_name)
rescue BlogDataNotFound
    STDERR.puts "Файл #{file_name} не найден"
rescue BlogDataFormatError
    STDERR.puts "В файле #{file_name} отсутствуют данные блога"
rescue Exception => exc
    STDERR.puts " Общая ошибка загрузки #{file_name}: #{exc.message}"
end
```

Выражения `rescue` могут непосредственно помещаться на самом внешнем уровне определения метода и не требуют заключения содержимого в блок `begin-end`.

Здесь наше краткое введение в управляющую логику завершается, и теперь у нас есть основные строительные блоки, из которых мы сможем выстраивать более крупные структуры.

4.4. Организационные структуры

Для организации методов в Ruby есть два основных понятия: классы и модули. Мы рассмотрим по очереди каждое из них.

Классы

Определение класса имеет в Ruby следующий вид:

```
Строка
1    class Order < ActiveRecord::Base
-        has_many :line_items
-         def self.find_all_unpaid
-             self.where('paid = 0')
5         end
-         def total
-             sum = 0
-             line_items.each {|li| sum += li.total}
-             sum
10         end
-     end
```

Это определение начинается с ключевого слова `class`, за которым следует имя класса (которое должно начинаться с буквы в верхнем разряде). Данный класс `Order` определен как подкласс, относящийся к классу `Base`, который находится в модуле `ActiveRecord`.

Rails довольно интенсивно привлекается к определениям на уровне классов. В данном случае `has_many` — это метод, который определен в Active Record. Он вызывается при определении класса `Order`. Обычно методы такого рода позволяют судить о классе, поэтому в данной книге мы называем их *объявлениями*.

Внутри тела класса можно определить методы класса и методы экземпляра. Добавление к имени метода префикса `self.` (как это сделано в строке 3) делает его

методом класса: он может быть вызван во всем классе. В данном случае можно из любого места нашего приложения сделать следующий вызов:

```
to_collect = Order.find_all_unpaid
```

Объекты класса хранят свое состояние в *переменных экземпляра*. Эти переменные, имена которых всегда начинаются с символа @, доступны всем методам экземпляра определенного класса. Каждый объект получает свой собственный набор переменных экземпляра.

Непосредственный доступ к переменным экземпляра за пределами класса невозможен. Для доступа к ним нужно создавать методы, возвращающие их значения:

```
class Greeter
    def initialize(name)
        @name = name
    end

    def name
        @name
    end

    def name=(new_name)
        @name = new_name
    end
end

g = Greeter.new("Барни")
g.name        #=> Барни
g.name = "Бетти"
g.name        #=> Бетти
```

Для создания этих методов доступа Ruby предоставляет свои, очень удобные методы (это должно понравиться тем, кто уже устал от написания всех этих извлекателей и установщиков значений свойств):

```
class Greeter
attr_accessor :name        # создание методов чтения и записи
attr_reader :greeting      # создание только метода чтения
attr_writer :age           # создание только метода записи
```

По умолчанию методы экземпляра класса являются открытыми (`public`) и могут быть вызваны отовсюду. Возможно, вам потребуется отменить эту установку для тех методов, которые предназначены для использования только другими методами экземпляра класса:

```
class MyClass
    def m1                 # Это открытый метод
    end

    protected

    def m2             # Это защищенный метод
    end
```

```
    private

    def m3              # это закрытый метод
    end
end
```

Самой строгой является директива `private`; закрытые методы могут быть вызваны только внутри того же самого экземпляра. Защищенные методы могут быть вызваны как из того же самого экземпляра, так и из других экземпляров того же класса и его подклассов.

Классы не являются в Ruby единственной организационной структурой. Еще одной такой структурой является модуль.

Модули

Модули похожи на классы тем, что они содержат коллекцию методов, констант, а также определений других модулей и классов. В отличие от классов, создавать объекты на основе модулей невозможно.

Модули служат двум целям. Во-первых, они работают как пространства имен, позволяя определять методы, чьи имена не будут конфликтовать с именами методов, определенных в каком-нибудь другом месте. Во-вторых, они позволяют иметь для классов общие функциональные возможности — если класс *смешивается* с модулем, методы экземпляра модуля становятся доступны, как будто они были определены в самом классе. С одним и тем же модулем могут смешиваться несколько классов, вместе используя функциональные возможности модуля без привлечения механизма наследования. Можно также смешать с отдельным классом сразу несколько модулей.

Примером использования модулей в Rails могут послужить вспомогательные методы. Rails автоматически смешивает вспомогательные модули с соответствующими шаблонами представления. Например, если нужно написать вспомогательный метод, который может быть вызван из представления, вызванного контроллером `store`, можно в файле store_helper.rb каталога app/helpers определить следующий модуль:

```
module StoreHelper
    def capitalize_words(string)
        string.split(' ').map {|word| word.capitalize}.join(' ')
    end
end
```

Модуль YAML, входящий в стандартную библиотеку Ruby, заслуживает особого упоминания, с учетом его использования в Rails.

YAML

YAML[1] — это рекурсивный акроним, означающий YAML Ain't Markup Language (YAML — не язык разметки). Применительно к Rails YAML используется как

[1] http://www.yaml.org/

удобный способ определения конфигураций баз данных, тестовых данных и преобразований. Например:

```
development:
    adapter: sqlite3
    database: db/development.sqlite3
    pool: 5
    timeout: 5000
```

В YAML важную роль играют отступы, поэтому здесь определяется, что `development` (разработка) имеет набор из четырех пар ключ-значение, с двоеточием в качестве разделителя.

Хотя YAML является одним из способов представления данных, особенно в случаях общения с людьми, Ruby предоставляет более универсальный способ представления данных для их использования в приложениях.

4.5. Маршализированные объекты

Ruby способен конвертировать какой-либо объект в поток байтов, который может быть сохранен вне приложения. Этот процесс называется *маршализацией*. Затем сохраненный объект может быть прочитан другим экземпляром приложения (или вообще другим приложением), в результате чего может быть восстановлена копия оригинального объекта.

При использовании маршализации существуют две потенциальные проблемы. Во-первых, некоторые объекты не поддаются разложению в двоичную форму: если в подвергаемых обработке объектах содержатся какие-нибудь связывания, объекты процедур или методов, экземпляры класса `IO` или синглтон-объекты либо если предпринимается попытка перевести в двоичную форму безымянные классы или модули, будет выдано исключение `TypeError`.

Во-вторых, когда загружаются маршализированные объекты, Ruby нужно знать определение класса этого объекта (и всех содержащихся в нем объектов).

Rails использует маршализацию для хранения данных сессии. Если полагаться на Rails при динамической загрузке классов, может получиться, что конкретный класс на момент восстановления данных сессии не будет определен. Именно поэтому в контроллере используется объявление `model` для перечисления всех моделей, подвергающихся маршализации. Это объявление приводит к предварительной загрузке классов, необходимых для работы маршализации.

Ознакомившись с основами Ruby, давайте смешаем все изученное в более объемном, прокомментированном примере, объединившем сразу несколько понятий. Затем будет дан краткий обзор характерных особенностей, которые помогут программированию в среде Rails.

4.6. А теперь все вместе

Рассмотрим пример того, как Rails сводит в единое целое сразу несколько свойств Ruby, чтобы сделать программный код, нуждающийся в поддержке, более

описательным. Этот пример будет снова показан в главе 6, в разделе «Создание временных платформ». А сейчас основное внимание будет уделено тем аспектам примера, которые связаны с языком Ruby.

```
class CreateProducts < ActiveRecord::Migration
    def change
        create_table :products do |t|
            t.string :title
            t.text :description
            t.string :image_url
            t.decimal :price, precision: 8, scale: 2

            t.timestamps
        end
    end
end
```

Даже не зная языка Ruby, можно понять, что этот код создает таблицу по имени `products`. При создании этой таблицы определяются поля `title`, `description`, `image_url` и `price`, а также несколько временных меток, `timestamps` (они будут рассмотрены в разделе 22.1).

Давайте посмотрим на этот пример с точки зрения Ruby. В нем определяется класс по имени `CreateProducts`, который наследуется из класса `Migration` модуля `ActiveRecord`. Определяется один метод по имени `change()`. Этот метод вызывает метод `create_table()` (определенный в `ActiveRecord::Migration`), передавая ему имя таблицы в форме обозначения.

Вызову `create_table()` также передается блок, который должен быть вычислен до создания таблицы. При вызове этого блока ему передается объект по имени `t`, который используется для сбора списка полей. Для этого объекта Rails определяет ряд методов, которые носят названия стандартных типов данных. При вызове эти методы просто добавляют определения полей к накапливаемому набору имен.

Определение `decimal` также принимает ряд необязательных параметров, представленных хэшем.

Тем, кто не знаком с Ruby, кажется, что для решения такой простой проблемы привлекается слишком много сложных механизмов, а для тех, кто знаком с этим языком, ни один из этих механизмов не кажется особенно сложным. В любом случае Rails широко использует средства, предоставляемые Ruby, для того чтобы сделать операции определения (например, миграционных задач) как можно проще и описательнее. Даже такие незначительные особенности языка, как необязательные круглые и фигурные скобки, используются для достижения повсеместной удобочитаемости и удобства творчества.

И наконец, существует множество мелких особенностей, точнее идиоматических сочетаний свойств, которые зачастую не столь очевидны для тех, кто плохо знаком с языком Ruby. Они и станут заключительной темой данной главы.

4.7. Идиомы, используемые в Ruby

Многие отдельные свойства Ruby могут быть объединены довольно интересным образом, и значение такого идиоматического использования зачастую не сразу бросается в глаза тем, кто плохо знаком с языком. В данной книге будут использованы следующие широко распространенные Ruby-идиомы.

Методы типа `empty!` и `empty?`

В Ruby имена методов могут заканчиваться восклицательным знаком (метод-модификатор) или вопросительным знаком (метод-предикат). Методы-модификаторы обычно вносят изменения в получатель. Методы-предикаты в зависимости от определенных условий возвращают `true` или `false`.

`a || b`

Инструкция `a || b` вычисляет `a`. Если результат не равен `false` или `nil`, вычисление прекращается и инструкция возвращает `a`. В противном случае возвращается `b`. Это весьма распространенный способ возвращения значения по умолчанию, если первое значение не установлено.

`a ||= b`

Инструкция присваивания поддерживает целый набор сокращений: сокращение вида `a` *оператор*`= b` эквивалентно выражению `a = a` *оператор* `b`. Это сокращение работает с большинством операторов:

```
count += 1            # эквивалентно count = count + 1
price *= discount     # price = price * discount
count ||= 0           # count = count || 0
```

Таким образом, инструкция `count ||= 0` присваивает переменной `count` значение 0, если у `count` еще нет значения.

`obj = self.new`

Иногда метод класса должен создать экземпляр этого класса:

```
class Person < ActiveRecord::Base
        def self.for_dave
            Person.new(name: 'Дэйв')
        end
end
```

Все это прекрасно работает, возвращая новый объект `Person`. Но позже кто-нибудь может создать подкласс нашего класса:

```
class Employee < Person
        # ..
end
dave = Employee.for_dave      # возвращает Person
```

Метод `for_dave()` был жестко запрограммирован на возвращение объекта `Person`, поэтому вызов `Employee.for_dave` возвращает именно его. При использовании вместо него метода `self.new` возвращается новый объект класса-получателя, то есть `Employee`.

`lambda`

Оператор `lambda` превращает блок в объект типа `Proc`. Альтернативный синтаксис, введенный в Ruby 1.9, имеет вид `->`. Оба этих варианта синтаксиса рассматриваются в подразделе «Области действия» раздела 19.3.

```
require File.expand_path('../../config/environment',__FILE__)
```

Ruby-метод `require` загружает в приложение файл из внешнего источника. Это используется для включения кода библиотеки и классов, от которых зависит работа приложения. При обычном использовании Ruby находит эти файлы, проводя поиск в каталогах, перечисленных в списке `LOAD_PATH`.

Иногда нужно точно указать, какой файл следует включить. Это можно сделать, передав методу полное путевое имя файла. Проблема в том, что мы не знаем, каким будет это путевое имя, — наши пользователи могут установить наш код куда угодно.

В какое бы место ни было в конечном итоге установлено наше приложение, относительный путь между файлом, из которого делается запрос, и запрашиваемым файлом будет одним и тем же. Зная это, мы можем выстроить абсолютный путь к запрашиваемому файлу с помощью метода File.expand_path(), передавая ему относительный путь к этому файлу, а также получив абсолютный путь к файлу, из которого делается запрос (этот путь доступен в специальной переменной `__FILE__`).

Дополнительно в Интернете имеется множество хороших информационных ресурсов, в которых показываются используемые в Ruby идиомы и фокусы. Вот лишь некоторые из них:

- http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/
- http://en.wikipedia.org/wiki/Ruby_programming_language
- http://www.zenspider.com/Languages/Ruby/QuickRef.html

Теперь у нас появился прочный фундамент, на котором можно вести строительство. Мы установили Rails, проверили ее работоспособность на простом приложении, получили некоторое представление о том, что такое Rails, и повторили (а некоторые впервые изучили) основы языка Ruby. Настало время применить эти знания при создании более крупного приложения.

II

Создание приложения

5 Интернет-магазин

Основные темы:

- поэтапная разработка;
- примеры использования, последовательность страниц, структура данных;
- приоритеты.

Можно, конечно, потратить весь день на совместный разбор простых тестовых приложений, но это не поможет нам оплатить счета. Поэтому давайте нацелимся на что-нибудь более существенное и создадим на основе веб-технологий интернет-магазин под названием Depot.

Конечно, в мире и так достаточно всевозможных интернет-магазинов, но сотни разработчиков продолжают создавать все новые и новые подобные приложения. А чем мы хуже?

А если серьезно, то наша корзина покупателя поможет проиллюстрировать многие особенности Rails-разработки. Мы увидим, как создаются простые страницы обслуживания покупателей, как они привязываются к таблицам базы данных, как обрабатываются сессии и как создаются формы. На протяжении следующих 12 глав мы также будем обращаться к внешним темам, таким как блочное тестирование, безопасность и разметка страницы.

5.1. Поэтапная разработка

Разработка нашего приложения будет вестись поэтапно. Мы не станем пытаться определить буквально все до начала написания программного кода. Лучше мы выработаем объем спецификации, позволяющий приступить к программированию, и тут же создадим нечто работающее. Мы будем проводить практическую

апробацию идей, обобщать отзывы и продолжать работу, переходя к другому циклу мини-проектирования и разработки.

Придерживаться такого стиля программирования удается не всегда. Здесь требуется тесное сотрудничество с пользователями приложения, поскольку по мере продвижения разработки нам нужно собирать отзывы о работе продукта. Ошибаться можем и мы, и заказчик, обнаруживший, что, требуя от нас одно, он на самом деле хотел получить совсем другое. Не важно, кто стал виновником ошибки, но чем раньше мы ее обнаружим, тем меньше сил будет затрачено на ее исправление. В общем, применение этого стиля разработки характеризуется внесением многочисленных изменений в процессе работы.

Поэтому нам нужно воспользоваться таким набором инструментов, который не будет усложнять работу при изменении взглядов на создаваемый продукт. Если мы решили, что к таблице базы данных нужно добавить еще один столбец или изменить систему переходов между страницами, нам нужно иметь возможность войти в приложение и сделать все необходимое без объемных работ по программированию или долгой возни с настройками конфигурации. Вы сможете убедиться в том, что когда дело касается внесения изменений, Ruby on Rails всегда на высоте — это идеальная среда для гибкого программирования.

Попутно мы будем создавать и поддерживать в рабочем состоянии набор тестов. Они помогут нам убедиться в том, что приложение всегда делает только то, для чего оно предназначено. Rails не только допускает создание подобных тестов, но и предоставляет вам исходный набор тестов при каждом определении нового контроллера.

Но вернемся к приложению.

5.2. Для чего предназначен Depot

Начнем с написания основной спецификации приложения Depot. Для этого рассмотрим самые общие примеры практики использования и составим краткое описание переходов по веб-страницам. Попытаемся также определить, какие данные понадобятся приложению (отдавая себе отчет в том, что наши исходные предположения могут оказаться неверными).

Примеры использования

Пример использования — это обыкновенное утверждение о том, как кто-нибудь использует систему. Консультанты изобретают подобные фразы, чтобы навесить ярлыки на вполне очевидные понятия, и то, что необычные слова всегда ценятся выше, чем обычные, которые на самом деле куда значимее, — это издержки бизнеса.

Примеры использования приложения Depot довольно просты (что для некоторых прозвучит трагически). Начнем с определения двух разных ролей, или действующих субъектов: *покупателя* и *продавца*.

Покупатель использует Depot для просмотра продаваемого товара, выбора покупок и предоставления информации, необходимой для оформления заказа.

Продавец использует Depot для учета продаваемого товара, для определения заказов, ожидающих доставки, и для пометки доставленных заказов. (Продавцы также используют Depot, чтобы заработать кучу денег, выйти на пенсию и жить на тропических островах, но это уже тема какой-нибудь другой книги.)

И пока это все, что нам может понадобиться. Можно, конечно, мучительно уточнять, что такое «учет товара» и что подразумевается под «заказом, готовым к доставке», но зачем? Если есть какие-то неясные моменты, то вскоре, по мере демонстрации заказчику последовательных этапов нашей работы, все само собой прояснится.

Что касается отзывов, не забудем получить их прямо сейчас, чтобы убедиться в том, что наши начальные (весьма схематичные) примеры использования вполне отвечают представлениям нашего пользователя. Предположив, что наши примеры использования были приняты: давайте определим, как наше приложение будет работать с точки зрения различных пользователей.

Последовательность страниц

В любом случае нам нужно выработать замысел устройства основных страниц нашего приложения и составить примерное представление о том, как пользователи будут переходить со страницы на страницу. На ранней стадии разработки эта последовательность страниц, скорее всего, будет неполной, но она все же поможет нам сфокусироваться на том, что должно быть сделано, и определить последовательность действий.

Возможно, кому-то нравится создавать макеты страниц веб-приложения в Photoshop, Word или (как ни странно) с помощью HTML. А мне нравится работать с карандашом и бумагой. Так быстрее, и клиент также может включиться в работу, взяв карандаш и набросав варианты на бумаге.

Первый набросок последовательности страниц для покупателя показан на рис. 5.1. В нем нет ничего особенного. Покупатель видит страницу каталога, на которой он поштучно выбирает товар. Каждый выбранный товар попадает в корзину, которая демонстрируется после каждого выбора. Покупатель может продолжить покупки, используя страницы каталога, или подтвердить содержимое корзины и купить находящийся в ней товар. В процессе подтверждения мы получаем контактные и платежные сведения, а затем выставляем счет. Мы еще не знаем, как будем оформлять платеж, поэтому его детали в этой последовательности не имеют ясных очертаний.

Последовательность страниц продавца показана на рис. 5.2. Она также не отличается особой сложностью. После входа в программу продавец видит меню, позволяющее ему создать запись о товаре или просмотреть существующую запись, а также отправить существующие заказы. При просмотре товара продавец может выборочно отредактировать информацию о товаре или вовсе удалить касающуюся его запись.

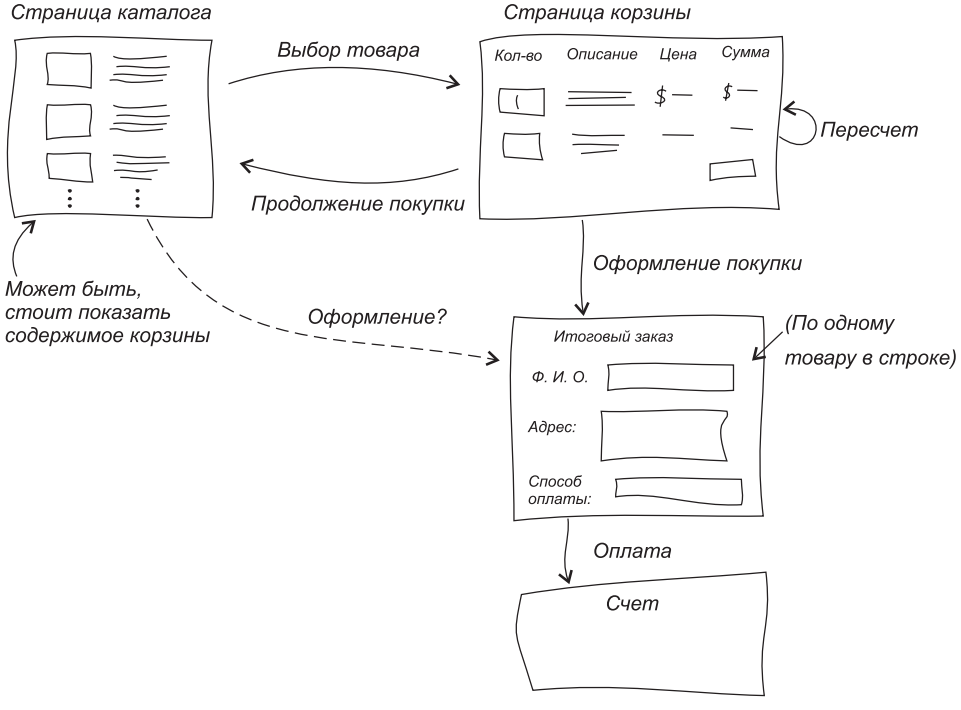


Рис. 5.1. Последовательность страниц, используемых покупателями

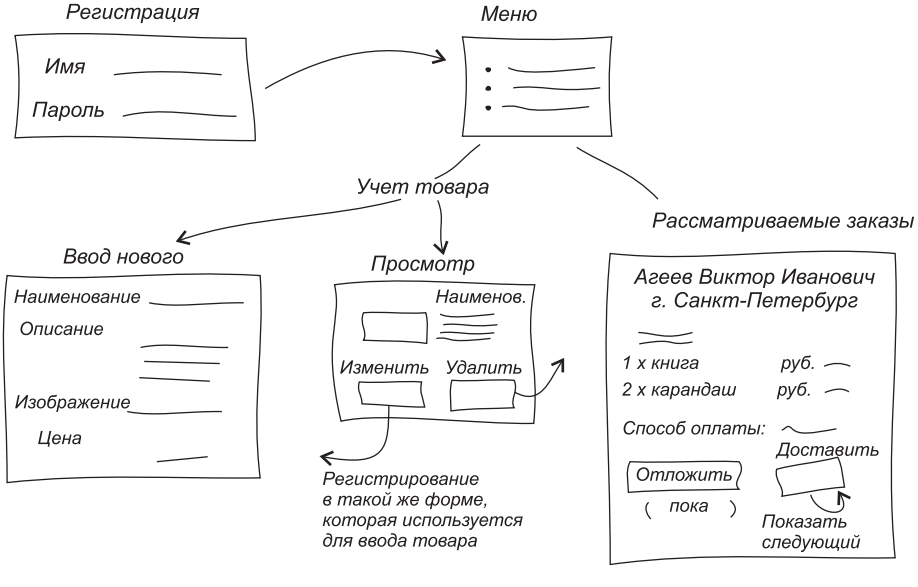


Рис. 5.2. Последовательность страниц, используемых продавцом

Вариант отправки сильно упрощен. Он предусматривает постраничный вывод каждого еще не отправленного заказа. Продавец может пропустить текущий заказ и перейти к следующему или может выбрать отправку заказа, используя, при необходимости, информацию со страницы.

Понятно, что функция отправки не выдержит никакого испытания практикой, но отправка относится к тем областям, реальное представление о которых составить довольно трудно. Проводить ее детальную проработку пока преждевременно, мы неизбежно наделаем массу ошибок.

Пока оставим все как есть, пребывая в уверенности, что сможем внести изменения, как только пользователи наберутся опыта работы с нашим приложением.

Данные

И наконец, нужно подумать о данных, с которыми мы собираемся работать.

Заметьте, что здесь мы не употребляем такие слова, как «схемы» или «классы». Мы также не говорим о базах данных, таблицах, ключах и тому подобных вещах. Речь идет только о данных. На этой стадии разработки мы еще не знаем, будут ли использоваться базы данных.

Похоже, что на основании примеров использования и последовательности страниц мы будем иметь дело с теми данными, которые изображены на рис. 5.3. На мой взгляд, и здесь намного проще будет использовать карандаш и бумагу, а не какой-нибудь мудреный инструмент, но вы можете воспользоваться тем, что вам больше по душе.

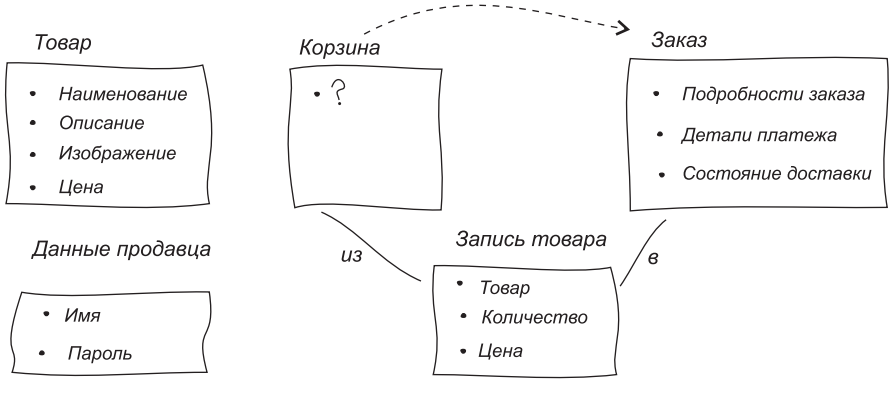


Рис. 5.3. Исходные предположения о составе данных, используемых в приложении

Работа с диаграммой, отображающей состав данных, наводит на некоторые размышления. Поскольку пользователь что-то покупает, нам нужно куда-то складывать покупки, поэтому я добавил корзину. Но что такое корзина, кроме как место для временного хранения списка товаров, представляется весьма смутно — я не в состоянии найти что-либо существенное, что можно было бы в ней хранить. Для обозначения этой неопределенности я поместил внутри прямоугольника,

изображающего на диаграмме корзину, вопросительный знак. Смею предположить, что эту неопределенность мы развеем в процессе создания приложения Depot.

В результате размышлений о данных общего характера возникает вопрос, какие сведения должны попадать в заказ. И тут я снова оставляю вопрос открытым — мы проясним его чуть позже, как только начнем демонстрировать пользователю наши первые шаги.

В конечном итоге вы могли заметить, что я продублировал цену товара в строке выбора. Здесь я несколько отступаю от правила «в начальной стадии все должно выглядеть как можно проще», но это нарушение продиктовано моим личным опытом. Если цена товара изменится, то это изменение не должно отражаться на том товаре, который попал в строку выбора текущих открытых заказов, поэтому в каждой строке выбора должна быть указана такая цена товара, которая существовала на момент оформления заказа.

Здесь я снова вместе с заказчиком проверил свои рассуждения и понял, что нахожусь на верном пути. (Вероятнее всего, мой заказчик сидел рядом со мной, пока я рисовал все эти три диаграммы.)

5.3. А теперь приступим к программированию

Итак, посидев рядом с заказчиком и проведя некоторый предварительный анализ, мы готовы сесть за компьютер и приступить к разработке с опорой на замысел, отображенный в трех исходных диаграммах. Но вскоре мы, наверное, их просто выбросим — они устареют, как только мы начнем получать отзывы о создаваемом приложении. Кстати, именно поэтому мы и не стали тратить на них слишком много времени — так будет проще с ними расстаться.

В следующих главах мы начнем создавать приложение, основываясь на нашем текущем представлении. Но пока мы не перевернули эту страницу, нужно ответить еще на один вопрос. С чего начать?

Я предпочитаю работать с заказчиком, поэтому мы можем выработать приоритеты совместными усилиями. В данном случае я обратил его внимание на то, что трудно разрабатывать что-нибудь другое, пока у нас не будет некоторого задела основных товаров, присутствующих в системе, поэтому я предложил потратить пару часов на то, чтобы получить начальную версию работоспособного средства по учету товаров. И он, конечно же, согласился.

ОБЩИЙ СОВЕТ ПО ВОЗВРАЩЕНИЮ НА ПРАВИЛЬНЫЙ КУРС

Весь материал данной книги был протестирован. При точном соблюдении описанного в ней сценария с использованием рекомендованных версий Rails и SQLite3 на Linux, Mac OS X или Windows все должно работать в соответствии с описаниями. Но могут случаться и отклонения от заданного маршрута. Никто из нас не застрахован от опечаток, и сопутствующие исследования не только возможны, но и категорически приветствуются. При этом следует знать, что они могут сбить вас с нужного курса. Но не стоит этого бояться: в конкретных разделах,

где часто возникают подобные проблемы, будут появляться описания конкретных действий по возвращению на правильный курс. А общие советы на этот счет даются прямо сейчас.

Перезапускать сервер нужно будет только в нескольких случаях, когда это действие конкретно оговорено в данной книге. Но когда вы попадете в полный тупик, можно будет попробовать перезапустить сервер.

«Волшебная» команда rake db:migrate:redo, о существовании которой вам нужно знать, подробно рассмотрена в части III. Она выполняет отмену и повторное использование последней миграции.

Если ваш сервер не примет какие-нибудь введенные в форму данные, обновите форму на своем браузере и повторите отправку данных.

6

Задача A: создание приложения

Основные темы:

- создание нового приложения;
- конфигурирование базы данных;
- создание моделей и контроллеров;
- добавление таблиц стилей;
- обновление разметки и представления.

Нашей первой задачей в разработке приложения станет создание веб-интерфейса, позволяющего вести учет товаров — вводить информацию о новых товарах, редактировать сведения об уже имеющихся, удалять ненужные товары и т. д. Мы разобьем разработку приложения на шаги, каждый из которых будет по времени выполнения занимать всего лишь несколько минут.

Обычно наши шаги будут состоять из нескольких этапов, например шаг B будет состоять из этапов B1, B2, B3 и т. д. В данном случае у шага будет два этапа. Итак, приступим.

6.1. Шаг A1: создание приложения по учету товаров

Основой приложения Depot является база данных. Если эта база уже установлена, сконфигурирована и протестирована, это может избавить вас от массы проблем. Если вы не уверены в том, что именно вам нужно, проще всего положиться на

настройки по умолчанию. Если вы уже знаете, что вам нужно, Rails упростит вам описание выбранной конфигурации.

Создание Rails-приложения

В главе 2 мы уже видели, как создается новое Rails-приложение. Здесь мы сделаем то же самое. Перейдите в окно командной строки и наберите команду `rails new`, указав после нее имя нашего проекта. В данном случае наш проект называется `depot`, поэтому убедитесь в том, что вы не находитесь в каталоге уже существующего приложения, и наберите следующую команду:

```
work> rails new depot
```

Мы увидим, как по экрану побегут строки вывода. Когда этот процесс завершится, мы обнаружим новый каталог по имени depot. Именно с ним мы и будем работать.

```
work> cd depot
depot> dir /w
[.]            [..]            [app]          [bin]      [config]
config.ru       [db]           Gemfile        Gemfile.lock   [lib]
[log]          [public]        Rakefile       README.rdoc    [test]
[tmp]          [vendor]
```

Пользователям Windows вместо команды `ls -p` нужно воспользоваться командой `dir /w`.

Создание базы данных

Для данного приложения будет использоваться база данных с открытым исходным кодом SQLite (которая вам понадобится, если вы намерены придерживаться представленного программного кода). В данной книге описывается использование SQLite версии 3.

SQLite 3 является базой данной, используемой при разработке Rails-приложений по умолчанию, и она была установлена вместе с Rails в главе 1 «Установка Rails». При работе с SQLite 3 не нужны никакие шаги по созданию базы данных, и поэтому нет никаких специальных учетных записей пользователей или паролей. Итак, сейчас вы начинаете убеждаться в преимуществах следования общему течению (или, в соответствии с избитой фразой пользователей Rails, следования соглашению о конфигурации).

Если вам важно использовать другой сервер базы данных, не SQLite 3, то команды, необходимые для создания базы данных, и порядок наделения полномочиями будут другими. Ряд полезных советов на этот счет можно найти на информационном ресурсе «Getting Started Rails Guide»[1].

[1] http://guides.rubyonrails.org/getting_started.html#configuring-a-database

Генерирование временной платформы

Ранее, на рис. 5.3, «Исходные предположения о составе данных, используемых в приложении», мы дали кратное описание основного содержимого таблицы товаров `products`. Давайте воплотим все это в реальность. Нам нужно создать таблицу базы данных и *модель* Rails, которая позволит нашему приложению использовать эту таблицу, а также создать ряд *представлений* для формирования пользовательского интерфейса и *контроллер*, управляющий приложением.

Итак, давайте создадим для нашей таблицы `products` модель, представления, контроллер и миграцию. Работая с Rails, все это можно сделать с помощью одной команды, попросив Rails сгенерировать то, что называется *временной платформой* (scaffold) для заданной модели. Заметьте, что слово в командной строке, которая вскоре последует, используется в форме единственного числа — `Product`. В Rails модель автоматически отображается на таблицу базы данных, чье имя является формой множественного числа класса модели. В нашем случае мы запросили модель под названием `Product`, поэтому Rails связывает ее с таблицей по имени `products`. (А как же она найдет эту таблицу? Где ее искать, в Rails подскажет запись `development` в файле config/database.yml. Для пользователей SQLite 3 это будет файл в каталоге db.)

```
depot> rails generate scaffold Product \
        title:string description:text image_url:string price:decimal
    invoke active_record
    create     db/migrate/ 20121130000001_create_products.rb
    create     app/models/product.rb
    invoke     test_unit
    create     test/models/product_test.rb
    create         test/unit/product_test.rbcreate
    create         test/fixtures/products.yml
    invoke resource_route
     route     resources :products
    invoke jbuilder_scaffold_controller
    create     app/controllers/products_controller.rb
    invoke     erb
    create         app/views/products
    create         app/views/products/index.html.erb
    create         app/views/products/edit.html.erb
    create         app/views/products/show.html.erb
    create         app/views/products/new.html.erb
    create         app/views/products/_form.html.erb
    invoke     test_unit
    create         test/ controllers/products_controller_test.rb
    invoke     helper
    create         app/helpers/products_helper.rb
    invoke         test_unit
    create             test/helpers/products_helper_test.rb
    invoke  jbuilder
    exist     app/views/products
    create    app/views/products/index.json.jbuilder
    create    app/views/products/show.json.jbuilder
    invoke assets
```

```
    invoke     coffee
    create         app/assets/javascripts/products.js.coffee
    invoke       scss
    create         app/assets/stylesheets/products.css.scss
    invoke     scss
    create         app/assets/stylesheets/scaffolds.css.scss
```

Генератор создает целый пакет файлов. Нас в первую очередь будет интересовать файл *миграции*, а именно 20121130000001_create_products.rb.

Миграция представляет изменение, которое нужно внести в данные, выраженное в исходном файле в терминах, независимых от применяемой базы данных. Такие изменения могут обновить как схему базы данных, так и данные в ее таблицах. Эти миграции применяются для обновления нашей базы данных, и их применение можно отменить, чтобы база данных вернулась к прежнему состоянию. Миграциям будет посвящена целая глава 22, а сейчас мы будем просто пользоваться ими без излишних комментариев.

У миграций имеется префикс с отметкой времени в формате UTC (20121130000001), имя (create_products) и расширение имени файла (.rb, поскольку это код Ruby).

Тот префикс с отметкой времени, который вы увидите, будет отличаться от этого. В действительности отметки времени, используемые в данной книге, не имеют ничего общего с настоящими. Обычно ваши отметки времени не будут выстроены в четкой последовательности, и они будут отражать время создания миграции.

Применение миграций

Хотя мы уже сообщили Rails об основных типах данных каждого свойства, давайте уточним определение цены (`price`), чтобы у нее было восемь цифр в значимой части и две цифры после десятичного знака.

rails40/depot_a/db/migrate/20121130000001_create_products.rb

```
class CreateProducts < ActiveRecord::Migration
    def change
        create_table :products do |t|
            t.string :title
            t.text :description
            t.string :image_url
▶           t.decimal :price, precision: 8, scale: 2

            t.timestamps
        end
    end
end
```

После внесения изменений нужно заставить Rails применить эту миграцию к нашей разработочной базе данных. Мы это сделаем с помощью команды `rake`. Эта команда — надежный помощник, который всегда под рукой: вы приказываете ей выполнить определенную задачу, и она ее исправно выполняет. В данном случае

мы приказали rake реализовать любую миграцию, которая еще не применялась к нашей базе данных:

```
depot> rake db:migrate
== CreateProducts: migrating ==============================================
-- create_table(:products)
    -> 0.0027s
== CreateProducts: migrated (0.0023s) =====================================
```

Вот и все. Rake находит миграции, которые еще не применялись к базе данных, и применяет их. В нашем случае к базе данных, определенной в разделе development файла database.yml, добавляется таблица products.

Основа заложена. Мы установили наше приложение Depot в виде проекта Rails. Мы создали разработочную базу данных и сконфигурировали наше приложение на подключение к этой базе данных. Мы создали контроллер products и модель Product и воспользовались миграцией для создания соответствующей таблицы products. И для нас были созданы несколько представлений. Настала пора посмотреть, как все это работает.

Просмотр перечня товаров

Тремя командами мы создали приложение и базу данных (или таблицу внутри существующей базы данных, если вы выбрали не SQLite 3, а какую-нибудь другую программу управления базами данных). Прежде чем разбираться, что произошло за кулисами, давайте испытаем наше только что созданное приложение в работе.

Сначала нужно запустить локальный сервер, предоставляемый Rails:

```
depot> rails server
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Run 'rails server -h' for more startup options
=> Ctrl-C to shutdown server
[2013-04-18 17:45:38] INFO  WEBrick 1.3.1
[2013-04-18 17:45:38] INFO  ruby 2.0.0 (2013-02-24) [i386-mingw32]
[2013-04-18 17:45:43] INFO  WEBrick::HTTPServer#start: pid=4908 port=3000
```

Эта команда запускает веб-сервер на нашем локальном хосте с использованием порта 3000, то есть происходит то же самое, что было с нашим приложением demo, рассмотренным в главе 2. Если при попытке запуска сервера будет получено сообщение об ошибке «Address already in use» (адрес уже используется), это будет означать, что на данной машине уже есть запущенный сервер Rails. Если вы выполняли все упражнения, предлагаемые в данной книге, сервер мог быть запущен для приложения «Hello, World!» из главы 2. Нужно найти консоль этого сервера и прекратить его работу с помощью комбинации клавиш Ctrl+C. Если вы работаете в Windows, при этом можно увидеть приглашение на подтверждение: «Завершить выполнение пакетного файла [Y(да)/N(нет)]?» (Terminate batch job (Y/N)?). Если оно появится, ответьте y.

Давайте подключимся нашему приложению. Вспомним, что URL-адрес, вводимый в наш браузер, содержит номер порта (`3000`) и имя контроллера в нижнем регистре (`products`), как показано на рис. 6.1.

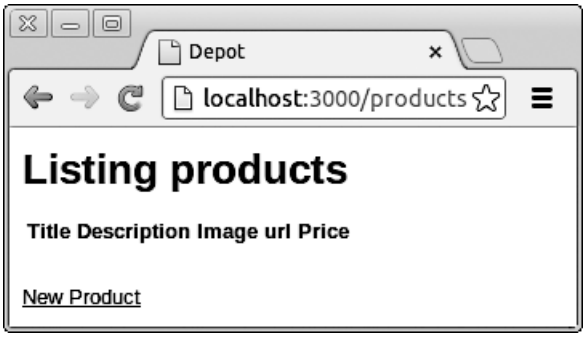


Рис. 6.1. Подключение к серверу

Поскольку перечень товаров пуст, ничего интересного мы там не увидим. Давайте этот перечень чем-нибудь наполним. Щелчок на ссылке New product (Новый товар) приведет к появлению формы, которую нужно будет заполнить (рис. 6.2).

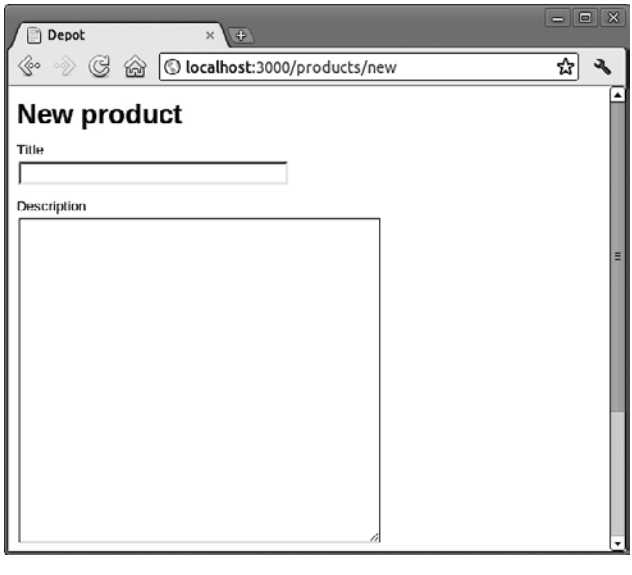


Рис. 6.2. Форма для добавления нового товара

Эти формы являются простыми HTML-шаблонами, похожими на те, которые создавались в разделе 2.2 «Привет, Rails!». Мы можем их изменить. Давайте изменим количество строк в поле `description` (описание):

rails40/depot_a/app/views/products/_form.html.erb

```
<%= form_for(@product) do |f| %>
    <% if @product.errors.any? %>
        <div id="error_explanation">
            <h2><%= pluralize(@product.errors.count, "error") %>
            prohibited this product from being saved:</h2>

            <ul>
            <% @product.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
            <% end %>
            </ul>
        </div>
    <% end %>

    <div class="field">
        <%= f.label :title %><br>
        <%= f.text_field :title %>
    </div>
    <div class="field">
        <%= f.label :description %><br>
        <%= f.text_area :description, rows: 6 %>
    </div>
    <div class="field">
        <%= f.label :image_url %><br>
        <%= f.text_field :image_url %>
    </div>
    <div class="field">
        <%= f.label :price %><br>
        <%= f.text_field :price %>
    </div>
    <div class="actions">
        <%= f.submit %>
    </div>
<% end %>
```

Более подробно все это будет рассмотрено в главе 8, «Задача B: Отображение каталога товаров». А сейчас, чтобы понять, что это такое, мы внесли изменение в одно из полей. Теперь продолжим работу и заполним форму (рис. 6.3).

Щелкните на кнопке Create (Создать), и вы увидите, что запись о новом товаре будет успешно создана. Если теперь щелкнуть на кнопке Back (Назад), вы увидите новый товар в перечне (рис. 6.4).

Может быть, это и не самый красивый интерфейс, но он работает, и мы можем предъявить его на утверждение нашему заказчику. Он может оценить работу других ссылок (показывающих подробности, позволяющих редактировать существующие сведения о товарах и т. д.). Нужно объяснить, что это всего лишь первый шаг и мы знаем, что он далек от совершенства, но нам хотелось получить отзыв заказчика как можно раньше. (И в какой-нибудь другой книге четырех команд для этого было бы, наверное, недостаточно.)

Рис. 6.3. Создание описания для нашего первого товара

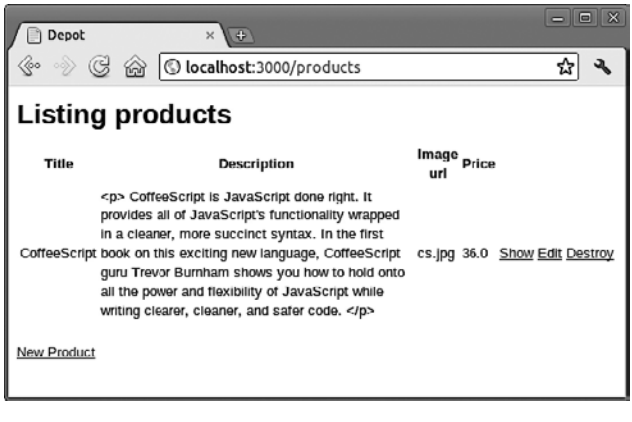


Рис. 6.4. Просмотр товара в том виде, в котором он появляется в базе данных

Всего лишь четыре команды позволили нам выполнить довольно большой объем работы. Прежде чем продолжить, попробуем воспользоваться еще одной командой:

```
rake test
```

В выводе этой команды должны присутствовать две строки, в каждой из которых сообщается: `0 failures, 0 errors` (0 сбоев, 0 ошибок). Это относится к тестам модели и контроллера, которые Rails генерирует вместе с созданием временной платформы. Пока эти тесты имеют минимальный объем, но сам факт их присутствия и успешного прохождения должен вселить в вас уверенность. По мере чтения глав части II запуск этой команды будет предлагаться довольно часто, поскольку это поможет вам выявить и отследить ошибки. Более подробно этот вопрос будет рассмотрен в разделе 7.2 «Шаг Б2: Блочное тестирование моделей».

Следует учесть, что при использовании базы данных, отличной от SQLite3, тестирование может не пройти. Проверьте содержимое своего файла database.yml и изучите материал главы 23.

6.2. Шаг А2: улучшение внешнего вида перечня товаров

У заказчика возникло еще одно требование (заказчикам всегда что-нибудь не нравится). По его мнению, перечень товаров имеет слишком неприглядный вид. Не можем ли мы его несколько «приукрасить»? И, если уж мы будем этим заниматься, нельзя ли наряду с URL-адресом изображения вывести само изображение товара?

Здесь у нас возникает дилемма. Как разработчики, мы учимся воспринимать подобные запросы, сделав резкий вдох, понимающе кивнув головой и вкрадчиво спросив: «А что бы вы хотели увидеть?» В то же время нам хочется продемонстрировать все, на что мы способны. В конце концов на первый план выходит та легкость, с которой подобные изменения делаются в Rails, и мы запускаем свой испытанный текстовый редактор.

Но перед тем как углубиться в работу, было бы неплохо обзавестись последовательным набором тестовых данных, с которыми можно работать. Мы *можем* воспользоваться нашим сгенерированным при создании временной платформы интерфейсом и ввести данные прямо в браузере. Но если мы это сделаем, будущие разработчики, работающие с основой нашего кода, вынуждены будут делать то же самое. И если мы работаем над этим проектом, представляя при этом только часть команды, каждому ее участнику придется вводить свои собственные данные. Было бы неплохо, если бы данные в таблицу можно было загрузить более управляемым способом. Оказывается, это в наших силах. Rails предоставляет нам возможность импортировать исходные данные.

Сначала мы просто внесем изменения в файл seeds.rb, который находится в каталоге db.

Затем мы добавим код для заполнения таблицы `products`. Для этого воспользуемся методом `create()` модели `Product`. Следующий код является извлечением из вышеупомянутого файла. Вместо того чтобы набирать содержимое файла вручную, можно загрузить файл из образца кода, имеющегося в Интернете[1].

[1] http://media.pragprog.com/titles/rails4/code/rails40/depot_a/db/seeds.rb

А заодно можно загрузить изображения[1] и поместить их в каталог app/assets/images вашего приложения. Но вы должны знать, что сценарий seeds.rb перед загрузкой новых данных удаляет из таблицы `products` все ранее находившиеся в ней данные. Если вы уже потратили несколько часов на ручной ввод своих данных в это приложение, возможно, вам не захочется запускать этот сценарий!

rails40/depot_a/db/seeds.rb

```
Product.delete_all
# . . .
Product.create!(title: 'Programming Ruby 1.9 & 2.0',
    description:
        %{<p>
            Ruby is the fastest growing and most exciting dynamic language
            out there. If you need to get working programs delivered fast,
            you should add Ruby to your toolbox.
        </p>},
    image_url: 'ruby.jpg',
    price: 49.95)
# . . .
```

Обратите внимание на то, что в коде используется элемент синтаксиса `%{…}`, являющийся альтернативой строковых литералов, взятых в двойные кавычки. Эта альтернатива удобна для использования с длинными строками. Также обратите внимание на то, что при использовании принадлежащего Rails метода `create()!` в случае невозможности вставки записей в базу данных из-за ошибок проверки данных будет выдано исключение.

Для заполнения таблицы `products` тестовыми данными нужно просто запустить следующую команду:

```
depot> rake db:seed
```

Теперь давайте приведем в порядок перечень товаров. Работа будет состоять из двух частей: определения набора стилевых правил и подключения этих правил к странице путем определения на ней HTML-атрибута `class`.

Нам нужно место, куда будут помещены наши определения стиля. Поскольку мы продолжаем работать с Rails, для нас в этой среде на данный счет есть соглашение, и ранее выданная команда `generate scaffolding` уже заложила всю нужную основу. Раз так, мы можем продолжить работу, заполняя пока еще пустую таблицу стилей products.css.scss, которая находится в каталоге app/assets/stylesheets.

rails40/depot_a/app/assets/stylesheets/products.css.scss

```
// Сюда помещаются все определения стилей для контроллера Products.
// Они будут автоматически включены в файл application.css.
// Что такое Sass (SCSS), можно узнать здесь: http://sass-lang.com/

.products {
    table {
        border-collapse: collapse;
    }
```

[1] http://media.pragprog.com/titles/rails4/code/rails40/depot_a/app/assets/images/

```
▶
▶  table tr td {
▶        padding: 5px;
▶        vertical-align: top;
▶    }
▶
▶    .list_image {
▶        width: 60px;
▶        height: 70px;
▶    }
▶
▶    .list_description {
▶        width: 60%;
▶
▶        dl {
▶            margin: 0;
▶        }
▶
▶        dt {
▶            color: #244;
▶            font-weight: bold;
▶            font-size: larger;
▶        }
▶
▶        dd {
▶            margin: 0;
▶        }
▶     }
▶
▶     .list_actions {
▶         font-size: x-small;
▶         text-align: right;
▶        padding-left: 1em;
▶     }
▶
▶     .list_line_even {
▶        background: #e0f8f8;
▶     }
▶
▶     .list_line_odd {
▶         background: #f8b0f8;
▶     }
▶ }
```

При выборе загрузки этого файла нужно убедиться в том, что его метка времени обновлена, в противном случае Rails не воспримет изменения до тех пор, пока сервер не будет перезапущен. Обновить метку времени можно, зайдя в свой любимый текстовый редактор и сохранив файл. На Mac OS X и на Linux для этого можно воспользоваться командой `touch`.

Если внимательно изучить эту таблицу стилей, можно заметить, что CSS-правила вложены друг в друга и правило для `dl` определено *внутри* правила для `.list_description`, которое, в свою очередь, определено внутри правила для

products. Тем самым правила избавляются от лишних повторений, их становится легче читать, понимать и обслуживать.

Пока вы были знакомы только с тем фактом, что в файлах, заканчивающихся на erb, происходит предварительная обработка встроенных выражений и инструкций Ruby. А эти файлы, если вы заметили, заканчиваются на scss, и вы, наверное, уже догадались, что данные файлы, перед тем как обслуживаться в качестве файлов css, проходят предварительную обработку в качестве продукта Sassy CSS[1]. И вы абсолютно правы!

Как и в случае с ERb, SCSS не конфликтует с написанным по всем правилам кодом CSS. Роль SCSS заключается в предоставлении дополнительного синтаксиса, позволяющего упростить разработку и обслуживание таблиц стилей. В ваших же интересах SCSS конвертирует все это в стандартный CSS, который понимает ваш браузер. Узнать больше о SCSS можно в книге «Pragmatic Guide to Sass».

И наконец, нам нужно определить класс products, используемый этой таблицей стилей. Если посмотреть на уже созданные файлы .html.erb, каких-либо ссылок на таблицы стилей вы в них не найдете. Вы даже не найдете HTML-раздел <head>, в котором обычно находятся такие ссылки. Вместо этого в Rails имеется отдельный файл, используемый для создания стандартной среды окружения страниц для всего приложения. Этот файл по имени application.html.erb является макетом Rails и находится в каталоге layouts:

rails40/depot_a/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Depot</title>
  <%= stylesheet_link_tag "application", media: "all",
    "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
▶ <body class='<%= controller.controller_name %>'>
<%= yield %>
</body>
</html>
```

Поскольку мы собрались загрузить все таблицы стилей сразу, нам нужно соглашение для ограничения распространения правил, указанных для контроллера, только на те страницы, которые связаны с этим контроллером. Выполнить эту задачу можно простым указанием в качестве имени класса значения controller_name, что мы здесь и сделали.

Теперь, когда все таблицы стилей находятся на своем месте, мы воспользуемся простым табличным шаблоном, отредактировав файл index.html.erb в каталоге app/views/products и заменив тем самым представление, сгенерированное при создании временной платформы:

[1] http://sass-lang.com/

rails40/depot_a/app/views/products/index.html.erb

```
<h1>Listing products</h1>

<table>
<% @products.each do |product| %>
    <tr class="<%= cycle('list_line_odd', 'list_line_even') %>">

        <td>
            <%= image_tag(product.image_url, class: 'list_image') %>
        </td>

        <td class="list_description">
            <dl>
                <dt><%= product.title %></dt>
                <dd><%= truncate(strip_tags(product.description),
                    length: 80) %></dd>
            </dl>
        </td>

        <td class="list_actions">
            <%= link_to 'Show', product %><br/>
            <%= link_to 'Edit', edit_product_path(product) %><br/>
            <%= link_to 'Destroy', product, method: :delete,
                        data: { confirm: 'Are you sure?' } %>
        </td>
    </tr>
<% end %>
</table>

<br />

<%= link_to 'New product', new_product_path %>
```

Даже в этом простом шаблоне используется ряд встроенных свойств Rails.

- Строки в перечне имеют чередующиеся фоновые цвета. Это делается с помощью вспомогательного метода Rails путем установки для каждой строки либо класса `list_line_even`, либо класса `list_line_odd`, что приводит к автоматическому переключению двух имен стилей для последовательных строк.
- Вспомогательный метод `truncate()` используется для отображения только первых восьмидесяти символов описания. Но перед вызовом метода `truncate()` мы вызываем метод `strip_tags()`, чтобы убрать из описания HTML-теги.
- Обратите внимание на то, что у строки ссылки `link_to 'Destroy'` есть параметр `data: { confirm: 'Вы уверены?' }`. Если щелкнуть на данной ссылке, Rails подстроится под ваш браузер для вывода диалогового окна, запрашивающего подтверждение на переход по ссылке и удаление записи о товаре. (Некоторые внутренние тонкости данного действия раскрыты в ближайшей врезке.)

Итак, мы загрузили в базу данных тестовые данные, переписали файл index.html.erb, отображающий перечень товаров, добавили таблицу стилей application.css.scss, и эта таблица стилей была загружена в нашу страницу с помощью файла разметки application.html.erb. Теперь вернемся в браузер и укажем в нем адрес http://localhost:3000/products. Появившийся в нем перечень товаров может выглядеть так, как показано на рис. 6.5.

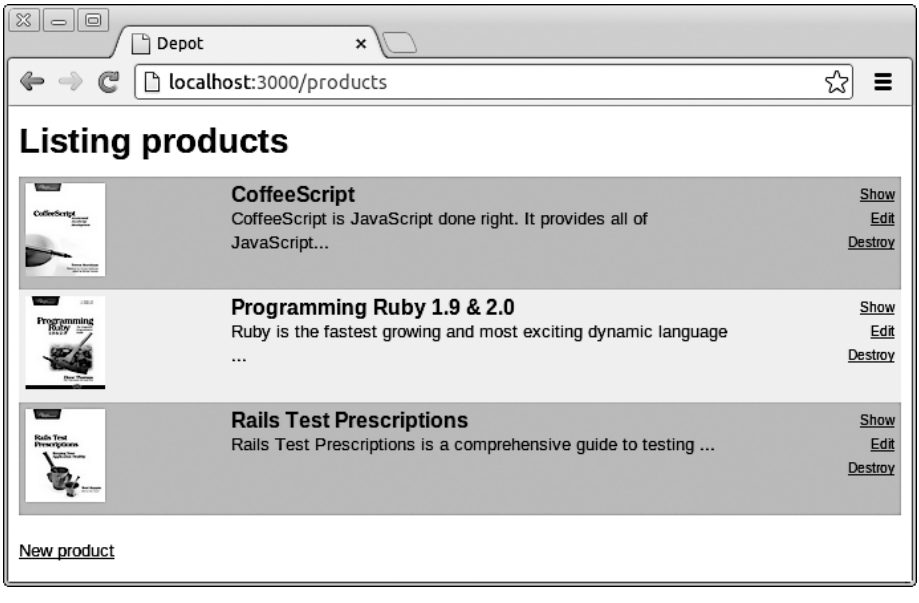


Рис. 6.5. Немного более привлекательный вид

Итак, мы с гордостью показываем заказчику новый перечень товаров, который его вполне удовлетворяет. Теперь пора приступить к созданию электронной витрины.

Наши достижения

В данной главе мы заложили основы для интернет-магазина.

- ☑ Создали базу данных, предназначенную для разработки.
- ☑ Использовали миграцию для создания и модификации схемы базы данных, предназначенной для разработки.
- ☑ Создали таблицу товаров `products` и воспользовались генератором временной платформы (scaffold) для создания приложения, с помощью которого можно вести учет товаров в этой таблице.
- ☑ Мы дополнили разметку, предназначенную для всего приложения, а также представление для конкретного контроллера, чтобы показать перечень товаров.

ЧТО ТАКОЕ METHOD: :DELETE?

Нетрудно заметить, что сгенерированная при создании временной платформы ссылка «Удалить» включает параметр method: :delete. Этот параметр определяет метод, вызываемый в классе ProductsController, а также влияет на используемый HTTP-метод.

Браузеры используют протокол HTTP для обмена данными с серверами. В протоколе HTTP определяется набор глаголов, которым браузер может воспользоваться, и определяется, когда и какой глагол может быть использован. Обычная гиперссылка, к примеру, использует запрос HTTP GET. Этот запрос определен протоколом HTTP для извлечения данных и не должен иметь никаких побочных эффектов. Использование данного параметра именно таким образом показывает, что для данной гиперссылки будет использован метод HTTP DELETE. Rails использует эту информацию для определения, какому действию в контроллере направить этот запрос.

Следует заметить, что при использовании внутри браузера, Rails заменит методы PUT и DELETE методом POST HTTP в процессе присоединения дополнительного параметра, чтобы маршрутизатор мог определить суть исходных намерений. В любом случае запрос не будет кэширован или вызван поисковым агентом Интернета.

Все сделанное нами не потребовало больших усилий и настроило нас на активное продолжение работы. Базы данных жизненно важны для данного приложения, но они не должны вас чем-то настораживать: в большинстве случаев выбор базы данных можно отложить на будущее и приступить к работе, используя ту базу данных, которая предоставлена Rails по умолчанию.

На данном этапе более важно правильно понять саму модель разработки. Как вскоре выяснится, простой выбор типов данных не всегда дает полное представление обо всех свойствах модели, даже в небольших приложениях, но с этим нам еще предстоит разобраться.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- Если вы уверены в своих силах, можете поэкспериментировать с откатом миграции. Просто наберите следующую команду:

```
depot> rake db:rollback
```

  Ваша схема переместится назад во времени, и таблица `products` исчезнет. Повторный вызов команды `rake db:migrate` создаст эту таблицу заново. Может также понадобиться заново заполнить таблицу тестовыми данными. Дополнительная информация о миграциях может быть получена в главе 22 «Миграции».

- Вопрос управления версиями уже затрагивался в соответствующем разделе главы 1, а теперь у нас появился серьезный повод для сохранения нашей работы. Если вы выбрали для этой цели систему Git (которую я, кстати, настоятельно рекомендую), сначала придется немного заняться настройками конфигурации — в основном придется всего лишь предоставить свое имя и адрес электронной почты.

```
depot> git config --global --add user.name "Sam Ruby"
depot> git config --global --add user.email rubys@intertwingly.net
```

Проверить конфигурацию можно с помощью следующей команды:

```
depot> git config --global --list
```

Rails также предоставляет файл по имени .gitignore, сообщающий системе Git о тех файлах, которые не попадают под управление версиями:

rails40/depot_a/.gitignore

```
# See http://help.github.com/ignore-files/ for more about ignoring files.
#
# If you find yourself ignoring temporary files generated by your text editor
# or operating system, you probably want to add a global ignore instead:
# git config --global core.excludesfile '~/.gitignore_global'

# Ignore bundler config.
/.bundle

# Ignore the default SQLite database.
/db/*.sqlite3
/db/*.sqlite3-journal

# Ignore all logfiles and tempfiles.
/log/*.log
/tmp
```

Из-за того, что имя этого файла начинается с точки, операционные системы на основе Unix не показывают его в листингах каталогов. Чтобы увидеть этот файл, нужно воспользоваться командой `ls -a`. Теперь все настройки завершены, и осталось только инициализировать репозиторий, добавив все файлы и передав их с сообщением о передаче:

```
depot> git init
depot> git add .
depot> git commit -m "Depot Scaffold"
```

Возможно, на данном этапе все это не кажется вам слишком увлекательным, но значение использования данной системы заключается в том, что у вас появляется возможность для свободного экспериментирования. Если будет переписан или удален не предназначавшийся для этого файл, можно будет вернуться к данной точке с помощью всего лишь одной команды:

```
depot> git checkout .
```

Задача Б: проверка приемлемости данных и блочное тестирование

7

Основные темы:

- выполнение проверки приемлемости данных и вывод сообщений об ошибках;
- блочное тестирование.

На данный момент у нас уже есть исходная модель для товара, а также завершенное приложение для ведения перечня товара, предоставленное нам временной платформой Rails. В этой главе мы собираемся сконцентрировать внимание на повышении надежности модели, то есть, перед тем как в следующих главах перейти к другим аспектам приложения Depot, преградить ошибкам, допущенным при вводе данных, путь в базу данных.

7.1. Шаг Б1: проверка приемлемости данных

Когда заказчик поработал с результатом, достигнутым при выполнении шага А1, он кое-что заметил. Если он вводил неверную цену или забывал давать товару описание, приложение без малейших сомнений принимало введенную форму и добавляло строку в базу данных. Отсутствие описания — это, конечно, еще полбеды, но вот нулевая цена — это уже вопрос денег заказчика, поэтому он попросил ввести в приложение проверку приемлемости данных. Товар не должен попадать в базу данных, если у него пустое поле названия или описания, или же неверный URL-адрес изображения, или неправильная цена.

Итак, куда же нам вставить проверку? Привратником между миром программного кода и базой данных является уровень модели. Все, что в нашем приложении выходит из базы данных и попадает туда на хранение, не проходит мимо модели. Поэтому модель становится идеальным местом, куда можно вставить проверку. И не важно, откуда поступают данные — из формы или в результате программных манипуляций в нашем приложении. Если модель проверяет данные перед их записью в базу данных, тогда эта база будет защищена от некачественных данных.

Взглянем еще раз на исходный код класса модели (который находится в файле app/models/product.rb):

```
class Product < ActiveRecord::Base
end
```

В добавлении проверки данных не должно быть ничего лишнего. Начнем с проверки наличия какого-нибудь содержимого во всех текстовых полях перед записью строки в базу данных. Для этого добавим в существующую модель следующую строку кода:

```
validates :title, :description, :image_url, presence: true
```

Метод `validates()` является в Rails стандартным средством проверки (валидатором). Он будет проверять одно или несколько полей модели на соблюдение одного или нескольких условий.

Часть инструкции `presence: true` предписывает валидатору проверять наличие данных в каждом из указанных полей. На рис. 7.1 можно увидеть, что получится, если мы попробуем отправить сведения о новом товаре, не заполнив ни одного поля. Картина будет весьма впечатляющей: поля с ошибками будут выделены, а сводка об ошибках помещена в красочном списке в верхней части формы. Неплохо для всего одной строки кода. Также можно заметить, что после редактирования и сохранения файла product.rb перезапускать приложение для проверки изменений не пришлось — перезагрузка страницы, заставившая ранее Rails заметить изменения в схеме данных, всегда приводит к тому, что используется самая последняя версия кода.

Нужно также проверить, что цена имеет допустимое, положительное числовое значение. Для проверки воспользуемся методом с выразительным именем `numericality`. Мы также передадим несколько многословному методу `greater_than_or_equal_to` (больше чем или равно) значение 0.01:

```
validates :price, numericality: {greater_than_or_equal_to: 0.01}
```

Теперь, если добавить товар с недопустимой ценой, появится соответствующее сообщение, показанное на рис. 7.2.

Почему проверка велась относительно значения 0.01, а не нуля? Потому что можно ведь ввести в это поле и такое значение, как 0.001. Поскольку база данных сохраняет только две цифры после десятичной точки, в ней окажется нуль, даже если поле пройдет проверку на ненулевое значение. Проверка того, что число, по крайней мере, равно 0.01, гарантирует, что будет сохранено только допустимое значение.

Рис. 7.1. Проверка наличия данных в полях

Нам нужно проверить еще два элемента. Сначала нам нужно убедиться в том, что у каждого товара есть свое уникальное название. Эта задача решается с помощью еще одной строки кода в модели `Product`. Проведем простую проверку, гарантирующую, что никакая другая строка в таблице `products` не имеет названия, указанного в той строке, которую мы собираемся сохранить:

```
validates :title, uniqueness: true
```

И наконец, нам нужно проверить приемлемость введенного URL-адреса изображения. Для этого воспользуемся методом `format`, который определяет соответствие значения поля регулярному выражению. В данный момент мы просто проверим, что URL-адрес заканчивается одним из расширений: `.gif`, `.jpg` или `.png`.

```
validates :image_url, allow_blank: true, format: {
    with: %r{\.(gif|jpg|png)\Z}i,
    message: 'URL должен указывать на изображение формата GIF, JPG или PNG.'
}
```

Рис. 7.2. Цена не проходит проверку на приемлемость

Заметьте, что во избежание получения нескольких сообщений об ошибках при пустом поле используется параметр `allow_blank`.

Может быть, позже нам захочется изменить эту форму, дав пользователю возможность выбора из перечня доступных изображений, но проверка приемлемости все равно будет сохранена, чтобы злоумышленники не смогли отправить неверные данные непосредственно из этого поля.

Итак, всего за пару минут мы добавили следующие виды проверок:

- поля названия, описания и URL-адреса изображения не оставлены пустыми;
- цена является допустимым числом, значение которого не меньше 0.01;
- название имеет уникальное значение среди всех товаров;
- URL-адрес изображения выглядит вполне приемлемо.

Обновленная модель `Product` должна приобрести следующий вид:

rails40/depot_b/app/models/product.rb

```
class Product < ActiveRecord::Base
    validates :title, :description, :image_url, presence: true
    validates :price, numericality: {greater_than_or_equal_to: 0.01}
    validates :title, uniqueness: true
    validates :image_url, allow_blank: true, format: {
        with: %r{\.(gif|jpg|png)\Z}i,
        message: 'must be a URL for GIF, JPG or PNG image.'
                 # URL должен указывать на изображение формата GIF, JPG или PNG
    }
end
```

Ближе к завершению данного цикла мы попросили заказчика поработать с приложением, и он остался доволен. Все это заняло лишь несколько минут, но простое добавление проверки приемлемости данных придало страницам ведения перечня товаров более внушительный вид.

Прежде чем двигаться дальше, еще раз протестируем приложение:

```
rake test
```

Как ни странно, на этот раз тест не прошел. Причем в двух местах: в `should create product` (нужно создать товар) и в `should update product` (нужно обновить товар). Понятно, что какие-то наши действия вынуждают что-нибудь сделать с созданием и обновлением товаров. Здесь нет ничего неожиданного. Если подумать, разве это в конечном счете является самым главным в проверке приемлемости данных?

Проблема решается предоставлением допустимых тестовых данных в файле test/controllers/products_controller_test.rb:

rails40/depot_b/test/controllers/products_controller_test.rb

```
require 'test_helper'
class ProductsControllerTest < ActionController::TestCase
    setup do
        @product = products(:one)
▶           @update = {
▶               title: 'Lorem Ipsum',
▶               description: 'Wibbles are fun!',
▶               image_url: 'lorem.jpg',
▶               price: 19.95
▶           }
    end
    test "should get index" do
        get :index
        assert_response :success
        assert_not_nil assigns(:products)
    end
    test "should get new" do
        get :new
        assert_response :success
    end
```

```
    test "should create product" do
        assert_difference('Product.count') do
            post :create, product: @update
        end
        assert_redirected_to product_path(assigns(:product))
    end

    # ...
    test "should update product" do
         patch :update, id: @product, product: @update
        assert_redirected_to product_path(assigns(:product))
    end

    # ...
end
```

После внесения этих изменений следует перезапустить тесты, и они отрапортуют о том, что все нормально. Но все это означает, что мы ничего не испортили. А нам нужно нечто большее. Теперь нужно убедиться в том, что только что добавленный код проверки приемлемости не только работает, но и будет работать после внесения будущих изменений. Более подробно тесты контроллеров будут рассмотрены в разделе 8.4 «Шаг В4: Функциональное тестирование контроллеров». А теперь настала пора написать несколько блочных тестов.

7.2. Шаг Б2: блочное тестирование моделей

Одно из реальных преимуществ среды Rails заключается в том, что тестирование в ней «выпекается» с запуском каждого нового проекта. Как мы уже видели, среда Rails приступает к созданию для вас тестовой инфраструктуры с момента создания нового приложения с помощью команды `rails`.

Заглянем в содержимое подкаталога models:

```
depot> dir test\models /w
[.]              [..]              product_test.rb
```

product_test.rb является файлом, который Rails создает, чтобы хранить блочные тесты для модели, ранее созданной с помощью сценария generate. Неплохо для начала, но это все, чем может помочь Rails.

Посмотрим, что полезного для тестирования Rails сгенерировала внутри файла test/models/product_test.rb при создании данной модели:

rails40/depot_b/test/models/product_test.rb

```
require 'test_helper'
class ProductTest < ActiveSupport::TestCase
    # test "the truth" do
    # assert true
    # end
end
```

Сгенерированный `ProductTest` является подклассом `ActiveSupport::TestCase`. Тот факт, что `ActiveSupport::TestCase` является подклассом класса `MiniTest::Unit::TestCase`, свидетельствует о том, что Rails генерирует тесты на основе структуры MiniTest[1], которая поступает в предустановленном виде вместе с Ruby.

Для нас это хорошая новость, поскольку это означает, что если мы уже тестировали наши Ruby-программы с помощью тестов MiniTest (а кто же от этого откажется?), то эти знания можно взять за основу тестирования Rails-приложений. Но если опыта работы с MiniTest нет, ничего страшного. Постепенно вы все поймете.

Внутри рассматриваемого теста Rails сгенерировала один-единственный закомментированный тест под названием `"the truth"`. Синтаксис `test...do` поначалу может показаться необычным, но здесь Active Support объединяет метод класса, необязательные круглые скобки и блок, определяющий тест-метод, тем самым всего лишь немного упрощая вашу работу. Иногда именно такие мелочи и имеют значение.

Строка `assert` в данном методе, по сути, и является настоящим тестом. Толку от него, конечно, немного, все, что он делает, — это проверяет, что истина (`true`) является истиной. Понятно, что это всего лишь заполнитель, предназначенный для замены настоящим тестом.

Настоящий блочный тест

Давайте займемся тестированием проверки приемлемости данных. В первую очередь, если товар будет создан без указания его свойств, ожидается, что он будет забракован, и планируется объявление ошибки, связанной с каждым полем. Чтобы увидеть, проводит ли модель проверку приемлемости данных, можно воспользоваться методами `errors()` и `invalid?()` этой модели, а для того, чтобы понять, есть ли ошибка, связанная с конкретным свойством, можно воспользоваться методом `any?()` списка ошибок.

Теперь, зная, *что* тестировать, нужно узнать, *как* сообщить среде тестирования о том, прошел наш код проверку или нет. Это делается с помощью *утверждений*. Утверждение — это обычный вызов метода, в котором среде тестирования сообщается, что именно для нас является истиной.

Простейшим утверждением является метод `assert`, который выдвигает предположение, что его аргументы должны быть истинными. Если это так, ничего особенного не происходит. Но если аргументы у `assert` ложные, утверждение не выполняется. Среда тестирования выведет сообщение и остановит выполнение метода тестирования, содержащего ошибку. В нашем случае ожидается, что незаполненная модель `Product` не пройдет проверку, и это можно выразить в виде утверждения, что товар действительно не пройдет проверку на приемлемость.

[1] http://www.ruby-doc.org/stdlib-2.0/libdoc/minitest/unit/rdoc/MiniTest.html

```
assert !product.valid?
```

Замените тест `the truth` следующим кодом:

rails40/depot_b/test/models/product_test.rb

```
test "product attributes must not be empty" do
    # свойства товара не должны оставаться пустыми
    product = Product.new
    assert product.invalid?
    assert product.errors[:title].any?
    assert product.errors[:description].any?
    assert product.errors[:price].any?
    assert product.errors[:image_url].any?
end
```

Перезапустить только блочные тесты можно с помощью команды `rake test:models`. Если это сделать, будет видно, что на этот раз тестирование выполнено успешно:

```
depot> rake test:models
.
Finished tests in 0.257961s, 3.8766 tests/s, 19.3828 assertions/s.
1 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Вполне очевидно, что проверка приемлемости данных включена и все утверждения пройдены.

Ясно, что здесь можно копнуть еще глубже и подвергнуть испытанию отдельные проверочные функции. Из множества возможных тестов мы рассмотрим только три.

Сначала протестируем, насколько хорошо работает проверка приемлемости цены:

rails40/depot_c/test/models/product_test.rb

```
test "product price must be positive" do
    # цена товара должна быть положительной
    product = Product.new(title:         "My Book Title",
                          description: "yyy",
                          image_url:     "zzz.jpg")
    product.price = -1
    assert product.invalid?
    assert_equal ["must be greater than or equal to 0.01"],

    product.errors[:price]
          # должна быть больше или равна 0.01
    product.price = 0

    assert product.invalid?
    assert_equal ["must be greater than or equal to 0.01"],
        product.errors[:price]
    product.price = 1
    assert product.valid?
end
```

В этом коде мы создаем новый товар, а затем пытаемся установить для него цену `-1`, `0` и `+1`, каждый раз проверяя его при этом. Если наша модель работает, то первые две проверки должны провалиться, и мы проверяем появление ожидаемого нами сообщения об ошибке, связанного с ценой.

Последняя цена является приемлемой, поэтому мы утверждаем, что теперь модель допустима. (Возможно, кто-то поместил бы эти три теста в три отдельных тестовых метода, и это было бы вполне разумно.)

Затем протестируем проверку окончания URL-адресов изображений одним из допустимых расширений: .gif, .jpg или .png:

rails40/depot_c/test/models/product_test.rb

```
def new_product(image_url)
    Product.new(    title:         "My Book Title",
                description:    "yyy",
                price: 1,
                image_url: image_url)
end

test "image url" do
    # url изображения
    ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
            http://a.b.c/x/y/z/fred.gif }
    bad = %w{ fred.doc fred.gif/more fred.gif.more }

    ok.each do |name|
        assert new_product(name).valid?, "#{name} shouldn't be invalid"
                                # не должно быть неприемлемым
    end

    bad.each do |name|
        assert new_product(name).invalid?, "#{name} shouldn't be valid"
                                # не должно быть приемлемым
    end
end
```

Здесь все несколько запутано. Вместо написания девяти отдельных тестов используется два цикла: один — для проверки вариантов, которые предположительно пройдут проверку на приемлемость, а второй — для проверки вариантов, которые предположительно не пройдут эту проверку. А общий код для этих двух циклов вынесен за их пределы.

Обратите внимание, что к вызову нашего метода `assert` добавлен еще один аргумент. Все тестовые утверждения принимают еще один замыкающий аргумент, содержащий строку, которая будет записана вместе с сообщением об ошибке, если утверждение не подтвердится, и будет полезной при диагностике нештатной ситуации.

И наконец, наша модель содержит проверку на уникальность всех названий товаров, хранящихся в базе данных. Чтобы протестировать эту проверку, нам нужно сохранить данные о товарах в базе данных.

Можно было бы воспользоваться таким тестом, который создает товар, сохраняет его, затем создает еще один товар с таким же названием и пытается его сохранить. Понятно, что такой тест был бы работоспособен. Но есть куда более простой способ — воспользоваться *испытательными стендами* Rails.

Испытательные стенды

В мире испытаний *стенды* являются той средой, в которой можно запустить тестирование. Если, к примеру, тестируется монтажная плата, можно установить ее на испытательном стенде, который подает на нее питание и входные сигналы, необходимые для управления тестируемой функции.

В мире Rails испытательный стенд — это просто спецификация исходного содержимого тестируемой модели (или моделей). Если, к примеру, мы хотим обеспечить, чтобы перед каждым блочным тестированием в начале таблицы товаров `products` хранились известные нам данные, мы можем определить это содержимое в испытательном стенде, а Rails позаботится обо всем остальном.

Данные стенда определяются в файлах каталога test/fixtures. В этих файлах данные содержатся в формате YAML. Каждый стендовый файл формата YAML содержит данные для одной модели. Имя стендового файла должно совпадать с именем таблицы базы данных. Поскольку нам нужны некоторые данные для модели `Product`, размещенные в таблице товаров, мы впишем их в файл, названный products.yml.

Rails уже создала этот стендовый файл одновременно с созданием модели:

rails40/depot_b/test/fixtures/products.yml

```
# Read about fixtures at http://api.rubyonrails.org/classes/Fixtures.html

one:
    title: MyString
    description: MyText
    image_url: MyString
    price: 9.99

two:
    title: MyString
    description: MyText
    image_url: MyString
    price: 9.99
```

Стендовый файл содержит запись для каждой строки, которую нужно вставить в базу данных. Каждой строке дается имя. В стенде, сгенерированном Rails, строки названы `one` и `two`. Для базы данных эти имена ничего не значат — они не вставляются в строки данных. Но вскоре будет показано, что имена дают нам удобный способ ссылки на тестовые данные внутри кода нашего теста. Эти имена используются также в сгенерированных комплексных тестах, поэтому пока мы их касаться не будем.

ДЭВИД ГОВОРИТ: ВЫБИРАЙТЕ ПОНЯТНЫЕ СТЕНДОВЫЕ ИМЕНА

Чаще всего нам хочется, чтобы имена переменных по возможности говорили сами за себя, то же самое относится и к стендовым данным. Тесты намного удобнее читаются, когда вы строите утверждение вида product(:valid_order_for_fred), и это без сомнения воспринимается как правильный заказ для Фреда. К тому же намного проще запомнить, какой стенд вы хотите протестировать, не выискивая названия вроде p1 или order4. Чем больше создано стендов, тем важнее дать им всем вполне понятные имена. Поэтому старайтесь с самого начала закладывать фундамент своего будущего успеха.

А что же делать со стендами, которым не так-то просто присвоить смысловые имена вроде valid_order_for_fred? Подберите естественное название, которое ближе всего подходит к их роли. Например, вместо того чтобы использовать order1, воспользуйтесь именем christmas_order. Вместо customer1 используйте fred. Как только у вас выработается привычка давать естественные имена, вскоре вы уже начнете сочинять небольшой рассказ о том, как Фред расплачивается за свой рождественский заказ (christmas_order) сначала неправильной кредитной картой (invalid_credit_card), а затем правильной кредитной картой (valid_credit_card) и в завершение выбирает отправку всего заказа своей тете Мэри (aunt_mary).

Истории, основанные на ассоциациях, являются ключом к запоминанию целого мира стендовых данных.

Внутри каждой записи вы увидите оформленный с отступом список, состоящий из пар имя-значение. Точно так же, как и в вашем файле config/database.yml, в начале каждой из строк данных можно использовать пробелы, но не символы табуляции, и все строчки, формирующие строку базы данных, должны иметь одинаковый отступ.

При внесении изменений нужно проявлять особую осторожность, поскольку необходимо обеспечить, чтобы в каждой записи использовались правильные имена столбцов — расхождение с именем столбца базы данных может привести к трудно отслеживаемому исключению.

Добавим к стендовому файлу данные, которые можно использовать при тестировании нашей модели `Product`:

rails40/depot_c/test/fixtures/products.yml

```
ruby:
    title: Programming Ruby 1.9
    description:
        Ruby is the fastest growing and most exciting dynamic
        language out there. If you need to get working programs
        delivered fast, you should add Ruby to your toolbox.
    price: 49.50
    image_url: ruby.png
```

При наличии стендового файла нужно, чтобы при запуске блочного теста среда Rails загружала тестовые данные в таблицу `products`. И Rails уже так и делает (тут опять для успеха всей нашей работы соглашение превалирует над настройкой конфигурации!), но какой из стендовых файлов нужно загружать, можно определить с помощью следующей строки в файле test/models/product_test.rb:

```
class ProductTest < ActiveSupport::TestCase
▶     fixtures :products
    #...
end
```

Инструкция `fixtures()` приводит к загрузке стендовых данных, соответствующих заданному имени модели в соответствующей таблице базы данных. Эти данные загружаются прежде, чем при тестировании будет запущен каждый тестовый метод. Имя стендового файла определяет загружаемую таблицу, поэтому указание `:products` приведет к тому, что будет использован стендовый файл products.yml.

Выразим эту мысль иным способом. В случае использования нашего класса `ProductTest` инструкция `fixtures` означает, что таблица `products` перед запуском каждого тестового метода будет очищена, а затем заполнена тремя строками, определенными в стендовом файле.

Следует заметить, что большинство временных платформ, генерируемых Rails, не содержит вызовов метода `fixtures`. Причина в том, что при тестировании по умолчанию перед запуском теста загружаются все стендовые данные. Поскольку обычно такое поведение по умолчанию вас вполне устраивает, ничего не нужно менять. И снова соглашение используется для избавления от ненужных настроек конфигурации.

До сих пор мы работали только в разработочной базе данных. Но теперь, когда мы запускаем тесты, среде Rails нужно использовать тестовую базу данных. Если посмотреть на содержимое файла database.yml в каталоге config, можно увидеть, что Rails уже создала конфигурацию для трех отдельных баз данных:

- db/development.sqlite3 будет нашей разработочной базой данных. В ней будет вестись вся наша работа по разработке программы;
- db/test.sqlite3 — тестовая база данных;
- db/production.sqlite3 — база данных готового изделия. Эта база будет использоваться нашим приложением, когда оно будет запущено в сетевой среде.

Каждый тестовый метод получает заново проинициализированную таблицу в тестовой базе данных, данные в которую загружаются из предоставленных нами стендовых файлов. Это происходит автоматически при запуске команды `rake test`, но может быть сделано и отдельно, путем запуска команды `rake db:test:prepare`.

Использование стендовых данных

Теперь, когда стало известно, как заполучить стендовые данные в базу данных, нужно понять, как их использовать в тестах.

Понятно, что одним из способов чтения данных будет применение методов поиска в модели. Но Rails предлагает более легкий способ. Для каждого загружаемого в тест стенда она определяет метод с тем же именем. Вы можете использовать этот метод для доступа к уже загруженным объектам модели, содержащим стендовые данные: стоит только передать ему имя записи в том виде, как оно определено в стендовом файле YAML-формата, и будет возвращен объект модели, содержащий данные этой записи. В случае с нашими данными о товаре вызов `products(:ruby)`вернет модель `Product`, содержащую данные, определенные

в стенде. Воспользуемся всем этим, чтобы протестировать проверку уникальности названий товаров.

rails40/depot_c/test/models/product_test.rb

```
test "product is not valid without a unique title" do
    # если у товара нет уникального названия, то он недопустим
    product = Product.new(title: products(:ruby).title,
        description:    "yyy",
        price:        1,
        image_url:    "fred.gif")

    assert product.invalid?

    assert_equal ["has already been taken"], product.errors[:title]
                # уже было использовано
end
```

В тесте предполагается, что база данных уже включает строку для книги по Ruby. Он берет название (`title`) из этой уже существующей строки, используя следующий код:

```
products(:ruby).title
```

Затем он создает новую модель `Product`, устанавливая для нее уже существующее название. Тест утверждает, что попытка сохранить эту модель будет неудачной и что свойство `title` имеет соответствующую связанную с ним ошибку.

Если для сообщения об ошибке Active Record использовать жестко заданную строку не хочется, можно подобрать соответствующее сообщение из встроенной таблицы ошибок:

rails40/depot_c/test/models/product_test.rb

```
test " product is not valid without a unique title - i18n" do
    product = Product.new(title:         products(:ruby).title,
                          description: "yyy",
                          price:        1,
                          image_url:    "fred.gif")

    assert product.invalid?

    assert_equal [I18n.translate('activerecord.errors.messages.taken')],
        product.errors[:title]
end
```

Функция `I18n` будет рассмотрена в главе 15 «Задача К: интернационализация».

Теперь мы можем быть уверены, что наш код проверки приемлемости не только работает, но и будет работать в дальнейшем. У нашего товара теперь есть модель, набор представлений, контроллер и набор блочных тестов. Это послужит хорошей основой, на которой можно будет построить все остальное приложение.

Наши достижения

Задействовав всего с десяток строчек кода, мы дополнили сгенерированный код проверкой приемлемости данных.

- ☑ Мы гарантировали наличие данных в требуемых полях.
- ☑ Мы гарантировали, что поле цены будет числовым и будет иметь значение, равное хотя бы одному центу.
- ☑ Мы гарантировали уникальность названий товаров.
- ☑ Мы гарантировали соответствие изображений заданному формату.
- ☑ Мы обновили блочные тесты, предоставленные Rails, чтобы они соответствовали наложенным на модель ограничениям, а также чтобы проверить новый код, который был добавлен.

Мы показали проделанную работу заказчику, и хотя он согласился, что все это вполне может быть использовано администратором, при этом заметил, что это, конечно же, совсем не то, что он считает удобным для использования его будущими клиентами. Понятно, что далее нам придется сконцентрироваться на создании пользовательского интерфейса.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи:

- ❍ Если вы пользуетесь системой Git, то сейчас, возможно, настал подходящий момент для фиксации нашей работы. Сначала можно посмотреть, какие файлы были изменены, воспользовавшись для этого командой `git status`:

```
depot> git status
# On branch master
# 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: app/models/product.rb
# modified: test/fixtures/products.yml
# modified: test/controllers/products_controller_test.rb
# modified: test/models/product_test.rb
# no changes added to commit (use "git add" and/or "git commit -a")
```

  Поскольку мы внесли изменения только в некоторые уже существующие файлы и не добавили никаких новых файлов, мы можем объединить команды `git add` и `git commit` и выдать единую команду `git commit` с ключом `-a`:

```
depot> git commit -a -m 'Validation!'
```

Как только это будет сделано, мы можем делать что угодно, зная, что в случае отказа все в любой момент можно будет вернуть к сохраненному состоянию, используя всего лишь одну команду `git checkout .`

- Ключ проверки приемлемости `:length` проверяет длину свойства модели. Добавьте эту проверку приемлемости к модели `Product`, чтобы проверить, что в названии товара присутствует не менее 10 символов.
- Измените сообщение об ошибке, связанное с одной из ваших проверок приемлемости данных.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

8

Задача B: отображение каталога товаров

Основные темы:

- создание своих собственных представлений;
- использование элементов разметки для улучшения внешнего вида страниц;
- включение CSS;
- использование помощников;
- создание функциональных тестов.

Пока в основном все у нас шло успешно. Были собраны начальные требования заказчика, отображены на бумаге основные потоки данных, выработан первый подход к необходимым данным и собрана страница для ведения перечня товаров интернет-магазина. У нас даже есть небольшой, но наращиваемый набор тестов.

И мы с воодушевлением переходим к решению следующей задачи. При обсуждении приоритетов с заказчиком он высказал пожелание увидеть приложение глазами покупателя. Поэтому следующей задачей будет создание простого отображения каталога товаров.

Мы также считаем это желание вполне обоснованным. Раз у нас уже есть товары, помещенные в базу данных после соответствующей проверки, вывести их на экран особого труда не составит. Тем самым будут заложены основы для дальнейшей разработки кода корзины покупателя.

Нужно будет также воспользоваться уже проделанной работой по ведению перечня товаров, ведь, по сути дела, отображение каталога товаров — не что иное, как вывод их привлекательно оформленного перечня.

И наконец, нам также нужно будет дополнить наши блочные тесты для модели несколькими функциональными тестами для контроллера.

8.1. Шаг B1: создание каталога товаров

У нас уже создан контроллер товаров, используемый продавцом для управления приложением Depot. Настало время создать второй контроллер, который будет работать с покупателями. Назовем его `Store`.

```
depot> rails generate controller Store index
    create app/controllers/store_controller.rb
     route get "store/index"
    invoke erb
    create app/views/store
    create app/views/store/index.html.erb
    invoke test_unit
    create test/controllers/store_controller_test.rb
    invoke helper
    create app/helpers/store_helper.rb
    invoke test_unit
    create test/helpers/store_helper_test.rb
    invoke assets
    invoke coffee
    create app/assets/javascripts/store.js.coffee
    invoke scss
    create app/assets/stylesheets/store.css.scss
```

Как и ранее при создании контроллера и связанной с ним временной платформы для работы с перечнем товаров, воспользуемся утилитой `generate`, но теперь потребуем от нее создать контроллер (класс `StoreController` в файле store_controller.rb), содержащий один метод действия — `index()`.

Поскольку для доступа к этому действию по адресу http://localhost:3000/store/index (можете попробовать!) все настройки уже произведены, мы можем кое-что усовершенствовать. Давайте упростим ситуацию для пользователя и сделаем этот URL-адрес корневым для веб-сайта. Отредактируем для этого файл config/routes.rb:

rails40/depot_d/config/routes.rb

```
Depot::Application.routes.draw do
    get "store/index"

    resources :products

    # The priority is based upon order of creation:
    # (Приоритет основан на порядке создания:)
    # first created -> highest priority.
    # (создан первым -> наивысший приоритет.)
    # See how all your routes lay out with "rake routes".
    # (Раскладку всех маршрутов можно увидеть с помощью команды "rake routes".)

    # You can have the root of your site routed with "root"
    # (Корневой маршрут к вашему сайту можно получить с помощью "root")
▶   root to: 'store#index', as: 'store'
    # ...
    end
```

В верхней части файла можно увидеть строки, добавленные для поддержки контроллеров магазина и товаров. Эти строки мы изменять не будем. Далее в файле вы увидите закомментированную строку, определяющую корневой адрес для веб-сайта. Можно либо убрать знак комментария из этой строки, либо добавить сразу же после нее новую строку. В этой строке будет изменено только название контроллера (с `welcome` на `store`) и добавлено `as: 'store'`. Последнее добавление заставит Rails создать метод доступа `store_path`, как это было в разделе 2.3 с `say_goodbye_path`.

Посмотрим, что получилось. Укажем браузеру адрес http://localhost:3000/ и выведем нашу веб-страницу (рис. 8.1).

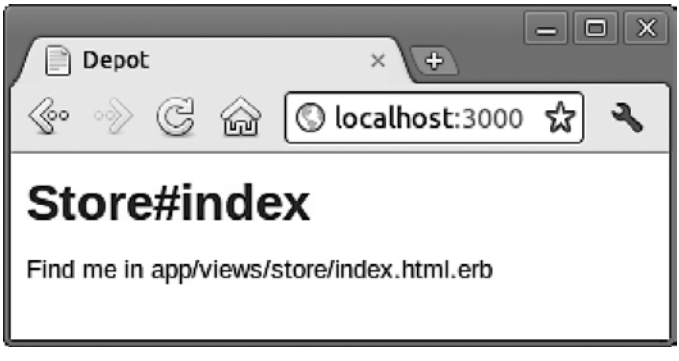


Рис. 8.1. Шаблон не найден

Вряд ли от этого мы станем богаче, но, по крайней мере, мы узнали, что все связано друг с другом правильно. Страница просто сообщает о том, где найти файл шаблона, который ее рисует.

Начнем с отображения простого перечня всех товаров, имеющихся в базе данных. Мы знаем, что в конечном счете понадобится более сложный подход с разбиением товаров на категории, к чему мы и будем стремиться.

Нам нужно извлечь перечень товаров из базы данных и сделать его доступным для кода представления, который и отобразит таблицу. Это означает, что мы должны внести изменения в метод `index()` в файле store_controller.rb. Мы хотим программировать на соответствующем уровне абстракции, поэтому просто предположим, что мы можем запросить у модели перечень продаваемых товаров:

rails40/depot_d/app/controllers/store_controller.rb

```
class StoreController < ApplicationController
    def index
▶       @products = Product.order(:title)
    end

end
```

Мы спросили у заказчика, в каком порядке он хочет увидеть товары, которые должны быть выведены в перечне, и приняли совместное решение: посмотреть,

что получится, если мы отобразим их в алфавитном порядке. Это делается путем добавления вызова метода `order(:title)` применительно к модели `Product`.

Теперь нужно создать шаблон представления. Для этого отредактируем файл index.html.erb в каталоге app/views/store. (Вспомним, что путевое имя к представлению выстраивается из имени контроллера `[store]` и имени действия `[index]`. Часть имени .html.erb означает, что это ERB-шаблон, генерирующий HTML.)

rails40/depot_d/app/views/store/index.html.erb

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

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
    <div class="entry">
        <%= image_tag(product.image_url) %>
        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>
        <div class="price_line">
            <span class="price"><%= product.price %></span>
        </div>
    </div>
<% end %>
```

Обратите внимание на использование метода `sanitize()` для описания (`description`). Он позволяет безопасно добавлять стилевое оформление HTML, повышающее интерес наших клиентов к описаниям. (Следует заметить, что данное решение открывает потенциальную дыру безопасности[1], но, поскольку описания готовятся людьми, работающими в самой компании, мы полагаем, что риск минимален.)

Здесь также используется вспомогательный метод `image_tag()`. Он генерирует HTML-тег `<img>`, используя свой аргумент в качестве источника изображения.

Затем мы добавим таблицу стилей, воспользовавшись тем фактом, что при выполнении шага A2 все было настроено таким образом, что в страницах, созданных `StoreController`, будет определен HTML-атрибут `class` по имени `store`:

rails40/depot_d/app/assets/stylesheets/store.css.scss

```
// Place all the styles related to the Store controller here.
// (Сюда помещаются все определения стилей для контроллера Store.)
// They will automatically be included in application.css.
// (Они будут автоматически включены в файл application.css.)
// You can use Sass (SCSS) here: http://sass-lang.com/
// (Что такое Sass (SCSS) можно узнать здесь: http://sass-lang.com/)

.store {
    h1 {
        margin: 0;
```

[1] http://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

```
        padding-bottom: 0.5em;
        font: 150% sans-serif;
        color: #226;
        border-bottom: 3px dotted #77d;
    }

    /* Запись в каталоге store */
    .entry {
        overflow: auto;
        margin-top: 1em;
        border-bottom: 1px dotted #77d;
        min-height: 100px;
        img {
            width: 80px;
            margin-right: 5px;
            margin-bottom: 5px;
            position: absolute;
        }
        h3 {
            font-size: 120%;
            font-family: sans-serif;
            margin-left: 100px;
            margin-top: 0;
            margin-bottom: 2px;
            color: #227;
        }
        p, div.price_line {
            margin-left: 100px;
            margin-top: 0.5em;
            margin-bottom: 0.8em;
        }
        .price {
            color: #44a;
            font-weight: bold;
            margin-right: 3em;
        }
    }
}
```

Щелчок на кнопке Обновить (Refresh) приведет к появлению картинки, показанной на рис. 8.2. Она все еще не отличается особой привлекательностью, чего-то в ней не хватает. Как раз во время наших размышлений мимо проходил заказчик, который выразил пожелание, чтобы на общедоступных страницах был привлекательный заголовок и боковая панель.

Если бы это все происходило на самом деле, нам, наверное, захотелось бы обратиться к дизайнеру. Так как всем нам попадалось на глаза слишком большое количество веб-сайтов, дизайн которых разрабатывался самими программистами, мы без малейших колебаний привлекли бы к этому делу кого-нибудь другого. Но наш веб-дизайнер в отпуске, черпает вдохновение на каком-нибудь пляже и вряд ли вернется до конца года, поэтому пока в этой графе поставим прочерк. А тем временем пришла пора сделать следующий шаг.

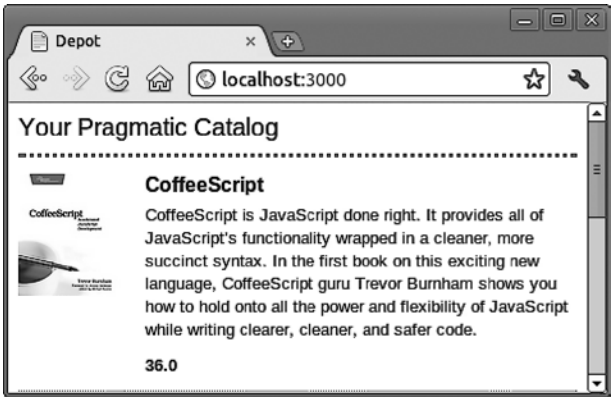


Рис. 8.2. Наша первая (черновая) страница каталога

8.2. Шаг B2: добавление макета страницы

Страницами типового веб-сайта часто используется один и тот же макет, поэтому разработчик должен создать стандартный шаблон, применяемый при размещении содержимого. Наша задача состоит в изменении данной страницы с целью добавления привлекательности каждой странице интернет-магазина.

До сих пор мы вносили лишь незначительные изменения в файл application.html.erb, в частности для добавления атрибута `class` в шаге A2. Поскольку этот файл является макетом, используемым всеми представлениями всех контроллеров, которые не предоставили свой собственный макет, мы можем изменить внешний вид всего сайта, отредактировав всего один файл. Это позволяет нам пока поставить в графе макета страницы прочерк, который можно будет убрать, как только наш дизайнер наконец-то вернется из отпуска на островах.

Давайте обновим этот файл, чтобы определить заголовок и боковую панель:

rails40/depot_e/app/views/layouts/application.html.erb

```
Строка 1    <!DOCTYPE html>
      -    <html>
      -    <head>
      -        <title>Pragprog Books Online Store</title>
      5        <%= stylesheet_link_tag "application", media: "all",
      -           "data-turbolinks-track" => true %>
      -        <%= javascript_include_tag "application",
      -           "data-turbolinks-track" => true %>
      -        <%= csrf_meta_tag %>
     10    </head>
      -    <body class="<%= controller.controller_name %>">
      -      <div id="banner">
      -        <%= image_tag("logo.png") %>
      -        <%= @page_title || "Pragmatic Bookshelf" %>
     15      </div>
      -      <div id="columns">
```

```
 -        <div id="side">
 -            <ul>
 -                <li><a href="http://www....">Home</a></li>
20                <li><a href="http://www..../faq">Questions</a></li>
 -                <li><a href="http://www..../news">News</a></li>
 -                <li><a href="http://www..../contact">Contact</a></li>
 -            </ul>
 -        </div>
25       <div id="main">
 -             <%= yield %>
 -         </div>
 -   </div>
 - </body>
30 </html>
```

Кроме обычного HTML-кода этот макет содержит 3 характерных для Rails элемента. В строке 5 используется вспомогательный метод Rails, предназначенный для генерирования тега `<link>`, ссылающегося на таблицу стилей нашего приложения, и указывается параметр для включения турбоссылок (turbolinks)[1], которые скрытно работают, ускоряя изменение внешнего вида страницы в вашем приложении. Точно так же в строке 7 генерируется тег `<link>`, ссылающийся на сценарии нашего приложения.

И наконец, в строке 9 устанавливаются все закулисные данные, необходимые для предотвращения атак путем подделки кросс-сайтовых запросов, которые выйдут на первый план, как только мы добавим формы в главе 12 «Задача Ж: оформление покупки».

В строке 14 для заголовка страницы установлено значение переменной экземпляра `@page_title`. Но настоящие чудеса начинаются в строке 26. При вызове действия `yield` Rails автоматически подставляет содержимое, специфическое для текущей страницы, то есть все, что сгенерировано представлением, вызванным данным запросом. В данном случае это будет страница каталога, сгенерированная с использованием файла index.html.erb.

Чтобы все это заработало, сначала переименуем файл application.css в application.css.scss. Если вы так и не решились воспользоваться Git, как предлагалось в разделе «Чем заняться на досуге» главы 6, то сейчас, возможно, для этого настал вполне благоприятный момент. Командой для переименования файла с помощью Git является `git mv`. После переименования этого файла с помощью либо Git, либо соответствующих команд операционной системы добавьте следующие строки:

rails40/depot_e/app/assets/stylesheets/application.css.scss

```
/*
* This is a manifest file that'll be compiled into application.css, which will
* include all the files listed below.
*
* Any CSS and SCSS file within this directory, lib/assets/stylesheets,
* vendor/assets/stylesheets, or vendor/assets/stylesheets of plugins, if any,
* can be referenced here using a relative path.
*
* You're free to add application-wide styles to this file and they'll appear
* at the top of the compiled file, but it's generally better to create a new
```

[1] https://github.com/rails/turbolinks

```
 * file per style scope.
 *
 *= require_self
 *= require_tree .
 */

#banner {
    background: #9c9;
    padding: 10px;
    border-bottom: 2px solid;
    font: small-caps 40px/40px "Times New Roman", serif;
    color: #282;
    text-align: center;

    img {
        float: left;
    }
}

#notice {
    color: #000 !important;
    border: 2px solid red;
    padding: 1em;
    margin-bottom: 2em;
    background-color: #f0f0f0;
    font: bold smaller sans-serif;
}

#columns {
    background: #141;

    #main {
        margin-left: 17em;
        padding: 1em;
        background: white;
    }

    #side {
        float: left;
        padding: 1em 2em;
        width: 13em;
        background: #141;

        ul {
            padding: 0;

            li {
                list-style: none;

                a {
                    color: #bfb;
                    font-size: small;
                }
            }
        }
    }
}
```

Как объяснено в комментариях, этот файл-манифест автоматически включит все таблицы стилей, доступные в данном каталоге и в любом из его подкаталогов. Это действие выполняется благодаря инструкции `require_tree`.

Вместо этого можно перечислить имена отдельных таблиц стилей, на которые нам нужна ссылка в `stylesheet_link_tag()`, но, поскольку мы находимся в макете всего приложения и поскольку этот макет уже настроен на загрузку всех таблиц стилей, пока оставим все без изменений.

Дизайн страницы состоит из трех основных областей экрана: баннера во всю ширину экрана, находящегося в верхней части, главной области в нижней правой части и боковой области слева от нее. Кроме того, здесь заданы некоторые заготовки указаний, касающихся внешнего вида. У каждой из областей есть отступы, поля, шрифты и цветовые настройки — в общем, все, что типично для таблицы CSS. Кроме того, баннер отцентрирован и содержит указание на то, что изображение должно быть помещено в его левой части. Внутри боковой области имеется обычное задание стилей для списка, а именно отключение полей и маркеров, а также указание другого шрифта и цвета.

И опять мы можем в полной мере воспользоваться Sass, такую возможность дает нам проведенное переименование файла. Например, в данном файле используется селектор `img`, вложенный в селектор `#banner`. Имеется также селектор, вложенный в селектор `#side`.

Щелчок на кнопке Обновить (Refresh) приведет к появлению картинки, показанной на рис. 8.3. Конечно, призов на конкурсе дизайна наш каталог не завоюет, но заказчик получит примерное представление о том, каким будет окончательный вид страницы.

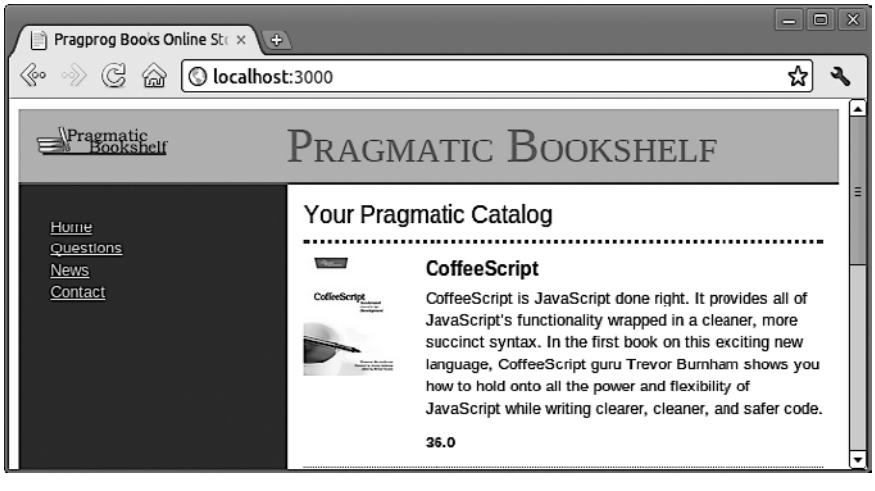


Рис. 8.3. Каталог с добавленным макетом

Рассматривая эту страницу, мы заметили небольшую проблему с отображением цены товара. База данных хранит цену в виде числа, но нам хотелось бы показать ее в долларах и центах. Цена 12.34 должна быть показана в виде $12.34, а цена 13 — в виде $13.00. Именно этим мы сейчас и займемся.

8.3. Шаг B3: использование помощника для форматирования цены

Ruby предоставляет функцию `sprintf()`, которая может использоваться для форматирования цен. Мы можем поместить логику, использующую эту функцию, непосредственно в представление. Например, можно воспользоваться следующим кодом:

```
<span class="price"><%= sprintf("$%0.02f", product.price) %></span>
```

Данный код будет работать, но он вставляет все сведения о текущем форматировании в представление. Если потребуется показать цену товаров в нескольких местах или чуть позже появится желание интернационализировать приложение, возникнут проблемы с его обслуживанием.

Давайте вместо этого воспользуемся вспомогательным методом для придания цене формата валюты. В Rails есть подходящий для этого встроенный помощник под названием `number_to_currency()`.

Этот помощник легко встраивается в представление, для этого нужно лишь в шаблоне index поменять этот код:

```
<span class="price"><%= product.price %></span>
```

на следующий:

rails40/depot_e/app/views/store/index.html.erb

```
<span class="price"><%= number_to_currency(product.price) %></span>
```

Я уверен, что после щелчка на кнопке Обновить (Refresh) появится красиво отформатированная цена, показанная на рис. 8.4.

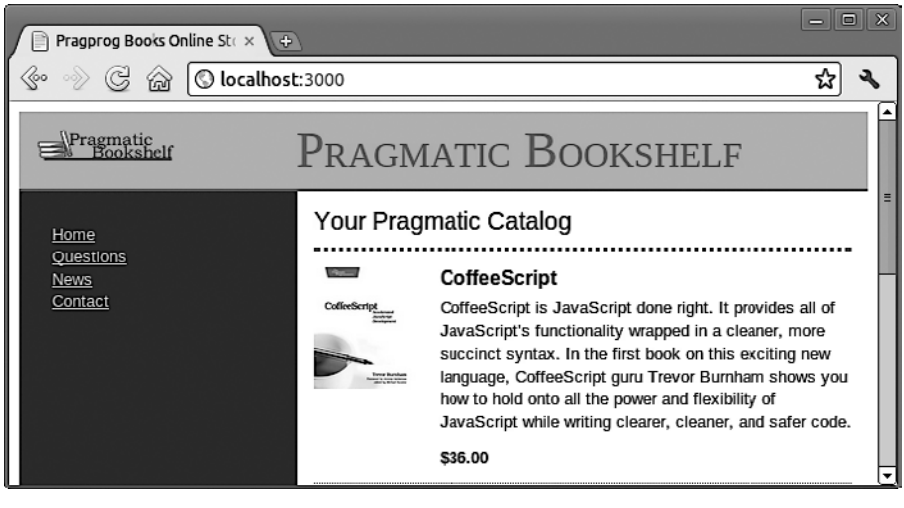


Рис. 8.4. Каталог с отформатированной ценой

Вроде бы все выглядит вполне прилично, но у нас возникает ноющее чувство, что нужно приступать к созданию тестов для всех новых функциональных возможностей, особенно после того, как мы уже приобрели опыт добавления логики к нашей модели.

8.4. Шаг B4: функциональное тестирование контроллеров

Теперь настал момент истины. Прежде чем сконцентрироваться на написании новых тестов, нам нужно определить, не было ли нами что-либо испорчено. Памятуя о нашем опыте работы после добавления к нашей модели логики проверки приемлемости данных, мы с чувством тревоги опять запустили наши тесты:

```
depot> rake test
```

На сей раз все прошло успешно. Несмотря на довольно многочисленные добавления, мы ничего не испортили. Можно облегченно вздохнуть, но наша работа на этом не закончена, нам все равно нужны тесты для всего, что только что было добавлено.

Ранее проведенное блочное тестирование моделей представлялось вполне достаточным. Мы вызывали метод и сравнивали возвращенные им данные с тем, что ожидали от него получить. Но теперь мы имеем дело с сервером, обрабатывающим запросы, и с пользователем, который просматривает ответы в браузере. Нам нужны *функциональные* тесты, проверяющие совместную работу модели, представления и контроллера. Но нам не о чем особо беспокоиться — среда Rails и здесь облегчила нашу задачу.

Сначала посмотрим на тот код, который Rails для нас сгенерировала:

rails40/depot_d/test/controllers/store_controller_test.rb

```
require 'test_helper'

class StoreControllerTest < ActionController::TestCase
    test "should get index" do
        get :index
        assert_response :success
    end

end
```

Тест «should get index» получает индекс и утверждает, что ожидается успешный ответ. Все это выглядит предельно просто. Для начала неплохо, но нам еще нужно проверить, содержит ли ответ наш макет, нашу информацию о товаре и наше числовое форматирование. Посмотрим, как это выглядит в виде программного кода:

rails40/depot_e/test/controllers/store_controller_test.rb

```
require 'test_helper'

class StoreControllerTest < ActionController::TestCase
```

```
    test "should get index" do
        get :index
        assert_response :success
▶       assert_select '#columns #side a', minimum: 4
▶       assert_select '#main .entry', 3
▶       assert_select 'h3', 'Programming Ruby 1.9'
▶       assert_select '.price', /\$[,\d]+\.\d\d/
    end

end
```

Четыре добавленные нами строки кода заглядывают *в* HTML-код, который возвращается при использовании нотации селекторов CSS. Следует напомнить, что селекторы, начинающиеся с символа решетки (`#`), соответствуют атрибутам `id`, селекторы, начинающиеся с точки (`.`), соответствуют атрибутам `class`, а селекторы вообще без префикса соответствуют названиям элементов.

Первый тест с селектором ищет элемент по имени `a`, который содержится в элементе с атрибутом `id`, имеющим значение `side`, который, в свою очередь, содержится внутри элемента с атрибутом `id`, имеющим значение `columns`. Этот тест проверяет, что у нас имеется минимум четыре таких элемента. Использование метода `assert_select()` является весьма эффективным средством, не правда ли?

В следующих трех строках проводится проверка отображения всех наших товаров. В первой строке проверяется наличие трех элементов, у которых есть атрибут `class` по имени `entry` в той части страницы, которая имеет атрибут `id` со значением `main`. В следующей строке проверяется наличие элемента `h3` с названием введенной нами ранее книги по Ruby. В третьей строке проверяется правильное форматирование цены. Эти утверждения основаны на тестовых данных, которые мы поместили в наши стенды:

rails40/depot_e/test/fixtures/products.yml

```
# Read about fixtures at http://api.rubyonrails.org/classes/Fixtures.html

one:
    title: MyString
    description: MyText
    image_url: MyString
    price: 9.99

two:
    title: MyString
    description: MyText
    image_url: MyString
    price: 9.99

ruby:
    title: Programming Ruby 1.9
    description:
       Ruby is the fastest growing and most exciting dynamic
       language out there. If you need to get working programs
       delivered fast, you should add Ruby to your toolbox.
    price: 49.50
    image_url: ruby.png
```

Если вы заметили, тип теста, выполняемого методом `assert_select()`, изменяется на основе типа его второго аргумента. Если это число, оно будет считаться количеством. Если это строка, она будет рассматриваться как ожидаемый результат. Еще один полезный тип теста вызывается регулярным выражением, использованным в нашем последнем утверждении. Мы проверяем наличие цены, имеющей значение, состоящее из знака доллара, за которым следует любое число (по крайней мере, одно), запятые или цифры, за которыми следует десятичная точка, после которой стоят две цифры.

Еще один заключительный нюанс: и проверка приемлемости данных, и функциональные тесты будут тестировать поведение только контроллера и не будут распространять обратное действие на объекты, уже существующие в базе данных или в стендах. В предыдущем примере два товара содержат одно и то же название. Такие данные не вызовут проблем и не будут обнаружены до тех пор, пока такие записи не будут изменены или сохранены.

Мы коснулись лишь нескольких возможностей метода `assert_select()`. Более подробную информацию можно найти в онлайн-документации[1]. Итак, мы получили массу проверок, добавив всего лишь несколько строк кода. Убедиться в работоспособности этого кода можно, перезапустив одни лишь функциональные тесты (ведь, по сути, только их мы и изменяли):

```
depot> rake test:controllers
```

Теперь у нас есть не только кое-что хорошо заметное на электронной витрине, но и тесты, гарантирующие, что все части приложения — модель, представление и контроллер — работают в тесном взаимодействии на выдачу желаемого результата. Звучит, конечно, внушительно, но добиться всего этого с Rails было совсем не трудно. По сути дела, в основном приложение состояло из HTML и CSS, и в нем было совсем немного программного кода или тестов. Перед тем как пойти дальше, давайте убедимся в том, что все это выстоит под натиском ожидаемых клиентов.

8.5. Шаг B5: Кэширование неполных результатов

Если все пойдет по плану, эта страница несомненно станет для сайта областью повышенного трафика. Чтобы отвечать на запросы, касающиеся данной страницы, нам понадобится извлекать каждый товар из базы данных и выводить его на экран. Но мы можем упростить ситуацию. В конечном счете каталог ведь не подвергается слишком частым изменениям, поэтому при каждом запросе не стоит начинать все сначала.

Поэтому мы можем продумать наши действия, и первое, что мы сделам, заключается в изменении настроек разработочной среды и включении кэширования.

[1] http://api.rubyonrails.org/classes/ActionDispatch/Assertions/SelectorAssertions.html

rails40/depot_e/config/environments/development.rb

```
config.action_controller.perform_caching = true
```

Как обычно, после изменения настроек нужно перезапустить сервер.

Затем нужно будет спланировать нашу атаку. По нашим прикидкам, требуется лишь заново вывести на экран все, что касается изменений товара, но даже тогда достаточно будет вывести только те товары, которые действительно подверглись изменениям. Сконцентрировавшись на первой части проблемы, нам нужно добавить код, возвращающий товар, который подвергся изменениям последним.

rails40/depot_e/app/models/product.rb

```
def self.latest
  Product.order(:updated_at).last
end
```

Далее мы пометили раздел нашего шаблона, который следует обновить при изменении каких-нибудь товаров, а внутри этого раздела пометили подраздел, необходимый нам для обновления любого конкретного товара, подвергшегося изменениям.

rails40/depot_e/app/views/store/index.html.erb

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

<h1>Your Pragmatic Catalog</h1>
▶ <% cache ['store', Product.latest] do %>
    <% @products.each do |product| %>
▶   <% cache ['entry', product] do %>
      <div class="entry">
        <%= image_tag(product.image_url) %>
        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>
        <div class="price_line">
         <span class="price"><%= number_to_currency(product.price) %></span>
        </div>
      </div>
▶   <% end %>
    <% end %>
▶ <% end %>
```

Кроме заключения в скобки раздела, мы обозначаем компоненты имени для каждой записи кэша. Мы решили назвать общую запись в кэше `store`, а отдельные записи в кэше — `entry`. Мы также связали товар с каждой записью, а именно самый последний товар, подвергшийся изменениям, с общей записью `store`, а отдельный товар мы выводим с помощью записи `entry`.

Разделы, заключенные в скобки, могут быть вложены друг в друга на произвольную глубину, и именно поэтому сообщество Rails решило назвать это матрешечным кэшированием (Russian doll caching)[1].

[1] http://37signals.com/svn/posts/3113-how-key-based-cache-expiration-works

И на этом наша работа будет завершена! Обо всем остальном, включая управление хранилищем и решение об аннулировании устаревших записей, позаботится Rails. Если вас действительно интересует данный вопрос, существует масса разнообразных элементов управления, которые можно включать, и путей выбора конкретной внешней памяти для кэширования. Сейчас вам беспокоиться обо всем этом не нужно, но может быть, все же стоит взять на заметку существование в RailsGuides[1] обзорной страницы Caching with Rails (кэширование с помощью Rails).

К сожалению, оценить по достоинству работоспособность данной технологии не представляется возможным. Если перейти на эту страницу, не будет видно никаких изменений, и это факт! Лучшее, что можно сделать — внести изменения в шаблон внутри блока, связанного с кэшированием, не обновляя никаких товаров, и проверить, что эти изменения ничем себя не проявляют, поскольку кэшированная версия страницы не подверглась обновлению.

После того как вы выяснили, что кэширование работает, выключите кэширование в режиме разработки приложения, чтобы предстоящие изменения шаблона могли тут же себя проявить.

rails40/depot_f/config/environments/development.rb

```
config.action_controller.perform_caching = false
```

Перезапустите сервер еще раз и убедитесь в том, что изменения, вносимые в шаблон, проявляются сразу же после их сохранения.

Наши достижения

Мы собрали воедино все, что составляет основу отображения каталога товаров интернет-магазина. При этом мы поэтапно выполнили следующие действия.

- ☑ Создали новый контроллер для работы с клиентом.
- ☑ Реализовали используемое по умолчанию действие `index()`.
- ☑ Добавили к контроллеру `Store` метод `default_scope()` для указания порядка перечисления элементов на веб-сайте.
- ☑ Реализовали представление (файл .html.erb) и содержащий его макет (еще один файл .html.erb).
- ☑ Воспользовались вспомогательным методом для нужного нам форматирования цен.
- ☑ Воспользовались каскадной таблицей стилей (CSS).
- ☑ Создали функциональные тесты для нашего контроллера.
- ☑ Реализовали фрагментарное кэширование для частей страницы.

Нужно все это проверить и перейти к выполнению следующей задачи, а именно к созданию корзины покупателя!

[1] http://guides.rubyonrails.org/caching_with_rails.html

Чем заняться на досуге

Попробуйте сделать все это без посторонней помощи.

- Добавить к боковой панели дату и время. Не нужно добиваться обновления их показаний, достаточно показать значения на момент отображения страницы.
- Поэкспериментировать с установкой различных аргументов вспомогательного метода `number_to_currency` и оценить влияние этих изменений на вывод вашего каталога товаров.
- Создать ряд функциональных тестов для ведения перечня товаров, воспользовавшись для этого методом `assert_select`. Тесты нужно будет поместить в файл test/controllers/products_controller_test.rb.
- Напоминаю, что завершение выполнения задания — хороший повод для сохранения своей работы с использованием системы Git. И если вы следовали всем предписаниям, на данный момент у вас уже есть все необходимое для этого действия. Выбор объектов резервного копирования будет рассмотрен при исследовании дополнительных функциональных возможностей системы в подразделе «Подготовка сервера для размещения вашего приложения», который находится в разделе 16.2.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlay Time.)

9 Задача Г: создание корзины покупателя

Основные темы:

- что такое сессии и как ими управлять;
- как добавить взаимодействие между моделями;
- как добавить кнопку для помещения товара в корзину.

Теперь, когда мы можем вывести каталог, содержащий все наши замечательные товары, было бы неплохо получить возможность их продавать. Заказчик с этим согласился, поэтому было принято совместное решение приступить к созданию корзины покупателя. Для этого мы собираемся применить ряд новых понятий, включая сессии, взаимодействие между моделями и добавление кнопки к представлению. Итак, давайте начнем.

9.1. Шаг Г1: обнаружение корзины

Просматривая наш онлайн-каталог, пользователи (как мы надеемся) выбирают покупаемые товары. По соглашению каждый выбранный товар будет добавлен к виртуальной корзине покупателя, которая поддерживается в нашем магазине. В какой-то момент у наших покупателей будет все необходимое, и они перейдут на нашем сайте к оформлению покупки, где они рассчитаются за товар, находящийся в их корзине.

Следовательно, наше приложение должно отслеживать все товары, добавленные покупателем в корзину. Для этого мы будем хранить корзину в базе данных и сохранять ее собственный идентификатор, `cart.id`, в сессии. При каждом

поступлении запроса мы можем восстановить идентичность из сессии и воспользоваться ею для обнаружения корзины в базе данных.

Давайте продолжим работу и создадим корзину:

```
depot> rails generate scaffold Cart
...
depot> rake db:migrate
== CreateCarts: migrating ====================================================
-- create_table(:carts)
-> 0.0012s
== CreateCarts: migrated (0.0014s) ===========================================
```

Для контроллера Rails делает текущую сессию похожей на хэш, поэтому мы сохраним идентификатор корзины в сессии путем индексирования его обозначением `:cart_id`.

rails40/depot_f/app/controllers/concerns/current_cart.rb

```
module CurrentCart
  extend ActiveSupport::Concern

  private

    def set_cart
      @cart = Cart.find(session[:cart_id])
    rescue ActiveRecord::RecordNotFound
      @cart = Cart.create
      session[:cart_id] = @cart.id
    end
end
```

Метод `set_cart()` начинается с получения `:cart_id` из объекта `session` с последующей попыткой обнаружения корзины, соответствующей данному идентификатору. Если такая запись корзины не найдется (а это произойдет, если идентификатор будет иметь значение, по каким-то причинам не подходящее, или значение `nil`), этот метод приступит к созданию нового объекта `Cart`, сохранит идентификатор созданной корзины в сессии, а затем вернет новую корзину.

Следует заметить, что мы поместили метод `set_cart()` в модуль `CurrentCart` и пометили его как закрытый (`private`). Подобная обработка позволяет использовать общий код (даже в столь малом объеме, как один метод!) между контроллерами и к тому же не позволяет Rails когда-либо делать его доступным в качестве действия контроллера.

9.2. Шаг Г2: связывание товаров с корзинами

Мы обратились к сессиям, потому что нам нужно где-то хранить нашу корзину покупателя. Более подробно сессии будут рассматриваться в подразделе «Сессии Rails» раздела 20.3, а сейчас мы продолжим реализацию корзины.

Не станем ничего усложнять. Корзина содержит набор товаров. На основе схемы исходных предположений о составе данных, показанной на рис. 5.3, в сочетании с краткой беседой с нашим заказчиком, мы можем приступить к генерации моделей Rails и к заполнению миграций с целью создания соответствующих таблиц:

```
depot> rails generate scaffold LineItem product:references cart:belongs_to
...
depot> rake db:migrate
== CreateLineItems: migrating ================================================
-- create_table(:line_items)
-> 0.0013s
== CreateLineItems: migrated (0.0014s) =======================================
```

Теперь у базы данных есть место для хранения связей между товарными позициями (`line item`), корзинами (`cart`) и товарами (`product`). А вот в приложении Rails эти взаимоотношения не фигурируют. Если посмотреть на сгенерированное определение класса `LineItem`, можно увидеть определения этих взаимосвязей.

rails40/depot_f/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :cart
end
```

На уровне модели разницы между простой ссылкой и связью «belongs to» не существует. Обе они реализуются с помощью метода `belongs_to()`. В модели LineItem два вызова belongs_to() сообщают Rails, что строки в таблице товарных позиций (`line_items`) являются дочерними по отношению к строкам в таблицах корзин (`carts`) и товаров (`products`). Пока не появятся соответствующие строки корзины и товара, строка товарной позиции существовать не сможет. Запомнить, где именно нужно постить инструкцию `belongs_to`, несложно: если у таблицы есть внешние ключи, соответствующая модель должна иметь для каждого из них инструкцию `belongs_to`.

А что же все-таки все эти разные инструкции делают? По сути, они добавляют к объектам модели возможности перемещения. Поскольку мы добавили к `LineItem` инструкцию `belongs_to`, теперь мы можем извлечь товар (`Product`), соответствующий позиции, и отобразить название книги:

```
li = LineItem.find(...)
puts "Эта товарная позиция относится к #{li.product.title}"
```

Чтобы иметь возможность прослеживать эти взаимоотношения в обоих направлениях, нужно к файлам нашей модели добавить несколько объявлений, указывающих на их обратные связи.

Откройте файл cart.rb в каталоге app/models и добавьте вызов метода has_many():

rails40/depot_f/app/models/cart.rb

```
class Cart < ActiveRecord::Base
    has_many :line_items, dependent: :destroy
end
```

Часть инструкции `has_many :line_items` говорит сама за себя: в корзине потенциально имеется множество (has many) связанных с ней товарных позиций. Они привязаны к корзине, потому что каждая товарная позиция содержит ссылку на идентификатор этой корзины. Часть инструкции `dependent: :destroy` показывает, что существование товарной позиции зависит от существования корзины. Если мы уничтожим корзину, удалив ее из базы данных, нам нужно, чтобы Rails также уничтожила все товарные позиции, связанные с этой корзиной.

Теперь, когда объявлено, что корзина (`Cart`) может иметь множество товарных позиций, мы можем сослаться на них (как на коллекцию) из объекта корзины:

```
cart = Cart.find(...)
puts "Количество товарных позиций в этой корзине: #{cart.line_items.count}"
```

А теперь для завершения общей картины нам нужно добавить инструкцию `has_many` к нашей модели товаров (`Product`). Ведь если имеется множество корзин, на каждый товар может ссылаться множество товарных позиций. На этот раз для предотвращения удаления товаров, на которые ссылаются товарные позиции, мы воспользуемся проверочным кодом.

rails40/depot_f/app/models/product.rb

```
class Product < ActiveRecord::Base
    has_many :line_items

    before_destroy :ensure_not_referenced_by_any_line_item

    #...

    private

    # убеждаемся в отсутствии товарных позиций, ссылающихся на данный товар
    def ensure_not_referenced_by_any_line_item
        if line_items.empty?
            return true
        else
            errors.add(:base, 'существуют товарные позиции')
            return false
        end
    end
end
```

Здесь объявляется, что у товара много товарных позиций, и определяется подключаемый метод по имени `ensure_not_referenced_by_any_line_item()`. Подключаемым называется такой метод, который Rails вызывает автоматически в определенный момент жизни объекта. В данном случае метод будет вызван перед тем, как Rails попытается удалить строку в базе данных. Если подключаемый метод возвращает `false`, строка не будет удалена.

Заметьте, что у нас есть непосредственный доступ к объекту `errors`. Это то же самое место, где `validates()` хранит сообщения об ошибках. Ошибки могут быть связаны с отдельными свойствами, но в данном случае мы связали ошибку с базовым объектом.

Более подробная информация о взаимодействиях внутри модели будет дана в подразделе «Определение отношений в моделях» раздела 19.2.

9.3. Шаг Г3: добавление кнопки

После всего, что уже сделано, настало время добавить к каждому представленному товару кнопку Добавить в корзину (Add to Cart).

Для этого не нужно создавать новый контроллер или даже новое действие. Если посмотреть на действия, предоставленные генератором временной платформы, то среди них можно найти `index()`, `show()`, `new()`, `edit()`, `create()`, `update()` и `destroy()`. Нашей операции соответствует действие `create()`. (Может показаться, что действие `new()` является аналогичным, но оно используется для получения формы, применяемой, чтобы запросить ввод для последующего действия `create()`.)

За принятием этого решения следуют все остальные решения. Что именно мы создаем? Конечно же, не корзину и даже не товар. Мы создаем товарную позицию `LineItem`. Посмотрев на комментарий, связанный с методом `create()` в файле app/controllers/line_items_controller.rb, вы увидите, что в результате этого выбора также определяется используемый URL-адрес (/line_items) и HTTP-метод (`POST`).

Этот выбор даже предлагает правильный элемент управления пользовательского интерфейса. Когда ранее мы добавляли ссылки, мы пользовались методом `link_to()`, но ссылки по умолчанию настроены на использование HTTP-метода GET. А нам нужно использовать метод POST, потому что на сей раз мы будем добавлять кнопку, а это значит, что мы будем использовать метод `button_to()`.

Мы можем подключить кнопку к товарной позиции, указав URL-адрес, но мы и в этот раз позволим Rails позаботиться об этом за нас, просто добавив `_path` к имени контроллера. В данном случае мы воспользуемся `line_items_path`.

Но при этом возникает проблема: как метод `line_items_path` узнает, *какой* товар добавлять к нашей корзине? Нам нужно передать ему идентификатор товара, соответствующего кнопке. Делается это довольно просто — нужно лишь к вызову `line_items_path()`добавить `:product_id`. Можно даже передать сам экземпляр товара — Rails знает, что при подобных обстоятельствах нужно извлечь из записи идентификатор.

В конечном итоге *одна* строка, которую нужно добавить к нашему файлу index.html.erb, имеет следующий вид:

rails40/depot_f/app/views/store/index.html.erb

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

<h1>Your Pragmatic Catalog</h1>
```

```
<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
        <%= image_tag(product.image_url) %>

        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>

        <div class="price_line">

            <span class="price"><%= number_to_currency(product.price) %></span>
            <%= button_to 'Add to Cart', line_items_path(product_id: product) %>
        </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

Здесь возникает еще один вопрос форматирования. Метод `button_to` создает HTML-тег `<form>`, и эта форма содержит HTML-тег `<div>`. Оба они формируют обычные блоковые элементы, которые должны появляться с новой строки. Мы хотим, чтобы они появлялись сразу же за ценой, поэтому, чтобы их встроить в ту же строку, нужно добавить немного CSS-волшебства:

rails40/depot_f/app/assets/stylesheets/store.css.scss

```
p, div.price_line {
    margin-left: 100px;
    margin-top: 0.5em;
    margin-bottom: 0.8em;

    form, div {
        display: inline;
    }
}
```

Теперь наша индексная страница выглядит так, как показано на рис. 9.1. Но перед тем как нажать эту кнопку, нам нужно изменить метод `create()` в контроллере товарных позиций, чтобы в качестве аргументов формы ожидался идентификатор товара. И здесь мы начинаем понимать всю важность поля идентификатора (`id`) в наших моделях. Rails идентифицирует объекты модели (и соответствующие строки базы данных) по их полям `id`. Если идентификатор передать методу `create()`, добавляемый товар получит уникальную идентификацию.

Почему используется метод `create()`? Для ссылки по умолчанию используется HTTP-метод GET, для кнопки — HTTP-метод POST, а Rails использует эти соглашения для определения вызываемого метода. Чтобы увидеть другие соглашения, взгляните на комментарии внутри файла app/controllers/line_items_controller.rb. Эти соглашения будут широко использоваться и внутри приложения Depot.

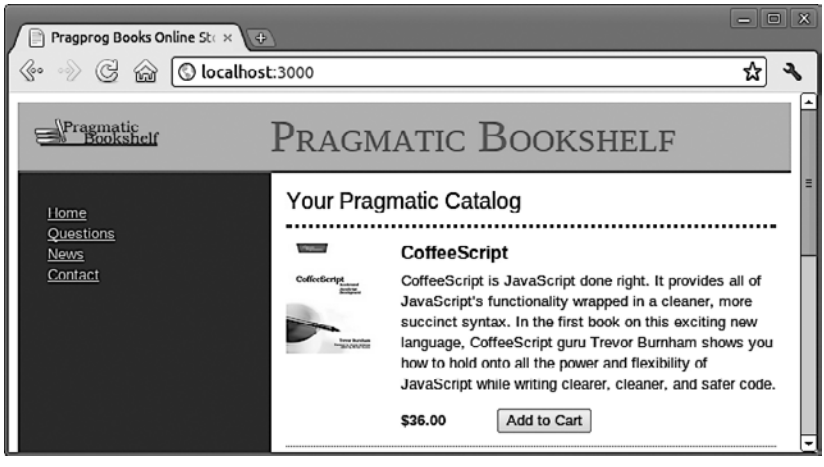


Рис. 9.1. Теперь на странице есть кнопка Добавить в корзину (Add to Cart)

Теперь давайте внесем изменения в `LineItemsController`, чтобы обнаружить корзину покупателя для текущей сессии (если еще нет ни одной корзины, корзину нужно добавить), добавить к этой корзине выбранный товар и отобразить содержимое корзины.

Чтобы найти (или создать) в сессии корзину, воспользуемся созданной на шаге Г1 структурой концерн (concern) в виде текущей корзины — `CurrentCart`.

rails40/depot_f/app/controllers/line_items_controller.rb

```
class LineItemsController < ApplicationController
▶  include CurrentCart
▶  before_action :set_cart, only: [:create]
   before_action :set_line_item, only: [:show, :edit, :update, :destroy]

   # GET /line_items
   #...
end
```

Мы включили модуль `CurrentCart` и объявили о том, что метод `set_cart()` должен быть вызван до действия `create()`. Более глубоко обратные вызовы действий рассматриваются в разделе «Обратные вызовы» главы 20, а сейчас нам достаточно знать, что Rails предоставляет возможность связывать вместе методы, которые вызываются до, после или даже и до и после действий контроллера.

Фактически, можно увидеть, что сгенерированный контроллер уже использует эту возможность для установки значения экземпляра переменной `@line_item` перед вызовами действия `show()`, `edit()`, `update()` или `destroy()`.

Теперь, зная о том, что переменной `@cart` присвоено значение текущей корзины, нам нужно лишь изменить несколько строк кода в методе `create()` в файле app/controllers/line_items_controller.rb[1], чтобы создать саму товарную позицию.

[1] Некоторые строки пришлось разрывать, чтобы они поместились на странице.

rails40/depot_f/app/controllers/line_items_controller.rb

```
def create
    product = Product.find(params[:product_id])
    @line_item = @cart.line_items.build(product: product)

    respond_to do |format|

        if @line_item.save
            format.html { redirect_to @line_item.cart,
                notice: 'Line item was successfully created.' }

            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Мы используем объект `params` для получения из запроса аргумента `:product_id`. Объект `params` играет в приложениях Rails довольно важную роль. Он содержит все аргументы, переданные в запросе браузера. Мы сохраняем результат в локальной переменной, потому что его не нужно делать доступным представлению.

Затем мы передаем найденный товар в `@cart.line_items.build`. Благодаря этому создается новая взаимосвязь товарной позиции между объектом `@cart` и товаром. Взаимосвязь можно создать с любого конца, и Rails позаботится об установке подключений с обеих сторон.

Получившуюся товарную позицию мы сохраняем в переменной экземпляра по имени `@line_item`.

Остальной код этого метода берет на себя обработку ошибок, более подробное рассмотрение которой будет в разделе 10.2 «Шаг Д2: обработка ошибок», и обработку JSON-запросов. Но сейчас нам нужно всего лишь изменить еще одну вещь: когда создана товарная позиция, нужно перенаправить вас на корзину, вместо того чтобы вернуть к товарной позиции. Поскольку объект товарной позиции знает, как найти объект корзины, нам нужно лишь добавить `.cart` к вызову метода.

Поскольку мы изменили функцию нашего контроллера, мы знаем, что нужно будет изменить соответствующий функциональный тест. Мы должны передать вызову идентификатор товара для создания и изменения того, что мы ожидаем в качестве цели переадресации. Это будет сделано путем обновления файла test/controllers/line_items_controller_test.rb.

rails40/depot_g/test/controllers/line_items_controller_test.rb

```
test "should create line_item" do
    assert_difference('LineItem.count') do
        post :create, product_id: products(:ruby).id
    end
    assert_redirected_to cart_path(assigns(:line_item).cart)
end
```

До сих пор мы еще ничего не говорили о методе `assigns`, но мы уже им пользуемся, поскольку он был сгенерирован автоматически командой `scaffold`. Этот метод дает нам доступ к переменной экземпляра, которая была определена (или могла быть определена) действиями контроллера для использования в представлении.

А теперь мы перезапустим этот набор тестов:

```
depot> rake test test/controllers/line_items_controller_test.rb
```

Будучи уверенными, что код работает в соответствии со своим предназначением, мы пробуем воспользоваться кнопками Добавить в корзину (Add to Cart) в нашем браузере.

И перед нами предстает картина, показанная на рис. 9.2.

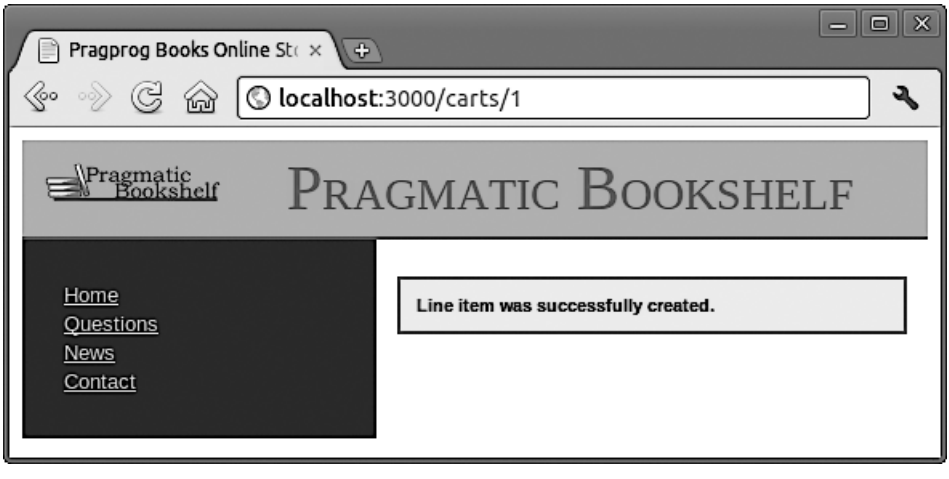


Рис. 9.2. Подтверждение обработки запроса

В восторг она, конечно, не приводит. Несмотря на то что мы создали для корзины временную платформу, при ее создании мы не предоставили никаких свойств, поэтому представлению нечего показывать. Теперь давайте создадим обычный шаблон (разукрасим его чуть позже):

rails40/depot_f/app/views/carts/show.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>
<h2>Your Pragmatic Cart</h2>
<ul>
    <% @cart.line_items.each do |item| %>
        <li><%= item.product.title %></li>
    <% end %>
</ul>
```

Итак, когда все это скомпоновано, давайте вернемся назад и еще раз щелкнем на кнопке Обновить (Refresh) своего браузера, чтобы увидеть, как выглядит наше простое представление (рис. 9.3).

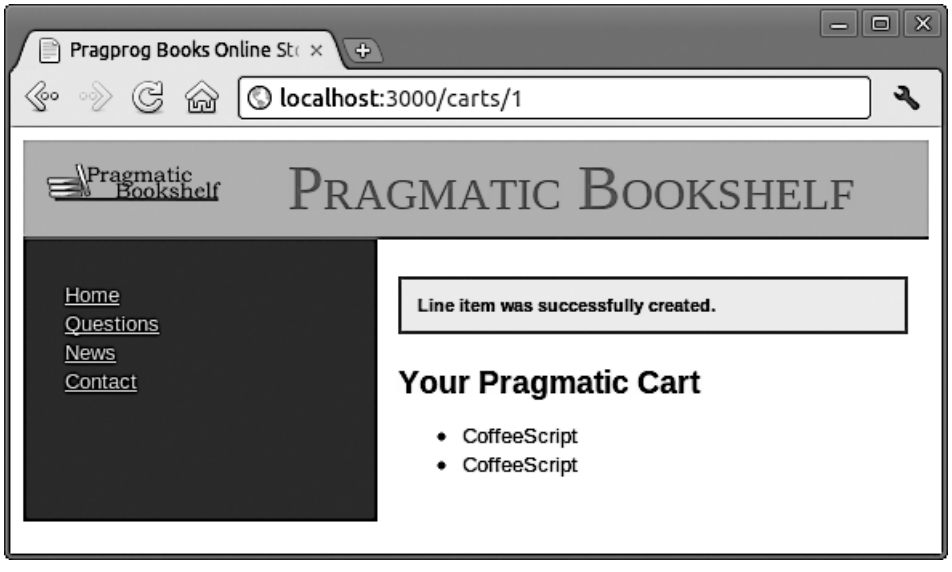


Рис. 9.3. Корзина с показанной новой записью

Вернитесь по адресу http://localhost:3000/ к странице главного каталога и добавьте к корзине другой товар. Вы увидите в вашей корзине две исходные записи плюс наш новый элемент. Похоже, что сессия работает. Настало время продемонстрировать все это заказчику, поэтому мы его позвали и с гордостью продемонстрировали нашу суровую новую корзину. Но он смутил нас своей реакцией, издав цокающий звук, предвещающий замеченное им наше явное упущение.

Заказчик объяснил, что в настоящей корзине покупателям не показывают двумя отдельными строками один и тот же товар. Вместо этого строка товара показывается всего один раз с указанием количества, которое в данном случае равно двум. Похоже, наметилась наша следующая задача.

Наши достижения

У нас был насыщенный событиями, продуктивный день. К магазину была добавлена корзина покупателя, и в процессе работы мы углубились в некоторые необходимые свойства Rails.

- ☑ В одном запросе мы создали объект корзины `Cart` и получили возможность успешно обнаружить эту самую корзину в последующих запросах, используя объект сессии.

- ☑ Мы добавили закрытый метод и поместили его в структуру концерн (concern), сделав доступным для всех наших контроллеров.
- ☑ Мы создали взаимосвязи между корзинами и товарными позициями и взаимосвязи между товарными позициями и товарами и получили возможность перемещения с использованием этих взаимосвязей.
- ☑ Мы добавили кнопку для переноса товара в корзину, которая заставляет создать новую товарную позицию.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- ❍ Добавьте новую переменную к сессии для записи количества обращений пользователя к действию `index` контроллера `store`. Учтите, что при первом обращении к данной странице вашего счетчика в сессии еще не будет. Этот факт можно проверить с помощью следующего кода:

```
if session[:counter].nil?
...
```

  Если переменная сессии отсутствует, ее нужно инициализировать. После этого появится возможность увеличивать ее значение.

- ❍ Передайте этот счетчик своему шаблону и отобразите его значение в верхней части страницы каталога. Подсказка: при формировании отображаемого сообщения можно воспользоваться вспомогательным методом `pluralize`, который переводит слово в множественную форму (определение метода дано в разделе 21.5).
- ❍ Сбросьте счетчик в нуль, когда пользователь что-нибудь добавит в свою корзину.
- ❍ Внесите в шаблон изменения, приводящие к отображению счетчика только в том случае, если он имеет значение больше пяти.

(Подсказки можно найти по адресу: http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

Задача Д: усовершенствованная корзина

10

Основные темы:

- изменение схемы и существующих данных;
- диагностика и обработка ошибок;
- флэш;
- ведение журнала.

Примитивная функциональность реализованной нами корзины предполагает большие доработки. Для начала нужно распознать добавление клиентами в корзину одного и того же товара. После этого придется также убедиться в том, что корзина может справляться с ошибочными ситуациями и доносить возникшие в процессе работы проблемы до клиента или до системного администратора, в зависимости от того, что больше подходит.

10.1. Шаг Д1: создание усовершенствованной корзины

Привязка счетчика к каждому товару в нашей корзине потребует от нас внести изменения в таблицу `line_items`. Ранее мы уже использовали миграции, например, для обновления схемы базы данных в разделе 6.1. Несмотря на то что это использование было частью создания исходной временной платформы для модели, базовые подходы остаются прежними.

```
depot> rails generate migration add_quantity_to_line_items quantity:integer
```

Исходя из имени миграции, Rails может сообщить, что вы добавляете один или несколько столбцов к таблице `line_items`, и может взять имена и типы данных для каждого столбца из последнего аргумента. Rails ищет соответствие двум шаблонам: `add_XXX_to_TABLE` и `remove_XXX_from_TABLE`, где значение `XXX` игнорируется. Значение имеет список имен столбцов и типов данных, появляющийся после имени миграции.

Rails не может только сообщить, какое приемлемое значение по умолчанию может быть выбрано для столбца. Во многих случаях подойдет нулевое значение, но давайте перед применением миграции внесем в нее изменение и сделаем это значение для существующих корзин равным 1:

rails40/depot_g/db/migrate/20121130000004_add_quantity_to_line_items.rb

```
class AddQuantityToLineItems < ActiveRecord::Migration
    def change
▶       add_column :line_items, :quantity, :integer, default: 1
    end
end
```

После внесения изменения запустим миграцию:

```
depot> rake db:migrate
```

Теперь в нашей корзине `Cart` нужен толковый метод `add_product()`, который будет проверять, имеется ли добавляемый товар в нашем списке позиций, и, если такой товар уже есть, будет повышать его количество, а если его не было, будет создавать новый объект `LineItem`:

rails40/depot_g/app/models/cart.rb

```
def add_product(product_id)
    current_item = line_items.find_by(product_id: product_id)
    if current_item
        current_item.quantity += 1
    else
        current_item = line_items.build(product_id: product_id)
    end
    current_item
end
```

Метод `find_by()` является модернизированной версией метода `where()`, который вместо возвращения массива результатов возвращает либо существующую товарную позицию `LineItem`, либо `nil`.

Чтобы воспользоваться этим методом, нам понадобится также внести изменения в контроллер товарных позиций:

Rails40/depot_g/app/controllers/line_items_controller.rb

```
def create
    product = Product.find(params[:product_id])
▶   @line_item = @cart.add_product(product.id)
    respond_to do |format|
        if @line_item.save
```

```
            format.html { redirect_to @line_item.cart,
                notice: 'Line item was successfully created.' }
            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
            status: :unprocessable_entity }
        end
    end
end
```

И еще одно небольшое изменение в представлении для использования этой новой информации:

Rails40/depot_g/app/views/carts/show.html.erb

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

<h2>Your Pragmatic Cart</h2>
<ul>
    <% @cart.line_items.each do |item| %>
        <li><%= item.quantity %> &times; <%= item.product.title %></li>
    <% end %>
</ul>
```

Когда все расставлено по местам, можно вернуться на страницу магазина и щелкнуть на кнопке Добавить в корзину (Add to Cart) напротив товара, уже имеющегося в корзине. Скорее всего, мы увидим смесь из единичных товаров, перечисленных отдельно, и один товар, напротив которого в списке будет указано количество, равное двум. Причина в том, что мы добавили количество, равное единице, к существующим столбцам, вместо того чтобы свернуть несколько строк в одну там, где это возможно. Стало быть, теперь нам нужно осуществить миграцию данных.

Начнем с создания миграции:

```
depot> rails generate migration combine_items_in_cart
```

На этот раз Rails не в состоянии предположить, что мы пытаемся сделать, поэтому мы не можем положиться на сгенерированный метод `change()`. Вместо этого нам нужно заменить этот метод отдельными методами `up()` и `down()`. Сначала создадим метод `up()`:

rails40/depot_g/db/migrate/20121130000005_combine_items_in_cart.rb

```
def up
    # замена нескольких записей для одного и того же товара в корзине одной записью
    Cart.all.each do |cart|
        # подсчет количества каждого товара в корзине
        sums = cart.line_items.group(:product_id).sum(:quantity)

        sums.each do |product_id, quantity|
            if quantity > 1
```

```
                # удаление отдельных записей
                cart.line_items.where(product_id: product_id).delete_all

                # замена одной записью
                item = cart.line_items.build(product_id: product_id)
                item.quantity = quantity
                item.save!
            end
        end
    end
end
```

Это, несомненно, самый объемный код из всех ранее попадавшихся. Давайте разберем его по частям.

- Сначала задается перебор содержимого каждой корзины.
- Для каждой корзины мы получаем сумму значений полей количества (`quantity`) для каждой из товарных позиций, связанных с этой корзиной, сгруппированных по полю идентификатора товара `product_id`. Получающиеся суммы станут списком упорядоченных пар значений полей `product_id` и количества `quantity`.
- Мы перебираем эти суммы, извлекая из каждой `product_id` и `quantity`.
- В тех случаях, когда количество превышает единицу, мы будем удалять все отдельные товарные позиции, связанные с этой корзиной и этим товаром, и заменять их одной товарной позицией с правильным количеством.

Заметьте, как просто и элегантно Rails позволила вам выразить этот алгоритм.

Имея данный код, мы применим эту миграцию тем же способом, что и любые другие миграции:

```
depot> rake db:migrate
```

Мы можем тут же увидеть результаты, взглянув на корзину, которая будет выглядеть, как показано на рис. 10.1. Хотя у нас есть повод для удовлетворения, работа еще не закончена.

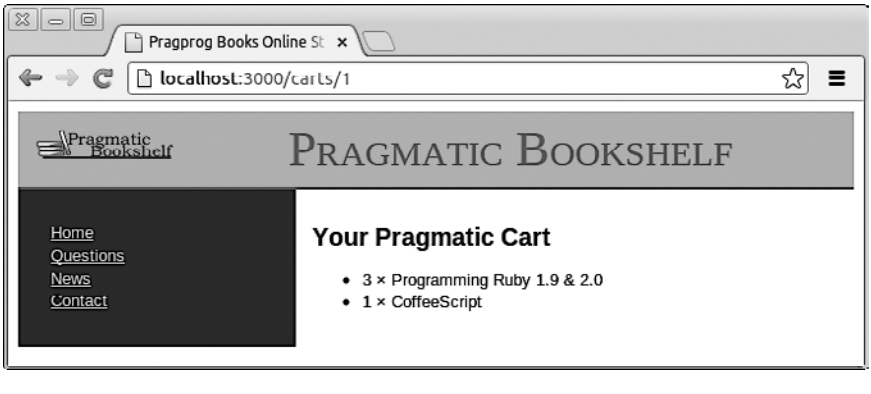


Рис. 10.1. Три товарных позиции LineItem объединены в одну

Важным принципом миграций является обратимость каждого шага, поэтому мы реализуем так же и метод `down()`. Этот метод находит товарные позиции со значением поля количества больше единицы; добавляет новые товарные позиции для этой корзины и товара, каждая из которых имеет значение поля количества, равное единице; и в завершение удаляет исходную товарную позицию. Все это делается с помощью следующего кода:

rails40/depot_g/db/migrate/20121130000005_combine_items_in_cart.rb

```
def down
    # разбиение записей с quantity>1 на несколько записей
    LineItem.where("quantity>1").each do |line_item|
        # add individual items
        line_item.quantity.times do
            LineItem.create cart_id: line_item.cart_id,
                product_id: line_item.product_id, quantity: 1
        end

        # удаление исходной записи
        line_item.destroy
    end
end
```

Теперь мы можем просто отменить нашу миграцию с помощью всего лишь одной команды:

```
depot> rake db:rollback
```

Rails предоставляет удобную `rake`-задачу, позволяющую проверить состояние ваших миграций.

```
depot> rake db:migrate:status
database: /home/rubys/work/depot/db/development.sqlite3
Status    Migration ID     Migration Name
--------------------------------------------------
  up      20130407000001   Create products
  up      20130407000002   Create carts
  up      20130407000003   Create line items
  up      20130407000004   Add quantity to line items
  down    20130407000005   Combine items in cart
```

В данный момент можно изменить и заново применить миграцию или даже удалить ее целиком. Результаты отката можно проверить, переместив миграцию в другой каталог и посмотрев на корзину (рис. 10.2).

Вернув файл миграции обратно и применив эту же миграцию еще раз (с помощью команды `rake db:migrate`), мы получим корзину, которая обслуживает счетчик для каждого содержащегося в ней товара, и мы получим представление, отображающее этот счетчик.

Обрадованные наличием презентабельного результата, мы позвали заказчика и показали ему результат нашей утренней работы. Он смог увидеть, как сайт стал превращаться в единое целое, и был доволен. Но он выразил беспокойство тем, что узнал из недавно прочитанной в прессе статьи о способах ежедневных атак

и взломов коммерческих сайтов. Он прочитал, что одним из способов атак является выдача веб-приложениям запросов с неверными аргументами в надежде на вскрытие ошибок и изъянов безопасности. Он заметил, что ссылка на корзину выглядит как carts/*nnn*, где *nnn* — наш внутренний идентификатор корзины. В роли злоумышленника он вручную набрал такой запрос в браузере, указав вместо идентификатора корзины слово *wibble*. Он был разочарован, когда наше приложение вывело страницу, показанную на рис. 10.3. Это считается проявлением крайнего непрофессионализма. Поэтому наш следующий шаг мы затратим на то, чтобы сделать приложение более стойким.

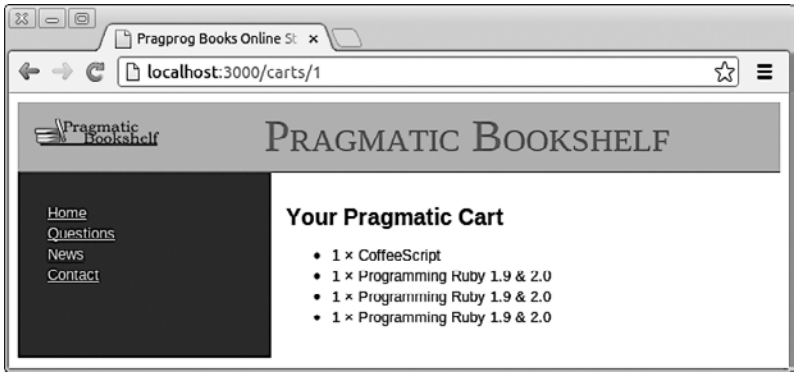


Рис. 10.2. Товарные позиции опять представлены обособленно

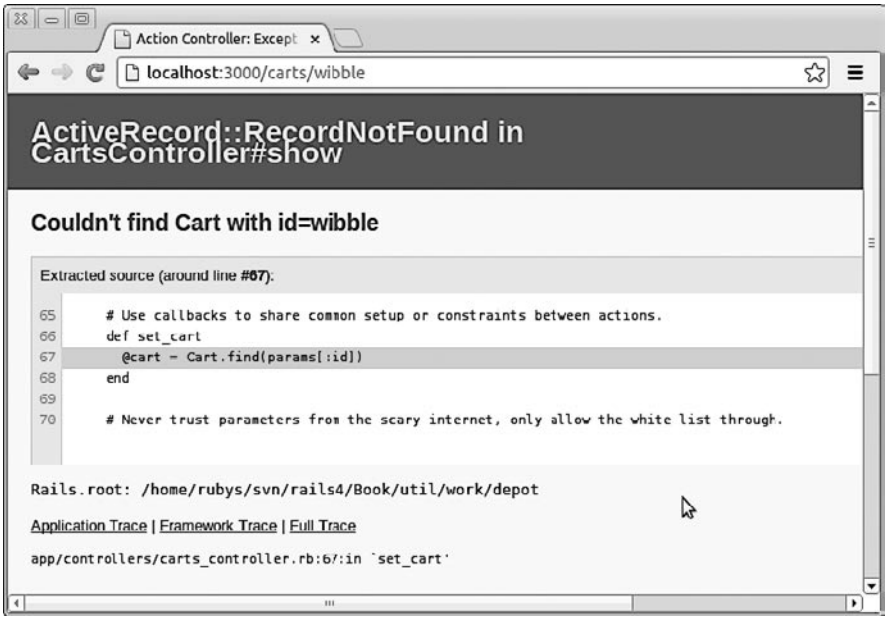


Рис. 10.3. Наше приложение раскрывает свое внутреннее устройство

10.2. Шаг Д2: обработка ошибок

Если посмотреть на страницу, показанную на рис. 10.3, станет понятно, что наше приложение выдало исключение в строке 67 контроллера корзин[1], то есть в строке:

```
@cart = Cart.find(params[:id])
```

Если корзина не может быть найдена, Active Record выдает исключение `RecordNotFound`, которое нам, очевидно, нужно обработать. Возникает вопрос, как это сделать?

Мы можем просто молча его проигнорировать. Может быть, с точки зрения безопасности это будет наилучшим выходом из ситуации, поскольку потенциальный злоумышленник не получает при этом никакой нужной для себя информации. Но это также означает, что при наличии в нашем коде ошибки, из-за которой генерируются неверные идентификаторы корзин, для внешнего мира наше приложение перестанет на что-либо реагировать, и о наличии ошибки так никто и не узнает.

Но мы поступим по-другому и при выдаче исключения предпримем два действия. Во-первых, мы зарегистрируем этот факт во внутреннем журнале регистрации, использовав имеющиеся в Rails соответствующие возможности[2]. Во-вторых, мы повторно выведем страницу каталога с коротким сообщением для пользователей (что-нибудь вроде «Несуществующая корзина»), чтобы они смогли продолжать пользоваться нашим сайтом.

В Rails имеется удобный способ обработки ошибок и уведомлений об их возникновении. Он заключается в определении структуры под названием *флэш*. Это область памяти (что-то близкое к хэшу), где вы можете хранить материал при обработке запроса. Содержимое флэш-области перед операцией автоматического удаления доступно следующему запросу в сессии. Обычно флэш используется для сбора сообщений об ошибках. К примеру, когда наш метод `show()` обнаружит, что ему был передан негодный идентификатор корзины, он может сохранить сообщение об ошибке во флэш-области и перенаправить ход выполнения программы на действие `index`, чтобы снова показать каталог. Представление, следующее за выполнением действия `index`, может извлечь сообщение об ошибке и показать его в верхней части страницы каталога. Информация флэш-памяти доступна из представления при использовании метода `flash`.

А почему мы не можем хранить информацию об ошибках в любой существующей переменной экземпляра? Не забывайте, что после того, как сообщение о неправильном идентификаторе отослано приложением браузеру, тот посылает в ответ приложению новый запрос. За то время, пока он будет идти, приложение не будет стоять на месте, и все переменные экземпляра, существовавшие во время предыдущих запросов, канут в Лету. А флэш-данные хранятся в сессии для того, чтобы быть доступными между запросами.

---

[1] У вас может быть другой номер строки. В наших исходных файлах форматирование иногда обусловлено размерами книжной страницы.

[2] http://guides.rubyonrails.org/debugging_rails_applications.html#the-logger

Вооружившись познаниями о флэш-данных, мы теперь можем создать метод `invalid_cart()`, предназначенный для выдачи отчета о наличии данной проблемы:

rails40/depot_h/app/controllers/carts_controller.rb

```
class CartsController < ApplicationController
    before_action :set_cart, only: [:show, :edit, :update, :destroy]
▶   rescue_from ActiveRecord::RecordNotFound, with: :invalid_cart
    # GET /carts
    # ...
    private
    # ...
▶   def invalid_cart
▶   logger.error "Attempt to access invalid cart #{params[:id]}"
▶   redirect_to store_url, notice: 'Invalid cart'
▶   end
end
```

Оператор `rescue_from` перехватывает исключение, выданное `Cart.find()`. А в обработчике исключения мы делаем следующее.

- Используем Rails-регистратор для записи ошибки. Свойство `logger` есть у каждого контроллера. В данном случае мы используем его для записи сообщения в регистрационный уровень `error`.
- Переадресовываем запрос на отображение каталога, используя метод `redirect_to()`. Аргумент `:notice` определяет сообщение, которое будет сохранено во флэш-области в качестве уведомления. А зачем нам переадресация, если можно просто отобразить каталог? Если мы применим переадресацию, в браузере будет выставлен URL-адрес магазина, а не http://.../cart/wibble. Таким образом, мы выставляем напоказ меньшую часть нашего приложения, а также оберегаем пользователя от повторного инициирования ошибки в случае перезагрузки страницы.

Имея такой код, мы можем перезапустить проблемный запрос нашего заказчика. На этот раз при вводе следующего URL-адреса:

```
http://localhost:3000/carts/wibble
```

мы уже не видим в браузере целую группу сообщений об ошибках. Вместо этого отображается страница каталога. Если посмотреть на последние записи регистрационного журнала (development.log в каталоге log), мы увидим наше сообщение:

```
    Started GET "/carts/wibble" for 127.0.0.1 at 2013-01-29 09:37:39 -0500
        Processing by CartsController#show as HTML
        Parameters: {"id"=>"wibble"}
        ^[[1m^[[35mCart Load (0.1ms)^[[0m SELECT "carts".* FROM "carts" WHERE
    "carts"."id" = ? LIMIT 1 [["id", "wibble"]]
▶   Попытка доступа к несуществующей корзине
    Redirected to http://localhost:3000/
    Completed 302 Found in 3ms (ActiveRecord: 0.4ms)
```

На рис. 10.4 показан более удачный способ.

На Unix-машинах для просмотра этого файла мы, скорее всего, воспользуемся командой `tail` или `less`. А в Windows можно воспользоваться привычным текстовым редактором. Неплохо было бы держать открытым окно, в котором отображаются добавляемые в файл новые строки. Для этого в Unix нужно воспользоваться командой `tail -f`. Вы можете загрузить команду `tail` для Windows[1] или получить средство с графическим интерфейсом[2]. И наконец, некоторые пользователи OS X для отслеживания регистрационных файлов используют Console.app. Для этого нужно в командной строке набрать команду `open` *имя*`.log`.

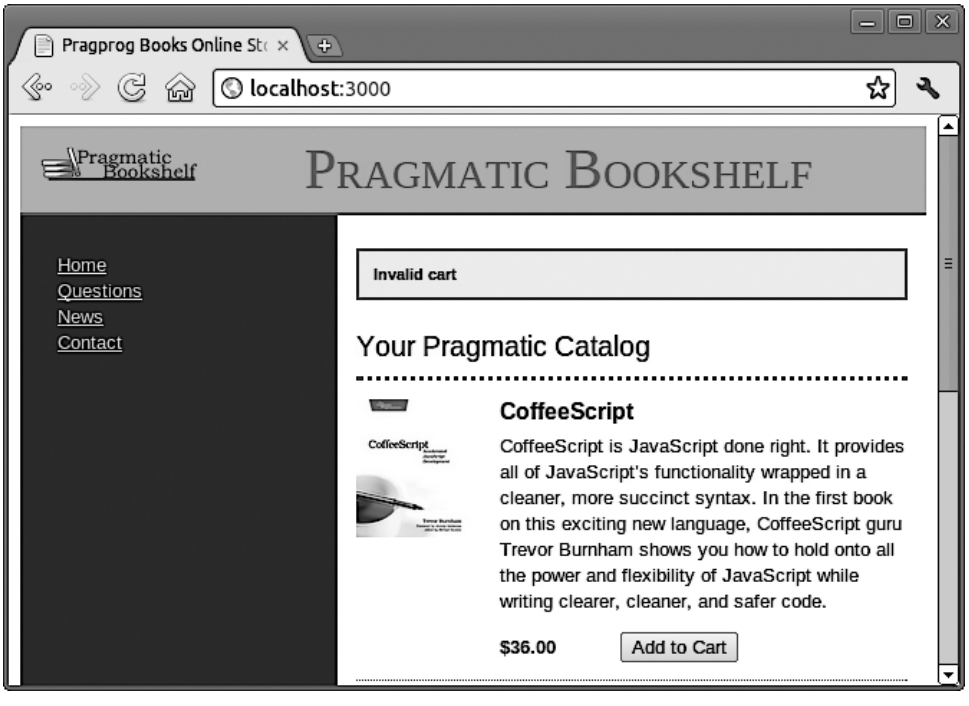


Рис. 10.4. Более удобное для пользователя сообщение об ошибке

Поскольку мы имеем дело с Интернетом, нельзя переживать только лишь за опубликованные веб-формы. Требуется также позаботиться о всевозможных интерфейсах, потому что взломщики могут получить предоставляемый нами исходный код HTML и попытаться подставить дополнительные параметры. И здесь недостоверные корзины перестанут быть самой большой проблемой, важнее будет предотвратить доступ к чужим корзинам.

И как всегда, первым рубежом обороны выступят контроллеры. Давайте начнем с того, что удалим `cart_id` из перечня разрешенных параметров.

[1] http://gnuwin32.sourceforge.net/packages/coreutils.htm
[2] http://tailforwin32.sourceforge.net/

rails40/depot_h/app/controllers/line_items_controller.rb

```
# Never trust parameters from the scary internet, only allow the white
# list through.
# (Не доверяйте параметрам, полученным из этого жуткого интернета,
# разрешайте использование только пустого списка.)
def line_item_params
▶  params.require(:line_item).permit(:product_id)
end
```

Посмотреть на все это в действии можно, перезапустив тесты нашего контроллера.

```
rake test:controllers
```

Хотя все тесты были пройдены, беглый просмотр нашего журнала log/test.log позволил обнаружить сорванную попытку взлома системы безопасности.

```
LineItemsControllerTest: test_should_update_line_item
--------------------------------------------------
    ^[[1m^[[36m (0.0ms)^[[0m ^[[1mbegin transaction^[[0m
    ^[[1m^[[35mLineItem Load (0.1ms)^[[0m SELECT "line_items".* FROM
"line_items" WHERE "line_items"."id" = ? LIMIT 1 [["id", 980190962]]
Processing by LineItemsController#update as HTML
    Parameters: {"line_item"=>{"product_id"=>nil}, "id"=>"980190962"}
    ^[[1m^[[36mLineItem Load (0.1ms)^[[0m ^[[1mSELECT "line_items".* FROM
"line_items" WHERE "line_items"."id" = ? LIMIT 1^[[0m [["id", "980190962"]]
▶  Unpermitted parameters: cart_id
    ^[[1m^[[35m (0.0ms)^[[0m SAVEPOINT active_record_1
    ^[[1m^[[36m (0.1ms)^[[0m ^[[1mRELEASE SAVEPOINT active_record_1^[[0m
Redirected to http://test.host/line_items/980190962
Completed 302 Found in 2ms (ActiveRecord: 0.2ms)
    ^[[1m^[[35m (0.0ms)^[[0m rollback transaction
```

После очистки условий тестирования проблема будет устранена.

rails40/depot_h/test/controllers/line_items_controller_test.rb

```
test "should update line_item" do
▶    patch :update, id: @line_item, line_item: { product_id: @line_item.product_id }
    assert_redirected_to line_item_path(assigns(:line_item))
end
```

Теперь мы очистим журналы тестирования и перезапустим тесты.

```
rake log:clear LOGS=test
rake test:controllers
```

При завершающем просмотре журналов проблемы не выявляются.

Периодический просмотр журналов имеет вполне определенный смысл, поскольку в них содержится немало полезной информации.

Чувствуя близость завершения очередного шага, мы позвали заказчика и показали ему, что теперь приложение осуществляет вполне приемлемую обработку ошибки. Это его обрадовало, и он продолжил испытание нашего приложения на прочность. При этом он обнаружил в нашем новом отображении корзины

небольшой изъян — отсутствие способа опустошения корзины. Эта небольшая доработка и станет нашим следующим шагом, который мы сделаем прежде, чем уйти с работы.

10.3. Шаг Д3: завершение разработки корзины

Теперь мы знаем, что для реализации функции «пустая корзина» нужно добавить ссылку на корзину и изменить метод `destroy()` в контроллере корзины, чтобы он очищал сессию. Начнем с шаблона и, чтобы поместить кнопку на страницу, опять воспользуемся методом `button_to()`:

rails40/depot_h/app/views/carts/show.html.erb

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

<h2>Your Pragmatic Cart</h2>
<ul>
    <% @cart.line_items.each do |item| %>
        <li><%= item.quantity %> &times; <%= item.product.title %></li>
    <% end %>
</ul>

<%= button_to 'Empty cart', @cart, method: :delete,
    data: { confirm: 'Are you sure?' } %>
```

Чтобы гарантировать, что перед перенаправлением на главную страницу с уведомительным сообщением пользователь удаляет содержимое своей собственной корзины (что немаловажно!), в контроллере будет изменен метод `destroy()`:

Rails40/depot_h/app/controllers/carts_controller.rb

```
def destroy
    @cart.destroy if @cart.id == session[:cart_id]
    session[:cart_id] = nil
    respond_to do |format|
        format.html { redirect_to store_url,
            notice: 'Теперь ваша корзина пуста!' }
        format.json { head :no_content }
    end
end
```

Мы также обновляем соответствующий тест в файле test/controllers/carts_controller_test.rb.

Rails40/depot_i/test/controllers/carts_controller_test.rb

```
test "should destroy cart" do
    assert_difference('Cart.count', -1) do
```

```
        session[:cart_id] = @cart.id
        delete :destroy, id: @cart
    end
    assert_redirected_to store_path
end
```

Теперь при просмотре корзины по щелчку на кнопке Опустошить корзину мы будем возвращены на страницу каталога, и небольшое аккуратное сообщение будет уведомлять нас о пустой корзине (рис. 10.5).

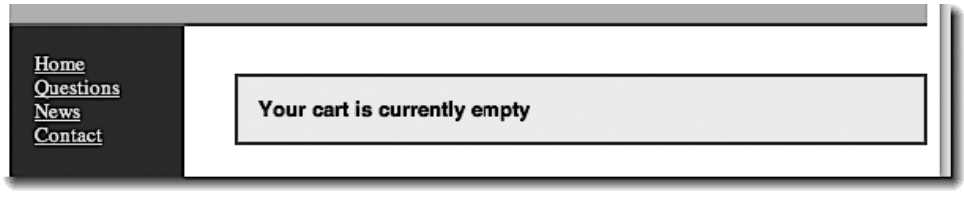


Рис. 10.5. Флэш-уведомление: ваша корзина пуста

Нам также нужно удалить флэш-сообщение, автоматически генерируемое при добавлении товарной позиции:

Rails40/depot_i/app/controllers/line_items_controller.rb

```
def create
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product.id)
    respond_to do |format|
        if @line_item.save
            format.html { redirect_to @line_item.cart }
            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
                status: :unprocessable_entity }
        end
    end
end
```

ДЭВИД ГОВОРИТ: БИТВА МАРШРУТОВ: PRODUCT_PATH ПРОТИВ PRODUCT_URL

Поначалу кажется, что будет довольно сложно узнать, когда при необходимости создать ссылку или перенаправление по заданному маршруту нужно использовать метод product_path, а когда метод product_url. Но на самом деле все довольно просто.

При использовании метода product_url вы получите полную начинку с протоколом и доменным именем, наподобие http://example.com/products/1. Его следует использовать, если осуществляется перенаправление redirect_to, поскольку спецификация HTTP при осуществлении перенаправлений с кодом 302 и им подобных требует указывать URL-адрес полностью. Полный URL-адрес нужен также при перенаправлении с одного домена на другой, например product_url(domain: "example2.com", product: product).

Во всех остальных случаях можно с успехом использовать product_path. Этот метод будет генерировать только часть пути /products/1, а для ссылок или указания форм вроде link_to "My lovely product", product_path(product) больше ничего и не нужно.

Путаница возникает из-за того, что снисходительность браузеров зачастую делает эти два метода взаимозаменяемыми. Перенаправление redirect_to можно производить с product_path, и такой вариант, скорее всего, сработает, но с точки зрения спецификации он будет считаться неправильным. Точно так же можно при ссылке link_to использовать product_url, но тогда ваш HTML будет засорен ненужными символами, что также нельзя признать удачным вариантом.

И наконец, мы наведем порядок в отображении корзины. Давайте вместо использования для каждой записи элементов `<li>` воспользуемся таблицей. Стили опять же будут задаваться с помощью CSS:

Rails40/depot_i/app/views/carts/show.html.erb

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

▶ <h2>Your Cart</h2>
▶ <table>
    <% @cart.line_items.each do |item| %>
▶     <tr>
▶       <td><%= item.quantity %>&times;</td>
▶       <td><%= item.product.title %></td>
▶       <td class="item_price"><%= number_to_currency(item.total_price) %></td>
▶     </tr>
     <% end %>

▶   <tr class="total_line">
▶     <td colspan="2">Total</td>
▶     <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
▶   </tr>
▶ </table>

<%= button_to 'Empty cart', @cart, method: :delete,
    data: { confirm: 'Are you sure?' } %>
```

Чтобы все это заработало, нам нужно к обеим моделям, `LineItem` и `Cart`, добавить метод, возвращающий общую стоимость, соответственно, для каждой товарной позиции и для всей корзины. Сначала займемся товарной позицией, где понадобится только простое умножение:

Rails40/depot_i/app/models/line_item.rb

```
def total_price
    product.price * quantity
end
```

Метод в модели `Cart` мы реализуем с использованием Rails-метода `Array::sum()`, который суммирует цены каждой записи, имеющейся в коллекции:

rails40/depot_i/app/models/cart.rb

```
def total_price
    line_items.to_a.sum { |item| item.total_price }
end
```

Затем нам нужно будет добавить небольшой фрагмент к нашей таблице стилей carts.css.scss:

rails40/depot_i/app/assets/stylesheets/carts.css.scss

```
// Place all the styles related to the Carts controller here.
// They will automatically be included in application.css.
// You can use Sass (SCSS) here: http://sass-lang.com/

.carts {
    .item_price, .total_line {
        text-align: right;
    }
    .total_line .total_cell {
        font-weight: bold;
        border-top: 1px solid #595;
    }
}
```

Улучшенное изображение корзины показано на рис. 10.6.

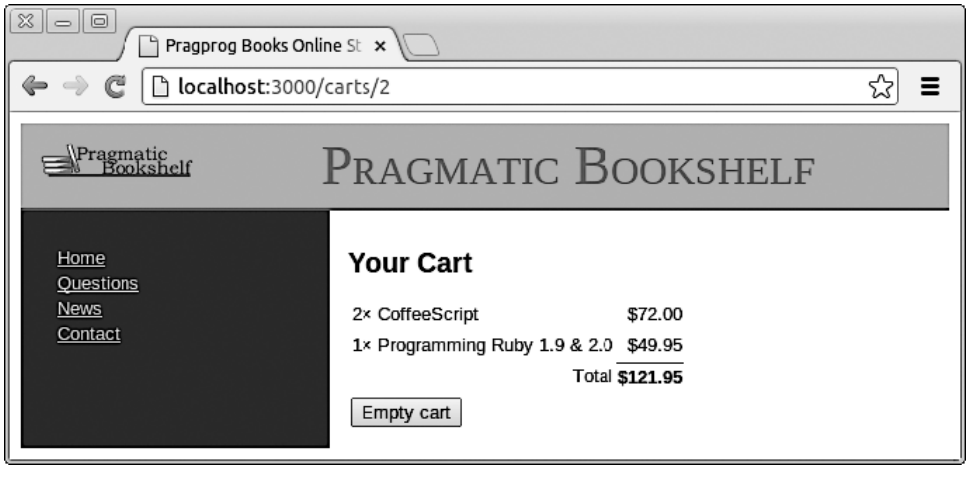


Рис. 10.6. Отображение корзины с общей суммой (Total)

Наши достижения

Теперь наша корзина покупателя может порадовать заказчика. При решении нашей задачи мы прошли следующие этапы:

- ☑ добавление к существующей таблице столбца, имеющего значение по умолчанию;
- ☑ миграция существующих данных в новый табличный формат;
- ☑ предоставление флэш-сообщения об обнаруженной ошибке;
- ☑ использование регистратора для записи событий в журнал;
- ☑ удаление записи;
- ☑ настройка способа отображения таблицы с использованием CSS.

Но когда мы стали подумывать о том, что дело в шляпе, наш заказчик просматривал экземпляр «Information Technology and Golf Weekly». Видимо, там была статья о новом стиле интерфейса «AJAX», используемого на стороне браузера, который позволяет обновлять содержимое на лету. Ну что ж, завтра на него и взглянем.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- ❍ Создайте миграцию, копирующую цену товара в товарную позицию, и измените метод `add_product()` в модели `Cart` для получения цены при создании новой товарной позиции.
- ❍ Добавьте блочные тесты, которые добавляют уникальные товары и дублируют товары. Учтите, что вам придется изменить стенд для ссылки на товары и корзины по имени, например `product: ruby`.
- ❍ Организуйте проверку товаров и товарных позиций в тех местах, где были бы уместны полезные для пользователей сообщения об ошибках.
- ❍ Добавьте возможность удаления из корзины отдельных товарных позиций. Для этого могут потребоваться кнопки в каждой строке, и эти кнопки должны будут ссылаться на действие `destroy()` в `LineItemsController`.

(Подсказки можно найти по адресу: http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

11 Задача E: добавление AJAX

Основные темы:

- использование парциальных шаблонов;
- перемещение корзины в макет страницы;
- динамическое обновление страниц с помощью AJAX и JavaScript;
- выделение изменений с помощью jQuery UI;
- сокрытие и отображение DOM-элементов;
- тестирование AJAX-обновлений.

Наш заказчик захотел добавить в код интернет-магазина поддержку AJAX. А что такое AJAX?

В прежние времена (примерно до 2005 года) считалось, что браузеры не в состоянии обрабатывать данные самостоятельно. При создании приложений, использующих браузер, последнему посылались данные, и больше в течение текущей сессии о нем даже не вспоминали. Временами пользователю нужно было заполнить поля формы или щелкнуть на гиперссылке, пробуждая приложение входящим запросом. В ответ оно посылало пользователю готовую страницу, и весь рутинный процесс возвращался в прежнее русло. Именно так до сих пор вело себя и наше приложение Depot.

Но, оказывается, браузеры не столь пассивны (а вы знали об этом?). Они могут работать с программным кодом. Почти все браузеры в состоянии работать с JavaScript. Оказалось также, что JavaScript на браузере может в фоновом режиме взаимодействовать с приложением на сервере, обновляя в результате этого процесса информацию, предоставляемую пользователю. Джесси Джеймс Гаррет (Jesse James Garrett) назвал такой метод взаимодействия AJAX (что означало когда-то

Asynchronous JavaScript and XML, а теперь означает технологию, позволяющую сократить информационные потребности браузера).

Итак, проведем «аяксификацию» нашей корзины покупателя. Вместо отдельной страницы для обслуживания корзины поместим отображение информации, принадлежащей текущей корзине, на боковую панель каталога. Затем мы применим волшебство AJAX-технологии для обновления корзины на боковой панели без повторного отображения всей страницы.

При работе с AJAX для начала лучше создать версию, не использующую эту технологию, а затем постепенно ввести в нее функции AJAX. Мы так и сделаем. Для начала переместим корзину с ее собственной страницы на боковую панель.

11.1. Шаг E1: перемещение корзины

На данный момент наша корзина отображается действием `show` в `CartController` и соответствующим шаблоном .html.erb. Мы же хотим переместить это отображение в боковую панель, то есть лишить корзину своей собственной страницы. Вместо этого мы отобразим ее в макете, который выводит весь каталог. Проще всего это сделать с помощью *парциальных шаблонов*.

Парциальные шаблоны

Языки программирования предоставляют возможность определения методов. Метод представляет собой поименованный фрагмент кода: стоит только вызвать метод по имени, как тут же будет запущен соответствующий фрагмент кода. И конечно же, методам можно передавать аргументы, позволяющие создавать один фрагмент кода, который может быть использован во многих различных обстоятельствах.

Парциальные шаблоны (или, для краткости, парциалы) Rails можно считать некой разновидностью методов для представлений. Парциал — это просто фрагмент представления, находящийся в своем собственном отдельном файле. Парциал можно вызвать (отобразить) из другого шаблона или из контроллера, и он отобразит самого себя и вернет результаты этого отображения. И, как и при работе с методами, парциалу можно передать аргументы, поэтому один и тот же парциал может отображать разные результаты.

В этом шаге парциалы будут применены дважды. Для начала взглянем на отображение корзины:

rails40/depot_i/app/views/carts/show.html.erb

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

<h2>Your Cart</h2>
<table>
    <% @cart.line_items.each do |item| %>
        <tr>
```

```
          <td><%= item.quantity %>&times;</td>
          <td><%= item.product.title %></td>
          <td class="item_price"><%= number_to_currency(item.total_price) %></td>
        </tr>
    <% end %>
    <tr class="total_line">
        <td colspan="2">Total</td>
        <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
    </tr>
</table>

<%= button_to 'Empty cart', @cart, method: :delete,
    data: { confirm: 'Are you sure?' } %>
```

Здесь создается список строк таблицы, по одной для каждой записи в корзине. Когда сталкиваешься с подобного рода повторениями, возникает вопрос, а не многовато ли логики в шаблоне? Оказывается, абстрагироваться от цикла можно с помощью парциалов (вскоре станет ясно, что тем самым мы подготовимся и к будущему применению магии AJAX).

Для этого воспользуемся возможностью передачи коллекции методу, отображающему парциальные шаблоны, и этот метод автоматически вызовет парциал по одному разу для каждого элемента коллекции. Давайте перепишем отображение корзины, чтобы воспользоваться этой возможностью:

rails40/depot_j/app/views/carts/show.html.erb

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

<h2>Your Cart</h2>
<table>
    <%= render(@cart.line_items) %>

    <tr class="total_line">
        <td colspan="2">Total</td>
        <td class="total_cell"><%= number_to_currency(@cart.total_price) %></td>
    </tr>

</table>

<%= button_to 'Empty cart', @cart, method: :delete,
    data: { confirm: 'Are you sure?' } %>
```

Ну вот, теперь все стало намного проще. Метод `render()` будет осуществлять перебор элементов переданной ему коллекции. Сам по себе парциальный шаблон — это просто еще один файл шаблона (который по умолчанию находится в том же самом каталоге, что и отображаемый объект, и использует в качестве имени имя таблицы). Но чтобы парциалы отличались от обычных шаблонов, Rails при поиске файла автоматически ставит перед именем парциала символ подчеркивания. Значит, наш парциал нужно назвать _line_item.html.erb и поместить его в каталог app/views/line_items.

rails40/depot_j/app/views/line_items/_line_item.html.erb

```
<tr>
    <td><%= line_item.quantity %>&times;</td>
    <td><%= line_item.product.title %></td>
    <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
</tr>
```

Здесь используется одна интересная особенность. Внутри парциального шаблона мы ссылаемся на текущий объект, используя для этого имя переменной, совпадающее с именем шаблона. В данном случае парциал называется `line_item`, следовательно, внутри парциала ожидается наличие переменной по имени `line_item`.

Итак, мы навели порядок с отображением корзины, но она еще не перемещена на боковую панель. Для ее перемещения обратимся еще раз к нашему макету. Имея парциальный шаблон, способный отобразить корзину, мы просто можем вставить в боковую панель следующий вызов:

```
render("cart")
```

Но знает ли сейчас парциал о том, где найти объект `cart`? Он мог бы выстроить предположение. В макете у нас есть доступ к переменной экземпляра `@cart`, которая была определена в контроллере. Оказывается, она также доступна внутри парциала, вызываемого из макета. Это чем-то напоминает вызов метода и передачу ему какого-то значения глобальной переменной. Все это, конечно, работает, но выглядит неважно, увеличивая к тому же взаимозависимость компонентов программы (делая ее менее устойчивой и трудной в поддержке).

Получив парциал для товарной позиции, давайте сделаем то же самое и для корзины. Создадим сначала шаблон _cart.html.erb. Его основу составит наш шаблон carts/show.html.erb, но с использованием `cart` вместо `@cart` и без уведомления. (Заметьте, что вызов из парциала других парциалов — в порядке вещей.)

rails40/depot_j/app/views/carts/_cart.html.erb

```
<h2>Your Cart</h2>
<table>
    <%= render(cart.line_items) %>

    <tr class="total_line">
        <td colspan="2">Total</td>
        <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
    </tr>

</table>

<%= button_to 'Empty cart', cart, method: :delete,
    data: { confirm: 'Are you sure?' } %>
```

Мантра Rails гласит: не допускайте повторений — «Don't Repeat Yourselves» (DRY). Но мы только что сделали это. Теперь эти два файла согласованы, и может показаться, что проблем с ними больше нет. Но проблемы вызывает наличие одного набора логики для вызовов AJAX и другого набора логики для обработки тех случаев, когда JavaScript отключен.

Давайте попробуем всего этого избежать и заменим исходный шаблон кодом, вызывающим отображение парциала:

rails40/depot_k/app/views/carts/show.html.erb

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

▶<%= render @cart %>
```

Теперь изменим макет приложения, включив этот новый парциал в боковую панель:

rails40/depot_k/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
    <title>Pragprog Books Online Store</title>
    <%= stylesheet_link_tag "application", media: "all",
       "data-turbolinks-track" => true %>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tags %>
</head>
<body class="<%= controller.controller_name %>">
    <div id="banner">
        <%= image_tag("logo.png") %>
        <%= @page_title || "Pragmatic Bookshelf" %>
    </div>
    <div id="columns">
        <div id="side">
▶           <div id="cart">
▶               <%= render @cart %>
▶           </div>
▶
            <ul>
                <li><a href="http://www....">Home</a></li>
                <li><a href="http://www..../faq">Questions</a></li>
                <li><a href="http://www..../news">News</a></li>
                <li><a href="http://www..../contact">Contact</a></li>
            </ul>
        </div>
        <div id="main">
            <%= yield %>
        </div>
    </div>
</body>
</html>
```

Далее нужно будет внести небольшие изменения в контроллер магазина. Макет вызывается при обращении к действию `index` контроллера магазина, а пока переменной `@cart` в этом действии присваивается неверное значение. Но это нетрудно исправить:

rails40/depot_k/app/controllers/store_controller.rb

```
class StoreController < ApplicationController
  include CurrentCart
  before_action :set_cart
   def index
     @products = Product.order(:title)
   end
end
```

И наконец, мы изменим инструкции стиля, которые сейчас применяются только к выводу, производимому `CartController`, чтобы они применялись также к таблице, когда она появляется на боковой панели. И здесь SCSS позволяет нам внести изменения только в одном месте, поскольку эта система позаботится о всех вложенных определениях.

rails40/depot_k/app/assets/stylesheets/carts.css.scss

```
// Place all the styles related to the Carts controller here.
// They will automatically be included in application.css.
// You can use Sass (SCSS) here: http://sass-lang.com/
.carts, #side #cart {
    .item_price, .total_line {
        text-align: right;
    }

    .total_line .total_cell {
        font-weight: bold;
        border-top: 1px solid #595;
    }
}
```

Несмотря на то что данные для корзины имеют общий характер и не зависят от того, куда именно помещается вывод, требования, что представление должно быть таким же независимым от того, куда помещается содержимое, не существует. Следует заметить, что черные надписи на зеленом фоне слишком плохо читаются, поэтому давайте предоставим дополнительные правила для этой таблицы при ее появлении на боковой панели:

rails40/depot_k/app/assets/stylesheets/application.css.scss

```
#side {
    float: left;
    padding: 1em 2em;
    width: 13em;
    background: #141;

    form, div {
        display: inline;
    }

    input {
        font-size: small;
    }
```

```
    #cart {
        font-size: smaller;
        color: white;

        table {
            border-top: 1px dotted #595;
            border-bottom: 1px dotted #595;
            margin-bottom: 10px;
        }
    }

    ul {
        padding: 0;
        li {
            list-style: none;
            a {
                color: #bfb;
                font-size: small;
            }
        }
    }
}
```

Если отобразить каталог после добавления товара в корзину, появится картинка, похожая на рис. 11.1. Остается только дождаться наград в номинации Webby Award.

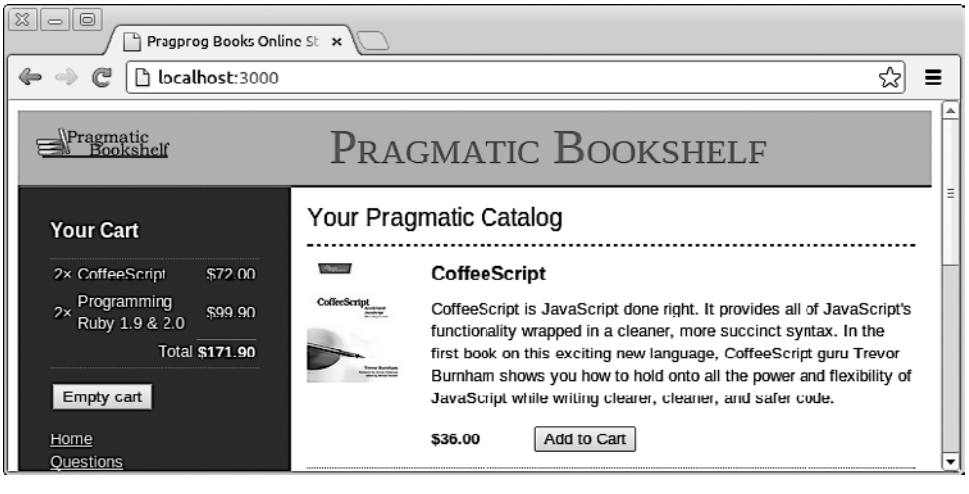


Рис. 11.1. Корзина на боковой панели

Смена направления

Теперь, когда корзина отображается на боковой панели, мы можем изменить характер работы кнопки Добавить в корзину (Add to Cart). Вместо отображения

отдельной страницы корзины ей нужно будет всего лишь обновить главную страницу каталога. Изменить ее функции несложно: в конце действия `create` мы просто перенаправим браузер обратно на каталог:

rails40/depot_k/app/controllers/line_items_controller.rb

```
def create
    @cart = current_cart
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product.id)
    respond_to do |format|
        if @line_item.save
▶           format.html { redirect_to store_url }
            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Итак, теперь у нас есть магазин с корзиной на боковой панели. Когда мы щелкнем на кнопке добавления товара к содержимому корзины, страница перезагрузится с отображением обновленной корзины. Тем не менее, если каталог слишком объемный, эта перезагрузка займет довольно много времени. При этом будут задействованы ресурсы интернет-канала и сервера. Улучшить создавшуюся ситуацию позволит технология AJAX.

11.2. Шаг E2: создание корзины на основе AJAX-технологии

AJAX позволяет создавать код, выполняемый на стороне браузера и взаимодействующий с приложением, находящимся на сервере. В нашем случае мы хотели бы заставить кнопку Добавить в корзину (Add to Cart) вызывать серверное действие в контроллере `LineItems` в фоновом режиме. Затем сервер сможет отправить обратно только лишь HTML-код, касающийся корзины, а мы сможем заменить корзину, присутствующую на боковой панели, ее серверным обновлением.

Сейчас было бы вполне естественно сделать это, написав код JavaScript, который запускается в браузере, и написав код на стороне сервера, который взаимодействует с этим кодом JavaScript (возможно, с помощью такой технологии, как JavaScript Object Notation (JSON)).

Но нам повезло, потому что в Rails все это делается вне поля нашего зрения. Все, что нам нужно, мы можем сделать с помощью Ruby (при широкой поддержке ряда вспомогательных методов Rails).

Вся хитрость добавления AJAX к приложению состоит в том, чтобы делать это мелкими шагами. Начнем с самого основного шага и изменим страницу каталога, чтобы она отправляла нашему серверному приложению AJAX-запрос, заставив приложение отвечать HTML-фрагментом, содержащим обновленную корзину.

На странице каталога для ссылки на действие `create` мы используем метод `button_to()`. Нам нужно внести изменения, чтобы отправить вместо этого AJAX-запрос. Для этого мы просто добавим к вызову фрагмент `remote: true`.

rails40/depot_l/app/views/store/index.html.erb

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

<h1>Your Pragmatic Catalog</h1>

<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
          <%= image_tag(product.image_url) %>
          <h3><%= product.title %></h3>
          <%= sanitize(product.description) %>
          <div class="price_line">
            <span class="price"><%= number_to_currency(product.price) %></span>
▶           <%= button_to 'Add to Cart', line_items_path(product_id: product),
▶               remote: true %>
          </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

Пока мы приспособили браузер для отправки AJAX-запроса нашему приложению. Следующим шагом нужно заставить приложение вернуть ответ. Наш план состоит в том, чтобы создать обновленный HTML-фрагмент, представляющий корзину, и заставить браузер вставить этот HTML в браузерное внутреннее представление структуры и содержимого отображаемого документа, то есть в объектную модель документа — Document Object Model (DOM). Путем воздействия на DOM мы заставим отображение измениться прямо на глазах пользователя.

Сначала мы не дадим действию `create` осуществить перенаправление на отображение главной страницы, если запрос предназначается для JavaScript. Мы сделаем это путем добавления вызова метода `respond_to()`, в котором ему сообщается, что нам нужно получить ответ в формате .js.

Поначалу этот синтаксис может показаться немного странным, но это просто вызов метода, которому в качестве аргумента передается необязательный блок. Описание блоков дается в главе 4, в разделе «Блоки и итераторы». А метод `respond_to()` будет более подробно рассмотрен в главе 20, в разделе «Выбор представления данных».

rails40/depot_l/app/controllers/line_items_controller.rb

```
def create
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product.id)

    respond_to do |format|
        if @line_item.save
            format.html { redirect_to store_url }
▶           format.js
            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Благодаря этому изменению, когда `create` завершает обработку AJAX-запроса, Rails будет искать для отображения шаблон `create template to render`.

Rails поддерживает шаблоны, генерирующие JavaScript, — JS означает JavaScript. Шаблон .js.erb — это способ заставить JavaScript сделать на браузере то, что нам нужно, и весь он создается с помощью кода Ruby на серверной стороне. Давайте напишем наш первый файл: create.js.erb. Он помещается в каталог app/views/line_items, как и все другие представления для товарных позиций:

rails40/depot_l/app/views/line_items/create.js.erb

```
$('#cart').html("<%= escape_javascript render(@cart) %>");
```

Этот простой шаблон заставляет браузер заменять содержимое элемента, имеющего идентификатор `id="cart"`, этим кодом HTML.

Проанализируем, как это у него получается.

Для простоты и краткости библиотека jQuery обозначается псевдонимом `$`, с которого начинаются практически все примеры использования jQuery.

Первый вызов — `$('#cart')` — предписывает jQuery найти HTML-элемент, имеющий `id` со значением `cart`. Затем вызывается метод `html()`[1], в качестве первого аргумента которого используется нужная замена содержимого этого элемента. Эта замена формируется за счет вызова метода `render()` в отношении объекта `@cart`. Вывод этого метода обрабатывается вспомогательным методом `escape_javascript()`, который превращает эту строку Ruby в формат, воспринимаемый как ввод для JavaScript.

Следует заметить, что этот сценарий выполняется в браузере. На сервере выполняется только та часть, которая находится между ограничителями `<%=` и `%>`.

И это работает? В книге, конечно, это трудно показать, но все это работает. Обязательно перезагрузите главную страницу, чтобы получить удаленную версию формы и загруженные в ваш браузер библиотеки JavaScript. Затем щелкните

[1] http://api.jquery.com/html/

на одной из кнопок Добавить в корзину (Add to Cart). Вы увидите, что корзина на боковой панели обновится. И вы *не увидите* в своем браузере ни одного признака перезагрузки страницы. Только что вы создали AJAX-приложение.

Возможные проблемы

Удивительная простота использования AJAX в Rails не защищает от неверных шагов. Поскольку мы имеем дело со свободной интеграцией ряда технологий, понять причины неработоспособности применяемых элементов AJAX-технологии будет непросто. Поэтому на каждом этапе мы добавляем к приложению не более одной функции AJAX.

Если приложение Depot отказывается от демонстрации магии AJAX, можно посоветовать выяснить следующее.

- Не нужны ли вашему браузеру какие-нибудь специальные настройки, чтобы заставить его перезагрузить все, что есть на странице? Иногда браузеры сохраняют в локальной кэш-памяти версию содержимого страницы, мешая тем самым нашему тестированию. Может быть, стоит сделать полную перезагрузку страницы.
- Не было ли каких-нибудь сообщений об ошибках? Загляните в файл development.log в каталоге logs. Загляните также в окно сервера Rails, поскольку некоторые сообщения об ошибках выводятся именно туда.
- Не видно ли при изучении регистрационного журнала входящих запросов к действию `create`? Если их нет, значит ваш браузер не выдает AJAX-запросов. Если были загружены библиотеки JavaScript (при использовании в браузере функции просмотра кода страницы будет показан код HTML), то, может быть, в браузере отключено выполнение JavaScript?
- Некоторые читатели сообщили, что для работы корзины на основе AJAX им пришлось остановить и снова запустить их приложения.
- Если вы пользуетесь браузером Internet Explorer, он может быть запущен в режиме quirks mode, который обеспечивает обратную совместимость со старыми версиями IE, но является таким же урезанным, как и они. Для переключения Internet Explorer в *стандартный режим*, который работает с компонентами AJAX гораздо лучше, в первой строке загружаемой страницы должен быть соответствующий заголовок DOCTYPE. В нашем макете используется следующий заголовок:

```
<!DOCTYPE html>
```

Заказчик вечно чем-то недоволен

Мы вполне довольны своими результатами, поскольку внесли изменения в группу программных строк, и наше скучное приложение, созданное по стандартам Веб 1.0, теперь оживилось благодаря скоростным возможностям AJAX-технологии из арсенала Веб 2.0. И вот, затаив дыхание, мы приглашаем заказчика. Не говоря ни

слова, мы с гордостью щелкаем на кнопке Добавить в корзину (Add to Cart) и следим за его реакцией, ожидая неизбежной похвалы. Но вместо этого наблюдаем лишь его удивление и слышим вопрос: «Вы что, позвали меня продемонстрировать свою ошибку? Ну щелкнули вы на этой кнопке, а что, собственно, изменилось?»

Мы терпеливо объясняем, что на самом деле произошла масса событий, что нужно взглянуть на корзину на боковой панели. Видите? Мы кое-что добавили, и количество изменилось, цифру 4 сменила цифра 5.

«Понятно, — сказал он. — А я и не заметил». Ну, раз он не заметил обновления страницы, стало быть, и другие заказчики тоже могут этого не заметить. Значит, с пользовательским интерфейсом надо что-то делать.

11.3. Шаг E3: выделение изменений

В Rails включено несколько JavaScript-библиотек. Одна из таких библиотек, а именно jQuery UI[1], позволяет украсить веб-страницы рядом интересных визуальных эффектов. Один из этих эффектов — небезызвестный (теперь уже) Yellow Fade Technique, постепенно исчезающая желтизна. С его помощью осуществляется подсветка элемента на странице браузера: по умолчанию элемент получает желтую фоновую подсветку, которая постепенно сменяется белой. Эффект Yellow Fade Technique в применении к нашей корзине показан на рис. 11.2: на заднем плане корзина изображена в исходном варианте. Когда пользователь щелкает на кнопке Добавить в корзину (Add to Cart) и количество товара меняется на 2, строка вспыхивает. Затем ее фон быстро возвращается к исходному.

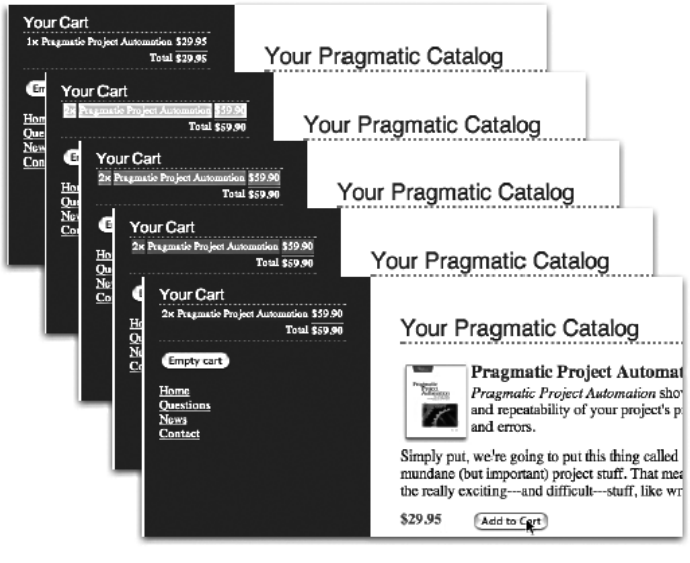


Рис. 11.2. Наша корзина с применением эффекта Yellow Fade Technique

[1] http://jqueryui.com/

Установить библиотеку jQuery UI довольно просто. Сначала нужно добавить одну строку к вашему файлу Gemfile.

rails40/depot_m/Gemfile

```
# Use jquery as the JavaScript library
   gem 'jquery-rails'
▶  gem 'jquery-ui-rails'
```

Установите gem-пакет, выполнив команду `bundle install`:

```
$ bundle install
```

Когда завершится выполнение этой команды, перезапустите свой сервер.

Теперь, когда библиотека `jQuery-UI` доступна нашему приложению, можно вставить эффект, который мы хотим использовать. Это делается путем добавления одной строки кода к файлу app/assets/javascripts/application.js.

rails40/depot_m/app/assets/javascripts/application.js

```
// This is a manifest file that'll be compiled into application.js, which will
// include all the files listed below.
//
// Any JavaScript/Coffee file within this directory, lib/assets/javascripts,
// vendor/assets/javascripts, or vendor/assets/javascripts of plugins, if any,
// can be referenced here using a relative path.
//
// It's not advisable to add code directly here, but if you do, it'll appear at
// the bottom of the compiled file.
//
// Read Sprockets README
// (https://github.com/sstephenson/sprockets#sprockets-directives) for details
// about supported directives.
//
//= require jquery
▶//= require jquery.ui.effect-blind
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

При выполнении шага A2 мы уже видели файл assets/stylesheets/application.css. А этот файл ведет себя точно так же, но только в отношении библиотек JavaScript, а не таблиц стилей. Обратите внимание на то, что в данной строке используется дефис, а не символ подчеркивания, стало быть, не все авторы библиотек следуют одним и тем же соглашениям о присвоении имен.

Давайте воспользуемся данной библиотекой для добавления к нашей корзине этого выделения и устроим вспышку фона при каждом изменении товарной позиции в корзине (как при ее добавлении, так и при изменении количества товара). Тогда пользователям станет понятнее, что произошли какие-то изменения, даже если не была обновлена вся страница.

Сначала нам нужно идентифицировать ту товарную позицию корзины, которая только что была обновлена. Сейчас каждая товарная позиция корзины представляет собой простой `<tr>`-элемент таблицы. Теперь нам нужно найти способ

пометить самые последние изменения. Начнем с `LineItemsController`. Давайте передадим текущую товарную позицию в шаблон, присвоив ее значение переменной экземпляра:

rails40/depot_m/app/controllers/line_items_controller.rb

```
def create
    @cart = current_cart
    product = Product.find(params[:product_id])
    @line_item = @cart.add_product(product.id)
    respond_to do |format|
        if @line_item.save
            format.html { redirect_to store_url }
            format.js { @current_item = @line_item }
            format.json { render action: 'show',
                status: :created, location: @line_item }
        else
            format.html { render action: 'new' }
            format.json { render json: @line_item.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Затем в парциале _line_item.html.erb мы проверим, не является ли отображаемая товарная позиция той самой, в которой только что произошли изменения. Если это так и есть, мы пометим ее с помощью атрибута `id` со значением `current_item`:

rails40/depot_m/app/views/line_items/_line_item.html.erb

```
<% if line_item == @current_item %>
<tr id="current_item">
<% else %>
<tr>
<% end %>
    <td><%= line_item.quantity %>&times;</td>
    <td><%= line_item.product.title %></td>
    <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
</tr>
```

В результате этих двух незначительных изменений элемент `<tr>` той товарной позиции, которая подверглась в корзине самым последним изменениям, будет помечен атрибутом `id="current_item"`. Теперь осталось только заставить JavaScript изменить фоновый цвет на тот, который сразу бросался бы в глаза, а затем постепенно вернуть его к прежнему состоянию. Это будет сделано в уже существующем шаблоне create.js.erb:

rails40/depot_m/app/views/line_items/create.js.erb

```
$('#cart').html("<%= escape_javascript render(@cart) %>");

$('#current_item').css({'background-color':'#88ff88'}).
animate({'background-color':'#114411'}, 1000);
```

Вы заметили, как мы идентифицировали элемент браузера, к которому захотели применить эффект, путем передачи аргумента `'#current_item'` функции `$`? Затем мы вызвали метод `css()` чтобы установить начальный цвет фона, а затем вызвали метод `animate()` для постепенного возвращения к исходному цвету, используемому нашим макетом за период в 1000 мс, широко известный как одна секунда.

После внесения этих изменений щелкните на кнопке Добавить в корзину (Add to Cart), и тогда вы увидите, как измененная товарная позиция вспыхивает светло-зеленым цветом, а затем постепенно возвращается к прежнему состоянию, сливаясь с общим фоном.

11.4. Шаг E4: предотвращение отображения пустой корзины

От нашего заказчика поступил еще один запрос. На данный момент корзина в боковой панели отображается постоянно, даже если в ней ничего нет. Нельзя ли настроить ее так, чтобы она появлялась только в том случае, если в ней что-то есть? Конечно, можно!

У нас есть из чего выбирать. Проще всего, наверное, включить HTML-код корзины только в том случае, когда в ней что-нибудь есть. Все это можно сделать внутри парциала `_cart`:

```
    <% unless cart.line_items.empty? %>
    <div class="cart_title">Your Cart</div>
    <table>
        <%= render(cart.line_items) %>

        <tr class="total_line">
          <td colspan="2">Total</td>
          <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
        </tr>
    </table>

    <%= button_to 'Empty cart', cart, method: :delete,
        confirm: 'Are you sure?' %>
    <% end %>
```

Несмотря на то что все это работает, пользовательский интерфейс получается грубоватым: при переходе от пустой корзины к полной происходит перерисовка всей боковой панели. Поэтому лучше этим кодом не пользоваться. Давайте сделаем все более изящным способом.

Библиотека jQuery UI также предоставляет переходный эффект появления элемента. Давайте воспользуемся аргументом `blind` метода `show()`, который плавно покажет корзину, сдвинув вниз все остальное содержимое боковой панели и освободив для нее место.

Неудивительно, что для вызова эффекта мы опять воспользуемся уже имеющимся у нас шаблоном .js.erb. Поскольку шаблон create вызывается только при добавлении в корзину каких-нибудь товарных позиций, мы знаем, что должны

показать корзину в боковой панели именно тогда, когда в ней точно есть одна товарная позиция (потому что это означает, что до этого корзина была пуста и, стало быть, скрыта). И поскольку корзина должна быть видна до эффекта выделения товарной позиции, мы добавим код показа корзины перед кодом, переключающим цвет фона товарной позиции.

Теперь шаблон имеет следующий вид:

rails40/depot_n/app/views/line_items/create.js.erb

```
▶if ($('#cart tr').length == 1) { $('#cart').show('blind', 1000); }
▶
$('#cart').html("<%= escape_javascript render(@cart) %>");

$('#current_item').css({'background-color':'#88ff88'}).
  animate({'background-color':'#114411'}, 1000);
```

Нам также нужно устроить все так, чтобы скрыть корзину, когда она опустеет. Это можно сделать двумя основными способами. Один из них проиллюстрирован кодом в начале этого раздела и заключается в том, чтобы вообще не генерировать никакого кода HTML. К сожалению, если мы так и сделаем, то при добавлении в корзину какого-нибудь товара и внезапного создания HTML-кода корзины мы будем наблюдать, как корзина на миг появится в браузере при первоначальном отображении, а затем исчезнет, чтобы медленно появиться снова под воздействием эффекта `blind`.

Лучше решить эту проблему, создав HTML-код корзины и установив при этом CSS-стиль `display: none`, если корзина еще пуста. Для этого нам нужно внести изменения в макет application.html.erb, который находится в каталоге app/views/layouts. Наша первая попытка будет иметь следующий вид:

```
<div id="cart"
        <% if @cart.line_items.empty? %>
            style="display: none"
        <% end %>
    >
    <%= render(@cart) %>
</div>
```

Этот код добавляет к тегу `<div>` CSS-атрибут `style=`, но только в том случае, если корзина пуста. Все вроде бы неплохо работает, но выглядит уж очень уродливо. Оторванный символ `>` выглядит так, как будто он не на своем месте (хотя на самом деле это и не так), и способ, которым логика вставляется в тело тега, создает языку шаблонов дурную славу. Давайте не будем засорять свой код подобной несуразицей. Лучше создадим абстракцию, которая все это скроет, — напишем вспомогательный метод.

Вспомогательные методы

Когда мы хотим удалить какой-то процесс обработки из представления (из любого вида представления), нам нужно написать вспомогательный метод.

Если вы взглянете на каталог app, вы увидите в нем шесть подкаталогов.

```
depot> dir app /w
[.]           [..]          [assets]      [controllers] [helpers]
[mailers]     [models]      [views]
```

Вполне очевидно, что наши вспомогательные методы находятся в каталоге helpers. Заглянув в этот каталог, вы увидите, что в нем уже содержатся несколько файлов:

```
depot> dir app\helpers /w
application_helper.rb line_items_helper.rb store_helper.rb
carts_helper.rb products_helper.rb
```

Генераторы Rails автоматически создали файл вспомогательных методов для каждого из наших контроллеров (`products` и `store`). Сама команда Rails (та, которая изначально создала приложение) создала файл application_helper.rb. Если хотите, можете разместить свои методы внутри файлов вспомогательных методов соответствующих контроллеров, но поскольку данный метод будет использован в макете приложения, давайте поместим его в файл вспомогательных методов, общих для всего приложения.

Давайте напишем вспомогательный метод по имени `hidden_div_if()`, который получает условие, дополнительный набор атрибутов и блок. Он помещает вывод, сгенерированный блоком в теге `<div>`, добавляя стиль `display: none`, если условие выполняется. В макете магазина его нужно использовать следующим образом:

rails40/depot_n/app/views/layouts/application.html.erb

```
<%= hidden_div_if(@cart.line_items.empty?, id: 'cart') do %>
    <%= render @cart %>
<% end %>
```

Контроллеру магазина созданный нами вспомогательный метод будет виден благодаря тому, что мы добавим его в файл application_helper.rb в каталоге app/helpers:

rails40/depot_n/app/helpers/application_helper.rb

```
module ApplicationHelper
▶def hidden_div_if(condition, attributes = {}, &block)
▶    if condition
▶        attributes["style"] = "display: none"
▶    end
▶    content_tag("div", attributes, &block)
▶end
end
```

Этот код использует стандартный вспомогательный метод Rails по имени `content_tag()`, который может использоваться для помещения вывода, создаваемого блоком в тег. Используя форму записи `&block`, мы заставляем Ruby передать блок, предоставленный `hidden_div_if()`, во вспомогательный метод `content_tag()`.

И наконец, нам нужно избавиться от флэш-сообщения, которое использовалось нами, когда пользователь очищал корзину. Теперь необходимость в нем отпала, поскольку корзина полностью исчезает из боковой панели при перерисовке страницы каталога. Но есть и еще одна причина для его удаления. Теперь, когда мы используем AJAX для добавления товара в корзину, основная страница не подвергается перерисовке между запросами в процессе покупок. Значит, сообщение о том, что корзина пуста, будет оставаться даже при том, что мы показываем корзину в боковой панели.

rails40/depot_n/app/controllers/carts_controller.rb

```
def destroy
    @cart.destroy
    session[:cart_id] = nil
    respond_to do |format|
        format.html { redirect_to store_url }
        format.json { head :no_content  }
    end
end
```

Теперь, добавив эти AJAX-усовершенствования, продолжим работу.

Хотя может показаться, что проделан довольно большой объем работы, на самом деле все сделанное умещается в двух основных шагах. Во-первых, мы заставили корзину прятаться и появляться, сделав CSS-стиль `display` изменяющимся в зависимости от количества товарных позиций в корзине. Во-вторых, мы установили инструкции JavaScript, предназначенные для вызова эффекта `blind`, когда корзина превращается из пустой в содержащую одну товарную позицию.

Пока что все эти изменения касались внешнего вида, но не функциональности приложения. Давайте приступим к изменению поведения страницы. Как насчет того, чтобы сделать восприимчивыми к щелчкам изображения с добавлением представляемого ими товара в корзину? Оказывается, сделать это с помощью JQuery совсем нетрудно.

11.5. Шаг E5: придание изображениям восприимчивости к щелчкам

Все, что делалось до сих пор, было реакцией на щелчок, и только на тех объектах интерфейса, которые реагировали на него по определению (а именно на кнопках и ссылках). В данном случае нам нужно обработать событие `onClick`, относящееся к изображению, и добиться в процессе этой обработки выполнения некоторых определяемых нами действий.

Иными словами, нам нужно получить сценарий, который выполняется при загрузке страницы, и заставить его найти все изображения и связать логику с этими изображениями, чтобы направить обработку события щелчка на кнопку Добавить в корзину (Add to Cart) для той же самой записи.

Сначала освежим в нашей памяти организацию рассматриваемой страницы:

rails40/depot_n/app/views/store/index.html.erb

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

<h1>Your Pragmatic Catalog</h1>

<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
        <%= image_tag(product.image_url) %>
        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>
        <div class="price_line">
            <span class="price"><%= number_to_currency(product.price) %></span>
            <%= button_to 'Add to Cart', line_items_path(product_id: product),
                remote: true %>
        </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

Используя эту информацию, мы продолжим работу, изменяя содержимое файла app/assets/javascripts/store.js.coffee:

rails40/depot_n/app/assets/javascripts/store.js.coffee

```
# Place all the behaviors and hooks related to the matching controller here.
# All this logic will automatically be available in application.js.
# You can use CoffeeScript in this file: http://coffeescript.org/
$(document).on "ready page:change", ->
    $('.store .entry > img').click ->
        $(this).parent().find(':submit').click()
```

CoffeeScript[1] — это еще один препроцессор, облегчающий написание активных компонентов. В данном случае CoffeeScript помогает выразить JavaScript в более сжатой форме. В сочетании с JQuery вы можете получить в результате весьма существенные эффекты, приложив для этого незначительные усилия.

В данном случае нам в первую очередь нужно определить функцию, выполняемую при загрузке страницы. Именно это и делается в первой строке сценария: в ней определяется функция с использованием оператора `->`, которая затем передается функции по имени `on`, связывающей определяемую функцию с двумя событиями: `ready` и `page:change`. Событие `ready` возникает, если кто-нибудь переходит на вашу страницу не из вашего сайта, а событие `page:change` инициируется турбоссылками (turbolinks)[2], если кто-нибудь переходит на вашу страницу из вашего сайта. Связывание сценария с обоими событиями гарантирует перекрытие любых путей перехода.

[1] http://jashkenas.github.com/coffee-script/

[2] https://github.com/rails/turbolinks/blob/master/README.md#turbolinks

Вторая строка задает поиск всех непосредственных дочерних изображений для элементов, определенных с атрибутом `class="entry"`, которые, в свою очередь, являются потомками элемента с атрибутом `class="store"`. Эта последняя часть играет особую роль, поскольку, как и в случае с таблицами стилей, Rails будет по умолчанию объединять все компоненты JavaScript в единый ресурс. Для каждого найденного изображения — а таких изображений при запуске в отношении других страниц нашего приложения может и не быть — определяется функция, связанная с событием щелчка на этом изображении.

Третья и последняя строка обрабатывает это событие щелчка. Она начинается с элемента, на котором происходит событие, а именно с элемента `this`. Затем осуществляется поиск родительского элемента, который будет контейнером `div` с атрибутом `class="entry"`. Внутри этого элемента мы находим подчиненную кнопку и переходим к щелчку на ней.

При обработке в браузере страница по внешнему виду ничем не отличается от той, что показана на рис. 11.1. Но ведет она себя по-другому. Щелчок на изображении приводит к добавлению товара в корзину. Удивительно то, что все это было выполнено с помощью всего лишь трех строк кода.

Разумеется, можно было все выполнить и непосредственно в JavaScript, но для этого понадобились бы пять дополнительных наборов круглых скобок, два набора фигурных скобок и в целом примерно на 50% больше символов. И это всего лишь малая толика того, на что способен препроцессор CoffeeScript. Для более близкого знакомства с ним есть хорошая книга «CoffeeScript: Accelerated JavaScript Development»[1].

Получается, что мы пока не уделили должного внимания тестированию. Както не чувствуется, что мы сделали многое в плане функциональных изменений, поэтому и беспокоиться об этом вроде бы не стоит. Но, ради своего спокойствия, мы снова запускаем наши тесты:

```
depot> rake test
.....E...F.EEEE.........EEEE..
```

Надо же, отказы и ошибки. Нехорошо. Видимо, нужно снова обратиться к тестированию. Этим мы сейчас и займемся.

11.6. Тестирование изменений, внесенных при добавлении AJAX

Мы посмотрели на отказы при тестировании и увидели несколько ошибок примерно следующего вида:

```
ActionView::Template::Error: undefined method 'line_items' for nil:NilClass
```

[1] *Trevor Burnham*. CoffeeScript: Accelerated JavaScript Development. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

Поскольку эта ошибка представляет большинство отмеченных проблем, давайте сначала займемся именно ею, чтобы потом сконцентрироваться на всех остальных ошибках. Согласно тесту, проблема возникает при получении индекса товара, и действительно, когда мы нацеливаем браузер на адрес http://localhost:3000/products/, то видим окно, показанное на рис. 11.3.

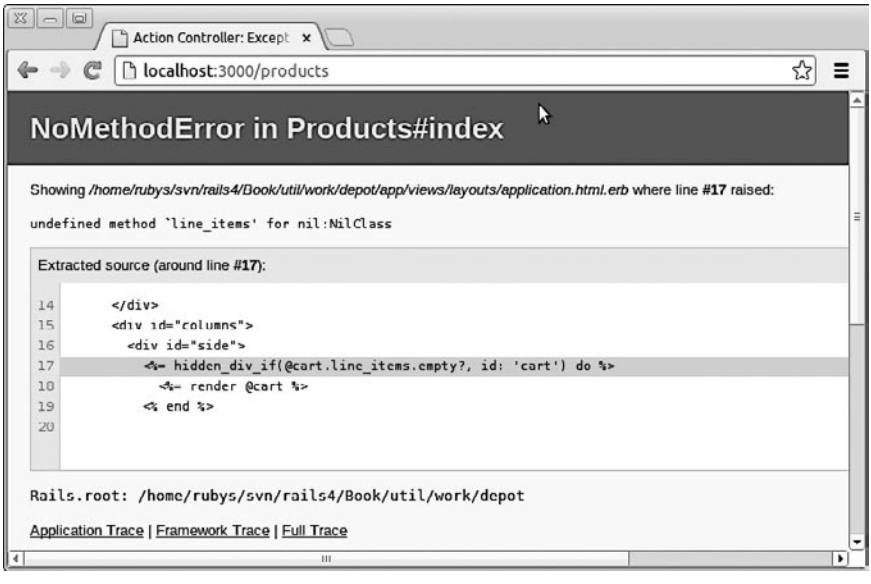


Рис. 11.3. Ошибка в макете может повлиять на все приложение

Это очень полезная информация. В сообщении указывается файл шаблона, который обрабатывался на момент возникновения ошибки (app/views/layouts/application.html.erb), номер строки, в которой произошла ошибка, и фрагмент шаблона, включающий строки вокруг той, в которой возникла ошибка. Из этого можно увидеть, что в момент возникновения ошибки вычислялось выражение `@cart.line_items` и было выдано сообщение о том, что метод `'line_items'` для `nil` не определен (`undefined method 'line_items' for nil`).

Итак, при выводе каталога наших товаров `@cart`, очевидно, имеет значение `nil`. И это вполне логично, поскольку значение для этой переменной задается только в контроллере магазина. Эту ошибку довольно просто исправить, для этого нужно лишь вообще отменить отображение корзины, пока этой переменной не будет присвоено значение:

rails40/depot_o/app/views/layouts/application.html.erb

```
▶<% if @cart %>
    <%= hidden_div_if(@cart.line_items.empty?, id: 'cart') do %>
      <%= render @cart %>
    <% end %>
▶<% end %>
```

После исправления этой ошибки мы перезапускаем тесты и видим, что количество ошибок сократилось до одной. Значение перенаправления было не тем, что ожидалось. Это произошло при создании товарной позиции. И действительно, мы внесли изменения в разделе 11.1, в подразделе «Смена направления». В отличие от последнего изменения, которое имело случайный характер, это изменение было преднамеренным, поэтому мы обновим соответствующие данные функционального теста:

rails40/depot_o/test/controllers/line_items_controller_test.rb

```
test "should create line_item" do
    assert_difference('LineItem.count') do
        post :create, product_id: products(:ruby).id
    end
▸   assert_redirected_to store_path
end
```

После внесения этого изменения наши тесты опять стали успешно проходить. Только представьте себе, что могло бы произойти. Изменение в одной части приложения в целях поддержки новых требований нарушает функцию, ранее реализованную в другой части приложения. При невнимательной работе это может произойти в таком небольшом приложении, как Depot. Но даже при весьма аккуратной работе такое может случиться и в большом приложении.

Но на этом наша работа еще не завершена. Мы не протестировали ни одно из наших AJAX-добавлений, к примеру, не проверили, что произойдет, когда мы щелкнем на кнопке Добавить в корзину (Add to Cart). Rails и здесь облегчает нам задачу.

У нас уже есть тест для создания товарной позиции `should create line item`, поэтому давайте добавим еще один тест для создания товарной позиции с помощью AJAX: `should create line item via ajax`:

rails40/depot_o/test/controllers/line_items_controller_test.rb

```
test "should create line_item via ajax" do
    assert_difference('LineItem.count') do
        xhr :post, :create, product_id: products(:ruby).id
    end

    assert_response :success
    assert_select_jquery :html, '#cart' do
        assert_select 'tr#current_item td', /Programming Ruby 1.9/
    end
end
```

Этот тест отличается по своему названию, по способу вызова из теста создания товарной позиции (`xhr :post` вместо простого `post`, где `xhr` означает трудно произносимый термин XML-HttpRequest), а также по ожидаемым результатам. Вместо перенаправления мы ожидаем успешный ответ, содержащий запрос на замену HTML-кода для корзины, и в этом HTML-коде мы ожидаем найти строку с идентификатором `current_item`, имеющую значение, совпадающее с `Programming Ruby 1.9`. Это достигается путем применения метода `assert_select_jquery()`

для извлечения соответствующего HTML с последующей обработкой этого HTML через те дополнительные утверждения, которые вам нужно применить.

И наконец, здесь был представлен CoffeeScript. Хотя тестирование кода, выполняемого в браузере, не входит в круг вопросов, рассматриваемых в данной книге, мы должны протестировать наличие разметки, от которой зависит этот сценарий. И в этом нет ничего сложного:

rails40/depot_o/test/controllers/store_controller_test.rb

```
test "markup needed for store.js.coffee is in place" do
    get :index
    assert_select '.store .entry > img', 3
    assert_select '.entry input[type=submit]', 3
end
```

Таким образом, если слишком активный веб-дизайнер изменит разметку страницы, повлияв тем самым на нашу логику, мы будем предупреждены об этом и сможем внести изменения еще до того, как код начнет применяться. Следует заметить, что `:submit` — это расширение CSS, относящееся только к jQuery, в нашем тесте нам нужно лишь произвести синтаксический разбор `input[type=submit]`.

Поддерживание теста на уровне, соответствующем изменениям кода, является важной составляющей сопровождения вашего приложения. Rails облегчает эту задачу. Сообразительные программисты делают тестирование неотъемлемой частью своей разработки. Некоторые из них заходят в этом деле настолько далеко, что сначала пишут тесты, а потом уже пишут первую строку своего кода.

Наши достижения

При выполнении данного шага мы добавили к нашей корзине поддержку AJAX.

- ☑ Мы переместили корзину покупателя в боковую панель. Затем мы настроили действие `create` на повторный вывод страницы каталога.
- ☑ Для вызова действия `LineItemsController.create()` с использованием AJAX мы воспользовались кодом `remote: true`.
- ☑ Затем мы использовали ERb-шаблон, чтобы создать JavaScript-код, выполняемый на стороне клиента. Чтобы обновить на странице только HTML-код корзины, в этом сценарии используется jQuery.
- ☑ Чтобы помочь пользователю увидеть изменения, связанные с корзиной, мы добавили эффект выделения, воспользовавшись для этого библиотекой jQuery-UI.
- ☑ Мы написали вспомогательный метод, скрывающий пустую корзину, и воспользовались jQuery для того, чтобы показать корзину при добавлении в нее товара.
- ☑ Мы написали тест, проверяющий не только создание товарной позиции, но также и содержимое ответа, возвращаемого в результате соответствующего запроса.

Главный урок состоит в том, что AJAX-разработку нужно вести постепенно. Начинать следует с традиционного приложения, добавляя AJAX-функции одну за другой. С отладкой AJAX могут возникнуть трудности, поэтому постепенное добавление его функций облегчает отслеживание тех изменений, которые нарушают работу приложения. Мы увидели, что первоначальное создание традиционного приложения существенно упрощает поддержку обоих вариантов его поведения: как с использованием AJAX, так и без его использования.

В завершение дадим вам пару советов. Во-первых, если вы собираетесь заняться широкомасштабными AJAX-разработками, вам, скорее всего, нужно будет ознакомиться со средствами отладки JavaScript, предоставляемыми браузером, и с имеющимися в нем средствами контроля объектной модели документов (DOM), такими как Firebug в Firefox, Developer Tools в Internet Explorer, Developer Tools в Google Chrome, Web Inspector в Safari или Dragonfly в Opera. И во-вторых, дополнительный модуль NoScript, имеющийся в Firefox, позволяет проверять работу приложения с JavaScript и без него одним щелчком мыши. Некоторые считают полезным пользоваться в процессе разработки двумя разными браузерами, на одном из которых JavaScript включен, а на другом отключен. Как только я добавляю какую-нибудь новую функцию, то сразу же ее «подбрасываю» обоим браузерам, чтобы убедиться, что она работает независимо от состояния поддержки JavaScript.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- Сейчас корзина пропадает при опустошении с помощью перерисовки всего каталога. Можете ли вы изменить приложение с целью использования вместо этого `blind`-эффекта из jQuery UI?
- Добавьте сразу после каждой товарной позиции корзины кнопку, щелчок на которой будет вызывать действие, уменьшающее количество товара и удаляющее его из корзины, если количество станет нулевым. Заставьте все это работать сначала без AJAX, а потом с добавлением функций AJAX.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

Задача Ж: оформление покупки 12

Основные темы:

- связывание таблиц внешними ключами;
- использование объявлений `belongs_to`, `has_many` и `:through`;
- создание форм на основе моделей (`form_for`);
- связывание форм, моделей и представлений;
- генерация RSS-канала с помощью метода `atom_helper`, примененного в отношении объектов моделей.

Подведем итоги. На данный момент мы собрали воедино основную систему ведения перечня товара, создали каталог и получили довольно привлекательную корзину покупателя. А теперь нам нужно дать покупателю реальную возможность приобрести содержимое корзины. Давайте реализуем функцию оформления покупки.

Мы не собираемся делать ничего сверхъестественного. На данном этапе все, что нам нужно, — это получить контактную информацию покупателя и избранную им форму оплаты товара. Использовав эти сведения, мы сформируем заказ в базе данных. В дальнейшем будет подробнее рассмотрена работа моделей, вопросы проверки данных и обработки форм.

12.1. Шаг Ж1: регистрация заказа

Заказ представляет собой набор товарных позиций, дополненный подробностями, необходимыми для оформления покупки. В нашей корзине уже есть товарные позиции, `line_items`, поэтому нам нужно лишь добавить к таблице `line_items`

столбец идентификатора заказа `order_id` и создать таблицу заказов `orders` на основе исходных предположений о составе данных, показанных на рис. 5.3, и кратких переговоров с заказчиком.

Сначала мы создадим модель заказов `order` и дополним таблицу `line_items`:

```
depot> rails generate scaffold Order name address:text email pay_type
depot> rails generate migration add_order_to_line_item order:references
```

Заметьте, что для трех из четырех столбцов не указан тип данных. Причина в том, что по умолчанию устанавливается тип данных `string`. Это еще один небольшой пример того, как Rails упрощает разработку вашего приложения в наиболее распространенных случаях, требуя приложения дополнительных усилий только в том случае, когда вам нужно конкретно указать тип данных.

Теперь, после создания миграций мы их можем применить:

```
depot> rake db:migrate
== CreateOrders: migrating ===================================
-- create_table(:orders)
-> 0.0014s
== CreateOrders: migrated (0.0015s) ==========================
== AddOrderIdToLineItem: migrating ===========================
-- add_column(:line_items, :order_id, :integer)
-> 0.0008s
== AddOrderIdToLineItem: migrated (0.0009s) ==================
```

Поскольку в базе данных нет записей для этих двух новых миграций в таблице `schema_migrations`, задача `db:migrate` применяет к базе данных обе миграции. Разумеется, мы можем применить их по отдельности, запустив задачу миграции после создания отдельных миграций.

Создание формы для ввода информации о заказе

Теперь, когда у нас есть все необходимые таблицы и модели, можно приступать к процессу оформления заказа. Сначала нам нужно добавить к корзине кнопку Оформить заказ (Checkout). Поскольку она будет создавать новый заказ, мы свяжем ее с действием `new` в нашем контроллере заказов:

rails40/depot_o/app/views/carts/_cart.html.erb

```
<h2>Your Cart</h2>
<table>
    <%= render(cart.line_items) %>

    <tr class="total_line">
        <td colspan="2">Total</td>
        <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
    </tr>

</table>
```

```
▶<%= button_to "Checkout", new_order_path, method: :get %>
   <%= button_to 'Empty cart', cart, method: :delete,
     confirm: 'Are you sure?' %>
```

Сначала нужно проверить, есть ли что-то в корзине. Для этого нам потребуется доступ к этой корзине. Если закладывать в планы еще и будущие потребности, то доступ к корзине нам также понадобится при создании заказа.

rails40/depot_o/app/controllers/orders_controller.rb

```
class OrdersController < ApplicationController
▶    include CurrentCart
▶    before_action :set_cart, only: [:new, :create]
     before_action :set_order, only: [:show, :edit, :update, :destroy]

     # GET /orders
     #...
end
```

Затем нужно будет добавить код, проверяющий корзину. Если в корзине ничего нет, мы отправим пользователя назад в каталог, оповестим о своих действиях и немедленно вернемся на исходную позицию. Тем самым пользователи не смогут непосредственно перейти к оформлению покупки и к созданию пустых заказов. Здесь важную роль играет инструкция `return`, без нее будет получена ошибка двойного вывода *double render error*, поскольку ваш контроллер попытается осуществить и перенаправление, и вывод.

rails40/depot_o/app/controllers/orders_controller.rb

```
def new
▶    if @cart.line_items.empty?
▶        redirect_to store_url, notice: "Your cart is empty"
▶        return
▶    end
▶
     @order = Order.new
end
```

ДЖО СПРАШИВАЕТ: А ГДЕ ЖЕ ОБРАБОТКА КРЕДИТНЫХ КАРТ?

В реальном мире нам, наверное, потребовалось бы, чтобы наше приложение справлялось с коммерческой стороной оформления заказа. У нас даже может появиться желание встроить в него обработку кредитных карт. Но объединение с серверными системами обработки платежей требует большого объема бумажной работы и преодоления многих трудностей. Это отвлекло бы от рассмотрения вопросов работы с Rails, поэтому мы пока собираемся отложить подробности решения этой задачи.

Мы вернемся к ней в разделе 25.1 «Обработка кредитных карт с помощью Active Merchant», где будут исследованы дополнительные модули, которые могут нам помочь в этом деле.

Мы добавим тест, требующий наличия товарной позиции в корзине `requires item in cart`, и изменим существующий тест для получения новой товарной позиции `should get new`, чтобы гарантировать ее наличие в корзине:

rails40/depot_o/test/controllers/orders_controller_test.rb

```
▸test "requires item in cart" do
▸    get :new
▸    assert_redirected_to store_path
▸    assert_equal flash[:notice], 'Your cart is empty'
▸end

 test "should get new" do
     item = LineItem.new
     item.build_cart
     item.product = products(:ruby)
     item.save!
     session[:cart_id] = item.cart.id
     get :new
     assert_response :success
 end
```

Теперь нам требуется новое действие, чтобы предоставить нашим пользователям форму, приглашающую их ввести информацию в таблицу `orders`: их имя, почтовый и электронный адреса и способ оплаты. Следовательно, нам нужно отобразить шаблон Rails, содержащий форму. Поля ввода этой формы нужно связать с соответствующими атрибутами объекта модели Rails, поэтому нам нужно создать в действии `new` пустой объект модели, чтобы дать этим полям что-нибудь, с чем они будут работать.

Нужно выполнить обычную для HTML-форм задачу: заполнить начальными значениями поля формы, а затем извлечь эти значения в наше приложение, когда пользователь щелкнет на кнопке передачи данных.

Для ссылки на новый объект модели `Order` в контроллере объявляется новая переменная экземпляра `@order`. Наши действия обусловлены тем, что представление заполняет форму, используя данные этого объекта. Само по себе это обстоятельство не представляет интереса: поскольку модель новая, все поля будут пустыми. Но рассмотрим какой-нибудь общий случай. Возможно, нам захочется отредактировать существующий заказ. Или пользователь может попытаться подтвердить заказ, но его данные не пройдут проверку. В таких случаях нам захочется, чтобы данные, существующие в модели, были показаны пользователю при отображении формы. Передача на данной стадии пустого объекта модели сохраняет совместимость со всеми этими случаями — представление всегда сможет иметь доступ к этому объекту модели.

Затем, когда пользователь щелкнет на кнопке передачи информации, желательно, чтобы новые данные извлекались из формы и, возвращаясь контроллеру, передавались в объект модели.

Нам повезло, поскольку Rails со всем этим довольно легко справляется. Она предоставляет в наше распоряжение массу *помощников форм*. Эти помощники взаимодействуют с контроллером и с моделями для осуществления общего решения по обработке формы. Перед тем как приступить к созданию окончательного варианта формы, рассмотрим простой пример:

```
Строка  1 <%= form_for @order do |f| %>
        2    <p>
```

```
3         <%= f.label :name, "Name:" %>
4         <%= f.text_field :name, size: 40 %>
5     </p>
6 <% end %>
```

В данном коде есть две интересные особенности. Начнем с того, что помощник `form_for()` в первой строке формирует стандартную HTML-форму. Но он делает не только это. Первый аргумент, `@order`, сообщает методу о переменной экземпляра, которую нужно использовать при создании имен полей и при организации передачи значений этих полей обратно контроллеру.

Как видите, `form_for` открывает Ruby-блок (который заканчивается на шестой строке). В этот блок можно поместить обычное содержимое шаблона (к примеру, тег `<p>`). Но для ссылки на содержимое формы можно также использовать параметр блока (в данном случае `f`). Мы воспользовались этим в четвертой строке, чтобы добавить в форму текстовое поле. Поскольку текстовое поле создано в контексте `form_for`, оно автоматически связывается с данными объекта `@order`.

В этих взаимосвязях нетрудно запутаться. Важно запомнить, что для связи с моделью Rails нужно знать и *имена* полей, и их *значения*. Эту информацию ей предоставляет комбинация `form_for` и различных помощников, предназначенных для работы с полями (таких как `text_field`). Этот процесс показан на рис. 12.1.

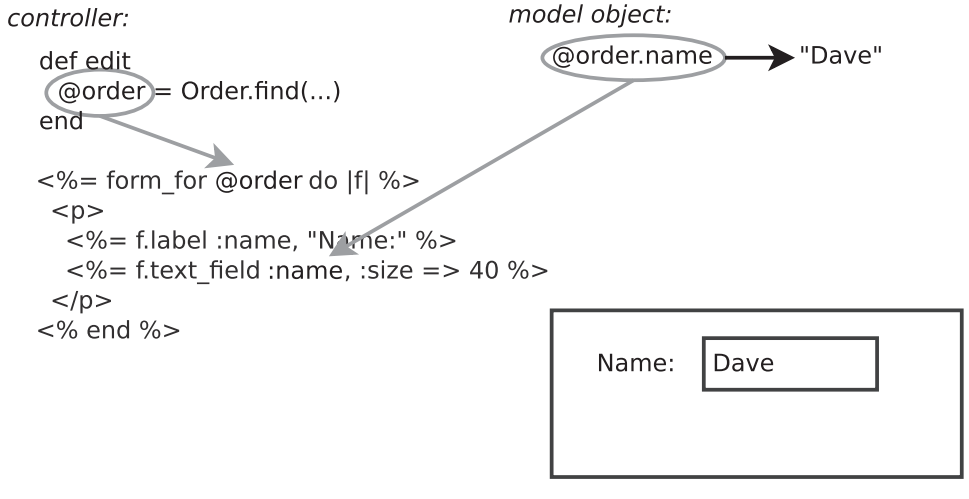


Рис. 12.1. Имена в form_for проецируются на объекты и свойства

Теперь мы можем обновить шаблон, предназначенный для формы, которая собирает данные о клиенте, необходимые для оформления заказа. Его вызов будет осуществляться действием `new` в контроллере заказов, поэтому файл шаблона называется new.html.erb и может быть найден в каталоге app/views/orders:

rails40/depot_o/app/views/orders/new.html.erb

```
<div class="depot_form">
    <fieldset>
```

```
        <legend>Please Enter Your Details</legend>
        <%= render 'form' %>
    </fieldset>
</div>
```

В этом шаблоне используется парциал _form:

rails40/depot_o/app/views/orders/_form.html.erb

```
<%= form_for(@order) do |f| %>
    <% if @order.errors.any? %>
        <div id="error_explanation">
            <h2><%= pluralize(@order.errors.count, "error") %>
            prohibited this order from being saved:</h2>
            <ul>
            <% @order.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
            <% end %>
            </ul>
        </div>
    <% end %>
    <div class="field">
        <%= f.label :name %><br>
▶       <%= f.text_field :name, size: 40 %>
    </div>
    <div class="field">
        <%= f.label :address %><br>
▶       <%= f.text_area :address, rows: 3, cols: 40 %>
    </div>
    <div class="field">
        <%= f.label :email %><br>
▶       <%= f.email_field :email, size: 40 %>
    </div>
    <div class="field">
        <%= f.label :pay_type %><br>
▶       <%= f.select :pay_type, Order::PAYMENT_TYPES,
▶           prompt: 'Select a payment method' %>
    </div>
    <div class="actions">
▶       <%= f.submit 'Place Order' %>
    </div>
<% end %>
```

В Rails имеются помощники для разных HTML-элементов формы. В предыдущем коде для получения имени, а также почтового и электронного адресов клиента использовались помощники `text_field`, `email_field` и `text_area`. Более подробно помощники форм рассматриваются в разделе 21.2 «Генерирование форм».

Единственная сложность связана со списком выбора. Подразумевается, что список возможных вариантов оплаты является свойством модели `Order`. И пока мы этого не забыли, лучше определим в файле модели `order.rb` дополнительный массив:

rails40/depot_o/app/models/order.rb

```
class Order < ActiveRecord::Base
    PAYMENT_TYPES = [ "Check", "Credit card", "Purchase order" ]
end
```

В шаблоне этот массив вариантов оплаты передается помощнику `select`. Ему также передается аргумент `:prompt`, добавляющий фиктивный выбор, содержащий текст приглашения.

Добавим немного магии CSS:

rails40/depot_o/app/assets/stylesheets/application.css.scss

```
.depot_form {
    fieldset {
        background: #efe;
        legend {
            color: #dfd;
            background: #141;
            font-family: sans-serif;
            padding: 0.2em 1em;
        }
    }
    form {
        label {
            width: 5em;
            float: left;
            text-align: right;
            padding-top: 0.2em;
            margin-right: 0.1em;
            display: block;
        }
        select, textarea, input {
            margin-left: 0.5em;
        }
        .submit {
            margin-left: 4em;
        }
        br {
            display: none
        }
    }
}
```

Теперь все готово для испытания нашей формы в деле. Добавьте что-нибудь в свою корзину и щелкните на кнопке Оформить заказ (Checkout). Вы должны увидеть что-нибудь похожее на окно, показанное на рис. 12.2.

Выглядит неплохо! Перед продолжением разработки давайте завершим действие `new`, добавив к нему ряд проверок приемлемости. Изменим модель `Order` для осуществления проверки заполнения клиентом всех полей ввода.

Проверим также, что способ оплаты представлен одним из допустимых значений.

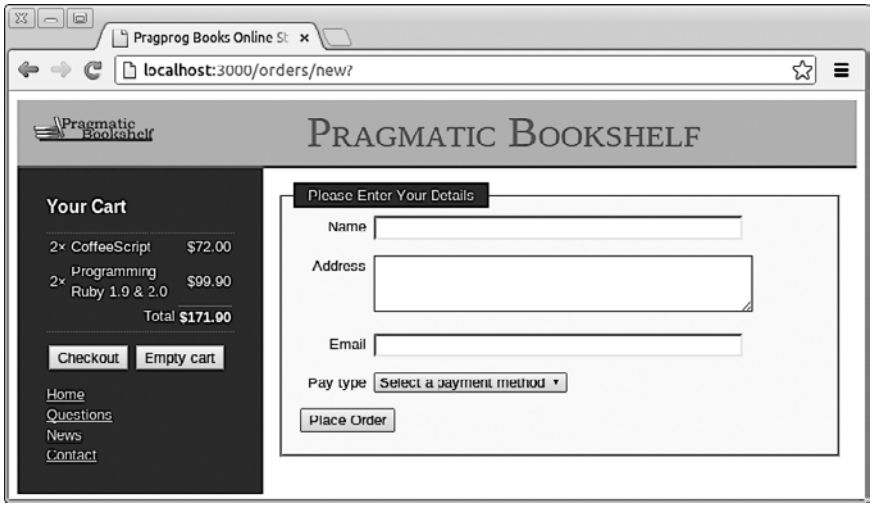


Рис. 12.2. Экран оформления заказа

Возможно, проверка способа оплаты вызовет у кого-то удивление, учитывая, что значения берутся из раскрывающегося списка, содержащего только допустимые варианты. Мы это делаем потому, что приложение не может всецело положиться на то, что ему предоставлены значения из им же созданных форм. Ничто не мешает злоумышленникам отправить данные формы непосредственно приложению в обход нашей формы. Если пользователь указал неизвестный способ оплаты, то вполне вероятно, что он сможет получить наши товары бесплатно.

rails40/depot_o/app/models/order.rb

```
class Order < ActiveRecord::Base
    # ...
▶   validates :name, :address, :email, presence: true
▶   validates :pay_type, inclusion: PAYMENT_TYPES
end
```

Следует заметить, что в начале страницы мы уже осуществили перебор всех значений, имеющихся в переменной `@order.errors`, обеспечивающий вывод сообщений об элементах, не прошедших проверку.

Поскольку правила проверки приемлемости данных были изменены, нам необходимо соответственно изменить и тестовый стенд:

rails40/depot_o/test/fixtures/orders.yml

```
# Read about fixtures at
# http://api.rubyonrails.org/classes/ActiveRecord/Fixtures.html

one:
▶   name: Dave Thomas
    address: MyText
▶   email: dave@example.org
▶   pay_type: Check
```

```
two:
    name: MyString
    address: MyText
    email: MyString
    pay_type: MyString
```

Кроме того, для создаваемого заказа нужно, чтобы в корзине был товар, поэтому нам требуется изменить также и стенд для тестирования товарных позиций:

rails40/depot_o/test/fixtures/line_items.yml

```
# Read about fixtures at
# http://api.rubyonrails.org/classes/ActiveRecord/Fixtures.html
one:
    product: ruby
▶   order: one

two:
    product: ruby
    cart: one
```

Можно также внести и другие изменения, но в функциональных тестах используется только первое из них. Чтобы эти тесты были пройдены, нам понадобится реализовать модель.

Получение детализации заказа

Давайте создадим в контроллере действие `create()`, которое должно будет делать следующее.

1. Взять значения из формы для заполнения нового объекта модели `Order`.
2. Добавить товарные позиции из нашей корзины в этот заказ.
3. Проверить и сохранить заказ. Если проверка не будет пройдена, вывести соответствующие сообщения и дать возможность пользователю внести поправки.
4. Как только заказ будет успешно сохранен, удалить корзину, снова показать страницу каталога и вывести сообщение, подтверждающее размещение заказа.

Сначала определим взаимосвязи и начнем с отношения товарной позиции к заказу:

rails40/depot_o/app/models/line_item.rb

```
class LineItem < ActiveRecord::Base
▶   belongs_to :order
    belongs_to :product
    belongs_to :cart
    def total_price
        product.price * quantity
    end
end
```

Затем определим связь заказа с товарной позицией, еще раз показав, что все товарные позиции, принадлежащие заказу, должны быть уничтожены при уничтожении заказа:

rails40/depot_o/app/models/order.rb

```
class Order < ActiveRecord::Base
▶    has_many :line_items, dependent: :destroy
        # ...
    end
```

В конечном итоге сам метод должен приобрести следующий вид:

rails40/depot_o/app/controllers/orders_controller.rb

```
def create
    @order = Order.new(order_params)
▶    @order.add_line_items_from_cart(@cart)

    respond_to do |format|
        if @order.save
▶           Cart.destroy(session[:cart_id])
▶           session[:cart_id] = nil
▶           format.html { redirect_to store_url, notice:
▶               'Thank you for your order.' }
            format.json { render action: 'show', status: :created,
                location: @order }
        else
▶           @cart = current_cart
            format.html { render action: 'new' }
            format.json { render json: @order.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Сначала создается новый объект `Order`, который инициализируется данными формы. В следующей строке кода к этому заказу добавляются товарные позиции, хранящиеся в корзине, — предназначенный для этого метод мы создадим буквально через минуту.

Затем объекту заказа будет дана команда на сохранение самого себя (и его дочерних элементов, товарных позиций) в базе данных. Попутно объект заказа выполнит проверку (но до нее мы доберемся через минуту). Если сохранение пройдет успешно, мы делаем две вещи. Сначала мы подготовимся к получению от этого же покупателя следующего заказа, удалив корзину из сессии. Затем мы снова выведем каталог и воспользуемся методом `redirect_to()` для отображения сообщения с благодарностью за сделанный заказ. А если сохранить данные не удастся, мы заново выведем форму оформления заказа с текущей корзиной.

В действии `create` мы предполагали наличие в объекте заказа метода `add_line_items_from_cart()`, поэтому давайте сейчас его и реализуем:

rails40/depot_p/app/models/order.rb

```
class Order < ActiveRecord::Base
    # ...
▶   def add_line_items_from_cart(cart)
▶       cart.line_items.each do |item|
▶           item.cart_id = nil
▶           line_items << item
▶       end
▶   end
end
```

Для каждой товарной позиции, переносимой нами из корзины в заказ, нам нужно сделать две вещи. Сначала мы присвоим свойству `cart_id` значение `nil`, чтобы товарная позиция не исчезла при удалении корзины.

Затем саму товарную позицию мы добавим к коллекции `line_items` для заказа. Заметьте, что нам не нужно делать ничего особенного с различными полями внешних ключей — например, устанавливать значения столбца `order_id` на строки товарных позиций для ссылки на только что созданную строку заказа. Rails создает эту связь для нас, используя объявления `has_many()` и `belongs_to()`, добавленные нами к моделям `Order` и `LineItem`. Добавление каждой новой товарной позиции к коллекции `line_items` возлагает ответственность за управление ключами на Rails.

ДЖО СПРАШИВАЕТ: А НЕ ВОЗНИКНУТ ЛИ У НАС ДУБЛИКАТЫ ЗАКАЗОВ?

Джо озабочен тем, что наш контроллер создает объекты модели Order в двух действиях: new и create. Он удивлен тем, что в результате этого в базе данных не появляются продублированные заказы.

Ответ прост: действие new создает в памяти объект Order только лишь затем, чтобы дать программному коду шаблона материал для работы. Когда ответ отправлен в браузер, этот объект остается не у дел, и в конечном счете он будет поглощен сборщиком мусора, имеющимся в Ruby. Он никогда не будет иметь отношение к базе данных.

Действие create также создает объект Order, заполняя его данными из полей формы. Этот объект и будет сохранен в базе данных.

Итак, объекты модели выполняют две роли: они отображают данные, которые извлекаются из базы данных и помещаются в нее, но они также являются и обыкновенными объектами, в которых содержатся какие-то данные, необходимые в процессе работы. Они обращаются к базе данных только тогда, когда им это предписывается, как правило, путем вызова метода save().

Придется также изменить тест, чтобы он соответствовал новому перенаправлению:

rails40/depot_p/test/functional/orders_controller_test.rb

```
test "should create order" do
    assert_difference('Order.count') do
      post :create, order: { address: @order.address, email: @order.email,
        name: @order.name, pay_type: @order.pay_type }
    end
▶   assert_redirected_to store_path
end
```

Итак, в качестве первой проверки всех этих изменений, щелкните на кнопке Разместить заказ (Place Order) на странице оформления заказа без заполнения полей формы. Должна быть выведена страница оформления заказа вместе с рядом сообщений об ошибках, жалующихся по поводу пустых полей, показанная на рис. 12.3.

Если ввести какие-нибудь данные, как показано в верхней части рис. 12.4, и щелкнуть на кнопке Разместить заказ (Place Order), мы окажемся снова на странице каталога, как показано в нижней части того же рисунка. Но все ли сработало? Взглянем на базу данных.

```
depot> sqlite3 -line db/development.sqlite3
SQLite version 3.7.4
Enter ".help" for instructions
sqlite> select * from orders;
          id = 1
        name = Dave Thomas
      address = 123 Main St
       email = customer@example.com
    pay_type = Check
  created_at = 2013-01-29 02:31:04.964785
  updated_at = 2013-01-29 02:31:04.964785
sqlite> select * from line_items;
         id = 10
 product_id = 2
    cart_id =
 created_at = 2013-01-29 02:30:26.188914
 updated_at = 2013-01-29 02:31:04.966057
   quantity = 1
      price = 36
   order_id = 1
sqlite> .quit
```

Рис. 12.3. Полный набор! Ни одно из полей не прошло проверку

Хотя увиденное вами будет различаться номерами версий и датами (и цена будет присутствовать только после выполнения вами упражнения, заданного в разделе «Чем заняться на досуге»), вы увидите отдельный заказ и одну или несколько товарных позиций, соответствующих вашему выбору.

Рис. 12.4. Ввод информации для оформления заказа приводит к выводу благодарности

Еще одно, последнее AJAX-изменение

После того как заказ принят, мы возвращаемся на страницу каталога с отображением благодарственного флэш-сообщения: «Спасибо за Ваш заказ». Если пользователь продолжает закупку и на его браузере включен JavaScript, мы будем заполнять его корзину, расположенную на боковой панели, без перерисовки главной страницы. Значит, флэш-сообщение так и будет оставаться на экране. Нужно, чтобы оно исчезало с добавлением в корзину первого же товара (как это и происходит, если JavaScript в браузере отключен). К счастью, все легко исправить: мы просто скроем блок `<div>`, в котором содержится флэш-сообщение, когда товар попадет в корзину.

ДЖО СПРАШИВАЕТ: А ПОЧЕМУ БЫЛ ВЫБРАН ATOM?

Существует несколько разных форматов RSS-каналов, наиболее примечательные из них RSS 1.0, RSS 2.0 и Atom, ставшие стандартами в 2000, 2002 и 2005 году соответственно. Все

эти три формата имеют весьма широкую поддержку. Чтобы помочь со сменой версий, некоторые сайты предоставляют несколько RSS-каналов для одного и того же сайта, но сейчас такой подход утратил актуальность, он еще больше запутывает пользователей и вообще не рекомендован к применению.

В языке Ruby предоставляется низкоуровневая библиотека, способная производить любой из этих форматов, а также несколько менее распространенных версий RSS. Для наилучшего результата нужно выбрать одну из этих трех основных версий.

Среда Rails всецело настроена на использование рациональных настроек по умолчанию, и в ней в качестве формата RSS-канала по умолчанию выбран Atom. Инженерной группой по развитию Интернета (IETF) он определен для интернет-сообщества как базовый стандартный протокол Интернета, и Rails предоставляет помощник высокого уровня по имени atom_feed, который берет на себя все детали, основываясь на знании используемого в Rails соглашения относительно присваивания имен для таких вещей, как идентификаторы и даты.

rails40/depot_p/app/views/line_items/create.js.erb

```
▶$('#notice').hide();
▶

if ($('#cart tr').length == 1) { $('#cart').show('blind', 1000); }

$('#cart').html("<%= escape_javascript render(@cart) %>");

$('#current_item').css({'background-color':'#88ff88'}).
    animate({'background-color':'#114411'}, 1000);
```

Учтите, что при нашем первом посещении магазина во флэш-сообщении нечего выводить, поэтому HTML-блок с атрибутом `id`, имеющим значение `notice`, не выводится. Следовательно, тег с `id`, имеющим значение `notice`, отсутствует, и вызов jQuery не соответствует ни одному элементу. Так как вызов `hide()` применяется к каждому соответствующему элементу, ничего не происходит и не возникает никакой проблемы. Именно этого мы и добивались, поэтому все в порядке.

После получения всех данных для оформления заказа настало время оповестить об этом отдел заказов. Это будет сделано с помощью записей в RSS-канале, а именно с применением потока ордеров в формате Atom.

12.2. Шаг Ж2: применение Atom-канала

Использование стандартного формата RSS-канала, например Atom, означает, что вы можете тут же получить все преимущества от широкого разнообразия уже существующих клиентов. Поскольку Rails уже знает об идентификаторах, датах и ссылках, она может освободить вас от всяческих забот об этих мелких подробностях и позволить вам сконцентрироваться на создании удобочитаемой сводки. Начнем с добавления нового действия к контроллеру товаров:

rails40/depot_p/app/controllers/products_controller.rb

```
def who_bought
    @product = Product.find(params[:id])
```

```
    @latest_order = @product.orders.order(:updated_at).last
    if stale?(@latest_order)
      respond_to do |format|
          format.atom
      end
  end
end
```

Вдобавок к извлечению товара мы проверяем, не является ли запрос *просроченным*. Помните, как в разделе 8.5, «Шаг B5: Кэширование неполных результатов», неполные результаты ответов кэшировались по той причине, что отображение каталога считалось областью повышенного трафика? Так вот, каналы также относятся к этой категории, но с другой схемой использования. Вместо большого количества разных клиентов, запрашивающих одну и ту же страницу, у нас есть небольшое количество клиентов, часто запрашивающих одну и ту же страницу. Если вам знакома идея использования кэшей браузеров, то она же применима и к RSS-агрегаторам.

Все работает благодаря тому, что ответы содержат метаданные, определяющие, когда содержимое изменялось в последний раз, и хэшируемое значение, которое называется `ETag`. Если последующий запрос возвращает те же данные, сервер получает возможность отправить ответ с пустым телом ответа и признаком того, что данные не подвергались изменениям.

И как обычно при использовании Rails, вам не нужно беспокоиться о механизме реализации. Достаточно указать источник содержимого, а Rails сделает все остальное. В данном случае мы используем последний заказ. А обычная обработка запроса проводится внутри инструкции `if`.

Добавляя `format.atom`, мы заставляем Rails искать шаблон по имени `who_bought.atom.builder`. Такой шаблон может использовать общую типовую XML-функциональность, предоставляемую инструментом Builder, а также использовать знания о RSS-формате Atom, предоставляемые помощником `atom_feed`:

rails40/depot_p/app/views/products/who_bought.atom.builder

```
atom_feed do |feed|
    feed.title "Who bought #{@product.title}"

    feed.updated @latest_order.try(:updated_at)

    @product.orders.each do |order|
        feed.entry(order) do |entry|
            entry.title "Order #{order.id}"
            entry.summary type: 'xhtml' do |xhtml|
                xhtml.p "Shipped to #{order.address}"

                xhtml.table do
                    xhtml.tr do
                        xhtml.th 'Product'
                        xhtml.th 'Quantity'
                        xhtml.th 'Total Price'
                    end
```

```
                order.line_items.each do |item|
                    xhtml.tr do
                        xhtml.td item.product.title
                        xhtml.td item.quantity
                        xhtml.td number_to_currency item.total_price
                    end
                end

                xhtml.tr do
                    xhtml.th 'total', colspan: 2
                    xhtml.th number_to_currency \
                        order.line_items.map(&:total_price).sum
                end
            end

            xhtml.p "Paid by #{order.pay_type}"
        end
        entry.author do |author|
            author.name order.name
            author.email order.email
        end
    end
  end
end
```

Дополнительные сведения, касающиеся инструмента Builder, могут быть найдены в разделе 24.1 «Генерация XML с помощью Builder».

На общем уровне канала нам нужно предоставить лишь два блока информации: заголовок и самую последнюю дату обновления. Если заказов нет, значение `updated_at` будет нулевым и Rails предоставит вместо этого значения текущее время.

Затем осуществляется последовательный перебор каждого заказа, связанного с данным товаром. Следует заметить, что прямой связи между этими двумя моделями не существует. Фактически эта связь является опосредованной. У товаров имеется много товарных позиций, а товарные позиции принадлежат заказу. Мы можем осуществить итерацию и проследить связи, но простым объявлением о существовании связи между товарами (`products`) и заказами (`orders`) *через* связь с товарными позициями (`line_items`) мы можем упростить наш код:

rails40/depot_p/app/models/product.rb

```
class Product < ActiveRecord::Base
    has_many :line_items
▶   has_many :orders, through: :line_items
    #...
end
```

Для каждого заказа мы предоставляем заголовок, сводку и автора. Сводка может быть полным XHTML, и мы используем это для создания таблицы названий товаров, заказанного количества и общих стоимостей. За этой таблицей мы поместим абзац, содержащий способ оплаты (`pay_type`).

Чтобы выполнить эту работу, нужно определить маршрут. Это действие будет отвечать за HTTP GET-запросы и будет работать с составляющими коллекции (иными словами, с отдельными товарами), а не со всей коллекцией сразу (что в данном случае будет означать все товары):

rails40/depot_p/config/routes.rb

```
Depot::Application.routes.draw do
    resources :orders
    resources :line_items
    resources :carts

    get "store/index"

▶   resources :products do
▶       get :who_bought, on: :member
▶   end

    # The priority is based upon order of creation:
    # first created -> highest priority.
    # See how all your routes lay out with "rake routes".
    # You can have the root of your site routed with "root"
    # ...
end
```

Мы сами можем испытать все это в работе:

```
depot> curl --silent http://localhost:3000/products/3/who_bought.atom
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
    <id>tag:localhost,2005:/products/3/who_bought</id>
    <link type="text/html" href="http://localhost:3000" rel="alternate"/>
    <link type="application/atom+xml"
        href="http://localhost:3000/info/who_bought/3.atom" rel="self"/>
    <title>Who bought Programming Ruby 1.9</title>
    <updated>2013-01-29T02:31:04Z</updated>
    <entry>
        <id>tag:localhost,2005:Order/1</id>
        <published>2013-01-29T02:31:04Z</published>
        <updated>2013-01-29T02:31:04Z</updated>
        <link rel="alternate" type="text/html"
            href="http://localhost:3000/orders/1"/>
        <title>Order 1</title>
        <summary type="xhtml">
            <div xmlns="http://www.w3.org/1999/xhtml">
                <p>Shipped to 123 Main St</p>
                <table>
                ...
                </table>
                <p>Paid by check</p>
            </div>
        </summary>
        <author>
            <name>Dave Thomas</name>
```

```
            <email>customer@pragprog.com</email>
        </author>
    </entry>
</feed>
```

Выглядит неплохо. Теперь на этот канал можно подписаться в предпочитаемом нами средстве чтения RSS-каналов.

Но лучше всего то, что наша работа понравилась заказчику. Мы реализовали ведение каталога товаров, вывод основного каталога и корзину покупателя, а теперь у нас есть еще и простая система обработки заказов. Наверное, нужно было бы еще создать некое приложение, относящееся к выполнению заказа, но оно подождет до какого-нибудь нового шага. (И этот шаг мы в данной книге пропустим: его реализацией мы вряд ли скажем что-либо новое о Rails.)

Наши достижения

За сравнительно короткий промежуток времени нам удалось сделать следующее.

- ☑ Мы создали форму для сбора сведений, касающихся оформления заказа и связали ее с новой моделью заказов.
- ☑ Мы добавили проверку приемлемости данных и использовали вспомогательные методы для отображения пользователю допущенных ошибок.
- ☑ Мы предоставили RSS-канал, чтобы администратор мог отслеживать заказы по мере их поступления.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- ❍ Получите представления, работающие с запросами `who_bought` в HTML-, XML- и JSON-формате. Поэкспериментируйте с включением информации о заказе в XML-представление путем визуализации `@product.to_xml(include: :orders)`. Сделайте то же самое для формата JSON.
- ❍ Что произойдет, если щелкнуть на кнопке Оформить заказ (Checkout) на боковой панели, когда экран оформления заказа уже отображен? Сможете ли вы найти способ отключения кнопки при таких обстоятельствах?
- ❍ Список возможных способов оплаты пока хранится в виде константы в классе `Order`. Можете ли вы переместить этот список в таблицу базы данных? Сможете ли вы по-прежнему проводить проверку приемлемости данных для соответствующего поля?

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

Задача 3: отправка электронной почты 13

Основные темы:

- электронная почта и
- комплексное тестирование.

На данный момент у нас есть веб-сайт, который будет отвечать на запросы и предоставлять RSS-потоки, позволяющие периодически проверять продажи отдельных наименований товаров. Но иногда возникает потребность в чем-то большем. На данный момент нам нужна возможность при наступлении определенного события отправлять конкретному человеку целевое сообщение. Может появиться потребность уведомить системного администратора о возникновении исключительной ситуации. Также это может быть форма отзыва, отправляемая пользователю. В данной главе мы выберем простую отправку подтверждающих сообщений по электронной почте клиентам, разместившим заказ. Когда задача будет выполнена, мы создадим тест для поддержки не только добавленных почтовых сообщений, но и для всего имеющегося на данный момент сценария работы пользователя с приложением.

13.1. Шаг 31: отправка подтверждающих электронных сообщений

Отправка сообщений электронной почты состоит в Rails из трех основных частей: конфигурации отправки электронных сообщений, определения момента отправки сообщения и указания того, что нужно сообщить. Рассмотрим эти части по очереди.

Конфигурация отправки электронных сообщений

Конфигурация отправки электронных сообщений является частью среды Rails-приложений, для чего используется блок `Depot::Application.configure`. Если нужно иметь одну и ту же конфигурацию для разработки, тестирования и реальной работы, добавьте конфигурацию в файл environment.rb в каталоге config; в противном случае добавьте разные конфигурации в соответствующие файлы в каталоге config/environments.

Внутри блока нужно поместить одну или несколько инструкций. Сначала нужно решить, как вы желаете доставлять электронные сообщения:

```
config.action_mailer.delivery_method = :smtp
```

Альтернативой `:smtp` служат аргументы `:sendmail` и `:test`.

Аргументы `:smtp` и `:sendmail` используются, когда нужно, чтобы попытки отправки электронной почты осуществляла система Action Mailer. При реальной работе приложения вам наверняка нужно будет воспользоваться одним из этих методов.

Установка аргумента `:test` подходит для блочного и функционального тестирования, которое будет использоваться в разделе «Тестирование отправки сообщений электронной почты». Электронная почта не будет доставляться, вместо этого она будет добавляться к массиву (доступному через свойство `ActionMailer::Base.deliveries`). В среде тестирования этот метод доставки используется по умолчанию. Интересно, что в среде разработки по умолчанию используется метод доставки `:smtp`. Если нужно, чтобы Rails доставляла электронную почту в процессе разработки вашего приложения, такая установка вполне подойдет. Если нужно отключить доставку электронной почты в режиме разработки, отредактируйте файл development.rb в каталоге config/environments, добавив к нему следующие строки:

```
Depot::Application.configure do
    config.action_mailer.delivery_method = :test
end
```

Настройка `:sendmail` возлагает доставку электронной почты на почтовую программу вашей локальной системы, которая предположительно находится в каталоге /usr/sbin. Этот механизм доставки не обладает достаточной переносимостью, поскольку в других операционных системах sendmail не всегда устанавливается в этот каталог. Эта настройка также зависит от поддержки вашей локальной программы sendmail ключей командной строки `-i` и `-t`.

Можно добиться большей переносимости, если оставить для этой настройки ее значение по умолчанию `:smtp`. При этом нужно также указать некоторые дополнительные настройки, сообщающие Action Mailer, где найти SMTP-сервер для обработки ваших исходящих сообщений электронной почты. Это может быть машина, запустившая ваше веб-приложение, или может быть отдельный блок (возможно, у вашего интернет-провайдера, если вы запустили Rails вне корпоративной среды). Настройки этих параметров могут быть предоставлены вашим системным администратором. Вы можете также определить их из своих собственных настроек клиента электронной почты.

Следующие настройки являются типовыми для Gmail. Если нужно, можете взять их за основу.

```
Depot::Application.configure do
    config.action_mailer.delivery_method = :smtp

    config.action_mailer.smtp_settings = {
        address:          "smtp.gmail.com",
        port:              587,
        domain:           "domain.of.sender.net",
        authentication:   "plain",
        user_name:        "dave",
        password:         "secret",
        enable_starttls_auto: true
    }
end
```

Как и при всех других изменениях конфигурации, если были внесены изменения в любой из файлов среды окружения, вам потребуется перезапустить ваше приложение.

Отправка сообщений электронной почты

Завершив настройку конфигурации, давайте напишем код для отправки сообщений электронной почты.

Теперь уже вы вряд ли удивитесь, узнав, что в Rails есть сценарий-генератор для создания почтовых отправлений. Почтовая система в Rails является классом, который хранится в каталоге app/mailers. Он содержит один или несколько методов, каждый из которых соответствует определенному шаблону сообщения электронной почты. Для создания тела сообщения эти методы по очереди используют представления (точно так же, как действия контроллера используют представления для создания HTML и XML). Итак, давайте создадим почтовую систему для нашего интернет-магазина. Она будет использоваться для отправки двух разных типов электронных сообщений: одного при размещении заказа, а другого при отправке заказа. Команда `rails generate mailer` получает имя почтового класса, а также имена методов, относящихся к действиям по отправке электронной почты:

```
depot> rails generate mailer OrderNotifier received shipped
    create app/mailers/order_notifier.rb
    invoke erb
    create   app/views/order_notifier
    create   app/views/order_notifier/received.text.erb
    create   app/views/order_notifier/shipped.text.erb
    invoke   test_unit
    create   test/mailers/order_notifier_test.rb
```

Заметьте, что мы создали класс `OrderNotifier` в app/mailers и два файла шаблонов, по одному для каждого типа электронных сообщений, в каталоге app/views/

notifier. (Мы также создали файл теста, который рассмотрим в разделе «Тестирование отправки сообщений электронной почты».)

Каждый метод в почтовом классе отвечает за настройку среды для отправки конкретного сообщения электронной почты. Перед тем как перейти к деталям, рассмотрим пример. Это код, который был сгенерирован для нашего класса `OrderNotifier` с одним изменением установки по умолчанию:

rails40/depot_q/app/mailers/order_notifier.rb

```
class OrderNotifier < ActionMailer::Base
▶    default from: 'Sam Ruby <depot@example.com>'

    # Subject can be set in your I18n file at config/locales/en.yml
    # with the following lookup:
    #
    # en.order_notifier.received.subject
    #
    def received
        @greeting = "Hi"

        mail to: "to@example.org"
    end
    # Subject can be set in your I18n file at config/locales/en.yml
    # with the following lookup:
    #
    # en.order_notifier.shipped.subject
    #
    def shipped
        @greeting = "Hi"

        mail to: "to@example.org"
    end
end
```

Если вам показалось, что это похоже на контроллер, то в принципе так оно и есть. Каждому действию соответствует один метод. Вместо вызова метода отображения `render()` вызывается метод отправки электронной почты `mail()`. Этот метод принимает ряд аргументов, включая `:to` (как показано в листинге), `:cc`, `:from` и `:subject`, каждый из которых вызывает выполнение намного большего количества полезных действий, чем вы могли бы ожидать. Значения, общие для всех вызовов `mail` в почтовом классе, могут быть установлены как значения по умолчанию, как это сделано для `:from` в самом начале определения этого класса. Вы можете перекроить этот класс под свои собственные нужды.

Комментарии, имеющиеся в классе, также свидетельствуют о том, что строки темы (subject) уже готовы к переводу, который будет рассмотрен в главе 15 «Задача К: локализация». А теперь мы просто будем использовать параметр `:subject`.

Как и при работе с контроллерами, шаблоны содержат отправляемый текст, а котроллеры и почтовые классы могут предоставлять значения, вставляемые в такие шаблоны посредством переменных экземпляра.

Шаблоны сообщений электронной почты

Сценарий-генератор создал в каталоге app/views/order_notifier два шаблона сообщений электронной почты, по одному для каждого действия в классе `OrderNotifier`. Это обычные файлы с расширением .erb. Мы будем пользоваться ими для создания обычных текстовых сообщений электронной почты (создание сообщений HTML-формата будет показано позже). Как и при работе с шаблонами для создания веб-страниц нашего приложения, в файлах содержится сочетание статического текста и динамического содержимого. Шаблон, содержащийся в файле received.text.erb, мы можем переделать под свои потребности. Это сообщение электронной почты будет отправляться для подтверждения заказа:

rails40/depot_q/app/views/order_notifier/received.text.erb

```
Уважаемый(ая) <%= @order.name %>

Администрация магазина Pragmatic Store благодарит Вас за недавний заказ.

Вы заказали следующий товар:

<%= render @order.line_items -%>

Когда Ваш заказ будет отправлен, мы пошлем Вам по электронной почте отдельное
сообщение.
```

Парциальный шаблон, выводящий товарную позицию, форматирует отдельную строку с указанием количества товара и его названия. Поскольку мы имеем дело с шаблоном, нам доступны все стандартные вспомогательные методы, такие как `truncate()`:

rails40/depot_q/app/views/line_items/_line_item.text.erb

```
<%= sprintf("%2d x %s",
            line_item.quantity,
            truncate(line_item.product.title, length: 50)) %>
```

Теперь мы должны вернуться обратно и заполнить метод `received()` в классе `OrderNotifier`:

rails40/depot_r/app/mailers/order_notifier.rb

```
def received(order)
    @order = order

    mail to: order.email, subject: 'Подтверждение заказа в Pragmatic Store'
end
```

Здесь мы добавили `order` в качестве аргумента для вызова метода `received`, добавили код для копирования переданного аргумента в переменную экземпляра и обновили вызов метода `mail()`, указав, куда отправлять сообщение электронной почты и какую строку темы использовать.

Генерация сообщений электронной почты

После настройки шаблона и определения почтового метода мы можем использовать их в наших обычных контроллерах для создания и (или) отправки сообщений электронной почты.

rails40/depot_r/app/controllers/orders_controller.rb

```
def create
    @order = Order.new(order_params)
    @order.add_line_items_from_cart(@cart)

    respond_to do |format|
        if @order.save
            Cart.destroy(session[:cart_id])
            session[:cart_id] = nil
▶           OrderNotifier.received(@order).deliver
            format.html { redirect_to store_url, notice:
                'Thank you for your order.' }
            format.json { render action: 'show', status: :created,
                location: @order }
        else
            format.html { render action: 'new' }
            format.json { render json: @order.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Нам также нужно обновить метод `shipped()` точно так же, как это было сделано с методом `received()`:

rails40/depot_r/app/mailers/order_notifier.rb

```
def shipped(order)
    @order = order

    mail to: order.email, subject: 'Заказ из Pragmatic Store отправлен'
end
```

На данный момент у нас есть достаточная база для размещения заказа и отправки самому себе обычного сообщения электронной почты — если предположить, что вы не отключили отправку сообщений в режиме разработки. А теперь давайте приправим сообщение электронной почты небольшой порцией форматирования.

Доставка сообщений с составным содержимым

Кто-то предпочитает получать сообщения электронной почты в простом текстовом формате, а кто-то — в формате HTML. Rails упрощает отправку сообщений, имеющих альтернативные форматы содержимого, позволяя пользователю (или клиенту электронной почты) решать, что именно он предпочитает увидеть.

В предыдущем разделе мы создали простое текстовое сообщение электронной почты. Файл представления для нашего действия `received` был назван received.text.erb. Это имя соответствует стандартному соглашению Rails по присваиванию имен. Мы также можем создать сообщения электронной почты в формате HTML.

Давайте попробуем это сделать для уведомления об отправке заказа. Нам не нужно вносить изменения в какой-либо код, нужно просто создать новый шаблон:

rails40/depot_r/app/views/order_notifier/shipped.html.erb

```
<h3>Заказ из Pragmatic Order отправлен</h3>
<p>
    Это сообщение уведомляет Вас, что мы отправили ваш недавний заказ:
</p>

<table>
    <tr><th colspan="2">К-во</th><th>Описание</th></tr>
<%= render @order.line_items -%>
</table>
```

Нам даже не нужно изменять парциал, поскольку с поставленной задачей вполне сможет справиться тот, что у нас уже имеется:

rails40/depot_r/app/views/line_items/_line_item.html.erb

```
<% if line_item == @current_item %>
<tr id="current_item">
<% else %>
<tr>
<% end %>
    <td><%= line_item.quantity %>&times;</td>
    <td><%= line_item.product.title %></td>
    <td class="item_price"><%= number_to_currency(line_item.total_price) %></td>
</tr>
```

Но для шаблонов сообщений электронной почты действует еще одно волшебство имен. Если будут созданы несколько шаблонов с одинаковыми именами, но с разными типами указаний на содержимое, которое вставлено в имена их файлов, Rails отправит все эти файлы в одном сообщении электронной почты, упаковав содержимое так, чтобы клиент электронной почты мог бы различить каждое из них.

Это означает, что вы можете либо обновить, либо удалить шаблон с обычным текстом, который Rails предоставила для уведомления об отправке заказа.

Тестирование отправки сообщений электронной почты

Когда мы использовали сценарий-генератор для создания нашей почтовой системы, обслуживающей заказы, он автоматически сконструировал соответствующий файл order_notifier_test.rb в каталоге приложения test/mailers. Он весьма прост по

содержанию, в нем просто вызывается каждое действие и проверяются избранные части созданного сообщения электронной почты. Поскольку мы уже приспособили сообщение электронной почты под свои нужды, давайте обновим и тест, чтобы он соответствовал изменениям:

rails40/depot_r/test/mailers/order_notifier_test.rb

```
require 'test_helper'

class OrderNotifierTest < ActionMailer::TestCase
    test "received" do
▶       mail = OrderNotifier.received(orders(:one))
▶       assert_equal "Подтверждение заказа в Pragmatic Store", mail.subject
▶       assert_equal ["dave@example.org"], mail.to
▶       assert_equal ["depot@example.com"], mail.from
▶       assert_match /1 x Programming Ruby 1.9/, mail.body.encoded
    end

    test "shipped" do
▶       mail = OrderNotifier.shipped(orders(:one))
▶       assert_equal "Заказ из Pragmatic Store отправлен", mail.subject
▶       assert_equal ["dave@example.org"], mail.to
▶       assert_equal ["depot@example.com"], mail.from
▶       assert_match /<td>1&times;<\/td>\s*<td>Programming Ruby 1.9<\/td>/,
▶       mail.body.encoded
    end
end
```

Метод тестирования приказывает почтовому классу создать (но не отправлять) сообщение электронной почты, и мы используем утверждение для проверки соответствия динамического содержимого ожидаемому. Обратите внимание на использование метода `assert_match()` для проверки только части содержимого тела сообщения. Ваши результаты могут различаться в зависимости от того, как вы настроили строку `default :from` в своем классе `OrderNotifier`.

На этой стадии мы проверили, что сообщение, которое мы намеревались создать, правильно сформатировано, но мы не проверили, что оно отправлено, когда клиент завершает процесс оформления заказа. Для этого мы воспользуемся комплексными тестами.

ДЖО СПРАШИВАЕТ: А НЕ МОГУ ЛИ Я ТАКЖЕ ПОЛУЧАТЬ ЭЛЕКТРОННУЮ ПОЧТУ?

Action Mailer упрощает написание Rails-приложений, обрабатывающих входящую электронную почту. К сожалению, вам необходимо найти способ для извлечения соответствующих сообщений электронной почты из среды окружения вашего сервера и вставки их в приложение — это потребует дополнительных усилий.

Упрощается сама обработка сообщений внутри приложения. Напишите в вашем классе Action Mailer метод экземпляра под названием receive(), который получает один аргумент. Этот аргумент будет объектом Mail::Message, соответствующим входящей электронной почте. Вы можете извлекать поля, текстовое тело и (или) вложения и использовать их в своем приложении.

Все обычные технологии перехвата входящих сообщений сводятся к запуску команды с передачей ей содержимого сообщения в качестве стандартного ввода. Если мы делаем Rails-сценарий runner командой, вызываемой при каждом поступлении электронной почты, мы можем настроить ее на передачу этой электронной почты коду нашего приложения, занимающемуся обработкой электронной почты. Например, используя перехват, основанный на использовании утилиты procmail, мы можем написать правило, похожее на приведенный далее пример. Используя загадочный синтаксис procmail, это правило копирует любое входящее почтовое сообщение, чье поле темы (subject) содержит «Bug Report», через наш сценарий runner:

```
RUBY=/opt/local/bin/ruby
TICKET_APP_DIR=/Users/dave/Work/depot
HANDLER='IncomingTicketHandler.receive(STDIN.read)'

:0 c
* ^Subject:.*Bug Report.*
| cd $TICKET_APP_DIR && $RUBY bin/rails runner $HANDLER
```

Метод класса receive() доступен всем классам Action Mailer. Он берет текст сообщения электронной почты, переводит его в объект Mail, создает новый экземпляр класса получателя и передает объект Mail методу экземпляра receive(), имеющемуся в этом классе.

13.2. Шаг 32: комплексное тестирование приложений

Rails систематизирует тесты, разделяя их на тесты модели, контроллера и комплексные тесты. Прежде чем объяснить суть комплексных тестов, давайте кратко повторим то, что уже было рассмотрено до сих пор.

Блочное тестирование моделей

Классы моделей содержат бизнес-логику. Например, когда мы добавляем товар в корзину, класс модели корзины осуществляет проверку с целью определения наличия такого же товара среди уже имеющихся в корзине товарных позиций. Если такой товар там уже имеется, то увеличивается количественный показатель соответствующей товарной позиции, а если нет, добавляется новая товарная позиция для данного товара.

Функциональное тестирование контроллеров

Контроллеры управляют показом. Они получают входящие веб-запросы (обычно пользовательский ввод), взаимодействуют с моделями для сбора состояния приложения, а затем составляют ответ, вызывая для пользователя отображение соответствующего представления.

Таким образом, при тестировании контроллеров мы убеждаемся в том, что заданный запрос получает соответствующий ответ. Нам по-прежнему нужны модели, но мы уже охватили их блочными тестами.

Следующий уровень тестирования заключается в использовании всего цикла работы нашего приложения. Во многом это похоже на тестирование одной из

историй, которую нам рассказал заказчик, когда мы только приступали к созданию кода приложения.

Например, мы могли услышать следующее:

> *«Пользователь заходит на страницу каталога магазина. Он выбирает товар, добавляя его в свою корзину. Затем он оформляет заказ, заполняя форму заказа. Затем он отправляет данные формы, в базе данных создается заказ, содержащий информацию о пользователе наряду с одной товарной позицией, соответствующей товару, добавленному пользователем в корзину. Как только заказ был получен, отправляется сообщение электронной почты, подтверждающее покупку».*

Это идеальный материал для комплексного теста. Комплексные тесты имитируют продолжительный сеанс работы нашего приложения с одним или несколькими виртуальными пользователями. Их можно использовать для отправки запросов, отслеживания реакции, следования перенаправления и т. д.

При создании модели или контроллера Rails создает соответствующий блочный или функциональный тест. Комплексные тесты автоматически не создаются, но для создания такого теста можно использовать генератор.

```
depot> rails generate integration_test user_stories
invoke test_unit
create test/integration/user_stories_test.rb
```

Следует учесть, что Rails автоматически добавляет к имени теста окончание `_test`.

Посмотрим на сгенерированный файл:

```
require 'test_helper'

class UserStoriesTest < ActionDispatch::IntegrationTest
  # test "the truth" do
  #   assert true
  # end
end
```

Давайте приступим к созданию теста для нашей истории. Поскольку тестироваться будет только покупка товара, нам понадобятся только лишь стендовые данные товаров — `products`.

Поэтому вместо загрузки всех стендовых данных, загрузим только одну группу:

```
fixtures :products
```

Теперь создадим тест по имени «покупка товара». Мы знаем, что к концу теста нам нужно, чтобы заказ был добавлен к таблице заказов `orders`, а товарная позиция была добавлена к таблице `line_items`, поэтому сначала давайте очистим эти таблицы от данных. И, поскольку мы часто использовали стендовые данные о книге по Ruby, давайте загрузим эти данные в локальную переменную:

rails40/depot_r/test/integration/user_stories_test.rb

```
LineItem.delete_all
Order.delete_all
ruby_book = products(:ruby)
```

Давайте возьмемся за первое предложение из нашей истории: «Пользователь заходит на страницу каталога магазина».

rails40/depot_r/test/integration/user_stories_test.rb

```
get "/"
assert_response :success
assert_template "index"
```

Это выглядит почти как функциональный тест. Основное различие заключается в методе `get`. В функциональном тесте мы проверяем наш контроллер, поэтому при вызове метода `get()` мы указываем только действие. Но в комплексном тесте мы можем путешествовать по всему приложению, поэтому нам нужно передать этому методу относительно полный URL-адрес для контроллера и вызываемого действия.

Следующее предложение в нашей истории гласит: «Он выбирает товар, добавляя его в свою корзину». Мы знаем, что наше приложение для добавления товаров в корзину использует AJAX-запросы, поэтому для вызова действия мы будем использовать метод `xml_http_request()`. Когда метод вернет управление, мы проверим факт наличия в корзине запрошенного товара:

rails40/depot_r/test/integration/user_stories_test.rb

```
xml_http_request :post, '/line_items', product_id: ruby_book.id
assert_response :success

cart = Cart.find(session[:cart_id])
assert_equal 1, cart.line_items.size
assert_equal ruby_book, cart.line_items[0].product
```

Далее в закрученном сюжете нашей истории попадается фраза: «Затем он оформляет заказ...». В нашем тесте это легко проверить:

Rails40/depot_r/test/integration/user_stories_test.rb

```
get "/orders/new"
assert_response :success
assert_template "new"
```

К этому времени пользователь заполнил форму заказа своими данными. Как только это произошло и данные были отправлены, наше приложение создает заказ и переходит на страницу каталога. Давайте начнем со стороны HTTP, отправив данные формы действию `save_order` и проверив, что мы были перенаправлены на главную страницу. Мы также проверим, что наша корзина теперь пуста. Вспомогательный тестовый метод `post_via_redirect()` генерирует post-запрос, а затем следует любым возвращенным перенаправлениям, пока не будет возвращен ответ без перенаправления.

rails40/depot_r/test/integration/user_stories_test.rb

```
post_via_redirect "/orders",
                  order: { name:         "Dave Thomas",
                           address:     "123 The Street",
```

```
                        email:       "dave@example.com",
                        pay_type:    "Check" }
assert_response :success
assert_template "index"
cart = Cart.find(session[:cart_id])
assert_equal 0, cart.line_items.size
```

Затем мы заглянем в базу данных и убедимся, что мы создали заказ и соответствующую товарную позицию и что в них содержатся верные сведения. Поскольку перед началом тестирования мы очистили таблицу `orders`, мы просто проверим, что теперь в ней содержится только лишь наш новый заказ:

rails40/depot_r/test/integration/user_stories_test.rb

```
orders = Order.all
assert_equal 1, orders.size
order = orders[0]

assert_equal "Dave Thomas", order.name
assert_equal "123 The Street", order.address
assert_equal "dave@example.com", order.email
assert_equal "Check", order.pay_type

assert_equal 1, order.line_items.size
line_item = order.line_items[0]
assert_equal ruby_book, line_item.product
```

И наконец, мы проверим, что само почтовое отправление правильно адресовано и имеет ожидаемую нами строку темы:

rails40/depot_r/test/integration/user_stories_test.rb

```
mail = ActionMailer::Base.deliveries.last
assert_equal ["dave@example.com"], mail.to
assert_equal 'Sam Ruby <depot@example.com>', mail[:from].value
assert_equal "Pragmatic Store Order Confirmation", mail.subject
```

Ну вот и все.

А вот полный исходный код комплексного теста:

rails40/depot_r/test/integration/user_stories_test.rb

```
require 'test_helper'

class UserStoriesTest < ActionDispatch::IntegrationTest
    fixtures :products

    # A user goes to the index page. They select a product, adding it to their
    # cart, and check out, filling in their details on the checkout form. When
    # they submit, an order is created containing their information, along with a
    # single line item corresponding to the product they added to their cart.

    test "buying a product" do
        LineItem.delete_all
        Order.delete_all
        ruby_book = products(:ruby)
```

```
        get "/"
        assert_response :success
        assert_template "index"

        xml_http_request :post, '/line_items', product_id: ruby_book.id
        assert_response :success

        cart = Cart.find(session[:cart_id])
        assert_equal 1, cart.line_items.size
        assert_equal ruby_book, cart.line_items[0].product

        get "/orders/new"
        assert_response :success
        assert_template "new"

        post_via_redirect "/orders",
                          order: { name:        "Dave Thomas",
                                   address:     "123 The Street",
                                   email:       "dave@example.com",
                                   pay_type:    "Check" }
        assert_response :success
        assert_template "index"
        cart = Cart.find(session[:cart_id])
        assert_equal 0, cart.line_items.size

        orders = Order.all
        assert_equal 1, orders.size
        order = orders[0]

        assert_equal "Dave Thomas",         order.name
        assert_equal "123 The Street",      order.address
        assert_equal "dave@example.com",    order.email
        assert_equal "Check",               order.pay_type

        assert_equal 1, order.line_items.size
        line_item = order.line_items[0]
        assert_equal ruby_book, line_item.product

        mail = ActionMailer::Base.deliveries.last
        assert_equal ["dave@example.com"], mail.to
        assert_equal 'Sam Ruby <depot@example.com>', mail[:from].value
        assert_equal "Pragmatic Store Order Confirmation", mail.subject
    end
end
```

Все вместе взятые блочные, функциональные и комплексные тесты позволяют вам провести гибкую проверку всех аспектов вашего приложения как по отдельности, так и в сочетании друг с другом. В разделе 25.3 «Поиск новых дополнений в RailsPlugins.org», будет рассказано, где можно будет найти дополнительные модули, переводящие все это на следующий уровень и позволяющие создавать простые текстовые описания поведения, которые может прочитать ваш заказчик и которые могут быть проверены в автоматическом режиме.

А теперь пора завершить выполнение этого шага, переговорить с заказчиком и выяснить, какие функциональные возможности ожидают наше приложение Depot.

Наши достижения

Использовав совсем небольшой объем кода и всего лишь несколько шаблонов,

- ☑ мы настроили все три среды нашего Rails-приложения: разработочную, тестировочную и эксплуатационную — на отправку исходящих сообщений электронной почты;
- ☑ мы создали и настроили почтовую систему, которая будет отправлять подтверждающие почтовые сообщения как в виде простого теста, так и в HTML-формате заказчикам нашего товара;
- ☑ мы создали как функциональный тест для создания сообщений электронной почты, так и комплексный тест, охватывающий весь сценарий, связанный с оформлением заказа.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи:

- ❍ Добавьте к таблице `orders` столбец `ship_date` (дата отправки) и отправьте уведомление, когда его значение будет обновлено контроллером `OrdersController`.
- ❍ Обновите приложение, чтобы оно отправляло сообщение электронной почты системному администратору — лично вам — при возникновении в приложении каких-нибудь отказов, например того отказа, который обрабатывался в разделе 10.2 «Шаг Д2: обработка ошибок».
- ❍ Добавьте комплексный тест для обоих предыдущих элементов.

Задача И: вход в административную область

14

Основные темы:

- добавление к моделям безопасных паролей;
- использование дополнительных проверок;
- добавление к сессии аутентификации;
- использование консоли `rails`;
- использование транзакций базы данных;
- написание метода отката с использованием Active Record.

Итак, заказчик доволен — в сравнительно короткие сроки мы совместными усилиями собрали воедино все необходимое для работы корзины покупателя, и он может приступать к ее демонстрации своим пользователям. Но ему захотелось увидеть еще одно дополнение. На данный момент получить доступ к административным функциям может кто угодно. Он хочет, чтобы мы добавили стандартную систему пользовательского администрирования, которая заставит вас пройти авторизацию для входа на административную часть веб-сайта.

Похоже, переговоры с заказчиком показали, что нам незачем создавать для приложения слишком сложную систему безопасности. Нужно лишь распознать некоторых людей, основываясь на пользовательских именах и паролях. Прошедшие систему опознавания получат доступ к административным функциям.

14.1. Шаг И1: добавление пользователей

Начнем с создания модели и таблицы базы данных, в которых будут содержаться имена и пароли пользователей с правами администратора. Пароли будут храниться не в виде простого текста, а в виде цифрового хэша. Тем самым мы гарантируем, что, даже при вскрытии базы данных, хэш не покажет исходный пароль и не сможет использоваться для входа в административную область в качестве пользователя с помощью форм.

```
depot> rails generate scaffold User name:string password:digest
```

Если вы запустили выпуск Rails, отличающийся от 3.1.0, обратитесь к wiki[1], чтобы увидеть, нет ли там каких-нибудь дополнительных инструкций о том, что нужно делать, чтобы включить метод `has_secure_password()`.

Мы объявили пароль (`password`) типа `digest`, и это еще одна из приятных особенностей, предоставляемых Rails. Теперь, как обычно, запустите миграцию:

```
depot> rake db:migrate
```

А теперь мы должны конкретизировать модель пользователя — `user`:

rails40/depot_r/app/models/user.rb

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

Мы проверяем наличие и уникальность имени (то есть в базе данных не должно быть повторяющихся имен).

Затем следует таинственный метод `has_secure_password()`.

Вам ведь знакомы такие формы, которые предлагают вам ввести пароль, а затем заставляют вас ввести его еще раз в отдельное поле, чтобы убедиться, что вы ввели именно то, что задумали? Именно этим для вас и занимается метод `has_secure_password()`: он заставляет Rails проверить совпадение двух паролей. Эта строка была добавлена потому, что при создании временной платформы вы указали `password:digest`.

Далее следует убрать символ комментария с указания gem-пакета `bcrypt-ruby` в вашем файле Gemfile.

rails40/depot_r/Gemfile

```
# Use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

Затем нужно установить gem-пакет.

```
depot> bundle install
```

И наконец, требуется перезапустить ваш сервер.

[1] http://pragprog.com/wikis/wiki/ChangesToRails

С этим кодом у нас появляется возможность показать в форме оба поля: пароля и его подтверждения, а также возможность провести аутентификацию пользователя с заданным именем и паролем.

Администрирование пользователей

В дополнение к настроенной модели и таблице у нас уже есть временная платформа, сгенерированная для администрирования модели. Давайте пройдемся по ней и проведем необходимые настройки.

Начнем с контроллера. В нем определяются стандартные методы: `index()`, `show()`, `new()`, `edit()`, `update()` и `delete()`.По умолчанию Rails не показывает непонятный пароль в представлении. Стало быть, при работе с пользователями нам особо нечего показывать с помощью метода `show()`, разве что имя. Поэтому исключим перенаправление на показ данных пользователя после операции создания. Вместо этого давайте осуществим перенаправление на перечень пользователей и добавим имя пользователя к флэш-уведомлению.

rails40/depot_r/app/controllers/users_controller.rb

```
def create
    @user = User.new(user_params)
    respond_to do |format|
        if @user.save
▶           format.html { redirect_to users_url,
▶               notice: "Пользователь #{@user.name} был успешно создан." }
            format.json { render action: 'show',
                status: :created, location: @user }
        else
            format.html { render action: 'new' }
            format.json { render json: @user.errors,
                status: :unprocessable_entity }
        end
    end
end
```

То же самое сделаем и для операции обновления:

```
def update
    respond_to do |format|
        if @user.update(user_params)
▶         format.html { redirect_to users_url,
▶         notice: "Сведения о пользователе #{@user.name} были успешно обновлены." }
          format.json { head :no_content }
        else
            format.html { render action: 'edit' }
            format.json { render json: @user.errors,
                status: :unprocessable_entity }
        end
    end
end
```

Раз уж мы здесь, выстроим пользователей, возвращаемых в перечне, по именам:

```
def index
    @users = User.order(:name)
end
```

После внесения изменений в контроллер обратимся к представлению. На данный момент представление не выводит уведомительную информацию, поэтому давайте ее добавим:

rails40/depot_r/app/views/users/index.html.erb

```
<h1>Listing users</h1>
    <% if notice %>
    <p id="notice"><%= notice %></p>
    <% end %>

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

<br>

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

И наконец, нам нужно обновить форму, используемую как для создания нового пользователя, так и для обновления сведений об уже имеющемся пользователе. Учтите, что теперь форма настроена на вывод полей пароля и подтверждения пароля. Чтобы улучшить внешний вид страницы, добавим теги `legend` и `fieldset`. И в завершение, поместим вывод в тег `<div>` с атрибутом `class`, значение которого мы ранее определили в нашей таблице стилей.

rails40/depot_r/app/views/users/_form.html.erb

```
<div class="depot_form">

<%= form_for @user do |f| %>
    <% if @user.errors.any? %>
```

```
        <div id="error_explanation">
            <h2><%= pluralize(@user.errors.count, "error") %>
                , сведения об этом пользователе не могут быть сохранены:</h2>
            <ul>
            <% @user.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
            <% end %>
            </ul>
        </div>
    <% end %>

    <fieldset>
    <legend>Введите сведения о пользователе</legend>

    <div class="field">
       <%= f.label :name, 'Name:' %>
       <%= f.text_field :name, size: 40 %>
    </div>

    <div class="field">
        <%= f.label :password, 'Пароль:' %>:
        <%= f.password_field :password, size: 40 %>
    </div>

    <div class="field">
        <%= f.label :password_confirmation, 'Подтверждение:' %>:
        <%= f.password_field :password_confirmation, size: 40 %>
    </div>

    <div class="actions">
        <%= f.submit %>
    </div>

    </fieldset>
<% end %>

</div>
```

Давайте испытаем это в работе. Перейдите на адрес http://localhost:3000/users/new. Наш выдающийся пример дизайна страницы показан на рис. 14.1.

После щелчка на кнопке Создать пользователя (Create User) перечень выводится еще раз с позитивным флэш-уведомлением:

```
depot> sqlite3 -line db/development.sqlite3 "select * from users"
            id = 1
            name = dave
    password_digest = $2a$10$lki6/oAcOW4AWg4A0e0T8uxtri2Zx5g9taBXrd4mDSDVl3rQRWRNi
         created_at = 2013-01-29 14:40:06.230622
         updated_at = 2013-01-29 14:40:06.230622
```

Так же как и раньше, нам необходимо обновить наши тесты, чтобы они отвечали внесенным изменениям в проверке и перенаправлении. Сначала обновим тест для метода `create()`:

rails40/depot_r/test/controllers/users_controller_test.rb

```
    test "should create user" do
        assert_difference('User.count') do
            post :create, user: { name: 'sam', password: 'secret',
               password_confirmation: 'secret' }
        end

        assert_redirected_to users_path
    end
```

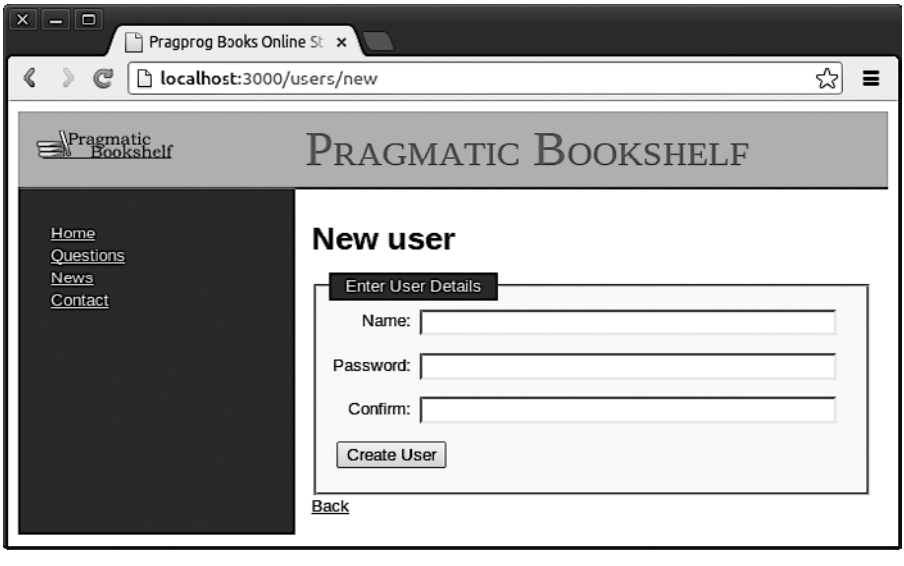


Рис. 14.1. Ввод сведений о пользователе

Поскольку изменилось также и перенаправление на метод `update()`, его тест также нуждается в изменении.

```
test "should update user" do
   patch :update, id: @user, user: { name: @user.name, password: 'secret',
   password_confirmation: 'secret' }
  assert_redirected_to users_path
end
```

Кроме этого, нужно обновить тестовые стенды, чтобы убедиться в отсутствии дубликатов имен.

rails40/depot_r/test/fixtures/users.yml

```
# Read about fixtures at
# http://api.rubyonrails.org/classes/ActiveRecord/Fixtures.html

one:
  name: dave
   password_digest: <%= BCrypt::Password.create('secret') %>
```

```
two:
  name: susannah
    password_digest: <%= BCrypt::Password.create('secret') %>
```

Заметьте, что в тестовых стендах используются динамически вычисляемые значения, в частности для значения `password_digest`. Этот код был также вставлен командой создания временной платформы, и в нем используется та же самая функция, которая в Rails служит для вычисления пароля.

Теперь, когда мы можем управлять нашими пользователями, нам нужно сначала провести аутентификацию пользователей, а затем ограничить доступ к административным функциям, предоставив его только пользователям с правами администратора.

14.2. Шаг И2: аутентификация пользователей

Что означает добавление поддержки входа в административную область для администраторов вашего магазина?

- Нам нужно предоставить форму, позволяющую им ввести имя пользователя и пароль.
- После входа в административную область нужно каким-то образом зафиксировать этот факт для всей остальной сессии (или до тех пор, пока они не выйдут из административной области).
- Нам нужно ограничить доступ к административной области приложения, разрешив управлять магазином только тем, кто вошел в административную область.

Всю логику можно поместить в один контроллер, но имеет смысл разбить ее на два контроллера: на контроллер сессии для поддержки входа в административную область и выхода из нее и на контроллер для работы администраторов.

```
depot> rails generate controller Sessions new create destroy
depot> rails generate controller Admin index
```

Действие `SessionsController#create` должно будет сделать запись где-нибудь в сессии, чтобы было известно, что в административную область вошел администратор. Пусть в ней хранится идентификатор ее объекта `User` с использованием ключа `:user_id`. Код входа в административную область имеет следующий вид:

rails40/depot_r/app/controllers/sessions_controller.rb

```
def create
    user = User.find_by(name: params[:name])
    if user and user.authenticate(params[:password])
        session[:user_id] = user.id
        redirect_to admin_url
```

```
    else
        redirect_to login_url, alert: "Неверная комбинация имени и пароля"
    end
end
```

Здесь мы делаем и кое-что новое: использование формы, которая не связана напрямую с объектом модели. Чтобы посмотреть, как это работает, рассмотрим шаблон для действия `sessions#new`:

rails40/depot_r/app/views/sessions/new.html.erb

```
<div class="depot_form">
    <% if flash[:alert] %>
        <p id="notice"><%= flash[:alert] %></p>
    <% end %>

    <%= form_tag do %>
        <fieldset>
            <legend>Зарегистрируйтесь, пожалуйста</legend>

            <div>
                <%= label_tag :name, 'Имя:' %>
                <%= text_field_tag :name, params[:name] %>
            </div>

            <div>
                <%= label_tag :password, 'Пароль:' %>
                <%= password_field_tag :password, params[:password] %>
            </div>

            <div>
                <%= submit_tag "Login" %>
            </div>
        </fieldset>
    <% end %>
</div>
```

Эта форма отличается от тех, которые мы видели раньше. Вместо использования метода `form_for` в ней используется метод `form_tag`, который просто со-здает обычную HTML-форму с тегами `<form>`. Внутри этой формы используются `text_field_tag` и `password_field_tag`, два вспомогательных метода, создающих HTML-теги `<input>`. Каждый вспомогательный метод получает два аргумента. Первый — это имя, которое дается полю, и второй — это значение, которым нужно заполнить поле. Эта разновидность формы позволяет нам связать значения в структуре `params` непосредственно с полями формы, при этом объект модели не требуется. В нашем случае мы выбрали использование объекта `params` непосредственно в форме. Альтернативным вариантом могло бы быть объявление контроллером переменных экземпляра.

Здесь также используются помощники `label_tag` для создания HTML-тегов `<label>`. Этот помощник также получает два аргумента. В первом содержится имя поля, а во втором — выводимая надпись.

Посмотрите на рис. 14.2 и обратите внимание на то, как значение поля формы передается между контроллером и представлением при помощи хэша `params`: представление получает значение для отображения в поле из `params[:name]`, и, когда пользователь отправляет данные формы, новое значение поля делается доступным контроллеру все тем же способом.

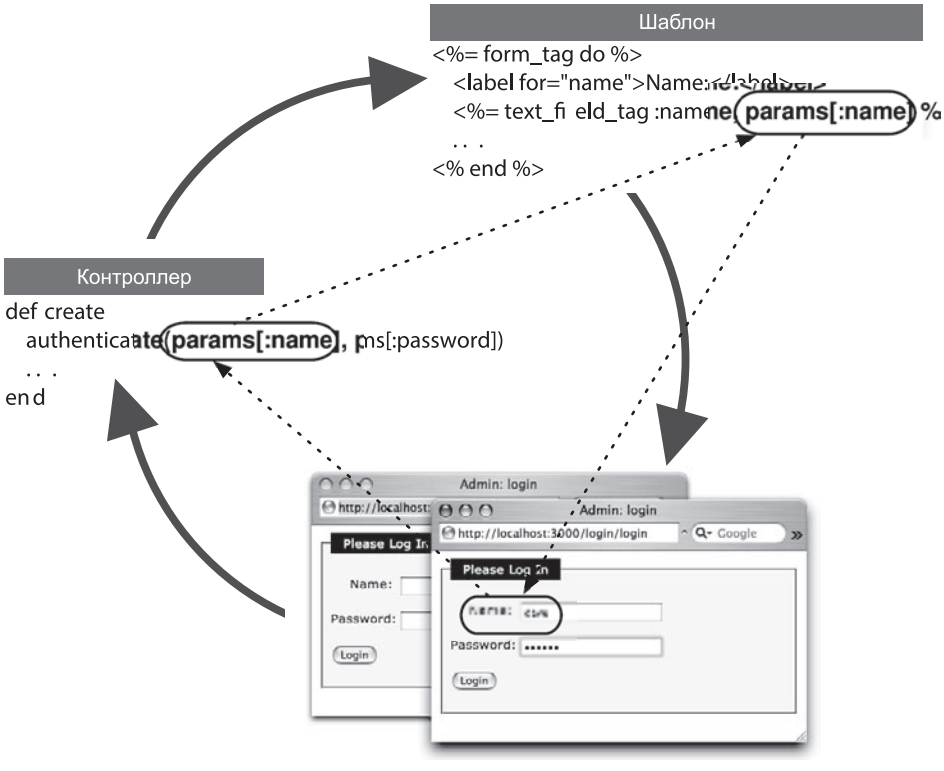


Рис. 14.2. Перенос аргументов между контроллерами, шаблонами и браузерами

Если пользователь успешно вошел в административную область, мы сохраняем идентификатор записи пользователя в данных сессии. Присутствие этого значения в сессии будет служить признаком того, что в административную область вошел пользователь с правами администратора.

Как вы, наверное, и ожидали, действия контроллера для выхода из административной области будут выглядеть намного проще:

rails40/depot_r/app/controllers/sessions_controller.rb

```
def destroy
▶    session[:user_id] = nil
▶    redirect_to store_url, notice: "Сеанс работы завершен"
end
```

И наконец, настала пора добавить страницу перечня, первый экран, который видит администратор после входа в административную область. Давайте заставим эту страницу приносить дополнительную пользу — показывать общее количество заказов, имеющихся в нашем магазине. Создайте шаблон в файле index.html.erb в каталоге app/views/admin. (В этом шаблоне используется помощник `pluralize()`, который в данном случае генерирует строку `order` или `orders` в зависимости от количества, указанного в его первом аргументе.)

rails40/depot_r/app/views/admin/index.html.erb

```
<h1>Добро пожаловать</h1>

Сейчас <%= Time.now %>
У нас имеется <%= pluralize(@total_orders, "order") %>.
```

Количество устанавливается в действии `index()`:

rails40/depot_r/app/controllers/admin_controller.rb

```
class AdminController < ApplicationController
    def index
▶       @total_orders = Order.count
    end
end
```

Перед тем как этим воспользоваться, нам нужно выполнить еще одну задачу. Если прежде мы зависели от генератора временной платформы, который создавал нашу модель и маршруты для нас, то на этот раз мы просто сгенерируем контроллер, поскольку для этого контроллера нет модели, основанной на базе данных. К сожалению, если не следовать соглашению, применяемому при генерации временных платформ, Rails не может узнать, какие действия для этого контроллера отвечают за GET-запросы, а какие отвечают за POST-запросы и т. д.

Нам нужно предоставить эту информацию путем редактирования нашего файла config/routes.rb:

rails40/depot_r/config/routes.rb

```
Depot::Application.routes.draw do
▶   get 'admin' => 'admin#index'
▶   controller :sessions do
▶       get 'login' => :new
▶       post 'login' => :create
▶       delete 'logout' => :destroy
▶   end

    get "sessions/create"
    get "sessions/destroy"
    resources :users
    resources :orders
    resources :line_items
    resources :carts

    get "store/index"
    resources :products do
```

```
        get :who_bought, on: :member
    end
    # The priority is based upon order of creation:
    # first created -> highest priority.
    # See how all your routes lay out with "rake routes".
    # You can have the root of your site routed with "root"
    root to: 'store#index', as: 'store'
    # ...
end
```

Мы уже сталкивались с этим, когда добавляли инструкцию `root` в разделе 8.1, «Шаг B1: создание каталога товаров». Команда генерации добавит к этому файлу только универсальные инструкции `get` для каждого указанного действия. Вы можете (и должны) удалить маршруты, предоставляемые для sessions/new, sessions/create и sessions/destroy.

В случае использования admin мы сократим URL-адрес, который пользователю нужно ввести (удалив часть /index), и отобразим его на полное действие. В случае действий, связанных с сессией, мы полностью изменим URL-адрес (заменив адреса вида session/create простым адресом login), а также перенастроим соответствующее HTTP-действие. Заметьте, что login отображается на оба действия, `new` и `create`, различающиеся запросом: HTTP GET или HTTP POST.

Мы также воспользуемся кратчайшим путем: поместим объявления маршрутов сессии в блок и передадим его методу класса `controller()`. Это сократит количество набираемых символов и упростит чтение маршрутов. Все, что вы можете сделать в этом файле, будет рассмотрено в разделе 20.1 «Диспетчеризация запросов к контроллерам».

Располагая этими маршрутами, мы можем испытать радость входа в область администрирования на правах администратора и увидеть окно, показанное на рис. 14.3.

Рис. 14.3. Интерфейс администрирования

Нам нужно заменить функциональные тесты в контроллере сессии, чтобы привести все в соответствие с только что реализованными функциями:

rails40/depot_r/test/functional/sessions_controller_test.rb

```
require 'test_helper'

class SessionsControllerTest < ActionController::TestCase
    test "should get new" do
        get :new
        assert_response :success
    end

▶   test "should login" do
▶       dave = users(:one)
▶       post :create, name: dave.name, password: 'secret'
▶       assert_redirected_to admin_url
▶       assert_equal dave.id, session[:user_id]
▶   end

▶   test "should fail login" do
▶       dave = users(:one)
▶       post :create, name: dave.name, password: 'wrong'
▶       assert_redirected_to login_url
▶   end

▶   test "should logout" do
▶       delete :destroy
▶       assert_redirected_to store_url
▶   end
end
```

Мы показали последние результаты заказчику, и он заметил, что мы до сих пор не контролируем доступ к страницам области администрирования (что в конечном счете и было целью всех этих усилий).

14.3. Шаг И3: ограничение доступа

Нам нужно отказать в доступе на страницы области администрирования всем, кто не зарегистрировался в качестве администратора. Оказывается, упростить реализацию этого замысла нам помогут имеющиеся в Rails *обратные вызовы* (callback).

Обратные вызовы в Rails позволяют перехватывать вызовы методов действий, добавляя свою собственную обработку перед тем, как методы будут реально вызваны, после окончания их работы или и в том, и другом случае. Сейчас мы воспользуемся обратным вызовом *перед действием,* чтобы перехватить все вызовы действий, находящиеся в нашем контроллере `admin`. Перехватчик может проверять значение `session[:user_id]`. Если оно установлено и соответствует пользователю, зарегистрированному в базе данных, приложение знает, что в систему вошел администратор и вызов можно продолжить. Если значение не установлено, перехватчик может осуществить переадресацию, в нашем случае на страницу авторизации.

А куда же нам следует поместить этот метод? Он может находиться непосредственно в контроллере `admin`, но в силу причин, очевидность которых скоро станет

понятна, мы поместим его вместо этого в ApplicationController — родительский класс всех наших контроллеров. Он находится в файле application_controller.rb каталога app/controllers. К тому же обратите внимание на то, что нам нужно ограничить доступ к этому методу, чтобы он не был виден в качестве действия конечным пользователям.

rails40/depot_r/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
    before_action :authorize
    # ...

    protected

        def authorize
            unless User.find_by(id: session[:user_id])
            redirect_to login_url, notice: "Пожалуйста, пройдите авторизацию"
        end
    end
end
```

Строка before_action() заставляет вызывать метод authorize() перед каждым действием в нашем приложении.

Это уже перебор. Нам нужно всего лишь ограничить доступ к области администрирования магазина, поэтому такое положение вещей нас не устраивает.

Мы можем вернуться и пометить только те методы, которые конкретно нуждаются в авторизации. Такой подход называется занесением в черный список, но при его использовании можно допустить ошибку и что-нибудь пропустить. Намного лучше применить подход «белого списка», то есть перечисления методов или контроллеров, для которых авторизация не требуется. Мы сделаем это путем простой вставки метода skip_before_action() в StoreController:

rails40/depot_r/app/controllers/store_controller.rb

```
class StoreController < ApplicationController
    skip_before_action :authorize
```

То же самое мы сделаем и для класса SessionsController:

rails40/depot_r/app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController
    skip_before_action :authorize
```

Но это еще не все. Теперь нам нужно разрешить создавать, обновлять и удалять корзины:

rails40/depot_r/app/controllers/carts_controller.rb

```
class CartsController < ApplicationController
    skip_before_action :authorize, only: [:create, :update, :destroy]
     # ...
     private
     # ...
```

```
        def invalid_cart
           logger.error "Attempt to access invalid cart #{params[:id]}"
           redirect_to store_url, notice: 'Invalid cart'
        end
end
```

И мы разрешим им создавать товарные позиции:

rails40/depot_r/app/controllers/line_items_controller.rb

```
class LineItemsController < ApplicationController
    skip_before_action :authorize, only: :create
```

и создавать заказы (включая доступ к new):

rails40/depot_r/app/controllers/orders_controller.rb

```
class OrdersController < ApplicationController
    skip_before_action :authorize, only: [:new, :create]
```

Располагая логикой авторизации, теперь мы можем перейти по адресу http://localhost:3000/products. Метод обратного вызова осуществит перехват на пути к перечню товаров и покажет вместо него экран входа в область администрирования.

К сожалению, внесенное изменение сильно испортило большинство наших функциональных тестов, поскольку большинство операций вместо выполнения желаемой функции теперь будет перенаправляться на экран входа в административную область. Но, к счастью, мы можем воспользоваться глобальным обращением, создав в test_helper метод `setup()`. И раз уж мы оказались в этом файле, определим заодно и несколько вспомогательных методов, позволяющих пользователю войти в административную область под определенным именем — `login_as()` и выйти из нее — `logout()`.

rails40/depot_r/test/test_helper.rb

```
class ActiveSupport::TestCase
    # ...

    # Add more helper methods to be used by all tests here...
    def login_as(user)
        session[:user_id] = users(user).id
    end

    def logout
        session.delete :user_id
    end

    def setup
        login_as :one if defined? session
    end
end
```

Обратите внимание на то, что метод `setup()` будет вызывать `login_as()`, только если определена сессия. Тем самым авторизация не будет исполняться

в тестах, которые не задействуют контроллер. Мы показали результаты заказчику и были награждены его широкой улыбкой и новым запросом: нельзя ли добавить боковую панель, поместив на нее ссылки на функции администрирования пользователей и товаров? И раз уж мы этим займемся, нельзя ли добавить возможность просмотра перечня и удаления пользователей с правами администратора? Конечно, можно!

14.4. Шаг И4: добавление боковой панели и дополнительных административных функций

Начнем с добавления в макет ссылок на различные функции администрирования на боковую панель и обеспечения их вывода только в том случае, если в сессии есть сведения об идентификаторе пользователя — `:user_id`:

rails40/depot_r/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
    <title>Pragprog Books Online Store</title>
    <%= stylesheet_link_tag "application", media: "all",
        "data-turbolinks-track" => true %>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tag %>
</head>
<body class="<%= controller.controller_name %>">
    <div id="banner">
        <%= image_tag("logo.png") %>
        <%= @page_title || "Pragmatic Bookshelf" %>
    </div>
    <div id="columns">
        <div id="side">
            <% if @cart %>
                <%= hidden_div_if(@cart.line_items.empty?, id: 'cart') do %>
                    <%= render @cart %>
                <% end %>
            <% end %>
            <ul>
                <li><a href="http://www....">Home</a></li>
                <li><a href="http://www..../faq">Questions</a></li>
                <li><a href="http://www..../news">News</a></li>
                <li><a href="http://www..../contact">Contact</a></li>
            </ul>
▶
▶           <% if session[:user_id] %>
▶               <ul>
▶                   <li><%= link_to 'Orders', orders_path %></li>
▶                   <li><%= link_to 'Products', products_path %></li>
```

```
▶                    <li><%= link_to 'Users', users_path %></li>
▶                </ul>
▶                <%= button_to 'Logout', logout_path, method: :delete %>
▶            <% end %>
         </div>
         <div id="main">
             <%= yield %>
         </div>
     </div>
 </body>
 </html>
```

Теперь все начало собираться воедино. Мы можем войти в административную область и, щелкнув на ссылке в боковой панели, просмотреть перечень пользователей. Давайте посмотрим, не сломано ли что-нибудь при этом.

Давайте все-таки оставим последнего администратора

Мы вызываем экран списка, который изображен на рис. 14.4, затем щелкаем на ссылке Удалить, которая стоит рядом с именем `dave`, чтобы удалить этого пользователя. Будучи уверены, что наш пользователь удален, мы с удивлением обнаруживаем, что попали на экран входа в административную область. Мы только что удалили из системы единственного пользователя. При поступлении следующего запроса авторизация не удалась, поэтому приложение отказалось нас впускать в административную область. Перед тем как воспользоваться функциями администрирования, мы должны снова войти в административную область. Но теперь перед нами возникла непростая проблема: в базе данных не осталось ни одного пользователя с правами администратора, поэтому мы не можем пройти авторизацию.

К счастью, мы можем быстренько добавить пользователя к базе данных из командной строки. Если вызвать команду `rails console`, Rails вызовет утилиту Ruby irb, но сделает это в контексте вашего Rails-приложения. А это означает, что вы можете взаимодействовать с кодом своего приложения, набирая инструкции Ruby и наблюдая за возвращаемыми значениями.

Мы можем воспользоваться этим обстоятельством для непосредственного вызова нашей модели `user`, заставив ее добавить для нас пользователя в базу данных.

```
depot> rails console
Loading development environment.
>> User.create(name: 'dave', password: 'secret',
       password_confirmation: 'secret')
=> #<User:0x2933060 @attributes={...} ... >
>> User.count
=> 1
```

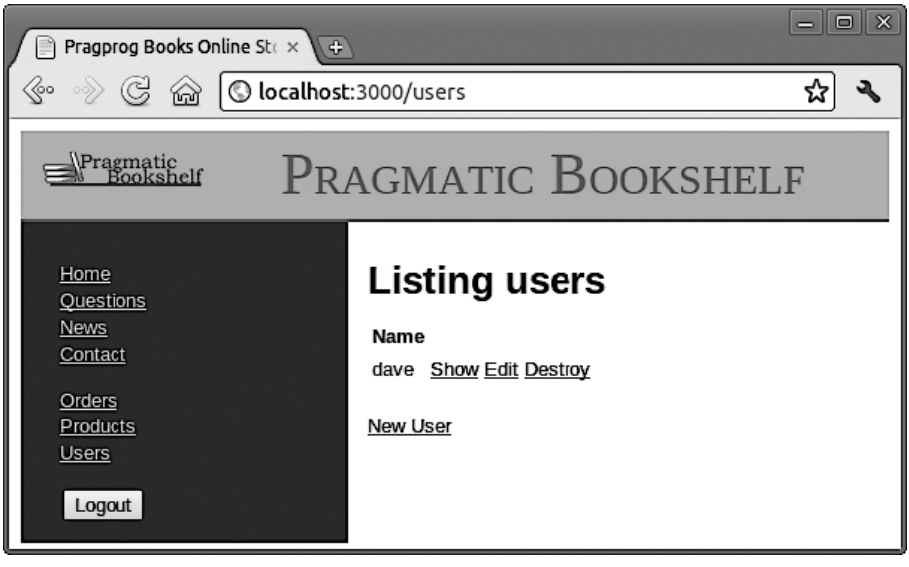


Рис. 14.4. Вывод списка пользователей

Последовательность символов `>>` представляет собой приглашение на ввод данных. После появления первого приглашения мы вызываем класс `User`, чтобы создать нового пользователя, а после появления второго приглашения мы вызываем его еще раз, чтобы отобразить тот факт, что у нас действительно есть один пользователь, занесенный в базу данных. После каждой введенной команды консоль Rails выводит возвращенное кодом значение (в первом случае это объект модели, а во втором — показания счетчика).

Паника закончилась, и мы можем опять пройти авторизацию и войти в административную область. Но как предотвратить повторение подобной ситуации? Для этого есть несколько способов. Например, мы можем написать код, предохраняющий нас от удаления нашей собственной учетной записи. Но этот прием может не сработать: теоретически некто А может удалить какого-то Б, а в то же самое время некто Б может удалить этого А. Применим другой подход. Мы будем удалять пользователя в процессе транзакции базы данных. Если после удаления пользователя в базе данных никого не останется, мы отменим транзакцию, восстановив пользователя, который только что был удален.

Для этого мы воспользуемся методом отката, принадлежащим Active Record. Один из таких методов мы уже рассматривали: откат `validate` вызывался Active Record для проверки состояния объекта. Оказывается, Active Record определяет около шестнадцати методов отката, каждый из которых вызывается в определенной точке жизненного цикла объекта. Мы воспользуемся откатом `after_destroy()`, который вызывается после выполнения SQL-инструкции `delete`. Если метод с таким именем является открытым и находится в области видимости, то он, что для нас вполне подходит, будет вызван в той же самой транзакции, что

и `delete`, поэтому, если он выдает исключение, транзакция будет отменена. Метод отката выглядит следующим образом:

rails40/depot_s/app/models/user.rb

```
after_destroy :ensure_an_admin_remains

private
    def ensure_an_admin_remains
        if User.count.zero?
            raise "Последний пользователь не может быть удален"
        end
    end
```

Для нас ключевым положением является использование исключения для индикации ошибки при удалении пользователя. Это исключение преследует две цели. Во-первых, поскольку оно выдается внутри транзакции, то становится причиной автоматического отката. Выдавая исключение в том случае, если таблица `users` после удаления опустошается, мы отменяем операцию удаления и возвращаем на место запись, относящуюся к этому последнему пользователю.

Во-вторых, исключение сигнализирует контроллеру об ошибке там, где мы используем блок `begin-end` для его обработки, и выводит пользователю флэш-сообщение об ошибке. Если нужно только отменить транзакцию, но не сигнализировать о возникновении исключения, следует вместо этого выдать исключение `ActiveRecord::Rollback`, поскольку это единственное исключение, которое не будет передано кодом `ActiveRecord::Base.transaction`.

rails40/depot_s/app/controllers/users_controller.rb

```
def destroy
    begin
        @user.destroy
        flash[:notice] = "Пользователь #{@user.name} удален"
    rescue StandardError => e
        flash[:notice] = e.message
    end

    respond_to do |format|
        format.html { redirect_to users_url }
        format.json { head :no_content }
    end
```

В этом коде все же сохраняется потенциальная проблема синхронизации: не исключено, что два администратора будут синхронно удалять двух последних пользователей. Но для устранения этой проблемы понадобятся такие тонкости при работе с базой данных, для которых в данной книге места не нашлось.

На самом деле рассмотренная в данной главе система входа в административную область довольно примитивна. В большинстве современных приложений для этого используется специальный дополнительный модуль. Существует сразу несколько дополнительных модулей, предоставляющих уже готовые решения, которые не только мощнее той логики аутентификации, которая была здесь показана,

но в целом требуют меньшего объема кода и усилий, предпринимаемых с вашей стороны. Несколько примеров будут показаны в разделе 26.3 «Поиск новых дополнений в RailsPlugins.org».

Наши достижения

К окончанию данного шага нам удалось сделать следующее.

- ☑ Воспользоваться методом `has_secure_password` для хранения закодированной версии пароля в базе данных.
- ☑ Установить контролируемый доступ к функциям администрирования с использованием обратных вызовов перед действием для вызова метода `authorize()`.
- ☑ Посмотреть, как используется команда `rails console` для непосредственного взаимодействия с моделью (и для вытаскивания нас из ямы, в которую мы провалились, удалив последнего пользователя).
- ☑ Посмотреть, как транзакция может помочь в предотвращении удаления последнего пользователя.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- ❍ Модифицируйте функцию обновления данных о пользователе, чтобы она запрашивала и проверяла текущий пароль, прежде чем позволить пользователю сменить этот пароль.
- ❍ Когда система только что установлена на новой машине, в базе данных нет никаких сведений об администраторах, следовательно, ни один администратор не может войти в административную область. Но если на это не способен ни один администратор, следовательно, никто не может создать пользователя с правами администратора.
- ❍ Внесите в код изменения, позволяющие при отсутствии в базе данных сведений об администраторах входить в административную область под любым именем пользователя (что позволит быстро создать настоящего пользователя с правами администратора).
- ❍ Поэкспериментируйте с командой `rails console`. Попробуйте создать товары, заказы и товарные позиции. Проследите за возвращаемым значением при сохранении объекта модели — когда проверка допустимости не будет пройдена, вы увидите, что будет возвращено значение `false` returned. Определите причины следующих ошибок:

```
>> prd = Product.new
=> #<Product id: nil, title: nil, description: nil, image_url:
nil, created_at: nil, updated_at: nil, price:
```

```
#<BigDecimal:246aa1c,'0.0',4(8)>>
>> prd.save
=> false
>> prd.errors.full_messages
=> ["Image url must be a URL for a GIF, JPG, or PNG image",
"Image url can't be blank", "Price should be at least 0.01",
"Title can't be blank", "Description can't be blank"]
```

- Посмотрите на метод `authenticate_or_request_with_http_basic()` и примените его в своем обратном вызове `:authorize`, если `request.format` не относится к `Mime::HTML`. Проверьте его работоспособность путем доступа к RSS-каналу Atom:

```
curl --silent --user dave:secret \
    http://localhost:3000/products/2/who_bought.atom
```

- Добившись работоспособности наших тестов при входе в административную область, мы еще не написали тесты, проверяющие, что доступ к конфиденциальным данным требует входа в эту область. Напишите хотя бы один тест, проверяющий это путем вызова метода `logout()` с последующей попыткой извлечь и обновить некоторые данные, требующие для этого аутентификации пользователя.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

15 Задача K: локализация

Основные темы:

- шаблоны локализации;
- разбор конструкции базы данных для I18n.

Теперь у нас есть работоспособная основная корзина покупателя, и наш заказчик начинает рассуждать о других языках, кроме английского, заметив, что у компании есть хорошие перспективы на развивающихся рынках. Пока мы не сможем предоставить что-нибудь на языке, понятном посетителям веб-сайта нашего заказчика, наш покупатель оставит деньги у себя. Допустить мы этого не можем.

Первая проблема заключается в том, что никто из нас не является профессиональным переводчиком. Заказчик убеждает нас, что беспокоиться не о чем, поскольку для этой части работы будет нанят сторонний специалист. На нас возлагается только *возможность* перевода. Более того, нам не стоит пока волноваться о страницах администрирования, поскольку все наши администраторы говорят по-английски. А нам нужно сконцентрироваться на магазине.

ДЖО СПРАШИВАЕТ: НУЖНО ЛИ ЧИТАТЬ ЭТУ ГЛАВУ, ЕСЛИ Я СОБИРАЮСЬ ИСПОЛЬЗОВАТЬ ТОЛЬКО ОДИН ЯЗЫК?

Если коротко, то нет. Фактически, многие Rails-приложения предназначены для небольших или однородных групп и никогда не нуждаются в переводе. Но следует сказать, что практически каждый, обнаруживший необходимость в переводе, соглашается с тем, что было бы лучше, если бы такая возможность была заложена как можно раньше. Поэтому, если вы не уверены в том, что перевод никогда не понадобится, мы рекомендуем вам хотя бы получить представление, о том, что за этим стоит, чтобы вы могли принять взвешенное решение.

Уже проще, но задача все равно нелегкая. Нам надо предоставить способ, позволяющий пользователю выбрать язык, предоставить сам перевод, а также внести изменения и в представления, чтобы воспользоваться этим переводом. Мы готовы к выполнению задачи, вооружены скромными познаниями в испанском на уровне средней школы — пора действовать!

15.1. Шаг К1: выбор региона

Начнем с нового файла конфигурации, в котором заключаются наши знания о доступных регионах и о том, какой из них будет использован по умолчанию.

rails40/depot_s/config/initializers/i18n.rb

```
#encoding: utf-8
I18n.default_locale = :en

LANGUAGES = [
    ['English', 'en'],
    ["Espa&ntilde;ol".html_safe, 'es']
]
```

Этот код выполняет две задачи.

Прежде всего, он приводит к использованию модуля `I18n` для установки региона по умолчанию. `I18n` выглядит забавно, но зато обеспечивает неизменно правильный набор термина, эквивалентного слову *internationalization* (локализация). Ведь слово «internationalization» начинается на «i», заканчивается на «n» и имеет восемнадцать букв между ними.

Затем определяется список связей между отображаемыми и локальными именами. К сожалению, на данный момент мы располагаем только лишь клавиатурой с раскладкой, предназначенной для США, а в слове español имеется символ, который невозможно ввести с нашей клавиатуры напрямую. Разные операционные системы обладают разными способами решения этой проблемы, и зачастую проще всего скопировать и вставить правильный текст с веб-сайта. При этом нужно убедиться в том, что ваш редактор настроен для работы с UTF-8. Тем временем мы решили использовать HTML-эквивалент испанского символа «n с тильдой». Если не сделать ничего другого, будет показана сама разметка. Но за счет вызова метода `html_safe` мы информируем Rails о том, что строка может быть безопасно интерпретирована как содержащая HTML.

Чтобы Rails восприняла эти изменения конфигурации, нужно перезапустить сервер.

Поскольку каждая страница, подвергаемая переводу, будет иметь `en-` и `es-` версию (на данный момент многое еще предстоит добавить), есть смысл включить это в URL-адрес. Давайте спланируем поместить указание на регион авансом, сделать его необязательным и по умолчанию использовать текущий регион, который, в свою очередь, будет по умолчанию настроен на английский.

Для реализации этого хитрого плана давайте начнем с изменения файла config/routes.rb:

rails40/depot_s/config/routes.rb

```
Depot::Application.routes.draw do
    get 'admin' => 'admin#index'

    controller :sessions do
        get 'login' => :new
        post 'login' => :create
        delete 'logout' => :destroy
    end

    get "sessions/create"
    get "sessions/destroy"

    resources :users

    resources :products do
      get :who_bought, on: :member
    end

▶   scope '(:locale)' do
       resources :orders
       resources :line_items
       resources :carts
       root 'store#index', as: 'store', via: :all
▶   end
end
```

Здесь мы вложили наши объявления ресурсов и исходных точек входа в объявление области видимости для `:locale`. Кроме того, обозначение `:locale` взято в круглые скобки, что является способом сообщить о его необязательном характере. Обратите внимание на то, что мы не стали помещать в эту область видимости административные функции и функции сессии, поскольку пока не собираемся их переводить.

Это значит, что при наборе адреса http://localhost:3000/ будет использован регион по умолчанию (English), и поэтому будет указан точно такой же маршрут, как и при указании адреса http://localhost:3000/en. При наборе адреса http://localhost:3000/es маршрут приведет к тому же контроллеру и действию, но нам нужно будет, чтобы это вызвало другую установку локализации.

Здесь мы внесли множество изменений в config.routes, а с вложенными и всеми дополнительными частями пути представить себе структуру довольно сложно. Но вам беспокоиться не о чем: при запуске сервера в разработочном режиме Rails предоставляет средство визуализации. Нужно лишь перейти по адресу http://localhost:3000/rails/info/routes, и, как показано на рис. 15.1, будет виден перечень всех ваших маршрутов. Дополнительные сведения о полях, показанных в таблице, можно найти в описании команды `rake routes` в главе 20, в разделе «REST: передача репрезентативного состояния».

Разобравшись с маршрутизацией, можно заняться извлечением местоположения из параметров и предоставлением его приложению.

Для этого нужно создать обратный вызов `before_action` и установить `default_url_options`. Логика для выполнения всего этого помещается в общий базовый класс для всех наших контроллеров, то есть в `ApplicationController`.

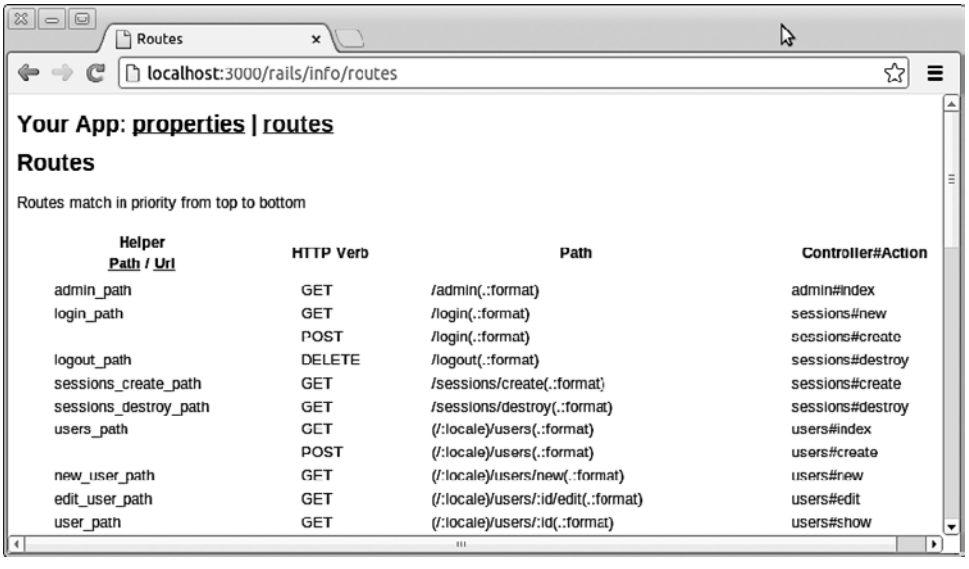


Рис. 15.1. Перечень всех активных маршрутов

rails40/depot_s/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
    before_action :set_i18n_locale_from_params
    # ...
    protected
        def set_i18n_locale_from_params
            if params[:locale]
                if I18n.available_locales.map(&:to_s).include?(params[:locale])
                    I18n.locale = params[:locale]
                else
                    flash.now[:notice] =
                        "#{params[:locale]} translation not available"
                                        # перевод недоступен
                    logger.error flash.now[:notice]
                end
        end
    end

    def default_url_options
        { locale: I18n.locale }
    end
end
```

Этот метод `set_i18n_locale_from_params` делает практически все в соответствии со своим названием: устанавливает локальные настройки из параметров, но

только в том случае, если они там есть, в противном случае он оставляет текущие локальные настройки без изменений. Позаботился он и об оповещении в случае невозможности установки локальных настроек как пользователя, так и администратора.

Метод `default_url_options` также делает практически то, что указано в его имени, — предоставляет хэш из URL-дополнений, которые считаются предоставленными, если они на самом деле не предоставлены. В данном случае мы предоставляем значение для параметра `:locale`. Это необходимо, когда представление, находящееся на странице, не имеющей указаний на регион, пытается составить ссылку на страницу, имеющую такие указания. Как это используется, мы скоро увидим.

При наличии всего этого мы получим результат, показанный на рис. 15.2.

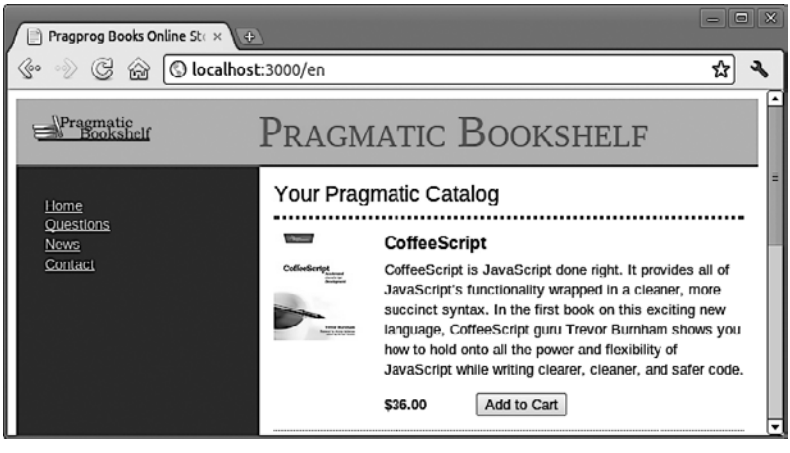


Рис. 15.2. Английская версия первой страницы

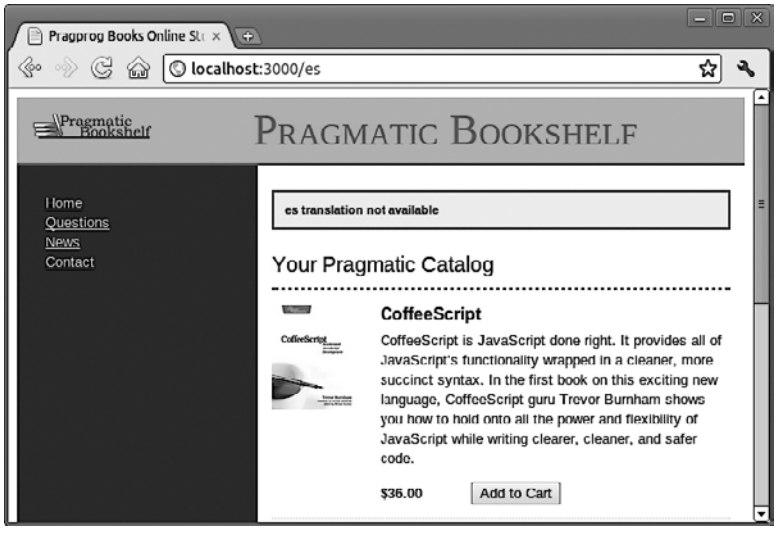


Рис. 15.3. Перевод недоступен

На данный момент английская версия страницы доступна при обращении как к главной странице веб-сайта, так и к странице, начинающейся с /en. Кроме этого, у нас еще есть код, выводящий на экран сообщение о том, что перевод недоступен (показанное на рис. 15.3), а также оставляющий в регистрационном журнале сообщение, свидетельствующее о том, что файл не был найден. Оно не передает именно этот смысл, но все равно приносит пользу.

15.2. Шаг К2: перевод каталога товаров

Настало время предоставить переведенный текст. Начнем с макета, как с самого наглядного кода. Заменим любой тест, нуждающийся в переводе, вызовом `I18n.translate`. У этого метода не только есть удобный псевдоним `I18n.t`, но для него также предоставлен помощник по имени `t`.

Аргумент для функции перевода является уникальным именем, обозначенным точкой. Можно выбрать любое понравившееся имя, но если используется предоставленная вспомогательная функция `t`, имена, начинающиеся с точки, должны быть сначала расширены с использованием имени шаблона. Давайте так и сделаем.

rails40/depot_s/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
    <title>Pragprog Books Online Store</title>
    <%= stylesheet_link_tag "application", media: "all",
      "data-turbolinks-track" => true %>
    <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
    <%= csrf_meta_tags %>
</head>
<body class="<%= controller.controller_name %>">
    <div id="banner">
        <%= image_tag("logo.png") %>
▶       <%= @page_title || t('.title') %>
    </div>
    <div id="columns">
        <div id="side">
            <% if @cart %>
                <%= hidden_div_if(@cart.line_items.empty?, id: 'cart') do %>
                    <%= render @cart %>
                <% end %>
            <% end %>

            <ul>
▶               <li><a href="http://www....."><%= t('.home') %></a></li>
▶               <li><a href="http://www..../faq"><%= t('.questions') %></a></li>
▶               <li><a href="http://www..../news"><%= t('.news') %></a></li>
▶               <li><a href="http://www..../contact"><%= t('.contact') %></a></li>
            </ul>
            <% if session[:user_id] %>
```

```
                <ul>
                    <li><%= link_to 'Orders', orders_path %></li>
                    <li><%= link_to 'Products', products_path %></li>
                    <li><%= link_to 'Users', users_path %></li>
                </ul>
                <%= button_to 'Logout', logout_path, method: :delete %>
            <% end %>
        </div>
        <div id="main">
            <%= yield %>
        </div>
    </div>
</body>
</html>
```

Поскольку это представление называется layouts/application.html.erb, английское отображение будет расширено до en.layouts.application. А вот соответствующий файл локализации:

rails40/depot_s/config/locales/en.yml

```
en:

    layouts:
        application:
            title:         "Pragmatic Bookshelf"
            home:         "Home"
            questions:     "Questions"
            news:         "News"
            contact:     "Contact"
```

и следующий файл на испанском:

rails40/depot_s/config/locales/es.yml

```
es:

    layouts:
        application:
            title:         "Publicaciones de Pragmatic"
            home:         "Inicio"
            questions:     "Preguntas"
            news:         "Noticias"
            contact:     "Contacto"
```

В них используется формат YAML — такой же формат использовался для конфигурирования баз данных. YAML состоит из расположенных с отступами имен и значений, где доступ в данном случае соответствует структуре, создаваемой в наших именах.

Чтобы заставить Rails распознать наличие новых YAML-файлов, нужно перезапустить сервер.

На рис. 15.4 мы можем увидеть реально переведенный текст, появляющийся в окне нашего браузера.

Рис. 15.4. Робкие шаги: перевод заголовков и боковой панели

Следующим нужно обновить основной заголовок, а также кнопку Add to Cart (Добавить в корзину). И то и другое можно найти в шаблоне каталога магазина:

rails40/depot_s/app/views/store/index.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>
<h1><%= t('.title_html') %></h1>
<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
        <%= image_tag(product.image_url) %>
        <h3><%= product.title %></h3>
        <%= sanitize(product.description) %>
        <div class="price_line">
            <span class="price"><%= number_to_currency(product.price) %></span>
            <%= button_to t('.add_html'), line_items_path(product_id: product),
                remote: true %>
        </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

А вот как выглядят соответствующие обновления файлов локализации, сначала на английском:

rails40/depot_s/config/locales/en.yml

```
en:

    store:
```

```
    index:
        title_html:     "Your Pragmatic Catalog"
        add_html:     "Add to Cart"
```

а затем на испанском:

rails40/depot_s/config/locales/es.yml

```
es:

  store:
      index:
          title_html:     "Su Cat&aacute;logo de Pragmatic"
          add_html:     "A&ntilde;adir al Carrito"
```

Следует заметить, что, поскольку `title_html` и `add_html` заканчиваются символами `_html`, мы можем свободно использовать имена, существующие в HTML для обозначения символов, не встречающихся на нашей клавиатуре.

Если вы не дали ключу перевода подобного имени, на странице будет видна разметка. Это еще одно соглашение, принятое в Rails для облегчения вашей программистской жизни. Rails будет также рассматривать имена, содержащие `html` в качестве компонента (иными словами, содержащие строковое значение `.html.`) ключевых имен HTML.

Обновив страницу в окне браузера, мы увидим результаты, показанные на рис. 15.5.

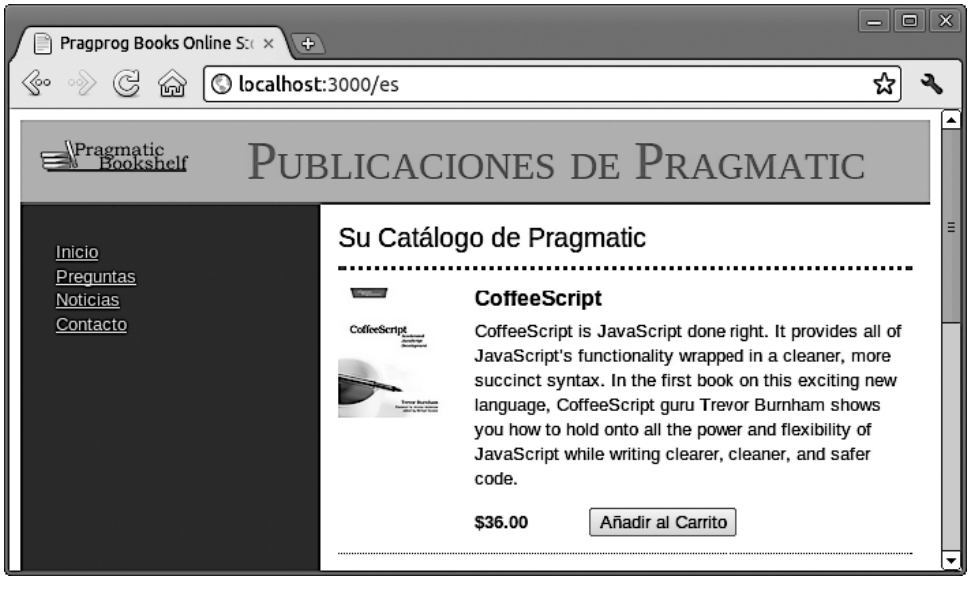


Рис. 15.5. Переведены заголовок и кнопка

Обретя уверенность, возьмемся за парциал корзины:

rails40/depot_s/app/views/carts/_cart.html.erb

```
▶<h2><%= t('.title') %></h2>
 <table>
     <%= render(cart.line_items) %>

     <tr class="total_line">
         <td colspan="2">Total</td>
         <td class="total_cell"><%= number_to_currency(cart.total_price) %></td>
     </tr>
 </table>

▶<%= button_to t('.checkout'), new_order_path, method: :get %>
▶<%= button_to t('.empty'), cart, method: :delete,
     data: { confirm: 'Are you sure?' } %>
```

И опять переводы:

rails40/depot_s/config/locales/en.yml

```
en:

    carts:
        cart:
            title:         "Your Cart"
            empty:         "Empty cart"
            checkout:     "Checkout"
```

rails40/depot_s/config/locales/es.yml

```
es:

    carts:
        cart:
            title:         "Carrito de la Compra"
            empty:         "Vaciar Carrito"
            checkout:     "Comprar"
```

Обновив страницу, мы увидим, что заголовки корзины и кнопки были переведены (рис. 15.6).

И теперь мы заметили нашу первую проблему. Локализация затрагивает не только язык, но и представление валюты. Изменяется также и общепринятый способ представления чисел.

Поэтому мы сначала обмениваемся мнениями с заказчиком и убеждаемся в том, что пока нам не нужно возиться с обменными курсами, поскольку этим займутся компании, обслуживающие кредитные карты и (или) безналичную торговлю, но при показе результата на испанском нам придется выводить после числового значения строку «USD» или «$US».

Еще одним изменением будет способ вывода самих чисел. Десятичные значения отделяются запятой, а разделителем тысячных позиций служит точка.

С валютой все немного сложнее, чем кажется на первый взгляд, поэтому нужно принять множество решений. К счастью, Rails в курсе, что для получения этой

информации нужно заглянуть в ваш файл перевода, нам нужно лишь все это предоставить. Сначала для en:

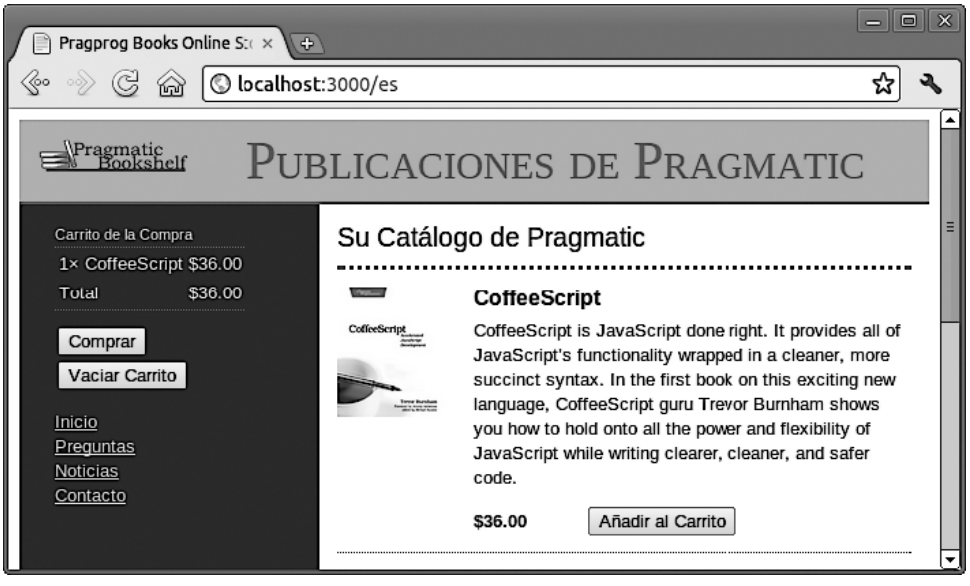


Рис. 15.6. Carrito bonita (Красивая корзина)

rails40/depot_s/config/locales/en.yml

```
en:

    number:
        currency:
            format:
                unit:          "$"
                precision:      2
                separator:      "."
                delimiter:      ","
                format:         "%u%n"
```

А затем для es:

rails40/depot_s/config/locales/es.yml

```
es:

    number:
        currency:
            format:
                unit:          "$US"
                precision:      2
                separator:      ","
                delimiter:      "."
                format:         "%n %u"
```

Мы указали денежную единицу (`unit`), точность (`precision`), разделитель десятичной части (`separator`) и разделитель тысячных позиций (`delimiter`) для `number.currency.format`. Все это вполне очевидно. А вот формат (`format`) выглядит немного сложнее: `%n` — это указатель места вставки самого числа, ` ` — это символ неразрывного пробела, уберегающий это значение от разбиения на несколько строк, а `%u` — это указатель места вставки денежной единицы. Результат указания денежной единицы показан на рис. 15.7.

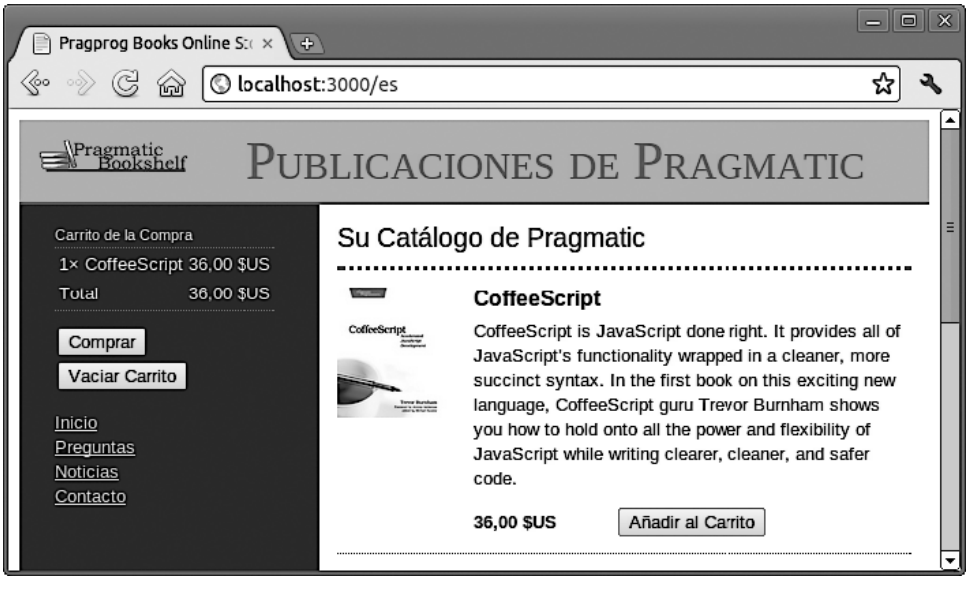


Рис. 15.7. Mas dinero, por favor (Еще денег, пожалуйста)

15.3. Шаг К3: перевод оформления заказа

Теперь мы на финишной прямой. Следующей будет страница заказа:

rails40/depot_s/app/views/orders/new.html.erb

```
<div class="depot_form">
    <fieldset>
        <legend><%= t('.legend') %></legend>
        <%= render 'form' %>
    </fieldset>
</div>
```

Вот форма, используемая этой страницей:

rails40/depot_s/app/views/orders/_form.html.erb

```
<%= form_for(@order) do |f| %>
    <% if @order.errors.any? %>
        <div id="error_explanation">
```

```
            <h2><%= pluralize(@order.errors.count, "error") %>
            prohibited this order from being saved:</h2>

            <ul>
            <% @order.errors.full_messages.each do |msg| %>
                <li><%= msg %></li>
            <% end %>
            </ul>
        </div>
    <% end %>

    <div class="field">
        <%= f.label :name %><br>
        <%= f.text_field :name, size: 40 %>
    </div>
    <div class="field">
        <%= f.label :address, t('.address_html') %><br>
        <%= f.text_area :address, rows: 3, cols: 40 %>
    </div>
    <div class="field">
        <%= f.label :email %><br>
        <%= f.email_field :email, size: 40 %>
    </div>
    <div class="field">
        <%= f.label :pay_type %><br>
        <%= f.select :pay_type, Order::PAYMENT_TYPES,
            prompt: t('.pay_prompt_html') %>
    </div>
    <div class="actions">
        <%= f.submit t('.submit') %>
    </div>
<% end %>
```

А вот как выглядят соответствующие определения локализации:

rails40/depot_s/config/locales/en.yml

```
en:

    orders:
        new:
            legend:           "Please Enter Your Details"
        form:
            name:             "Name"
            address_html:     "Address"
            email:            "E-mail"
            pay_type:         "Pay with"
            pay_prompt_html:  "Select a payment method"
            submit:           "Place Order"
```

rails40/depot_s/config/locales/es.yml

```
es:

    orders:
        new:
```

```
        legend:           "Por favor, introduzca sus datos"
    form:
        name:             "Nombre"
        address_html:     "Direcci&oacute;n"
        email:            "E-mail"
        pay_type:         "Forma de pago"
        pay_prompt_html:  "Seleccione un m&eacute;todo de pago"
        submit:           "Realizar Pedido"
```

Готовая форма показана на рис. 15.8.

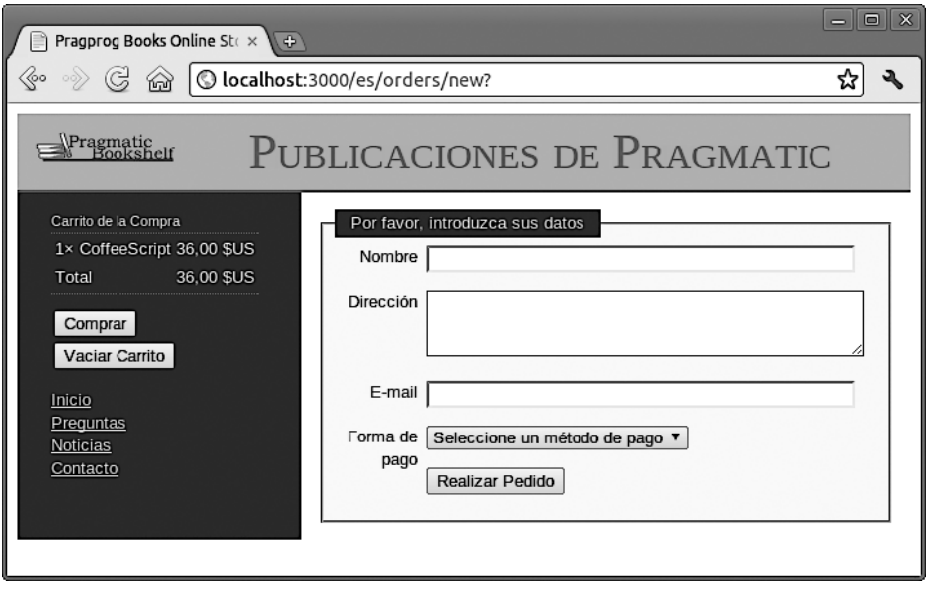


Рис. 15.8. Выражение готовности получить ваши деньги на испанском

Все выглядело неплохо, пока мы преждевременно не щелкнули на кнопке Realizar Pedido и не увидели уведомления, показанного на рис. 15.9. Сообщение об ошибке, создаваемое Active Record, также может быть переведено, нам нужно будет только предоставить перевод:

rails40/depot_s/config/locales/es.yml

```
es:

activerecord:
    errors:
        messages:
            inclusion:    "no est&aacute; incluido en la lista"
            blank:        "no puede quedar en blanco"

errors:
    template:
        body:             "Hay problemas con los siguientes campos:"
```

```
    header:
        one:   "1 error ha impedido que este %{model} se guarde"
        other: "%{count} errores han impedido que este %{model} se guarde"
```

Обратите внимание на то, что сообщения с числами имеют, как правило, две формы: `errors.template.header.one` — это сообщение, создаваемое при возникновении одной ошибки, и `errors.template.header.other` — сообщение, создаваемое во всех остальных случаях. Это позволяет переводчикам предоставить правильную множественную форму существительных и соответствие глаголов существительным.

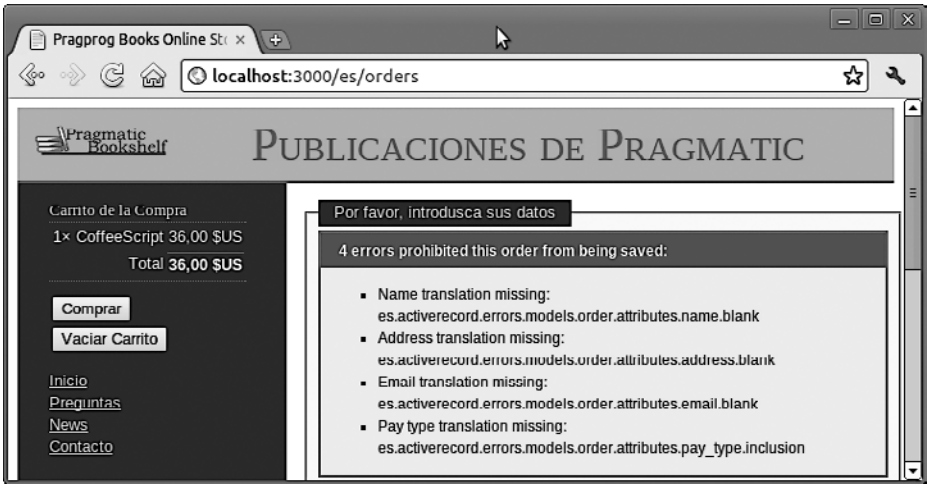


Рис. 15.9. Перевод отсутствует

Поскольку мы опять используем HTML-объекты, нам нужно, чтобы эти сообщения об ошибках выводились как есть (в понятиях Rails — в необработанном виде). Нам также нужно перевести сообщения об ошибках. Модифицируем форму еще раз:

rails40/depot_t/app/views/orders/_form.html.erb

```
<%= form_for(@order) do |f| %>
    <% if @order.errors.any? %>
        <div id="error_explanation">
▶           <h2><%=raw t('errors.template.header', count: @order.errors.count,
▶               model: t('activerecord.models.order')) %>.</h2>
▶           <p><%= t('errors.template.body') %></p>

            <ul>
            <% @order.errors.full_messages.each do |msg| %>
▶               <li><%=raw msg %></li>
            <% end %>
            </ul>
        </div>
```

```
    <% end %>
<!-- ... -->
```

Обратите внимание на то, что вызову метода для перевода заголовка шаблона ошибок мы передаем количество ошибок и имя модели (которая сама по себе допускает возможность перевода).

После внесения этих изменений мы повторяем нашу попытку и видим улучшения, показанные на рис. 15.10.

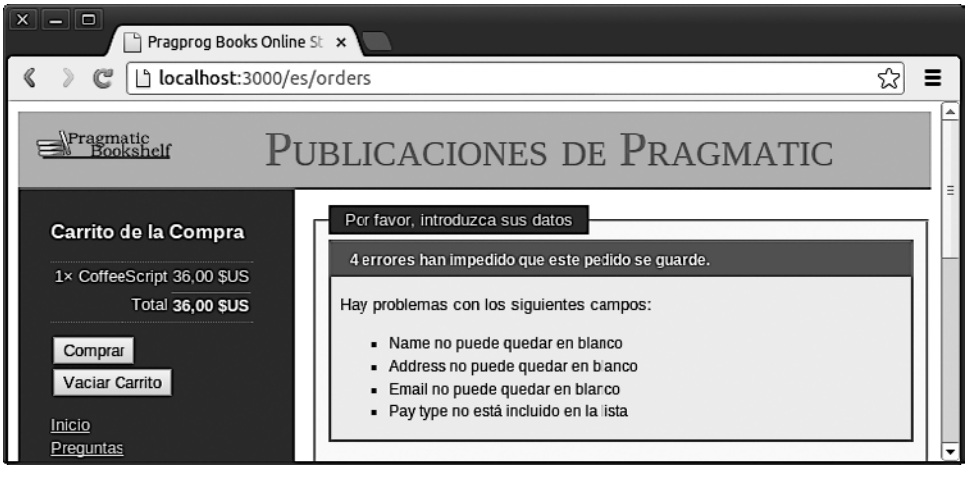


Рис. 15.10. Английские существительные в испанских предложениях

Уже лучше, но через интерфейс просачиваются имена модели и свойств. Это вполне допустимо на английском, поскольку названия подбирались для работы на этом языке. Нам нужно предоставить перевод для каждого имени.

Он также должен быть в YAML-файле:

rails40/depot_t/config/locales/es.yml

```
es:

    activerecord:
        models:
            order: "pedido"
        attributes:
            order:
                address:      "Direcci&oacute;n"
                name:         "Nombre"
                email:        "E-mail"
                pay_type:     "Forma de pago"
```

Обратите внимание на то, что нам не нужно предоставлять для всего этого английские эквиваленты, потому что эти сообщения встроены в Rails.

Нам приятно видеть переведенные имена модели и свойств на рис. 15.11; мы заполнили форму, отправили заказ и получили сообщение «Thank you for your order» («Спасибо за ваш заказ»).

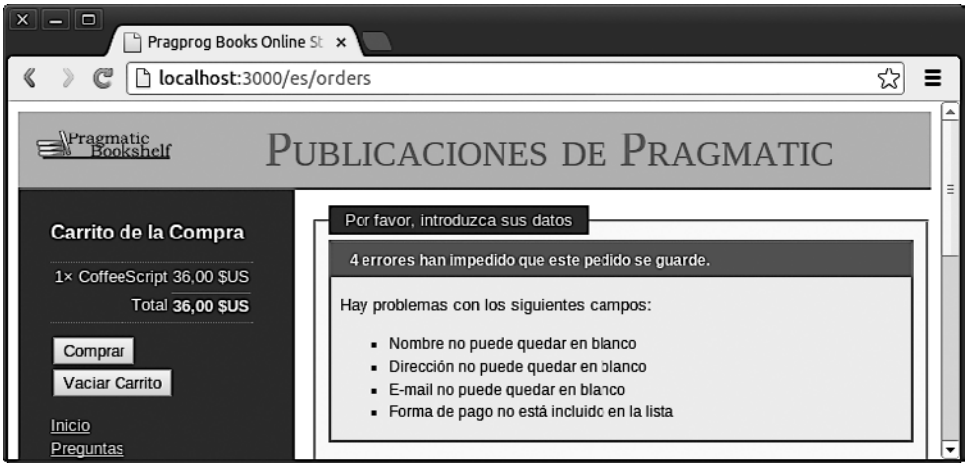


Рис. 15.11. Теперь переведены и имена модели

Нам нужно обновить флэш-сообщения:

rails40/depot_t/app/controllers/orders_controller.rb

```
def create
    @order = Order.new(order_params)
    @order.add_line_items_from_cart(@cart)

    respond_to do |format|
        if @order.save
            Cart.destroy(session[:cart_id])
            session[:cart_id] = nil
            OrderNotifier.received(@order).deliver
            format.html { redirect_to store_url, notice:
▶               I18n.t('.thanks') }
            format.json { render action: 'show', status: :created,
                location: @order }
        else
            @cart = current_cart
            format.html { render action: 'new' }
            format.json { render json: @order.errors,
                status: :unprocessable_entity }
        end
    end
end
```

И наконец, предоставим переводы:

rails40/depot_t/config/locales/en.yml

```
en:

    thanks: "Thank you for your order"
```

rails40/depot_t/config/locales/es.yml

```
es:

    thanks: "Gracias por su pedido"
```

Благодарственное сообщение показано на рис. 15.12.

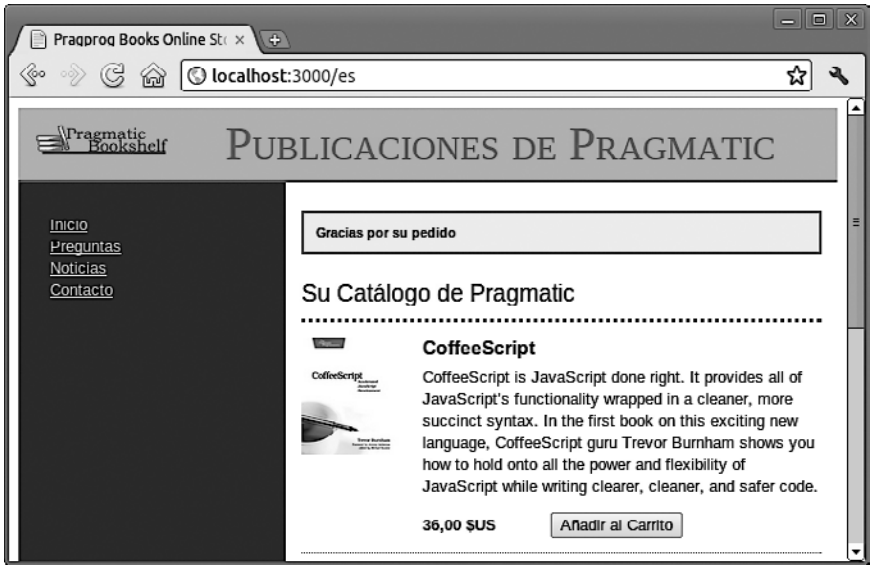


Рис. 15.12. Благодарим пользователя на испанском

15.4. Шаг К4: добавление переключателя локализации

Мы завершили выполнение задачи, но теперь нам нужно дополнительно обозначить ее возможности. Мы присмотрели одну неиспользуемую область в верхней правой части макета, поэтому дополним форму непосредственно перед `image_tag`:

rails40/depot_t/app/views/layouts/application.html.erb

```
<div id="banner">
▶     <%= form_tag store_path, class: 'locale' do %>
```

```
▶        <%= select_tag 'set_locale',
▶            options_for_select(LANGUAGES, I18n.locale.to_s),
▶            onchange: 'this.form.submit()' %>
▶        <%= submit_tag 'submit' %>
▶        <%= javascript_tag "$('.locale input').hide()" %>
▶    <% end %>
     <%= image_tag("logo.png") %>
     <%= @page_title || t('.title') %>
</div>
```

Методу `form_tag` указывается путь к магазину в качестве страницы, которая должна быть заново выведена при отправке формы. Атрибут `class` позволяет нам связать форму с кодом CSS.

Метод `select_tag` используется для определения одного поля ввода для этой формы, а именно `locale`. Это поле со списком, основанное на значениях массива `LANGUAGES`, который определен в файле конфигурации, со значением по умолчанию, соответствующим текущему региону (которое также сделано доступным благодаря модулю `I18n`). Мы также настраиваем обработчик событий `onchange`, который будет отправлять значение этой формы при каждом изменении ее значения. Этот обработчик будет работать, только если включен JavaScript, но он очень удобен.

Затем мы добавляем метод `submit_tag` на тот случай, когда JavaScript отключен. Для обработки случая, когда JavaScript включен и кнопка отправки данных не нужна, мы добавляем еще немного кода JavaScript, который скроет каждый из тегов ввода в форме локализации, даже если мы знаем, что в ней только один такой тег.

Затем мы внесем изменения в контроллер магазина для перенаправления на путь к магазину, связанный с заданной локализацией, если используется форма `:set_locale`:

rails40/depot_t/app/controllers/store_controller.rb

```
def index
▶    if params[:set_locale]
▶        redirect_to store_url(locale: params[:set_locale])
▶    else
▶        @products = Product.order(:title)
▶    end
End
```

И в заключение добавим немного CSS-кода:

rails40/depot_t/app/assets/stylesheets/application.css.scss

```
.locale {
    float: right;
    margin: -0.25em 0.1em;
}
```

Созданный переключатель показан на рис. 15.13. Теперь мы можем переключать языки в любом направлении простым щелчком мыши.

Теперь мы можем размещать заказы на двух языках, и наши дальнейшие мысли направлены на реальное развертывание приложения. Но поскольку у нас был весьма нелегкий день, пора сложить инструменты и расслабиться. А с утра мы приступим к развертыванию.

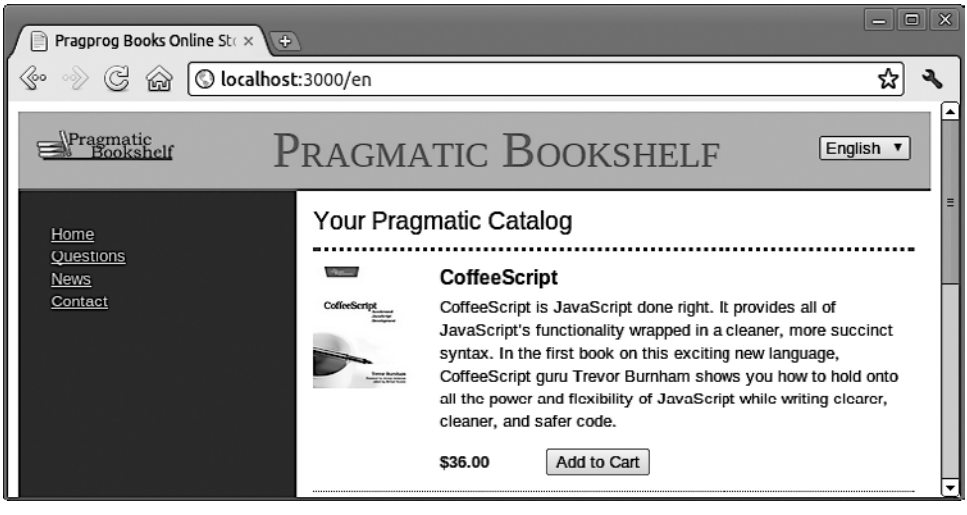


Рис. 15.13. Переключатель локализации в верней правой части окна

Наши достижения

К окончанию данного шага нам удалось сделать следующее.

- ☑ Мы установили для своего приложения локализацию по умолчанию и предоставили пользователю средства выбора альтернативной локализации.
- ☑ Мы создали файлы переводов для текстовых полей, указания количества денег, ошибок и имен модели.
- ☑ Мы изменили макеты и представления, чтобы вызвать модуль `I18n`, используя вспомогательный метод `t()` с целью перевода текстовых составляющих интерфейса.

Чем заняться на досуге

Попробуйте проделать все это без посторонней помощи.

- ❍ Добавьте столбец `locale` к таблице базы данных `products` и настройте представление каталога на выбор только тех товаров, которые соответствуют локализации. Настройте представление товаров, чтобы можно было про-

сматривать, вводить и изменять значение этого нового столбца. Введите несколько товаров для каждой локализации и посмотрите, что получилось.

- Определите текущий курс обмена между американскими долларами и евро и локализируйте вывод валюты для отображения евро, когда выбрана локализация `ES_es`.
- Переведите способы оплаты `Order::PAYMENT_TYPES`, показанные в раскрывающемся списке. Но при этом выбранное значение (отправляемое серверу) должно оставаться прежним. Измениться должно только отображение.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

Задача Л: развертывание и эксплуатация

16

Основные темы:

- запуск нашего приложения на эксплуатационном веб-сервере;
- конфигурирование базы данных для MySQL;
- использование Bundler и Git для управления версиями и
- развертывание нашего приложения с использованием Capistrano.

Развертывание в жизненном цикле приложения знаменует радостную веху. Именно в этот момент его код, который так бережно создавался, выкладывается на сервер, чтобы им могли пользоваться другие люди. Предполагается, что пиво и шампанское должны литься рекой, а стол ломиться от закусок. Вскоре о приложении напишут в журнале *Wired magazine*, а вы сами внезапно станете новой фигурой в избранном обществе.

Но реальность зачастую требует двусторонней проработки, чтобы добиться надежного и повторяемого развертывания вашего приложения.

Когда мы обдумывали содержание данной главы, наша установка имела вид, показанный на рис. 16.1.

В настоящее время мы проделываем всю работу на одной машине, в то время как взаимодействие пользователя с нашим веб-сервером должно осуществляться на отдельной машине. На рисунке машина пользователя находится в центре, а веб-сервер WEBRick находится слева. На этом сервере используются SQLite3, установленные вами различные gem-пакеты и код вашего приложения. На данный момент ваш код может либо находиться на Git-системе, либо нет — в любом

случае к концу данной главы он там будет, а также там будут используемые вами gem-пакеты.

Репозиторий Git будет продублирован на эксплуатационном сервере, который опять же может быть совершенно другой машиной, хотя и необязательно.

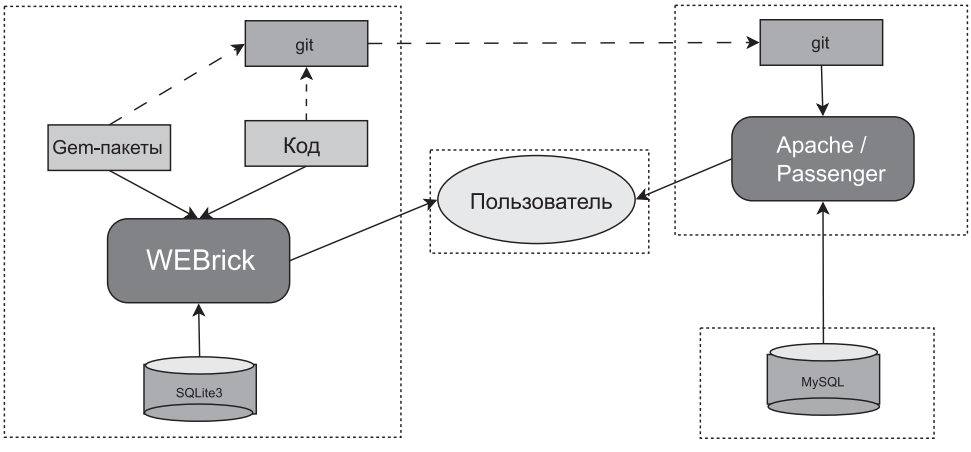


Рис. 16.1. Дорожная карта развертывания приложения

На этом сервере будет запущена комбинация, состоящая из Apache httpd и Phusion Passenger. Этот код будет обращаться к базе данных MySQL, и это может привести к появлению четвертой машины.

Capistrano будет использоваться для удаленного, безопасного и однообразного обновления сервера или серверов с развернутым приложением при содействии нашей разработочной машины.

Здесь немало движущихся частей!

Вместо того чтобы выполнить все сразу, мы разобьем работу на три шага. Первый шаг приведет приложение Depot в состояние готовности к работе с Apache, MySQL и Passenger — настоящим оборудованием веб-сервера, предназначенным для эксплуатации.

На второй шаг мы оставим работу с Git, Bundler и Capistrano. Эти инструментальные средства позволят отделить нашу разработочную деятельность от среды развертывания. Это означает, что к моменту, когда все будет сделано, мы проведем развертывание дважды, но это произойдет только в этот — первый — раз. И только для того, чтобы убедиться, что каждая из частей работает независимо. Это также позволит нам в любой момент времени сконцентрироваться на меньшем наборе переменных, что упростит процесс решения любых проблем, которые могут возникнуть.

Третий шаг будет заключаться в решении различных административных задач и наведении порядка. Итак, приступим!

16.1. Шаг Л1: развертывание с использованием Phusion Passenger и MySQL

До сих пор при развертывании Rails-приложения на нашей локальной машине мы, вероятнее всего, при запуске сервера использовали WEBrick. По большому счету это не имеет никакого значения. Команда `rails server` была наиболее подходящим способом запуска нашего приложения в разработочном режиме на порте 3000. Но развернутое Rails-приложение работает несколько иначе. Мы не можем просто запустить единственный серверный Rails-процесс и переложить всю работу на него. Точнее, можем, но это далеко не идеальное решение. Причина в том, что Rails является однопоточной средой и может одновременно работать только с одним запросом.

ДЖО СПРАШИВАЕТ: А НЕЛЬЗЯ ЛИ РАЗВЕРНУТЬ ПРИЛОЖЕНИЕ ПОД MICROSOFT WINDOWS?

Хотя развертывание приложений в среде Windows вполне допустимо, огромное количество инструментария Rails и общедоступного опыта предполагает использование операционных систем на базе Unix, например Linux или Mac OS X. Одно из таких инструментальных средств, Phusion Passenger, настоятельно рекомендовано командой разработчиков Ruby on Rails и будет рассмотрено в данной главе.

Рассматриваемые здесь технологии могут использоваться при таком вот развертывании под Linux или Mac OS X.

Но Интернет — это сугубо параллельная среда. Эксплуатационные веб-серверы, такие как Apache, Lighttpd и Zeus, могут одновременно обрабатывать сразу несколько запросов, счет которых может идти на десятки или даже сотни. Имеющий всего один процесс однопоточный веб-сервер на базе языка Ruby, наверное, с этим не справится. К счастью, ему с этим и не нужно справляться. Вместо этого способом развертывания Rails-приложения и введения его в эксплуатацию станет использование внешнего сервера, такого как Apache, который будет обрабатывать запросы, поступающие от клиентов. Затем HTTP прокси-сервер Passenger будет задействован для отправки запросов, предназначенных для обработки средой Rails, любому количеству серверных прикладных процессов.

Настройка второй машины

Хорошо, если у вас под рукой найдется вторая машина. Если таковой не найдется, то можно воспользоваться виртуальной машиной. В этих целях можно задействовать массу свободно распространяемого программного обеспечения, например VirtualBox и Ubuntu. Если остановите выбор на Ubuntu, то мы рекомендуем версию 12.04 LTS.

Настройте эту машину, воспользовавшись инструкцией в главе 1, «Установка Rails». Если хотите, можете пропустить этап установки Rails и установить вместо нее Bundler.

```
$ gem install bundler
```

Затем нужно скопировать весь каталог, содержащий приложение Depot, с вашей первой машины. На второй машине перейдите в этот каталог и воспользуйтесь программой Bundler для установки всего программного обеспечения, от которого зависит работа приложения.

```
$ bundle install
```

Проверьте работоспособность приложения, используя комбинацию следующих команд:

```
$ rake about
$ rake test
$ rails server
```

Теперь можно будет запустить браузер на любой машине и увидеть свое приложение. Как только удостоверитесь в том, что ваше приложение работает корректно, остановите сервер.

Конечно, не хотелось бы, чтобы разработчикам приложений приходилось выполнять все эти шаги по копированию каталогов, а также по запуску и остановке серверов, и когда эта глава будет закончена, все это уже будет автоматизировано. А сейчас знание того, какие шаги предпринять и какие промежуточные результаты получить, создадут ту самую базу, на основе которой мы сможем построить наше развертывание.

Установка Passenger

Затем нужно выполнить следующий шаг и убедиться, что на нашей второй машине установлен и запущен веб-сервер Apache. Пользователи Linux уже должны были установить Apache в ходе изучения раздела 1.3 «Установка под Linux». У тех, кто пользуется Mac OS X, он уже установлен вместе с операционной системой, но его нужно включить. Для Mac OS версий до 10.8 это может быть сделано путем перехода в System Preferences ▸ Sharing и включения Web Sharing. Начиная с версии Mac OS X 10.8 это делается через приложение Terminal.

```
$ sudo apachectl start
$ sudo launchctl load -w /System/Library/LaunchDaemons/org.apache.httpd.plist
```

Следующим шагом будет установка Passenger:

```
$ gem install passenger --version 4.0.8
$ passenger-install-apache2-module
```

Если не будет выявлено никаких необходимых зависимостей, последняя команда скажет, что нужно делать. Например, при работе под Ubuntu 13.04 (Raring Ringtail) окажется, что нужно установить libcurl4-openssl-dev, apache2-prefork-dev,

libapr1-dev и libaprutil1-dev. Если такое произойдет, следуйте предоставляемым инструкциям и запустите команду установки Passenger еще раз.

Как только все зависимости будут удовлетворены, эта команда заставит провести компиляцию сразу нескольких источников и обновление конфигурационных файлов. В течение этого процесса нам будут выдаваться запросы на обновление конфигурации нашего Apache-сервера. Первым будет запрос на включение ваших только что созданных модулей, и он будет заключаться в добавлении показанных далее строк к нашей Apache-конфигурации.

ПРИМЕЧАНИЕ

Passenger подскажет, какие конкретно строки нужно скопировать и вставить в файл, поэтому нужно использовать подсказанные им строки, а не те, что показаны здесь. Нам также пришлось сократить описание пути в строке LoadModule, чтобы оно поместилось на странице. Воспользуйтесь именно тем описанием пути, которое будет предоставлено Passenger.

```
LoadModule passenger_module /home/rubys/.rvm/.../ext/apache2/mod_passenger.so
PassengerRoot /home/rubys/.rvm/gems/ruby-2.0.0-p0/gems/passenger-4.0.1
PassengerDefaultRuby /home/rubys/.rvm/wrappers/ruby-2.0.0-p0/ruby
```

Чтобы определить, где именно находится конфигурационный файл Apache, попробуйте воспользоваться следующей командой:

```
$ apachectl -V | grep HTTPD_ROOT
$ apachectl -V | grep SERVER_CONFIG_FILE
```

На некоторых системах эта команда называется `apache2ctl`, на других она называется `httpd`. Пробуйте, пока не найдете правильную команду.

Вместо того чтобы вносить в этот файл непосредственные изменения, можно воспользоваться тем, что современные системы позволяют обслуживать расширения отдельно. На Mac OS X, к примеру, в конце файла httpd.conf можно увидеть следующую строку:

```
Include /private/etc/apache2/other/*.conf
```

Если она имеется в вашем файле httpd.conf, то можно поместить строки, которые предоставляются программой Passenger, в файл passenger.conf, разместив его в этом каталоге. В Ubuntu эти строки можно поместить в файл /etc/apache2/conf .d/passenger.

Локальное развертывание нашего приложения

Следующий шаг заключается в развертывании нашего приложения. В то время как предыдущий шаг выполняется только один раз для каждого сервера, этот шаг выполняется один раз для каждого приложения. Подставьте в следующую строку `ServerName` имя своего хоста и путь к каталогу вашего приложения:

```
<VirtualHost *:80>
    ServerName depot.yourhost.com
```

```
    DocumentRoot /home/rubys/deploy/depot/public/
    <Directory /home/rubys/deploy/depot/public>
        AllowOverride all
        Options -MultiViews
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Обратите внимание на то, что параметр `DocumentRoot` настроен на наш каталог public в нашем Rails-приложении и на то, что каталог public отмечен как читаемый.

И опять в вашей копии Apache могут быть соглашения о наилучшем месте для помещения этих инструкций. На Mac OS X нужно проверить файл httpd.conf на наличие следующей строки (которая может быть закомментирована):

```
#Include /private/etc/apache2/extra/httpd-vhosts.conf
```

Если такая строка имеется, стоит подумать над тем, чтобы снять с нее комментарий и заменить dummy-host.example.com именем вашего хоста.

На Ubuntu существует соглашение о помещении этих строк в файл в каталоге /etc/apache2/sitesavailable для последующего отдельного включения сайта. Например, если файл был назван depot, то сайт включается следующей командой:

```
sudo a2ensite depot
```

Если у вас есть несколько приложений, нужно повторить этот блок `VirtualHost` по числу приложений, настраивая значения параметров `ServerName` и `DocumentRoot` в каждом блоке. Нужно также проверить, что в файлах конфигурации уже имеется следующая строка:

```
NameVirtualHost *:80
```

Если этой строки нет, ее нужно добавить перед строкой, в которой содержится текст `Listen 80`.

Завершающим шагом будет перезапуск вашего веб-сервера Apache:

```
$ sudo apachectl restart
```

Теперь вам нужно настроить конфигурацию вашего клиента, чтобы он отображал выбранное вами имя хоста на правильную машину. Это делается в файле по имени /etc/hosts. На Windows-машине этот файл можно найти в каталоге C:\windows\system32\drivers\etc\. Чтобы отредактировать этот файл, вам нужно будет открыть его, пользуясь правами администратора.

Обычная строка файла /etc/hosts может иметь следующий вид:

```
127.0.0.1 depot.yourhost.com
```

Вот и все! Теперь мы можем обратиться к своему приложению, используя указанный нами хост (или виртуальный хост). Теперь в нашем URL-адресе указывать номер порта не нужно, если только не используется номер порта, отличный от 80.

Нужно также иметь в виду следующее.

- Если при перезапуске вашего сервера появляется сообщение о том, что адрес или порт указан неправильно («The address or port is invalid»), это означает, что строка `NameVirtualHost` уже присутствует, возможно, в другом конфигурационном файле в том же самом каталоге. Если это так, удалите добавленную вами строку, потому что эта директива должна присутствовать только один раз.
- Если нужно запустить другую, не эксплуатационную среду, мы можем включить директиву `RailsEnv` в каждый блок `VirtualHost` в конфигурации нашего Apache-сервера:

```
RailsEnv development
```

- Мы можем в любое время перезапустить наше приложение без перезапуска Apache путем создания файла по имени restart.txt в каталоге tmp нашего приложения:

```
$ touch tmp/restart.txt
```

- Вывод команды `passenger-install-apache2-module` сообщит нам, где можно найти дополнительную документацию.

Использование MySQL в качестве базы данных

На веб-сайте SQLite[1] на удивление честно рассказано, для каких целей лучше применять эту базу данных и на что она неспособна. В частности, SQLite не рекомендуется для объемных веб-сайтов с большим количеством параллельных запросов и большими наборами данных. И разумеется, мы хотим, чтобы у нашего веб-сайта были именно такие параметры.

Существует масса альтернатив базе данных SQLite, как бесплатных, так и коммерческих. Мы будем использовать MySQL. Она доступна из вашего исходного пакета инструментов в Linux и предоставляется для OS X в виде установочного пакета на веб-сайте MySQL[2].

Имеющаяся на Mac OS X версии 10.7 (x86, 64-разрядной) версия DMG Archive хорошо работает в версии 10.8. Если не хотите регистрироваться, найдите в нижней части страницы ссылку «No thanks, just take me to the downloads!» («Спасибо, не нужно, хочу просто попасть в раздел загрузки!»).

Кроме установки базы данных MySQL, вам также понадобится добавить в Gemfile указание на gem-пакет mysql:

rails40/depot_t/Gemfile

```
group :production do
    gem 'mysql2'
end
```

[1] http://www.sqlite.org/whentouse.html
[2] http://dev.mysql.com/downloads/mysql/

Если этот gem-пакет помещен в группу `production`, он не будет загружаться при запуске приложения в режиме разработки или тестирования. Если хотите, можете поместить gem-пакет `sqlite3` в отдельные группы разработки и тестирования — `development` и `test`.

Установите gem-пакет с помощью команды `bundle install`. Сначала может потребоваться найти и установить разработочные файлы базы данных MySQL для вашей операционной системы. К примеру, при работе под Ubuntu понадобится установить `libmysqlclient-dev`.

Для создания своей базы данных можно воспользоваться клиентом командной строки `mysql` или, если вам больше нравится такое средство, как `phpmyadmin` или `CocoaMySQL`, воспользуйтесь им:

```
depot> mysql -u root
mysql> CREATE DATABASE depot_production DEFAULT CHARACTER SET utf8;
mysql> GRANT ALL PRIVILEGES ON depot_production.*
    ->     TO 'username'@'localhost' IDENTIFIED BY 'password';
mysql> EXIT;
```

Если вы выбрали другое имя базы данных, его следует запомнить, поскольку вам понадобится настроить конфигурационный файл на соответствие выбранному имени. Теперь давайте взглянем на конфигурационный файл.

Файл config/database.yml содержит информацию о подключениях баз данных. Он состоит из трех разделов, каждый из которых предназначен для баз данных разработки, тестирования и эксплуатации. Текущий раздел эксплуатации имеет следующее содержимое:

```
production:
    adapter: sqlite3
    database: db/production.sqlite3
    pool: 5
    timeout: 5000
```

Мы заменим этот раздел чем-нибудь таким:

```
production:
    adapter: mysql2
    encoding: utf8
    reconnect: false
    database: depot_production
    pool: 5
    username: username
    password: password
    host: localhost
```

Нужно будет изменить имя пользователя, пароль и поля базы данных, указав необходимые значения.

Загрузка базы данных

Далее мы применим наши миграции:

```
depot> rake db:setup RAILS_ENV="production"
```

При этом произойдет одно или два события. Если все настроено правильно, вы увидите примерно такой вывод:

```
-- create_table("carts", {:force=>true})
    -> 0.1722s
-- create_table("line_items", {:force=>true})
    -> 0.1255s
-- create_table("orders", {:force=>true})
    -> 0.1171s
-- create_table("products", {:force=>true})
    -> 0.1172s
-- create_table("users", {:force=>true})
    -> 0.1255s
-- initialize_schema_migrations_table()
    -> 0.0006s
-- assume_migrated_upto_version(20121130000008, "db/migrate")
    -> 0.0008s
```

Если вместо этого вы увидите какие-нибудь ошибки, не нужно паниковать! Это, скорее всего, простые проблемы конфигурации. Можете попробовать следующее.

- Проверьте имя, которое вы дали базе данных в разделе `production:` файла database.yml. Оно должно быть таким же, как и имя созданной вами базы данных (с помощью `mysqladmin` или какого-нибудь другого средства администрирования базы данных).
- Проверьте, что имя пользователя и пароль в файле database.yml соответствуют тем, которые были использованы при создании базы данных.
- Проверьте, запущен ли сервер вашей базы данных.
- Проверьте возможность подключения к ней из командной строки. Если используется MySQL, запустите следующую команду:

```
depot> mysql depot_production
mysql>
```

- Если подключение из командной строки возможно, можете ли вы создать таблицу-пустышку `dummy`? (Так можно будет проверить, имеет ли пользователь базы данных достаточные права доступа к базе данных.)

```
mysql> create table dummy(i int);
mysql> drop table dummy;
```

- Если вы можете создавать таблицы из командной строки, а команда `rake db:migrate` не проходит, еще раз проверьте файл database.yml. Если в файле есть директивы `socket:`, попробуйте их закомментировать, поставив перед каждой символ решетки (`#`).
- Если вы видите ошибку отсутствия указанного файла или каталога («No such file or directory...») и имя файла, вызвавшего ошибку mysql.sock, значит, ваши Ruby-библиотеки MySQL не могут найти вашей базы данных MySQL. Это может случиться, если вы установили библиотеки до установки базы данных или если установили библиотеки, используя двоичный пакет

установки, который сделал неверное предположение о месте нахождения сокет-файла MySQL. Чтобы исправить положение, лучше всего переустановить Ruby-библиотеки MySQL. Если это неподходящий вариант, еще раз проверьте, что в строке `socket:` в вашем файле database.yml содержится правильный путь к MySQL-сокету на вашей системе.

- Если получена ошибка отсутствия загрузки MySQL (Mysql not loaded), это означает, что вы запустили старую версию Ruby-библиотеки MySQL. Среде Rails нужна как минимум версия 2.5.
- Некоторые читатели также сообщают о получении сообщения о том, что клиент не поддерживает протокол аутентификации, запрашиваемый сервером («Client does not support authentication protocol requested by server»), что предполагает обновление MySQL-клиента. Чтобы избавиться от этой несовместимости установленной версии MySQL и библиотек, используемых для обращения к этой базе данных, выполните инструкции, имеющиеся на веб-сайте http://dev.mysql.com/doc/mysql/en/old-client.html, и запустите MySQL-команду вроде `set password for 'some_user'@'some_host' = OLD_PASSWORD('newpwd');`.
- Если вы используете в Windows MySQL под Cygwin, могут появиться проблемы, если указать хост как `localhost`. Попробуйте вместо этого использовать `127.0.0.1`.
- И наконец, у вас могут появиться проблемы в формате файла database.yml. Библиотека YAML, которая читает этот файл, очень странно реагирует на символы табуляции.

  Если в вашем файле содержатся знаки табуляции, ждите проблем. (А вы думали, что предпочли язык Ruby языку Python, потому что вам не нравилась имеющаяся в Python значимость пробельных символов?)

Запустите команду rake db:setup столько раз, сколько понадобится для исправления любых имеющихся проблем конфигурации.

Если вас это пугает, не стоит переживать. В действительности подключение к базе данных в основном работает безотказно. И как только Rails сможет работать с базой данных, вам не придется больше об этом беспокоиться.

Ну, все готово к работе. Все выглядит точно так же, как и при запуске в однопользовательском режиме. Разница становится заметной только тогда, когда у вас появляется большое количество одновременно обращающихся пользователей или когда база данных становится слишком большой.

Следующий шаг заключается в обособлении нашей разработки от нашей эксплуатационной машины.

16.2. Шаг Л2: удаленное развертывание с помощью Capistrano

Если у вас большой магазин, обладающий целой линейкой выделенных серверов, которые вы администрируете, чтобы все работало, как следует, вам следует быть

уверенным, что на них запущена одна и та же версия необходимого программного обеспечения. Для более скромных запросов подойдет и общий сервер, но придется уделить особое внимание тому факту, что установленные версии программного обеспечения не всегда могут соответствовать той версии, которая установлена на нашей разработочной машине.

Не волнуйтесь, мы расскажем обо всем по порядку.

Подготовка вашего сервера развертывания

Хотя помещение нашей программы во время ее разработки под управление версиями — действительно весьма неплохая идея, когда дело доходит до развертывания, то отказ от использования управления версиями превращается в абсолютное безрассудство. Достаточно сказать, что выбранная для управления развертыванием программа, а именно Capistrano, практически требует использования этого управления.

На данный момент существует множество систем управления версиями программ (software configuration management — SCM). Например, есть достаточно хорошая система Subversion. Но если вы еще не сделали свой выбор, остановитесь на системе Git, которую легко установить и которая не требует отдельного серверного процесса. Показанные далее примеры основаны на использовании системы Git, но если вы выбрали другую SCM-систему, не стоит переживать. Capistrano в принципе все равно, какую именно систему вы выбрали, лишь бы она была выбрана.

Первый шаг заключается в создании пустого репозитория на машине, доступной вашим серверам развертывания. Фактически, если у нас только один сервер для развертывания приложения, ничто не препятствует тому, чтобы он выполнял двойную обязанность, будучи еще и вашим Git-сервером. Итак, войдите в этот сервер и запустите следующие команды:

```
$ mkdir -p ~/git/depot.git
$ cd ~/git/depot.git
$ git --bare init
```

Следующее, что вам нужно знать: даже если SCM-сервер и ваш веб-сервер являются одной и той же физической машиной, Capistrano будет обращаться к нашему программному обеспечению SCM так, как будто оно находится на удаленной машине. Мы можем сгладить это путем генерации открытого ключа (если у вас еще нет такого ключа) с его последующим использованием для получения разрешения на доступ к своему собственному серверу:

```
$ test -e ~/.ssh/id_dsa.pub || ssh-keygen -t dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Протестируйте этот метод путем использования безопасного сетевого протокола SSH на своем собственном сервере. Кроме всего прочего, это гарантирует нам обновление вашего файла known_hosts.

И, пока мы здесь, нужно проследить еще за одной вещью. Capistrano вставит каталог по имени current между каталогом с именем нашего приложения и подкаталогами Rails, включая и подкаталог public. Это означает, что вам нужно внести

поправки в строки `DocumentRoot` и `Directory` в вашем файле httpd.conf, если вы управляете своим собственным сервером, или в панель управления для вашего общего хоста:

```
DocumentRoot /home/rubys/deploy/depot/current/public/
<Directory /home/rubys/deploy/depot/current/public>
```

Перезапустите свой сервер Apache. Должно появиться предупреждение о том, что каталога depot/current/public не существует. Ничего страшного, скоро мы его создадим.

И наконец, проверьте то, что изменения, внесенные в ваши файлы Gemfile и config/database.yml, скопированы из приложения Depot на вашей второй машине в приложение Depot на вашей первой машине.

Работа с сервером закончена! Впредь все будет делаться с вашей машины для разработки.

Получение контроля над приложением

Сначала мы собираемся обносить наш Gemfile, чтобы показать, что мы используем Capistrano.

rails40/depot_t/Gemfile

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

Всем пользователям нужно будет убрать символ комментария с этой одной строки. А пользователям RVM нужно будет там, где показано, добавить символы `rvm-`.

Теперь мы можем установить Capistrano, используя `bundle install`. Мы использовали эту команду в главе 12, в шаге Ж3 для установки gem-пакета bcrypt-ruby.

Если вы еще не поставили ваше приложение под контроль версий, сделайте это сейчас:

```
$ cd your_application_directory
$ git init
$ git add .
$ git commit -m "initial commit"
```

Следующий шаг носит необязательный характер, но может стать полезным, если у вас нет полного контроля над сервером развертывания или если у вас большое количество серверов развертывания, которые нужно контролировать. Мы собираемся воспользоваться вторым свойством программы Bundler, а именно командой `package`. Она поместит версию программного обеспечения, от которого вы зависите, в репозиторий:

```
$ bundle package
$ git add Gemfile.lock vendor/cache
$ git commit -m "bundle gems"
```

Дополнительные свойства программы Bundler будут рассмотрены в разделе 24.3 «Управление зависимостями с помощью Bundler».

Отсюда будет проще выложить весь ваш код на сервер:

```
$ git remote add origin ssh://user@host/~/git/depot.git
$ git push origin master
```

Не забудьте вместо `user` и `host` указать настоящее имя вашего пользователя и имя хоста на удаленной машине.

Благодаря этим нескольким шагам вы получили контроль над тем, что развертывается. Вы управляете тем, что относится к вашему локальному репозиторию. Вы задаете момент, когда нужно выложить все на свой сервер. Кроме того, вы будете управлять вводом этого кода в эксплуатацию.

Удаленное развертывание приложения

Ранее мы уже развертывали приложение локально на сервере. Теперь мы собираемся провести второе развертывание, на этот раз удаленное.

Вся подготовительная работа уже проведена. Теперь наш код находится на SCM-сервере, где он может быть доступен серверу приложения. Здесь снова имеет значение не то, являются ли эти оба сервера одной и той же машиной, а то, какие роли они выполняют.

Чтобы заставить Capistrano совершить волшебство и добавить необходимые для проекта файлы, запустите следующую команду:

```
$ capify .
[add] writing './Capfile'
[add] writing './config/deploy.rb'
[done] capified!
```

Из выведенной информации видно, что программа Capistrano установила два файла. Первый — Capfile — это принадлежащий Capistrano аналог Rakefile. Вам нужно убрать символ комментария с одной строки. После этого больше этот файл трогать не нужно.

rails40/depot_t/Capfile

```
load 'deploy' if respond_to?(:namespace) # cap2 differentiator

# Uncomment if you are using Rails' asset pipeline
load 'deploy/assets'
load 'config/deploy' # remove this line to skip loading any of the default tasks
```

Второй файл, config/deploy.rb, содержит наборы команд, необходимые для развертывания нашего приложения. Capistrano предоставляет нам минимальную версию этого файла, но следующий файл является более сложной версией, которую вы можете загрузить и использовать в качестве отправной точки:

rails40/depot_t/config/deploy.rb

```
require 'bundler/capistrano'
```

```
# be sure to change these
set :user, 'rubys'
set :domain, 'depot.pragprog.com'
set :application, 'depot'

# adjust if you are using RVM, remove if you are not
set :rvm_type, :user
set :rvm_ruby_string, 'ruby-2.0.0-p247'
require 'rvm/capistrano'

# file paths
set :repository, "#{user}@#{domain}:git/#{application}.git"
set :deploy_to, "/home/#{user}/deploy/#{application}"

# distribute your applications across servers (the instructions below put them
# all on the same server, defined above as 'domain', adjust as necessary)
role :app, domain
role :web, domain
role :db, domain, :primary => true

# you might need to set this if you aren't seeing password prompts
# default_run_options[:pty] = true
# As Capistrano executes in a non-interactive mode and therefore doesn't cause
# any of your shell profile scripts to be run, the following might be needed
# if (for example) you have locally installed gems or applications. Note:
# this needs to contain the full values for the variables set, not simply
# the deltas.
# default_environment['PATH']='<your paths>:/usr/local/bin:/usr/bin:/bin'
# default_environment['GEM_PATH']='<your paths>:/usr/lib/ruby/gems/1.8'
# miscellaneous options
set :deploy_via, :remote_cache
set :scm, 'git'
set :branch, 'master'
set :scm_verbose, true
set :use_sudo, false
set :rails_env, :production

namespace :deploy do
    desc "cause Passenger to initiate a restart"
    task :restart do
        run "touch #{current_path}/tmp/restart.txt"
    end

    desc "reload the database with seed data"
    task :seed do
        run "cd #{current_path}; rake db:seed RAILS_ENV=#{rails_env}"
    end
end

after "deploy:update_code", :bundle_install
desc "install the necessary prerequisites"
task :bundle_install, :roles => :app do
    run "cd #{release_path} && bundle install"
end
```

Чтобы привести все в соответствие с нашим приложением, нам нужно будет отредактировать несколько свойств. Нам, конечно, понадобится изменить свойства `:user`, `:domain` и `:application`. Свойство `:repository` соответствует месту, куда мы ранее поместили Git-файл. Свойство `:deploy_to` может потребовать настройки на соответствие той информации, которую мы сообщили серверу Apache (о том, где он может найти каталог public для приложения).

Мы также включили несколько строк, чтобы показать, как нужно проинструктировать программу Capistrano, чтобы она воспользовалась RVM[1].

Если диспетчер версии Ruby (RVM) был установлен на вашей машине развертывания как корневой, измените строку `set :rvm_type`, чтобы указать `:system` вместо `:user`. Внесите изменение в строку `:rvm_ruby_string`, чтобы она соответствовала версии интерпретатора Ruby, которую вы установили и хотите использовать.

Если вы вообще не используете RVM, удалите эти строки.

Настройки `default_run_options` и `default_environment` должны использоваться только тогда, когда у вас возникли специфические проблемы. Предоставленные смешанные настройки («miscellaneous options») основаны на Git, и с их помощью выключаются некоторые средства логики обработки, предназначенные для предыдущих версий Rails.

Определены две задачи: одна сообщает Capistrano, как перезапустить Passenger, а другая перезагружает эту базу данных исходными данными. Вы можете изменять эти задачи по собственному усмотрению.

При первом развертывании нашего приложения нам необходимо выполнить дополнительные шаги для настройки основной структуры каталогов развертывания на сервере:

```
$ cap deploy:setup
```

При выполнении этой команды Capistrano выведет приглашение на ввод пароля сервера. Если программа этого не сделает и не сможет войти в систему, может понадобиться убрать символ комментария со строки `default_run_options` в нашем файле deploy.rb и повторить попытку. Как только появится подключение, необходимые каталоги будут созданы. После выполнения этой команды мы можем проверить нашу конфигурацию на наличие любых других проблем:

```
$ cap deploy:check
```

Как и раньше, нам может потребоваться убрать символы комментария и настроить строки `default_environment` в нашем файле deploy.rb. Мы можем повторять эту команду до тех пор, пока она не будет успешно завершена, решая любые проблемы, которые она сможет выявить.

И еще одна последняя задача: нужно загрузить исходные («seed») данные, содержащие наши товары.

```
$ cap deploy:seed
```

На этом гонки должны закончиться.

---

[1] http://beginrescueend.com/integration/capistrano/

Прополоскать. Промыть. Повторить

Если вы добрались до этих строк, значит сервер в любое время готов заполучить новую версию развернутого на нем приложения. Нужно только проверить, внесены ли все необходимые изменения в версию, находящуюся в репозитории, а затем провести повторное развертывание. К этому моменту у нас есть два еще не проверенных файла Capistrano. Хотя сервер приложений в них не нуждается, мы все же можем воспользоваться ими для проверки процесса развертывания:

```
$ git add .
$ git commit -m "add cap files"
$ git push
$ cap deploy
```

Первые три команды проведут обновление SCM-сервера. Когда вы познакомитесь с Git поближе, вам может потребоваться более строгий контроль того, когда и какие файлы добавляются, вам может понадобиться до развертывания передать с наращиванием сразу несколько изменений и т. д. Но только последняя команда обновляет серверы нашего приложения, а также веб-сервер и сервер базы данных.

Если по каким-то причинам нам понадобится отступить и вернуться к предыдущей версии нашего приложения, можно воспользоваться этой командой:

```
$ cap deploy:rollback
```

Теперь у нас есть полностью развернутое приложение, и мы можем при необходимости провести развертывание с целью обновления кода, запущенного на сервере. При каждом развертывании нашего приложения новая версия проверяется на сервере, некоторые символьные ссылки обновляются, и процессы Passenger перезапускаются.

16.3. Шаг Л3: проверка работы развернутого приложения

Когда приложение будет развернуто, вам, несомненно, понадобится периодически проверять, как оно работает. Для этого есть два основных способа. Первый из них заключается в отслеживании содержимого различных файлов, содержащих регистрационные журналы, заполняемые как внешними веб-серверами, так и сервером Apache, запустившем наше приложение. Второй способ состоит в подключении к приложению с помощью команды `rails console`.

Просмотр регистрационных файлов

Чтобы взглянуть, что происходит в приложении, можно использовать команду `tail`, чтобы изучить регистрационные файлы на наличие запросов, сделанных в адрес нашего приложения. Обычно самые интересные данные находятся

в регистрационных журналах самого приложения. Даже если Apache запускает сразу несколько приложений, регистрируемый вывод для каждого приложения помещается в файл production.log для этого приложения.

Предполагая, что наше приложение развернуто в том месте, которое мы показывали ранее, на наши запущенные регистрационные файлы можно взглянуть следующим образом:

```
# На нашем сервере
$ cd /home/rubys/deploy/depot/current
$ tail -f log/production.log
```

Иногда нам нужна информация низкого уровня, чтобы понять, что происходит с данными в нашем приложении. Когда это происходит, настает время выйти на сцену наиболее полезному отладочному средству работающего сервера.

Использование консоли для наблюдения за работающим приложением

В классах моделей приложения создано большое количество разнообразных функций. Разумеется, все они были созданы для использования контроллерами приложения. Но с ними можно общаться и напрямую. Воротами в этот мир служит сценарий консоли Rails. Мы можем запустить его на нашем сервере с помощью следующих команд:

```
# На нашем сервере
$ cd /home/rubys/deploy/depot/current/
$ rails console production
Loading production environment.
irb(main):001:0> p = Product.find_by(title: "CoffeeScript")
=> #<Product:0x24797b4 @attributes={. . .}
irb(main):002:0> p.price = 29.00
=> 29.00
irb(main):003:0> p.save
=> true
```

Поскольку у нас есть открытый сеанс работы с консолью, мы можем обращаться ко всему разнообразию методов в наших моделях. Мы можем создавать, проверять и удалять записи. В некотором смысле это похоже на пульт управления приложением.

После запуска приложения в эксплуатацию нам нужно позаботиться о нескольких рутинных операциях, чтобы наше приложение работало без сбоев. Эти рутинные операции не будут выполняться за нас автоматически, но, к счастью, мы можем их автоматизировать.

Работа с регистрационными файлами

При работе приложения в его регистрационном файле происходит постоянное накопление информации. Со временем регистрационные файлы могут разбухать

до невероятных размеров. Для решения этой проблемы многие регистрационные решения способны *преобразовывать* регистрационные файлы для создания последовательного набора регистрационных файлов в хронологически возрастающем порядке. При этом регистрационные файлы разбиваются на удобные для использования фрагменты, которые могут быть помещены в архив или даже удалены по прошествии некоторого времени.

Преобразование поддерживается классом `Logger`. Нам требуется определить, сколько регистрационных файлов нам нужно (или как часто они нужны) и размер каждого файла, используя строку в файле config/environments/production.rb, похожую на следующую:

```
config.logger = Logger.new(config.paths['log'].first, 'daily')
```

Или, возможно, этот код:

```
require 'active_support/core_ext/numeric/bytes'
config.logger = Logger.new(config.paths['log'].first, 10, 10.megabytes)
```

Учтите, что в данном случае требуется явный запрос `active_support`, поскольку эта инструкция обрабатывается на ранней стадии при инициализации вашего приложения, до того как были включены библиотеки Active Support. Фактически одна из настроек конфигурации, предоставляемая Rails, заключается в полном отказе от включения библиотек Active Support:

```
config.active_support.bare = true
```

Вместо этого мы можем направить наши регистрационные записи в системные журналы для нашей машины:

```
config.logger = SyslogLogger.new
```

Дополнительные настройки можно найти на веб-сайте http://rubyonrails.org/deploy.

Начало и продолжение эксплуатации

После настройки первичного развертывания мы готовы к завершению разработки нашего приложения и запускаем его в эксплуатацию. Возможно, вам потребуется устанавливать дополнительные серверы для развертывания приложений и уроки, извлеченные из первого развертывания, будут весомой подсказкой для структурирования дальнейшего развертывания. К примеру, может оказаться, что Rails — один из самых медленных компонентов вашей системы, потому что основная часть времени при обработке запросов тратится на ожидание окончания работы Rails с базой данных или с файловой системой. Это свидетельствует о том, что нужно повысить скорость путем добавления компьютеров и распределения между ними нагрузки на Rails.

Но может оказаться, что основная часть времени при обработке запроса тратится на работу базы данных. В таком случае потребуется обратить внимание на ее оптимизацию. Возможно, нужно будет изменить систему доступа к данным или

самостоятельно создать ряд SQL-инструкций и заменить ими код Active Record, используемый по умолчанию.

Одно известно наверняка: каждое приложение потребует за время эксплуатации свой собственный набор доводок. Самое главное — постоянно прислушиваться к его работе и определять необходимые действия. Ваша работа не заканчивается запуском приложения в эксплуатацию. Фактически этим она только начинается.

Несмотря на то что после первого развертывания нашего приложения в эксплуатационный режим работа только начинается, мы завершили наш тур по приложению Depot. После краткого повторения всего проделанного в данной главе, давайте посмотрим, чего мы достигли (что примечательно, при помощи совсем небольшого количества строк программного кода).

Наши достижения

В этой главе мы выполнили большой объем работы. Мы взяли наш код, который работал локально для одного пользователя на нашей машине для разработки, и поместили его на разные машины, запустили его на другом веб-сервере, открыли доступ к другой базе данных и, возможно, даже запустили его под управлением другой операционной системы.

Для достижения этого результата мы воспользовались рядом программных продуктов.

- ☑ Мы установили и настроили Phusion Passenger и Apache httpd, веб-сервер, пригодный для применения в эксплуатационном режиме.
- ☑ Мы установили и настроили MySQL, пригодный для применения в эксплуатационном режиме сервер базы данных.
- ☑ Мы установили контроль за всеми зависимостями нашего приложения, воспользовавшись системами Bundler и Git.
- ☑ Мы установили и настроили Capistrano, программу, позволяющую нам надежно и повторяемо развернуть наше приложение.

Чем заняться на досуге

Попробуйте проделать это без посторонней помощи.

- ❍ Если к нашей разработке привлечено несколько разработчиков, мы могли бы испытывать чувство тревоги, помещая детали конфигурации нашей базы данных (потенциально включающие пароли!) в нашу систему управления конфигурированием. Чтобы решить эту проблему, скопируйте готовый файл `database.yml` в каталог `shared` и напишите задачу, предписывающую Capistrano копировать этот файл в ваш каталог `current` при каждом развертывании.

- ❍ Хотя в данной главе все внимание было сконцентрировано на стабильных, проверенных и, возможно, несколько консервативных вариантах

развертывания, в этой области произошло множество инновационных изменений. Пока что Capistrano и Git представляются практически безальтернативным выбором. А все остальное можно просто испробовать. Займитесь следующим:

- попробуйте вместо rvm применить rbenv[1] и ruby-build[2];
- попробуйте вместо mysql использовать PostgreSQL[3];
- попробуйте вместо пары Phusion Passenger и Apache httpd использовать пару Unicorn[4] и nginx[5].

Гибкость означает не только умение сделать правильный выбор. Она также требует применения адаптивного планирования и быстрого и гибкого реагирования на всевозможные изменения.

(Подсказки можно найти по адресу http://www.pragprog.com/wikis/wiki/RailsPlayTime.)

[1] https://github.com/sstephenson/rbenv/#readme
[2] https://github.com/sstephenson/ruby-build#readme
[3] http://www.postgresql.org/
[4] http://unicorn.bogomips.org/
[5] http://wiki.nginx.org/Main

17 Ретроспектива Depot

Основные темы:

- обзор концепций Rails: модель, представление, контроллер, конфигурация, тестирование и развертывание;
- документирование проделанной работы.

Поздравляем! Добравшись до этого места, вы получили твердые знания основ каждого из приложений Rails. Многое еще предстоит изучить, к чему мы вернемся в части III. А теперь расслабимся и подведем итог всему увиденному в части II.

17.1. Концепции Rails

В главе 3 «Архитектура Rails-приложений» было дано введение в модели, представления и контроллеры. Теперь давайте посмотрим, как мы применили каждую из этих концепций в приложении Depot. Затем мы исследуем, как нами использовались конфигурация, тестирование и развертывание.

Модель

Модель — это то самое место, где происходит управление всеми постоянными данными, хранящимися в вашем приложении. При разработке приложения Depot мы создали пять моделей: `Cart`, `LineItem`, `Order`, `Product` и `User`.

По умолчанию у всех моделей есть свойства `id`, `created_at` и `updated_at`. К нашим моделям мы добавляли свойства типа строка — `string` (примеры: `title`, `name`), целое число — `integer` (`quantity`), текст — `text` (`description`, `address`),

десятичное число — `decimal` (`price`) и внешние ключи (`product_id`, `cart_id`). Мы даже создали виртуальное свойство, которое никогда не сохраняется в базе данных, имеется в виду `password`.

Мы создали связи `has_many` и `belongs_to`, которые могут использоваться для переходов между нашими объектами моделей, например из `Carts` в `LineItems` и в `Products`.

Мы применили миграции для обновления баз данных не только для введения новой схемы хранения информации, но и для изменения существующих данных. Мы показали, что миграции могут быть применены в полностью реверсивном порядке.

Созданные нами модели не были просто пассивным хранилищем наших данных. Для начинающих пользователей они активно проверяли приемлемость данных, не допуская распространения ошибок. Мы создали средства, проверяющие наличие, включение, отношение к числам, вхождение в диапазон, уникальность, формат и подтверждение. (А также длину, если вы выполняли упражнения.) Мы создали специальные проверки, чтобы убедиться, что удаляемые товары не ссылаются на какую-нибудь товарную позицию. Мы использовали подключаемый метод Active Record, чтобы гарантировать, что остается хотя бы один администратор, а транзакцию, чтобы можно было осуществить откат незавершенных обновлений в случае сбоя.

Мы также создали логику для добавления товара в корзину, добавления всех товарных позиций из корзины к заказу, для раскодирования и аутентификации пароля и для вычисления различных итоговых результатов.

И наконец, мы создали порядок сортировки по умолчанию для товаров при их отображении.

Представление

Представления управляют способом, которым наше приложение преподносит само себя внешнему миру. По умолчанию временные платформы Rails предоставляют возможности работы с элементами данных: редактирования (`edit`), вывода перечня (`index`), создания нового элемента (`new`) и демонстрации элемента (`show`), а также предоставляют парциал по имени `form`, который по назначению находится между `edit` и `new`. Некоторые из них мы изменили, а также создали новые парциалы для корзин и товарных позиций.

В дополнение к представлениям ресурсов на основе моделей мы создали совершенно новые представления для контроллеров `admin`, `sessions` и самого `store`.

Мы обновили общий макет, чтобы получить однообразный внешний вид для всего сайта. Мы создали связи с таблицами стилей. Мы воспользовались шаблонами для генерирования JavaScript, использующего технологии Веб 2.0, для придания нашему веб-сайту большей интерактивности.

Мы воспользовались вспомогательными методами для определения момента, когда нужно убрать корзину из основного представления.

Мы локализовали представления для покупателей, чтобы они отображались как на английском, так и на испанском языке.

Хотя мы в основном сконцентрировались на HTML-представлениях, мы также создали представления, использующие обычный текст, и представления, использующие RSS-канал Atom. Мы разработали представления не только для браузеров, но также и для электронной почты, и эти представления способны использовать общие парциалы для отображения товарных позиций.

Контроллер

К моменту завершения нашей работы мы создали восемь контроллеров: по одному для каждой из пяти моделей и три дополнительных, чтобы поддержать представления для `admin`, `sessions` и самого `store`.

Эти контроллеры взаимодействуют с моделями несколькими способами: от поиска и извлечения данных с последующим помещением их в переменные экземпляра до обновления моделей и сохранения данных, введенных с помощью форм. После этого мы либо осуществляем перенаправление на другое действие, либо выводим представление. Мы выводили представления в HTML, JSON и Atom.

Мы создали функции обратного вызова, которые запускались перед выбранными действиями с целью запросов поиска корзины, установки нужного языка и авторизации. Мы поместили логику, общую для нескольких контроллеров, в концерн (consern), а именно — в модуль `CurrentCart`.

Мы управляли сессиями, отслеживали вошедшего в административную область пользователя (для администраторов) и корзины (для покупателей). Мы отслеживали текущий регион, используемый для локализации выводимой информации. Мы перехватывали ошибки, регистрировали их и информировали о них пользователей через уведомления.

Мы использовали фрагментарное кэширование для перечня товаров и кэширование на уровне страниц для RSS-канала Atom.

Мы также отправляли подтверждающие сообщения электронной почты о приеме заказа к исполнению.

Конфигурация

Хотя соглашения способствуют минимальному количеству необходимых настроек конфигурации для Rails-приложения, мы все же выполнили ряд настроек.

Мы изменили конфигурацию нашей базы данных, чтобы использовать в эксплуатационном режиме базу данных MySQL.

Мы определили маршруты для наших ресурсов, наших контроллеров `admin` и `session` и определили *главную страницу* нашего веб-сайта — наш каталог товаров. Мы определили в нашем ресурсе товаров действие `who_bought` (кто что купил), чтобы получить доступ к RSS-ленте Atom, содержащей информацию этого ресурса.

Мы создали инициализатор для локализации и обновления локальной информации как для английского (en), так и для испанского (es) языков.

Мы создали тестовые данные для нашей базы данных.

Мы создали сценарий Capistrano для развертывания, включая определение ряда специальных задач.

Тестирование

Мы привели в порядок и улучшили всевозможные тесты.

Для проверки методов мы воспользовались блочными тестами. Мы также протестировали увеличение количества заданной товарной позиции.

Rails предоставляет основные тесты для всех контроллеров временных платформ, которым придается должный вид по мере внесения изменений. В процессе разработки мы добавляли тесты для таких компонентов, как Ajax, и перед добавлением заказа убеждались, что в корзине есть товар.

Для подпитки наших тестов данными мы использовали стенды.

И наконец, мы создали комплексный тест для всего сценария, при котором пользователь добавляет товар в корзину, вводит данные для оформления заказа и получает подтверждающее сообщение по электронной почте.

Развертывание

Мы развернули наше приложение на эксплуатационном сервере (Apache httpd), используя подходящую для эксплуатации базу данных (MySQL). В процессе развертывания для запуска нашего приложения мы установили и настроили программу Phusion Passenger, для отслеживания зависимостей установили программу Bundler, а для управления конфигурированием нашего кода установили систему Git. Для дирижирования обновлением с нашей разработочной машины веб-серверов развертывания в эксплуатационный режим мы воспользовались программой Capistrano.

Для предотвращения влияния на эксплуатацию приложения в процессе его разработки мы воспользовались средами тестирования и эксплуатации. В нашей среде разработки мы воспользовались облегченным сервером базы данных SQLite и облегченным веб-сервером, скорее всего WEBrick. Наши тесты запускались в контролируемой среде с тестовыми данными, предоставляемыми испытательными стендами.

17.2. Документирование проделанной работы

Для завершения нашей ретроспективы давайте взглянем на код с двух новых точек зрения.

Для создания привлекательной документации программиста Rails упрощает запуск имеющейся в Ruby утилиты RDoc[1] в отношении всех имеющихся в приложении исходных файлов. Но перед генерированием этой документации нам, наверное, нужно создать привлекательную начальную страницу, чтобы будущее поколение разработчиков знало, чем занимается наше приложение.

Для этого нужно отредактировать файл README.rdoc и ввести в него все, что вы считаете полезным. Этот файл будет обработан RDoc, поэтому вы можете всецело воспользоваться гибкостями форматирования.

Для создания документации в HTML-формате можно воспользоваться следующей командой `rake`:

```
depot> rake doc:app
```

С ее помощью будет сгенерирована документация, которая будет помещена в каталог `doc/app`. Пример такой документации показан на рис. 17.1.

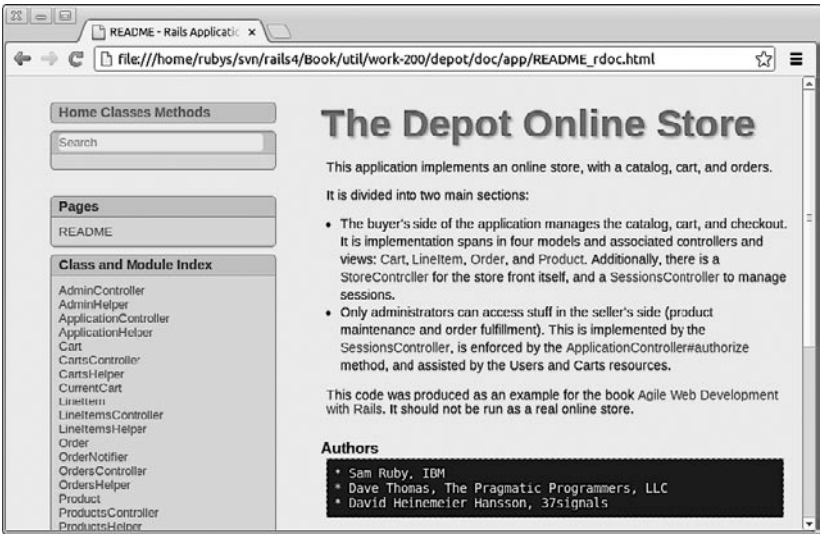


Рис. 17.1. Внутренняя документация нашего приложения

И наконец, нас может заинтересовать объем созданного программного кода. Для этого в Rake также имеется отдельная задача.

```
depot> rake stats
+----------------------+-------+-------+---------+---------+-----+-------+
| Name                 | Lines | LOC   | Classes | Methods | M/C | LOC/M |
+----------------------+-------+-------+---------+---------+-----+-------+
| Controllers          | 622   | 382   | 9       | 56      | 6   | 4     |
| Helpers              | 26    | 24    | 0       | 1       | 0   | 22    |
| Models               | 112   | 72    | 5       | 7       | 1   | 8     |
```

[1] http://rdoc.sourceforge.net/

```
| Mailers              | 29    | 11    | 1       | 2       | 2   | 3     |
| Javascripts          | 50    | 5     | 0       | 0       | 0   | 0     |
| Libraries            | 0     | 0     | 0       | 0       | 0   | 0     |
| Controller tests     | 404   | 283   | 8       | 0       | 0   | 0     |
| Helper tests         | 32    | 24    | 8       | 0       | 0   | 0     |
| Model tests          | 130   | 90    | 5       | 2       | 0   | 43    |
| Mailer tests         | 25    | 18    | 1       | 0       | 0   | 0     |
| Integration tests    | 198   | 138   | 2       | 9       | 4   | 13    |
+----------------------+-------+-------+---------+---------+-----+-------+
| Total                | 1628  | 1047  | 39      | 77      | 1   | 11    |
+----------------------+-------+-------+---------+---------+-----+-------+
Code LOC: 494     Test LOC: 553      Code to Test Ratio: 1:1.1
```

Если все это проанализировать, получится, что при весьма скромном объеме программного кода выполнен довольно большой объем работы. Более того, весьма существенная часть этого кода была для вас сгенерирована. В этом и заключается волшебство Rails.

III

Углубленное изучение Rails

18 Ориентация в мире Rails

Мы выдержали разработку проекта Depot, и настало время углубиться в изучение Rails. В оставшейся части книги мы будем изучать Rails тема за темой (а во многом — модуль за модулем). Большинство этих модулей нам уже встречалось в процессе предыдущей работы. Мы не только рассмотрим вопросы предназначения каждого модуля, но также узнаем, как расширить или даже заменить модуль и зачем это может понадобиться.

Главы третьей части охватывают все основные подсистемы Rails: Active Record, Active Resource, Action Pack (включая как Action Controller, так и Action View), а также Active Support. После этого вам будет предоставлено углубленное рассмотрение миграций. Затем мы собираемся углубиться во внутреннюю область Rails и показать, как компоненты объединяются в единое целое, как они запускаются и как они могут быть заменены.

Показывая, как могут быть объединены составные части Rails, мы завершим книгу обзором популярных взаимозаменяемых частей, многие из которых могут использоваться и за пределами Rails.

Но сначала следует подготовить почву. Эта глава охватывает весь общий материал, необходимый для понимания всего остального: структуру каталогов, конфигурацию и среду окружения.

18.1. Где что размещается

В Rails предполагается наличие конкретной схемы каталогов, используемой в процессе работы, и предоставляются приложения и генераторы временных платформ, создающие для вас эту схему. Например, если генерируется приложение my_app с использованием команды `rails new my_app`, то, как показано на рис. 18.1, для нашего нового приложения появляются каталоги верхнего уровня.

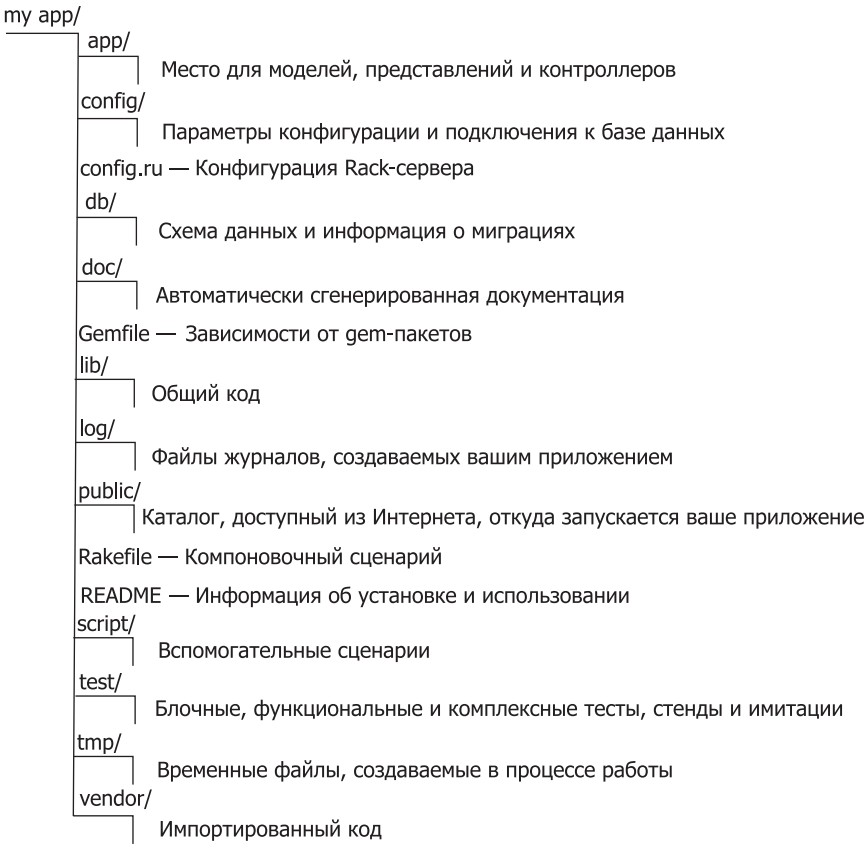


Рис. 18.1. У всех приложений Rails имеется эта структура каталогов верхнего уровня

ДЖО СПРАШИВАЕТ: ГДЕ ЖЕ ВСЕ-ТАКИ НАХОДИТСЯ RAILS?

Такой аспект Rails, как структура ее компонентов, представляет вполне определенный интерес. С точки зрения разработчика, все его время тратится на работу с высокоуровневыми модулями вроде Active Record и Action View. Но есть такой компонент под названием Rails, который постоянно находится за другими компонентами, молча дирижируя всем, что они делают, и заставляя все эти компоненты работать единым слаженным механизмом. Без Rails вряд ли что-то получится. Но вместе с тем лишь малая часть этой базовой инфраструктуры имеет отношение к повседневному труду разработчика. Именно эту часть мы и рассмотрим в данной главе.

Начнем с разбора текстовых файлов в верхней части каталога приложения.

- config.ru предназначен для конфигурации Rack Webserver Interface, либо для создания приложений Rails Metal, либо для использования в вашем Rail-

приложении Rack Middlewares. Дальнейшее рассмотрение данного файла доступно в Rails Guides[1].

- Gemfile определяет зависимости вашего Rails-приложения. Вы уже сталкивались с его использованием при добавлении gem-пакета `bcrypt-ruby` к приложению Depot. Зависимости приложения также включают базу данных, веб-сервер и даже сценарии, используемые для развертывания.

  Технически этот файл используется не самой средой Rails, а вашим приложением. Вызовы Bundler[2] можно найти в файлах config/application.rb и config/boot.rb.
- Gemfile.lock содержит записи о конкретных версиях каждого средства, от которого зависит ваше Rails-приложение. Этот файл обслуживается системой Bundler и должен быть зарегистрирован в вашем репозитории.
- Rakefile определяет задачи для запуска тестов, создания документации, извлечения текущей структуры вашей схемы и т. д. Полный список можно получить, если ввести в командную строку команду `rake -T`. Чтобы получить более подробное описание конкретной задачи, нужно набрать команду `rake -D` *задача*.
- README содержит общую информацию о самой среде Rails.

А теперь посмотрим, что попадает в каждый каталог (необязательно по порядку).

Место для нашего приложения

Основная часть нашей работы происходит в каталоге app. Как показано на рис. 18.2, основной код приложения размещается в подкаталогах каталога app. Более подробно структура каталога app будет рассмотрена в этой книге чуть позже, при изучении различных модулей Rails, таких как Active Record, Action Controller и Action View.

Место для наших тестов

Как мы уже видели в разделе 7.2 «Шаг Б2: Блочное тестирование моделей», в разделе 8.4 «Шаг В4: функциональное тестирование контроллеров» и в разделе 13.2 «Шаг 32: комплексное тестирование приложений», в Rails созданы вполне достаточные условия для проверки вашего приложения, и каталог test является домом для всего, что связано с тестированием, включая испытательные стенды, определяющие данные, которые используются в наших тестах.

---

[1] http://guides.rubyonrails.org/rails_on_rack.html
[2] https://github.com/carlhuda/bundler

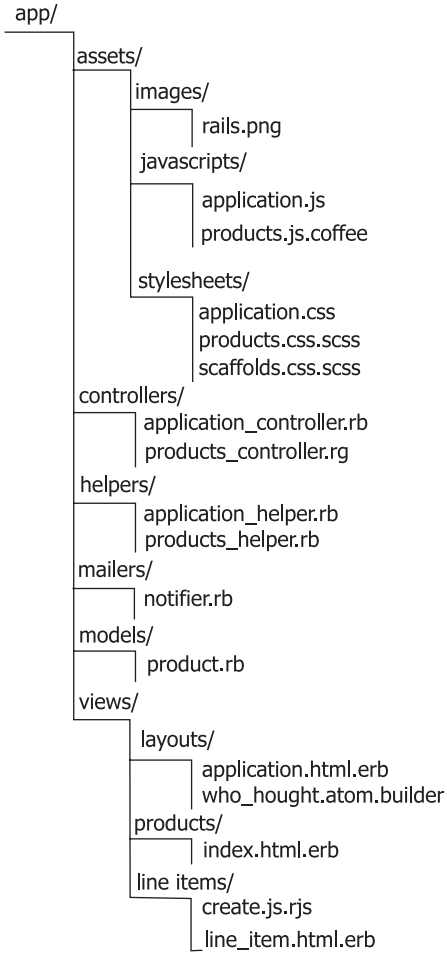


Рис. 18.2. Основной код нашего приложения находится в каталоге app

Место для документации

Хотя, как мы уже видели в разделе 17.2 «Документирование проделанной работы», каталог doc больше не входит в перечень обязательных каталогов, все же для генерирования документации, помещаемой в каталог doc/, Rails предоставляет Rake-задачу doc:app.

Кроме этой команды, в Rails предоставлена возможность выполнения других задач, генерирующих документацию: `doc:rails` создает документацию для запущенной вами версии Rails, а `doc:guides` создает руководства по использованию. Перед созданием руководств вам нужно будет добавить gem-пакет `redcarpet` в ваш Gemfile и запустить команду `bundle install`.

В Rails также предоставляется возможность запуска других задач, связанных с созданием документации. Чтобы просмотреть все эти задачи, следует набрать команду `rake -T doc`.

Место для поддерживающих библиотек

В каталоге lib содержится код приложения, который не вполне подходит для модели, представления или контроллера. Например, вы могли написать библиотеку, создающую кассовые чеки в формате PDF, которые могут быть загружены покупателями[1]. Эти чеки могут отправляться непосредственно из контроллера браузеру (используя метод `send_data()`). Код, который создает эти чеки в формате PDF, будет вполне естественно поместить в каталог lib.

Каталог lib также является хорошим местом для кода, совместно используемого моделями, представлениями или контроллерами. Возможно, вам понадобится библиотека, проверяющая контрольные суммы номеров кредитной карты или выполняющая какие-нибудь финансовые вычисления или вычисления даты Пасхи. Все, что не имеет непосредственного отношения к модели, представлению или контроллеру, должно попасть в каталог lib.

Не думайте, что вам нужно будет разместить пакет файлов непосредственно в самом каталоге lib. Вы можете по своему усмотрению создавать в каталоге lib подкаталоги, в которых группировать сходные по назначению функциональные средства. Например, на веб-сайте Pragmatic Programmer код, генерирующий чеки, документацию для отправки заказа покупателю и другие документы в формате PDF, находится в каталоге lib/pdf_stuff.

В предыдущих версиях Rails файлы в каталоге lib автоматически включались в путь загрузки, используемый для выполнения инструкций `require`. Теперь это элемент настройки, который нужно включить явным образом. Для этого в файл config/application.rb следует включить строку:

```
config.autoload_paths += %W(#{Rails.root}/lib)
```

Когда у вас будут файлы в каталоге lib и этот каталог будет добавлен в ваши пути автозагрузки (`autoload_paths`), вы сможете ими воспользоваться во всем остальном приложении. Если файлы содержат классы или модули и файлы названы с использованием имени класса или модуля в форме букв нижнего разряда, тогда Rails загрузит такой файл автоматически. Например, у нас может быть составитель чека в формате PDF, который находится в файле receipt.rb в каталоге lib/pdf_stuff.

Если только наш класс носит имя `PdfStuff::Receipt`, Rails сможет найти и загрузить его автоматически.

Для тех случаев, когда библиотека не может отвечать условиям автоматической загрузки, необходимо использовать имеющийся в Ruby механизм запроса, использующий инструкцию `require`. Если файл находится в каталоге lib, его можно запрашивать непосредственно по имени. Например, если наша библиотека вычисления

[1] ...что, собственно, и делается в магазине Pragmatic Programmer.

дня Пасхи находится в файле lib/easter.rb, мы можем включить ее в любую модель, представление или контроллер, используя следующую инструкцию:

```
require "easter"
```

Если библиотека находится в подкаталоге каталога lib, нужно не забыть включить имя этого подкаталога в инструкцию `require`. Например, для включения калькулятора доставки авиапочтой можно добавить следующую строку кода:

```
require "shipping/airmail"
```

Место для наших Rake-задач

В каталоге lib вы также найдете пустой подкаталог tasks. В этот подкаталог вы можете записывать свои собственные Rake-задачи, позволяющие вам добавлять автоматику к вашему проекту. Эта книга не посвящена Rake, поэтому мы не хотим здесь углубляться в эту тему, но простой пример приведем. Rails предоставляет Rake-задачу для сообщения о самой последней выполненной миграции.

Но может быть весьма полезно посмотреть на список *всех* выполненных миграций. Мы напишем Rake-задачу, выводящую версии, список которых находится в таблице `schema_migration`. Rake-задачи являются кодом на языке Ruby, но их нужно помещать в файлы с расширением .rake. Назовем нашу задачу db_schema_migrations.rake:

rails40/depot_t/lib/tasks/db_schema_migrations.rake

```
namespace :db do
    desc "Эта задача выводит примененные версии миграций "
    task :schema_migrations => :environment do
        puts ActiveRecord::Base.connection.select_values(
            'select version from schema_migrations order by version' )
    end
end
```

Эту задачу можно запустить из командной строки точно так же, как и любую другую Rake-задачу:

```
depot> rake db:schema_migrations
(in /Users/rubys/Work/...)
20121130000001
20121130000002
20121130000003
20121130000004
20121130000005
20121130000006
20121130000007
```

Дополнительные сведения о создании Rake-задач можно найти в документации по Rake по адресу: http://rubyrake.org/.

Место для наших регистрационных журналов

Как только Rails запускается, она производит целый ряд полезной регистрируемой информации. Эта информация сохраняется (по умолчанию) в каталоге log. В нем вы найдете три основных журнальных файла, которые называются development.log, test.log и production.log. Эти регистрационные журналы содержат не просто трассировочные строки, в них также содержится статистика затраченного времени, информация о кэшировании и расширения выполненных инструкций баз данных.

Какой из файлов используется, зависит от среды, в которой запущено ваше приложение (у нас еще будет что сказать о той или иной среде, когда речь пойдет о каталоге config в подразделе «Место для конфигурации»).

Место для статических веб-страниц

Каталог public является внешней поверхностью вашего приложения. Веб-сервер воспринимает этот каталог как основу вашего приложения. В него помещаются *статические* (иными словами, неизменяющиеся) файлы, как правило, связанные с запуском сервера.

Место для сценариев оболочек

Если вы сочтете полезным написание сценариев, запускаемых из командной строки и выполняющих различные задачи по обслуживанию вашего приложения, местом для оболочек, вызывающих такие сценарии, послужит каталог bin. Для заполнения этого каталога можно воспользоваться командой `bundle binstubs`.

В этом каталоге хранится также и сценарий Rails. Этот сценарий запускается, как только вы запускаете в командной строке команду `rails`. Первый аргумент, передаваемый вами этому сценарию, определяет выполняемую Rails функцию.

`console`
: Позволяет вам взаимодействовать с методами вашего Rails-приложения.

`dbconsole`
: Позволяет вам непосредственно взаимодействовать с вашей базой данных с использованием командной строки.

`destroy`
: Удаляет файлы, автоматически сгенерированные с помощью функции `generate`.

`generate`
: Генератор кода. Примечателен тем, что может создавать контроллеры, почтовые модули, модели, временные платформы и веб-службы. Для получения информации по использованию конкретного генератора нужно запустить функцию `generate` без всяких аргументов, например:

```
rails generate migration
```

`new`

Генерирует код Rails-приложения.

`runner`
: Выполняет метод в вашем приложении вне контекста сети. Является неинтерактивным эквивалентом консоли Rails (`rails console`). Эту функцию можно использовать для вызова из cron-задания методов, работающих со сроками кэширования, или для обработки входящих сообщений электронной почты.

`server`
: Запускает Rails-приложение на независимом веб-сервере, используя Mongrel (если он доступен в вашем блоке аппаратуры) или WEBrick. Мы использовали это в нашем приложении Depot в процессе его разработки.

Место для временных файлов

Наверное, тот факт, что Rails содержит свои временные файлы в каталоге tmp, не станет для вас сюрпризом. Там вы найдете подкаталоги для кэш-содержимого, сессий и сокетов. Обычно эти файлы очищаются Rails автоматически, но иногда, когда происходит какой-нибудь сбой, вам может понадобиться заглянуть туда и удалить старые файлы.

Место для кода сторонних разработчиков

Для кода сторонних разработчиков предназначен каталог vendor. Вы можете установить в каталог vendor Rails все средства, от которых зависит работа этой среды, как было показано в главе 16, в разделе «Получение контроля над приложением».

Если вы хотите вернуться к использованию версий gem-пакетов в масштабе всей системы, вы можете удалить каталог vendor/cache.

Место для конфигурации

Каталог config содержит файлы, конфигурирующие Rails. В процессе разработки Depot мы конфигурировали ряд маршрутов, конфигурировали базу данных, создавали инициализатор, изменяли локализацию и определяли инструкции по развертыванию. Вся остальная конфигурация была выполнена благодаря действующим в Rails соглашениям.

Перед запуском вашего приложения Rails загружает и исполняет файлы config/environment.rb и config/application.rb. Стандартная среда окружения, автоматически устанавливаемая этими файлами, включает следующие каталоги (относительно основного каталога вашего приложения) в пути загрузки вашего приложения.

- Каталог app/controllers и его подкаталоги.
- Каталог app/models.

- Каталог vendor и lib, содержащийся в каждом подкаталоге plugin.
- Каталоги app, app/helpers, app/mailers, app/services и lib.

Каждый из этих каталогов добавляется к пути загрузки, только если он существует на самом деле.

Кроме этого, Rails загрузит конфигурационный файл для каждой среды. Этот файл размещается в каталоге environments, и в него помещаются настройки конфигурации, изменяющиеся в зависимости от среды.

Так делается потому, что Rails понимает, что ваши потребности как разработчика сильно разнятся при написании кода, при его тестировании и при запуске этого кода в эксплуатацию. При написании кода вам требуется большой объем регистрирования, удобная перезагрузка изменившихся исходных файлов, назойливая демонстрация уведомления об ошибках и т. д. При тестировании вам нужна система, существующая в изоляции, чтобы получать повторяющиеся результаты. В эксплуатационном режиме ваша система должна быть настроена на производительность и пользователь ничего не должен знать об ошибках.

Переключатель, задающий среду выполнения, является внешним по отношению к вашему приложению. Это означает, что для изменений при переходе от разработки к тестированию и далее к эксплуатации не требуется никакого кода приложения. В главе 16 «Задача Л: Развертывание и эксплуатация», среда указывалась в команде `rake` с использованием аргумента `RAILS_ENV`, а для Phusion Passenger она указывалась с использованием строки `RailsEnv` в конфигурационном файле вашего сервера Apache. При запуске WEBrick с помощью команды `rails server` используется ключ `-e`:

```
depot> rails server -e development
depot> rails server -e test
depot> rails server -e production
```

Если у вас есть особые запросы, например если вы предпочитаете иметь какую-нибудь *промежуточную* среду, вы можете создавать свои собственные среды. Вам нужно будет добавить новый раздел к конфигурационному файлу базы данных и новый файл к каталогу config/environments.

Что будет помещено в эти конфигурационные файлы, целиком и полностью зависит от вас. Список параметров конфигурации, доступных для установки, можно найти в руководстве «Configuring Rails Applications», которое генерируется с помощью команды `doc:guides`, рассмотренной в разделе «Место для документации». Эта информация также доступна в Интернете[1].

18.2. Соглашения об именах

Для новичков иногда становится загадкой, как Rails автоматически обрабатывает имена. Они удивляются, каким образом, когда они называют класс своей модели `Person`, Rails знает, что нужно искать таблицу базы данных под названием

[1] http://guides.rubyonrails.org/configuring.html

people (мн. ч. от «person»). В этом разделе вы узнаете, как работает это неявное именование.

Приводимые здесь правила являются соглашениями по умолчанию, используемыми в Rails. Все эти соглашения можно переписать, используя настройки конфигурации.

Смешанный регистр, подчеркивание и применение слов во множественном числе

Зачастую переменные и классы именуются с использованием кратких фраз. В Ruby применяется соглашение, при котором имена переменных пишутся в нижнем регистре, а слова разделяются знаком подчеркивания. Классы и модули именуются по-другому: в их именах не используется знак подчеркивания, а каждое слово фразы (включая первое) пишется с заглавной буквы. (Мы по вполне понятным причинам будем называть это смешанным регистром.) Это соглашение приводит к тому, что имена переменных пишутся в виде order_status, а имена классов в виде LineItem.

Rails поддерживает это соглашение и расширяет его действие по двум направлениям. Во-первых, она предполагает, что имена таблиц баз данных уподобляются именам переменных, то есть пишутся в нижнем регистре и используют знак подчеркивания между словами. Во-вторых, Rails также предполагает, что в именах таблиц всегда используется слово во множественном числе. В результате получаются имена таблиц в виде orders и third_parties.

С другой стороны, Rails предполагает, что имена файлов пишутся в нижнем регистре со знаком подчеркивания.

Rails использует эти сведения, относящиеся к соглашению об именах, для автоматического преобразования имен. Например, ваше приложение может иметь класс модели, который обрабатывает товарные позиции. Вы определили этот класс, воспользовавшись соглашением об именах, имеющихся в Ruby, и назвали его LineItem. На основе этого имени Rails автоматически придет к следующему заключению:

- что соответствующая таблица базы данных будет названа line_items. Это имя класса, приведенное к нижнему регистру с подчеркиваниями между словами и во множественном числе;
- что определение класса нужно искать в файле с именем line_item.rb (в каталоге app/models).

Контроллеры Rails имеют дополнительные соглашения об именах. Если в нашем приложении есть контроллер store, происходит следующее.

- Rails предполагает, что класс называется StoreController и находится в файле store_controller.rb каталога app/controllers.
- Rails также ищет вспомогательный модуль по имени StoreHelper, который находится в файле store_helper.rb каталога app/helpers.

- Она станет искать шаблоны представлений для этого контроллера в каталоге app/views/store.
- Она будет по умолчанию брать код вывода на экран для этих представлений и рассматривать его как макеты шаблонов, находящихся в файле store.html.erb или store.xml.erb в каталоге app/views/layouts.

Все эти соглашения показаны в следующих таблицах.

Присвоение имен в модели	
Таблица	line_items
Файл	app/models/line_item.rb
Класс	LineItem

Присвоение имен в контроллере	
URL	http://../store/list
Файл	app/controllers/store_controller.rb
Класс	StoreController
Метод	list
Макет	app/views/layouts/store.html.erb

Присвоение имен в представлении	
URL	http://../store/list
Файл	app/views/store/list.html.erb (или .builder)
Помощник	module StoreHelper
Файл	app/helpers/store_helper.rb

Есть и еще один интересный эффект. Прежде чем в обычном Ruby-коде сослаться на классы и модули, находящиеся в исходных файлах Ruby, эти файлы нужно включить в программу с помощью ключевого слова `require`. Поскольку среде Rails известна связь между именами файлов и именами классов, инструкция `require` в Rails-приложении, как правило, не нужна. При первой же ссылке на класс или модуль Rails использует соглашение об именах для преобразования имени класса в имя файла и предпринимает попытку автоматически загрузить этот файл. В результате чего можно просто сослаться на класс модели (назвать его имя), и эта модель будет автоматически загружена в ваше приложение.

Группировка контроллеров в модули

До сих пор наши контроллеры находились в каталоге app/controllers. Иногда полезно навести порядок и в этой структуре. К примеру, наш магазин может в конечном счете содержать ряд контроллеров, осуществляющих схожие, но не пересекающиеся административные функции. Вместо того чтобы засорять

пространство имен высокого уровня, мы можем сгруппировать их в единое пространство имен admin.

Rails справляется с этим путем использования простого соглашения об именах. Если, скажем, входящий запрос ссылается на контроллер по имени admin/book, Rails будет искать в каталоге app/controllers/admin контроллер book_controller. Заключительная часть имени контроллера будет всегда превращаться в файл по имени *имя*_controller.rb, а любая лидирующая часть путевой информации будет использована для навигации по подкаталогам, вложенным в каталог app/controllers.

ДЭВИД ГОВОРИТ: ЗАЧЕМ ДЛЯ ИМЕН ТАБЛИЦ ИСПОЛЬЗУЕТСЯ МНОЖЕСТВЕННОЕ ЧИСЛО?

Поскольку они хорошо вписываются в разговор. Так и есть: «Выберите товар (Product) из товаров (products)». Или: «В заказе много товарных позиций (Order has_many :line_items)».

Прослеживается явное намерение навести мосты между программированием и разговором, создав некий специализированный язык, который можно будет использовать в обоих процессах. Наличие такого языка означает исключение мысленного перевода, при котором можно запутаться, обсуждая описания продукта с заказчиком, когда речь о нем идет действительно как о товаре. А подобные просчеты в общении ведут к ошибкам.

Если следовать стандартным соглашениям, Rails облегчает работу, бесплатно предоставляя основную массу настроек. Таким образом, разработчики поощряются за правильное поведение, причем здесь речь идет скорее не об отказе от «собственного пути», а о приобретении бесплатной производительности при разработке приложения.

Представьте, что в нашей программе имеются две подобные группы контроллеров (скажем, admin/xxx и content/xxx) и в обеих группах определен контроллер book. Значит, файл с именем book_controller.rb должен быть в обоих подкаталогах, и admin, и content, находящихся в каталоге app/controllers. В обоих этих файлах контроллеров определен метод класса по имени BookController. Если Rails не предпримет каких-нибудь дополнительных мер, эти два класса будут конфликтовать друг с другом.

Чтобы справиться с этой ситуацией, Rails предполагает, что контроллеры в подкаталогах каталога app/controllers находятся в модулях Ruby, названных по именам подкаталогов.

Таким образом, контроллер book в подкаталоге admin будет объявлен как

```
class Admin::BookController < ActionController::Base
    # ...
end
```

А контроллер book из подкаталога content будет находиться в модуле Content:

```
class Content::BookController < ActionController::Base
    # ...
end
```

Таким образом, эти два контроллера сохранят независимость друг от друга внутри приложения. Шаблоны для этих контроллеров появятся в подкаталогах каталога app/views. Так, шаблон представления, соответствующий запросу

```
http://my.app/admin/book/edit/1234
```

будет находиться в файле

```
app/views/admin/book/edit.html.erb
```

Вам будет приятно узнать о том, что генератор контроллеров разбирается в концепции контроллеров в модулях и позволяет создавать их командами вида

```
myapp> rails generate controller Admin::Book действие1 действие2 ...
```

Наши достижения

У всего в Rails есть свое место, и мы методично исследовали каждый из этих закоулков. В каждом месте содержащиеся в нем файлы и данные следуют соглашениям об именах, которые также были рассмотрены. Попутно мы заполнили несколько пробелов.

- ☑ Мы сгенерировали как API-документацию, так и руководство пользователя для самой Rails.
- ☑ Мы добавили Rake-задачу для распечатки версий выполненных миграций.
- ☑ Мы показали, как нужно конфигурировать каждую из сред использования Rails.

Далее последует рассмотрение основных подсистем Rails, начиная с самой большой — Active Record.

19 Active Record

Основные темы:

- метод `establish_connection`;
- таблицы, классы, столбцы и атрибуты;
- идентификаторы и связи;
- операции создания, чтения, обновления и удаления;
- внешние вызовы и транзакции.

Модуль Active Record — это надстройка, предоставляемая средой Rails для обеспечения объектно-реляционного отображения (ORM). Она является составной частью Rails, реализующей модель вашего приложения.

В данной главе мы будем основываться на отображении данных на строки и столбцы, как это делалось в приложении Depot. Затем будет рассмотрено использование Active Record для управления связями таблиц, а также работа этого компонента в процессе, охватывающем операции создания, чтения, обновления и удаления (которые в программировании обычно называются CRUD-методами). И наконец, мы углубимся в жизненный цикл объекта Active Record (включая внешние вызовы и транзакции).

19.1. Определение структуры ваших данных

В приложении Depot мы определили несколько моделей, включая и модель `Order`. Конкретно эта модель имеет ряд свойств, примером которых может послужить

свойство, содержащее адрес электронной почты — `email`, относящееся к типу `String`. Кроме определяемых нами свойств Rails предоставляет свойство по имени `id`, содержащее первичный ключ, предназначенный для записи. Rails также предоставляет несколько дополнительных свойств, включая свойства, отслеживающие время последнего обновления каждой строки. И наконец, Rails поддерживает связи между моделями, такие как связь между заказами и товарными позициями.

Если как следует во всем разобраться, можно понять, что Rails предоставляет широкую поддержку моделей. Давайте изучим все по порядку.

Формирование структуры с использованием таблиц и столбцов

Каждый подкласс, создаваемый на основе `ActiveRecord::Base`, например наш класс `Order`, вбирает в себя содержимое отдельной таблицы базы данных. По умолчанию Active Record допускает, что имя таблицы, связанной с конкретным классом, является множественной формой имени этого класса. Если имя класса содержит несколько слов, начинающихся с заглавных букв, то предполагается, что в имени таблицы между этими словами ставятся знаки подчеркивания.

Имя класса	Имя таблицы	Имя класса	Имя таблицы
`Order`	`orders`	`LineItem`	`line_items`
`TaxAgency`	`tax_agencies`	`Person`	`people`
`Batch`	`batches`	`Datum`	`data`
`Diagnosis`	`diagnoses`	`Quantity`	`quantities`

Эти правила отражают философию Rails: имена классов должны быть представлены в единственном числе, а имена таблиц — во множественном числе.

Хотя в Rails большинство неправильных множественных форм слов обрабатываются корректно, вы можете столкнуться со случаями неправильной обработки. Тогда можно будет добавить в Rails понимание особенностей и непоследовательностей английского языка путем внесения корректив в файл изменений форм слов:

rails40/depot_t/config/initializers/inflections.rb

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

# Add new inflection rules using the following format. Inflections
# are locale specific, and you may define rules for as many different
# locales as you wish. All of these examples are active by default:
# ActiveSupport::Inflector.inflections(:en) do |inflect|
# inflect.plural /^(ox)$/i, '\1en'
# inflect.singular /^(ox)en/i, '\1'
# inflect.irregular 'person', 'people'
# inflect.uncountable %w( fish sheep )
# end
```

```
# These inflection rules are supported but not enabled by default:
# ActiveSupport::Inflector.inflections(:en) do |inflect|
# inflect.acronym 'RESTful'
# end

ActiveSupport::Inflector.inflections do |inflect|
    inflect.irregular 'tax', 'taxes'
end
```

Если вам приходится использовать унаследованные таблицы или вам не нравится подобное поведение Rails, именем таблицы, связанной с конкретной моделью, можно управлять путем задания значения свойства `table_name` для конкретного класса:

```
class Sheep < ActiveRecord::Base
    self.table_name = "sheep"
end
```

Экземпляры классов Active Record соответствуют строкам в таблице базы данных.

У этих объектов есть свойства, соответствующие столбцам таблицы. Возможно, вы заметили, что в нашем определении класса `Order` не упоминаются какие-либо столбцы таблицы `orders`. Причина в том, что Active Record определяет их в динамическом режиме во время выполнения приложения. Для конфигурации классов, которые вбирают в себя содержимое таблиц, Active Record создает отображение схемы, составляющей внутреннее содержимое базы данных.

В приложении Depot наша таблица заказов `orders` определена следующей миграцией:

rails40/depot_r/db/migrate/ 20121130000007_create_orders.rb

```
class CreateOrders < ActiveRecord::Migration
    def change
        create_table :orders do |t|
            t.string :name
            t.text :address
            t.string :email
            t.string :pay_type
            t.timestamps
        end
    end
end
```

Чтобы поиграть с этой моделью, давайте воспользуемся вполне подходящей для этого командой `rails console`. Сначала запросим список имен столбцов:

```
depot> rails console
Loading development environment (Rails 4.0.0)
>> Order.column_names
=> ["id", "name", "address", "email", "pay_type", "created_at", "updated_at"]
```

Затем запросим подробную информацию о столбце `pay_type`:

```
>> Order.columns_hash["pay_type"]
=> #<ActiveRecord::ConnectionAdapters::SQLite3Column:0x00000003618228
@name="pay_type", @sql_type="varchar(255)", @null=true, @limit=255,
@precision=nil, @scale=nil, @type=:string, @default=nil,
@primary=false, @coder=nil>
```

Обратите внимание на то, что модуль Active Record собрал довольно большой объем информации о столбце `pay_type`. Ему известно, что этот столбец является строкой длиной не более десяти символов, у него нет значения по умолчанию, он не является первичным ключом, и он может содержать значение `null`. Rails получает эту информацию путем отправки запроса к основной базе данных при первой попытке использования класса `Order`.

Свойства экземпляра Active Record, как правило, соответствуют данным в соотносящейся с этим экземпляром строки таблицы базы данных. Например, в нашей таблице `orders` могут содержаться следующие данные:

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders limit 1"
         id  = 1
       name  = Dave Thomas
    address  = 123 Main St
      email  = customer@example.com
   pay_type  = Check
 created_at  = 2013-01-29 14:39:12.375458
 updated_at  = 2013-01-29 14:39:12.375458
```

ДЭВИД ГОВОРИТ: А ГДЕ ЖЕ НАШИ СВОЙСТВА?

Представление администратора базы данных (database administrator, DBA) в качестве отдельной роли, отличающейся от роли программиста, привело некоторых разработчиков к видению четкой границы между кодом и схемой. Active Record размывает это различие, и ни в чем это не проявляется столь очевидно, как в отсутствии четкого определения свойства в модели.

Но в этом нет ничего страшного. Практика показала, что взгляды на схему базы данных, отдельный XML-файл отображения или встраиваемые в модель свойства несколько различаются. По общему мнению, здесь имеют место такие же различия, которые уже встречались при рассмотрении структуры Модель—Представление—Контроллер, только они менее масштабны.

Как только дискомфорт от трактовки табличной схемы как части модели рассеется, вы начнете извлекать преимущества из соблюдения принципа отхода от повторений — DRY (Don't Repeat Yourself). Когда понадобится добавить в модель некое свойство, нужно будет просто создать миграцию и перезагрузить приложение.

Удаление этапа «конструирования» из процесса эволюции схемы данных так же укладывается в принцип ускоренной разработки, как и создание всего остального кода. Гораздо проще начинать с небольшой схемы, а потом по мере необходимости заниматься ее расширением и изменением.

Если мы извлечем эту строку в объект Active Record, у этого объекта будут семь свойств. Значением свойства `id` будет `1` (тип данных `Fixnum`), значением свойства `name` будет строка `"Dave Thomas"` и т. д.

Доступ к этим свойствам мы будем получать с помощью методов-аксессоров. При воспроизводстве схемы Rails автоматически создает как метод для чтения, так и метод для записи:

```
o = Order.find(1)
puts o.name                #=> "Dave Thomas"
o.name = "Fred Smith"      # присвоение имени
```

Присвоение свойству значения не приводит к изменениям в базе данных — чтобы эти изменения произошли, объект следует сохранить.

Значение, возвращаемое методами чтения свойства, по возможности приводится Active Record к соответствующему типу данных Ruby (к примеру, если столбец таблицы базы данных имел значение отметки времени, будет возвращен объект `Time`). Если вы хотите получить значение свойства, не подвергшееся преобразованию, добавьте к имени фрагмент `_before_type_cast`, как показано в следующем примере кода:

```
product.price_before_type_cast          #=> 34.95, число с плавающей точкой
product.updated_at_before_type_cast     #=> "2013-02-13 10:13:14"
```

Внутри программного кода модели вы можете воспользоваться локальными методами чтения и записи свойств `read_attribute()` и `write_attribute()`. Эти методы в качестве строкового аргумента получают имя свойства.

Отображение типов данных SQL на типы данных Ruby показано в табл. 19.1. При этом у столбцов `Decimal` и `Boolean` есть свои особенности.

Таблица 19.1. Отображение типов данных SQL на типы данных Ruby

Тип данных SQL	Класс Ruby	Тип данных SQL	Класс Ruby
int, integer	Fixnum	-float, double	Float
decimal, numeric	BigDecimal	char, varchar, string	String
interval, date	Date	datetime, time	Time
clob, blob, text	String	boolean	см. текст главы

Rails отображает столбцы с типом данных `Decimals` без дробных позиций на объекты с типом данных `Fixnum`; а при наличии таких позиций отображение происходит на объекты с типом данных `BigDecimal`, гарантируя при этом сохранение точности.

В случае с типом данных `Boolean` Rails предоставляет удобный метод в виде вопросительного знака, добавленного к имени столбца:

```
user = User.find_by(name: "Dave")
if user.superuser?
    grant_privileges
end
```

Вдобавок к тем свойствам, которые вы определили, есть еще ряд свойств, которые Rails или предоставляет автоматически, или придает им специальное значение.

Дополнительные столбцы, предоставляемые Active Record

Существует ряд имен столбцов, имеющих особое значение для Active Record. Вот их краткий перечень.

`created_at`, `created_on`, `updated_at`, `updated_on`
: Автоматически обновляются показателем времени создания строки или ее последнего обновления. Для этого нужно обеспечить способность исходного столбца базы данных принимать данные в формате даты, даты и времени или строки. По соглашению, в Rails-приложениях используется суффикс `_on` для столбцов с датой и суффикс `_at` для столбцов, включающих время.

`id`
: Имя, используемое по умолчанию для столбца первичного ключа таблицы (см. раздел «Идентификация отдельных строк»).

`xxx_id`
: Имя, используемое по умолчанию для ссылки внешнего ключа на таблицу с множественной формой *xxx*.

`xxx_count`
: Используется для поддержки кэша счетчика для дочерней таблицы *xxx*.

Дополнительные модули, такие как `act_as_list`[1], могут определять дополнительные столбцы.

Как первичные, так и внешние ключи играют жизненно важную роль в операциях с базой данных и заслуживают дополнительного рассмотрения.

19.2. Определение местоположения записей и прослеживание их связей

В приложении Depot объекты `LineItems` (товарные позиции) имеют непосредственные связи с тремя другими моделями: `Cart`, `Order` и `Product`. Кроме этого, модели могут иметь опосредованные связи через промежуточные объекты ресурсов. Примером такой связи может послужить связь между `Orders` и `Products` через `LineItems`.

Все это возможно благодаря идентификаторам.

[1] https://github.com/rails/acts_as_list

Идентификация отдельных строк

Классы Active Record соотносятся с таблицами базы данных. Экземпляры класса соотносятся с отдельными строками в таблице базы данных. Например, в результате вызова метода `Order.find(1)` возвращается экземпляр класса `Order`, в котором содержатся данные строки, имеющей значение первичного ключа, равное единице.

При создании новой схемы для Rails-приложения вам, скорее всего, не захочется что-либо менять, и вы согласитесь с тем, что ко всем вашим таблицам будет добавлен первичный ключ `id`. Но если нужно работать с уже существующей схемой, Active Record даст вам простой способ замены имени первичного ключа таблицы, используемого по умолчанию.

Например, вы можете работать с существующей устаревшей схемой, использующей в качестве первичного ключа для таблицы `books` столбец с именем `ISBN`.

Это нужно указать в нашей модели Active Record, используя примерно следующий код:

```
class LegacyBook < ActiveRecord::Base
    self.primary_key = "isbn"
end
```

Обычно модуль Active Record сам проявляет заботу о создании новых значений первичного ключа для тех записей, которые создаются и добавляются к базе данных — они будут возрастающими целыми числами (возможно, с некоторыми разрывами в последовательности). Но если заменить имя столбца первичного ключа, нужно также взять на себя и присвоение первичному ключу уникального значения перед тем, как сохранять новую строку. Как ни удивительно, для этого мы по-прежнему будем присваивать значение свойству по имени `id`. Пока за это отвечает Active Record, значение для свойства первичного ключа всегда присваивается с использованием свойства по имени `id`. Имя столбца, используемого в таблице, устанавливается с помощью объявления `primary_key=`. В следующем примере кода используется свойство по имени `id`, даже притом, что первичный ключ в таблице базы данных хранится в столбце `isbn`:

```
book = LegacyBook.new
book.id = "0-12345-6789"
book.title = "My Great American Novel"
book.save
# ...
book = LegacyBook.find("0-12345-6789")
puts book.title       # => "My Great American Novel"
p book.attributes     #=> {"isbn" =>"0-12345-6789",
                      #    "title"=>"My Great American Novel"}
```

Чтобы запутать все еще больше, свойства объекта модели имеют имена столбцов `isbn` и `title`, а имя `id` там даже не появляется. Если нужно присвоить значение первичному ключу, следует использовать имя `id`. Во всех остальных случаях используется настоящее имя столбца.

Объекты модели также переопределяют Ruby-методы `id()` и `hash()`, чтобы те ссылались на первичный ключ модели. Это означает, что объекты модели с правильными идентификаторами могут использоваться в качестве ключей хэша. Это также означает, что несохраненные объекты модели не могут использоваться в качестве надежных ключей хэша (поскольку у них еще нет правильного идентификатора).

И еще одно заключительное замечание: Rails рассматривает два объекта модели эквивалентными (используя метод `==`), если они являются экземплярами одного и того же класса и имеют один и тот же первичный ключ. Это означает, что несохраненные объекты модели могут сравниваться в качестве эквивалентных, даже если у них разные значения свойства. Если вам когда-нибудь придется сравнивать несохраненные объекты модели (что случается довольно редко), вам может потребоваться переписать метод `==`.

Как вы увидите, столбцы `id` также играют важную роль в установке связей между таблицами.

Определение связей в моделях

Active Record поддерживает три типа связей между таблицами: «один к одному», «один ко многим» и «многие ко многим». Эти связи указываются путем добавления к моделям объявлений `has_one`, `has_many`, `belongs_to` и еще одного объявления с удивительным названием — `has_and_belongs_to_many`.

Связи типа «один к одному»

Связь типа «один к одному» (или, точнее, связь «один к нулю или к одному») реализуется с использованием внешнего ключа в одной строке одной таблицы, который ссылается не более чем на одну строку в другой таблице. Эта связь может осуществляться между заказами (`orders`) и выставленными счетами (`invoices`): для каждого заказа не может быть больше одного счета (рис. 19.1).

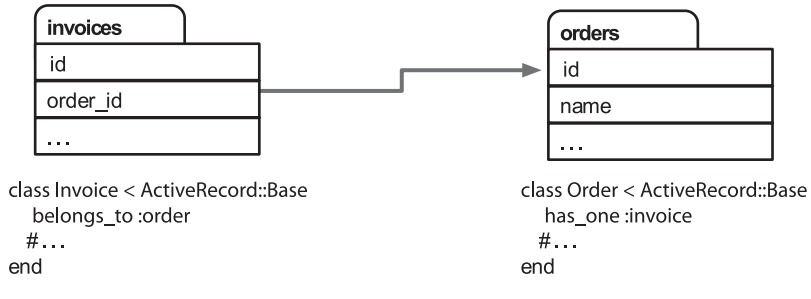


Рис. 19.1. Связь «один к одному»

Из примера видно, что эта связь указана в Rails добавлением к модели `Order` объявления `has_one` и добавлением к модели `Invoice` объявления `belongs_to`.

Здесь проиллюстрировано одно важное правило: модель для таблицы, имеющей внешний ключ, *всегда* содержит объявление `belongs_to`.

Связи типа «один ко многим»

Связь «один ко многим» позволяет представлять коллекцию объектов. Например, в заказе может быть любое число связанных с ним товарных позиций. В базе данных все строки товарных позиций, принадлежащих конкретному заказу, содержат столбец внешнего ключа, ссылающийся на этот заказ (рис. 19.2).

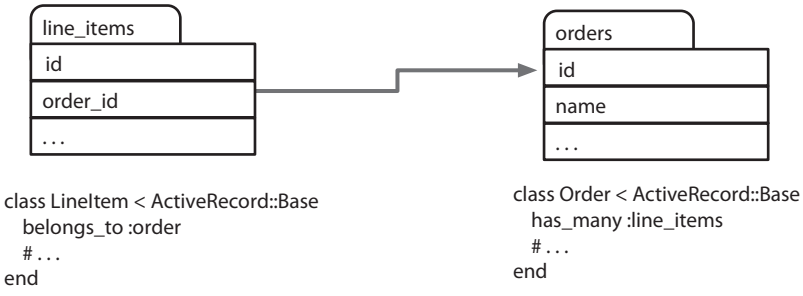


Рис. 19.2. Связь «один ко многим»

В Active Record родительский объект (тот самый, который логически содержит коллекцию дочерних объектов) использует для объявления своих связей с дочерней таблицей `has_many`, а для дочерней таблицы, чтобы указать на ее родителя, используется объявление `belongs_to`. В нашем примере класс `LineItem` принадлежит заказу: `belongs_to :order`, а класс, относящийся к таблице `orders`, содержит множество товарных позиций: `has_many :line_items`.

Опять-таки учтите: поскольку товарная позиция (`line item`) содержит внешний ключ, у нее имеется объявление о принадлежности `belongs_to`.

Связи типа «многие ко многим»

В завершение нужно определить категории наших товаров. Товар может принадлежать многим категориям, и каждая категория может содержать множество товаров. Это и является примером связи «многие ко многим». Похоже на то, что каждая из взаимосвязанных сторон содержит коллекцию элементов другой стороны (рис. 19.3).

В Rails это отношение выражается добавлением объявления `has_and_belongs_to_many` к обеим моделям. Связи «многие ко многим» являются симметричными, обе связываемые таблицы объявляют свою связь друг с другом, используя сокращение `habtm`.

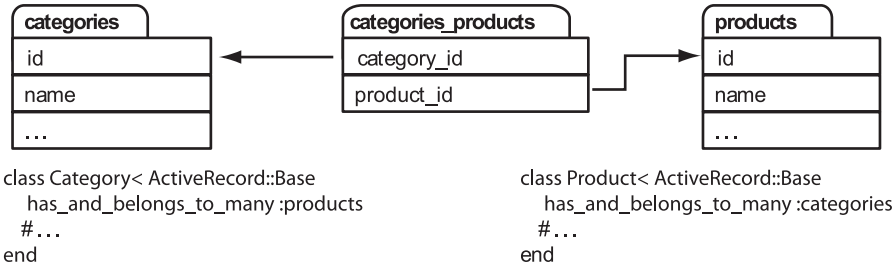


Рис. 19.3. Связь «многие ко многим»

Rails реализует связи «многие ко многим», используя промежуточную объединительную таблицу. В ней содержатся пары внешних ключей, связывающие две заданные таблицы. Active Record предполагает, что имя этой объединительной таблицы является объединением имен двух заданных таблиц, следующих в алфавитном порядке. В нашем примере объединяются таблицы `categories` и `products`, поэтому Active Record будет искать объединительную таблицу по имени `categories_products`.

Объединительные таблицы можно также определить непосредственным образом. В приложении Depot мы определили объединительную модель `LineItems`, которая объединяла `Products` либо с `Carts`, либо с `Orders`. Ее самостоятельное объявление также предоставило нам место для хранения дополнительных свойств, в частности `quantity` (количество).

После определений данных следующим вполне естественным побуждением будет доступ к данным, содержащимся в базе данных, — давайте к нему и приступим.

19.3. Создание, чтение, обновление, удаление (CRUD — Create, Read, Update, Delete)

Такие названия, как SQLite и MySQL, подчеркивают, что весь доступ к базе данных осуществляется путем использования языка структурированных запросов —Structured Query Language (SQL). В большинстве случаев Rails все возьмет на себя, но это целиком и полностью зависит от вас. Вскоре будет показано, что вы можете предоставлять выражения или даже полные SQL-инструкции, выполняемые базой данных.

Если вы уже знакомы с языком SQL, при чтении этого раздела обратите внимание на то, как Rails предоставляет места для знакомых выражений, таких как `select`, `from`, `where`, `group by` и т. д. Если вы еще не знакомы с SQL, вы узнаете, что одной из сильных сторон Rails является то, что вы можете отложить углубление в подобные области до тех пор, пока вам на самом деле не понадобится получить доступ к базе данных на этом уровне.

В этом разделе мы в качестве примера продолжим работу с моделью `Order` из приложения Depot. Методы Active Record будут использоваться для осуществления четырех основных операций с базой данных: создания, чтения, обновления и удаления.

Создание новых строк

С учетом того, что Rails представляет таблицы как классы, а строки как объекты, получается, что мы создаем строки в таблице путем создания новых объектов соответствующего класса. Мы можем создавать новые объекты, представляющие строки в нашей таблице заказов, путем вызова метода `Order.new()`. Затем мы можем заполнить значениями свойства (соответствующие столбцам в базе данных). В завершение мы можем вызвать принадлежащий объекту метод `save()` для сохранения заказа в базе данных. Без этого вызова заказ будет существовать только в нашей локальной памяти.

rails40/e1/ar/new_examples.rb

```
an_order = Order.new
an_order.name     = "Dave Thomas"
an_order.email    = "dave@example.com"
an_order.address  = "123 Main St"
an_order.pay_type = "check"
an_order.save
```

Конструкторы Active Record могут получать необязательный блок. Если такой блок присутствует, он вызывается вместе с только что созданным заказом в качестве аргумента. Этим можно воспользоваться, если нужно создать и сохранить заказ, не создавая новую локальную переменную.

rails40/e1/ar/new_examples.rb

```
Order.new do |o|
    o.name = "Dave Thomas"
    # . . .
    o.save
end
```

И наконец, конструкторы Active Record допускают использование в качестве аргумента хэша, состоящего из значений свойств. Каждая запись в этом хэше соответствует имени и значению устанавливаемого свойства. Это пригодится для выполнения таких задач, как сохранение значений из HTML-форм в строках базы данных.

rails40/e1/ar/new_examples.rb

```
an_order = Order.new(
    name:         "Dave Thomas",
    email:        "dave@example.com",
    address:      "123 Main St",
    pay_type:     "check")
an_order.save
```

Заметьте, что во всех этих примерах мы не устанавливаем для новой строки значение свойства `id`. Поскольку для первичного ключа использовался установленный Active Record по умолчанию столбец с целочисленными значениями, для него автоматически создавалось уникальное значение, а значение свойства `id` устанавливалось, как только строка сохранялась.

Впоследствии мы можем обнаружить это значение, запросив соответствующее свойство.

rails40/e1/ar/new_examples.rb

```
an_order = Order.new
an_order.name = "Dave Thomas"
# ...
an_order.save
puts "The ID of this order is #{an_order.id}"
```

Конструктор `new()` создает новый объект `Order` в памяти, и нам нужно не забыть своевременно сохранить его в базе данных. В Active Record есть очень удобный метод `create()`, который не только создает экземпляр объекта модели, но и сохраняет его в базе данных.

rails40/e1/ar/new_examples.rb

```
an_order = Order.create(
    name:        "Dave Thomas",
    email:       "dave@example.com",
    address:     "123 Main St",
    pay_type:    "check")
```

Методу `create()` можно передать массив, состоящий из хэшей свойств, тогда он создает сразу несколько строк в базе данных и возвращает массив соответствующих объектов модели:

rails40/e1/ar/new_examples.rb

```
orders = Order.create(
    [ { name:       "Dave Thomas",
        email:      "dave@example.com",
        address:    "123 Main St",
        pay_type:   "check"
      },
      { name:       "Andy Hunt",
        email:      "andy@example.com",
        address:    "456 Gentle Drive",
        pay_type:   "po"
      } ] )
```

Практический смысл в передаче методам `new()` и `create()` хэша значений обусловливается возможностью конструирования объектов модели непосредственно из параметров формы:

```
@order = Order.new(order_params)
```

Если эта строка вам что-то напоминает, так это потому, что вы уже ее видели ранее. Она появлялась в файле orders_controller.rb в приложении Depot.

Чтение существующих строк

Прежде чем читать данные из базы, нужно определить, какие конкретно строки данных нас интересуют, — нужно передать Active Record какие-то критерии, и вам будут возвращены объекты, содержащие данные из строки или строк, соответствующих этим критериям.

Простейшим способом поиска строки в таблице является указание значения ее первичного ключа. Каждый класс модели поддерживает метод `find()`, которому передается одно или более значений первичного ключа. Если ему передано всего лишь одно значение первичного ключа, он вернет объект, содержащий данные соответствующей строки (или выдаст исключение `ActiveRecord::RecordNotFound`, означающее, что строка не найдена). Если передать несколько значений первичного ключа, `find()` вернет массив, составленный из соответствующих объектов.

Учтите, что в этом случае исключение `RecordNotFound` будет выдано, если не будет найдено любое из этих значений `id` (поэтому если метод отработал без ошибки, длина полученного в результате его работы массива будет равна числу значений `id`, переданных в качестве аргументов):

```
an_order = Order.find(27) # find the order with id == 27

# Получение списка значений id из формы, а затем
# поиск связанных с ними товаров
product_list = Product.find(params[:product_ids])
```

И все-таки чаще всего требуется найти строку на основе критерия, не являющегося значением первичного ключа. Для осуществления таких запросов Active Record предоставляет дополнительные методы, позволяющие выражать более сложные запросы.

ДЭВИД ГОВОРИТ: ВЫДАВАТЬ ИЛИ НЕ ВЫДАВАТЬ ИСКЛЮЧЕНИЕ?

Когда ведется поиск по первичным ключам, ищется конкретная запись. Предполагается, что она существует. Вызов Person.find(5) основан на нашем знании таблицы people. Нам нужна строка с id, равным 5. Если вызов будет неудачным — запись с id, равным 5, уничтожена — мы попадем в исключительную ситуацию. Будут все основания для выдачи исключения, поэтому Rails выдаст RecordNotFound.

С другой стороны, чтобы найти соответствия, поисковые методы используют критерии поиска. Поэтому вызов Person.where(name: 'Dave').first равнозначен тому, что базе данных (выступающей в роли черного ящика) предписывается: «Выдай мне первую же строку таблицы, в которой есть имя Dave». Здесь отчетливо прослеживается иной подход к поиску — мы заранее не уверены, что получим какой-нибудь результат. Вполне возможно, что поиск пройдет впустую. Таким образом, вполне нормальным, неисключительным ответом будет возврат значения nil при поиске одной строки или возврат пустого массива, если велся поиск нескольких строк.

SQL и Active Record

Чтобы проиллюстрировать работу Active Record с SQL, передадим простую строку вызову метода `where()`, который соответствует SQL-условию `where`. Например,

чтобы был возвращен перечень всех заказов, сделанных человеком по имени `Dave` со способом оплаты `'po'`, можно воспользоваться следующей строкой:

```
pos = Order.where("name = 'Dave' and pay_type = 'po'")
```

Результат будет в объекте `ActiveRecord::Relation`, содержащем все соответствующие строки, каждая из которых будет аккуратно помещена в объект `Order`.

Это вполне подойдет для предопределенных условий, но что делать в ситуации, когда имя покупателя поступает извне (возможно, из веб-формы)?

Можно, конечно, подставить значение этой переменной в строку условий:

```
# получение имени из формы
name = params[:name]
# НЕ ДЕЛАЙТЕ ЭТОГО!!!
pos = Order.where("name = '#{name}' and pay_type = 'po'")
```

Из комментария следует, что это плохая затея. Почему? Да потому, что она создает огромную брешь в вашей базе данных для такого нехорошего явления, как инъекционные SQL-атаки, которые более подробно рассмотрены в руководствах по Rails. Их создание рассматривалось в главе 18, в разделе «Место для документации». А пока нужно поверить на слово, что подстановка строки из внешнего источника в SQL-инструкцию равнозначна открытию содержимого вашей базы данных для всего Интернета.

Лучше выбрать безопасный способ генерации динамических SQL-запросов и возложить всю обработку на Active Record. Это позволит Active Record создать полностью нейтральный код SQL, обладающий иммунитетом к инъекционным атакам. Посмотрим, как все это работает.

Если мы передаем вызову `where()` сразу несколько аргументов, Rails рассматривает первый аргумент как шаблон для генерации SQL-кода. Внутри этого SQL можно вставлять заменяемые элементы, на месте которых в процессе выполнения будут поставлены значения остальных элементов массива.

Один из способов установки заменяемых элементов заключается в расстановке в коде SQL одного или нескольких вопросительных знаков. Первый вопросительный знак будет заменен вторым элементом массива, второй — третьим и т. д. Например, предыдущему запросу можно придать следующий вид:

```
name = params[:name]
pos = Order.where(["name = ? and pay_type = 'po'", name])
```

Можно также использовать и поименованные заменяемые элементы. Каждый такой элемент должен иметь форму `:name`, а значение, которое его заменит, должно быть предоставлено в виде хэша, ключи которого соответствуют именам элементов в запросе:

```
name = params[:name]
pay_type = params[:pay_type]
pos = Order.where("name = :name and pay_type = :pay_type",
                  pay_type: pay_type, name: name)
```

Можно пойти еще дальше. Поскольку `params` на самом деле представляет собой хэш, его можно целиком передать в условия поиска. Если имеется форма, которую можно использовать для ввода критериев поиска, возвращаемый этой формой хэш значений можно использовать напрямую:

```
pos = Order.where("name = :name and pay_type = :pay_type",
                  params[:order])
```

Можно пойти и еще дальше. Если в качестве условия передать просто хэш, Rails сгенерирует условие `where`, используя ключи хэша в качестве имен столбцов, а значения хэша — в качестве значений для поиска соответствий. Благодаря этому предыдущий код можно записать еще короче:

```
pos = Order.where(params[:order])
```

Последнюю форму условия нужно применять с оглядкой: при конструировании условия здесь берутся все пары ключ-значение, имеющиеся в переданном хэше. Альтернативой может послужить указание, какие именно аргументы нужно использовать:

```
pos = Order.where(name: params[:name],
                  pay_type: params[:pay_type])
```

Независимо от формы заменяемых элементов, Active Record берет на себя всю заботу по расстановке кавычек и нейтрализации значений, подставляемых в SQL-код. При использовании этих форм динамического SQL Active Record сможет обезопасить вас от инъекционных атак.

Использование оператора Like

Если в условиях используются аргументированные операторы `like`, может появиться соблазн воспользоваться подобным кодом:

```
# Этот код работать не будет
User.where("name like '?%'", params[:name])
```

Rails не проводит синтаксический разбор SQL внутри условия, а поэтому не знает, что внутри строки заменяется имя. В результате Rails продолжает свою работу и ставит вокруг значения параметра `name` дополнительные кавычки. Правильно будет создать для оператора `like` весь аргумент и передать его в условие:

```
# Этот код работает
User.where("name like ?", params[:name]+"%")
```

Разумеется, при этом такие символы, как знак процента, если они появляются в значении аргумента имени, будут рассматриваться как символы-заместители.

Выделение подгрупп из возвращаемых записей

Узнав о том, как определяются условия, давайте обратим внимание на различные методы, поддерживаемые реляционным объектом класса `ActiveRecord::Relation`, начиная с `first()` и `all()`.

Как вы уже, наверное, догадались, `first()` возвращает первую строку из предоставленной информации. Если предоставлены пустые данные, этот метод возвращает `nil`. Следуя той же логике, метод `to_a()` возвращает все строки в виде массива. Объект класса `ActiveRecord::Relation` также поддерживает многие методы объектов `Array`, такие как `each()` и `map()`. При этом он сначала в неявном виде вызывает метод `all()`.

Важно понять, что запрос не вычисляется, пока не используется один из этих методов. Это позволяет нам изменять запрос несколькими способами путем вызова дополнительных методов до создания этого запроса. Теперь давайте посмотрим на эти методы.

order

SQL не гарантирует, что строки будут возвращены в каком-то определенном порядке, пока этот порядок не будет конкретно указан в запросе. Метод `order()` позволяет нам указать критерии, которые обычно добавляются после ключевых слов `order by`. Например, в результате выполнения следующего запроса будут возвращены все заказы, сделанные клиентом по имени `Dave`, отсортированные сначала по способу оплаты, а затем по дате отправки (в порядке убывания):

```
orders = Order.where(name: 'Dave').
    order("pay_type, shipped_at DESC")
```

limit

Вызов метода `limit()` позволяет ограничивать количество возвращаемых строк. Обычно при использовании метода ограничения возникает желание определить порядок сортировки, гарантирующий соответствующие результаты. Например, следующий код вернет первые десять соответствующих заказов:

```
orders = Order.where(name: 'Dave').
    order("pay_type, shipped_at DESC").
    limit(10)
```

offset

Метод `offset()` часто используется вместе с методом `limit()`. Он позволяет указать смещение первой строки в наборе возвращаемых результатов.

```
# Представление должно отобразить заказы, сгруппированные по страницам,
# на каждой из которых помещается группа заказов в количестве page_size.
# Этот метод возвращает заказы, предназначенные для страницы с номером
# page_num (счет начинается с нуля).

def Order.find_on_page(page_num, page_size)
    order(:id).limit(page_size).offset(page_num*page_size)
end
```

Чтобы перелистывать результаты запроса по `n` строк, можно использовать метод `offset` вместе с методом `limit`.

select

По умолчанию `ActiveRecord::Relation` извлекает из используемой базы данных все имеющиеся столбцы, применяя к ней инструкцию вида `select * from...`.

Отменить это положение можно с помощью метода `select()` с передачей строки, которая появится на месте символа * в инструкции `select`. Этот метод позволяет ограничить возвращаемые значения в том случае, если требуется вполне определенный набор данных, имеющихся в таблице. Например, таблица `podcasts` с описаниями звуковых файлов, предназначенных для интернет-вещания, может иметь сведения о заголовке, дикторе и дате, а также может содержать большой двоичный объект (blob) с речью в формате MP3. Если нужно составить только список записей, то загружать звуковые данные каждой строки будет незачем.

Метод `select()` позволяет выбрать загружаемые столбцы.

```
list = Talk.select("title, speaker, recorded_on")
```

joins

Метод `joins()` позволяет определить перечень дополнительных таблиц, присоединяемых к таблице, используемой по умолчанию. Он вставляет параметр в SQL-код сразу же после имени таблицы, связанной с моделью, и перед любыми условиями поиска, определяемыми первым параметром. Синтаксис присоединения зависит от типа используемой базы данных. Следующий код возвращает список всех товарных позиций, относящихся к книге с названием «Programming Ruby»:

```
LineItem.select('li.quantity').
    where("pr.title = 'Programming Ruby 1.9'").
    joins("as li inner join products as pr on li.product_id = pr.id")
```

readonly

Метод `readonly()` заставляет `ActiveRecord::Resource` вернуть объекты Active Record, которые не могут быть опять сохранены в базе данных.

При использовании метода `joins()` или `select()` объекты автоматически маркируются как `readonly`.

group

Метод `group()` добавляет оператор `group by` к SQL-коду:

```
summary = LineItem.select("sku, sum(amount) as amount").
                   group("sku")
```

lock

Метод `lock()` допускает в качестве необязательного аргумента строку, которая должна быть фрагментом кода SQL, имеющим синтаксис используемой базы данных, которым определяется вид блокировки. Например, при использовании MySQL блокировка *режима совместного доступа* дает самые последние данные, сохраненные в строке, и гарантирует, что никто другой не может внести в нее изменения до тех пор, пока она нами заблокирована. Можно создать код, который уменьшит средства на балансе счета в том случае, если их там достаточно для этой операции:

```
Account.transaction do
   ac = Account.where(id: id).lock("LOCK IN SHARE MODE").first
```

```
        ac.balance -= amount if ac.balance > amount
        ac.save
    end
```

Если не указать строковое значение или если дать методу `lock()` значение `true`, применяется блокировка с монополизацией, используемая в базе данных по умолчанию (обычно это блокировка для обновления данных — `"for update"`). Использование транзакций (которые будут рассмотрены в разделе 19.5 «Транзакции») зачастую избавляет от необходимости применения блокировки подобного рода.

Базы данных способны на большее, чем простой поиск и надежное извлечение данных, они могут также провести анализ, сокращающий объем необходимых данных. Rails предоставляет доступ и к этим методам.

Получение статистики столбцов

Rails имеет возможность выполнять статистические операции над значениями столбца. Например, применительно к таблице заказов можно вычислить следующее:

```
average  = Order.average(:amount)      # средняя стоимость заказов
max      = Order.maximum(:amount)
min      = Order.minimum(:amount)
total    = Order.sum(:amount)
number   = Order.count
```

Все эти строки соотносятся с функциями группировки используемой базы данных, но работают они в режиме, независимом от типа базы данных.

Как и прежде, методы можно объединять:

```
Order.where("amount > 20").minimum(:amount)
```

Эти функции группируют значения. По умолчанию они возвращают единственный результат, образующий, к примеру, минимальную стоимость заказа для тех заказов, которые отвечают некоторым условиям. Но если вы включили метод `group`, функции вместо этого выдают серию результатов, по одному результату для каждого набора записей, где группирующее выражение имеет одно и то же значение. Например, следующий код вычисляет максимальную стоимость заказа для каждого штата:

```
result = Order.group(:state).maximum(:amount)
puts result     #=> {"TX"=>12345, "NC"=>3456, ...}
```

Этот код возвращает упорядоченный хэш. Вы индексировали его, используя группирующий элемент (в нашем примере это `"TX"`, `"NC"` и т. д.). Можно также осуществить последовательный перебор записей, используя метод `each()`. Значение каждой записи становится значением, возвращаемым функцией группировки.

Методы `order` и `limit` при использовании групп выполняют присущие им задачи.

Например, следующий код возвращает строки, относящиеся к трем штатам, в которых сделаны самые дорогостоящие заказы, отсортированные по стоимости этих заказов:

```
result = Order.group(:state).
              order("max(amount) desc").
              limit(3)
```

Этот код уже не является независимым от применяемой базы данных, поскольку в методе `order` для сортировки по сгруппированным столбцам в функции группировки пришлось воспользоваться элементами синтаксиса SQLite (в данном случае `max`).

Области действия

По мере разрастания этой цепочки вызовов методов повторное использование таких цепочек становится проблематичным. Но Rails и здесь справляется с проблемой. Область действия Active Record может быть связана с Proc-объектом, вследствие чего может иметь аргументы:

```
class Order < ActiveRecord::Base
    scope :last_n_days, lambda { |days| where('updated < ?' , days) }
end
```

Такая поименованная область действия существенно упростит определение количества заказов за последнюю неделю:

```
orders = Order.last_n_days(7)
```

Более простые области действия вообще могут не иметь параметров.

```
class Order < ActiveRecord::Base
    scope :checks, -> { where(pay_type: :check) }
end
```

Области действия тоже могут объединяться. Можно также легко определить количество заказов за последнюю неделю, оплаченных чеками:

```
orders = Order.checks.last_n_days(7)
```

Кроме того, что области действия облегчают написание и чтение кода, они могут сделать ваш код более эффективным. Например, предыдущая инструкция реализуется в виде единого SQL-запроса.

Объекты `ActiveRecord::Relation` являются эквивалентом безымянной области действия:

```
in_house = Order.where('email LIKE "%@pragprog.com"')
```

Разумеется, реляционные объекты тоже могут быть объединены:

```
in_house.checks.last_n_days(7)
```

Области действия не ограничены условиями `where`; мы можем сделать практически все что угодно, вызывая методы `limit`, `order`, `join` и т. д. Нужно только учесть, что Rails не знает, как обрабатывать сразу несколько условий `order` или `limit`, поэтому в каждой цепочке вызовов нужно использовать только по одному соответствующему методу.

Практически в каждом случае будет достаточно тех методов, которые были рассмотрены. Но Rails не останавливается на том, что может справиться практически с каждым случаем, поэтому для тех случаев, которые требуют запросов, составленных вручную, также имеется соответствующий интерфейс прикладного программирования.

Написание собственного кода SQL

Каждый из рассмотренных методов вносит свой вклад в конструирование полной строки SQL-запроса. А метод `find_by_sql()` позволяет вашему приложению получить полный контроль над этим процессом. Он допускает использование одного аргумента, содержащего SQL-инструкцию `select` (или содержащего массив, состоящий из кода SQL и значений заменяемых элементов, как для метода `find()`), и возвращает (возможно, пустой) массив объектов модели из результирующего множества. Свойствами в этих объектах модели будет набор из столбцов, возвращенных запросом. Для возвращения всех столбцов таблицы обычно используется форма `select *`, но это необязательно.

rails40/e1/ar/find_examples.rb

```
orders = LineItem.find_by_sql("select line_items.* from line_items, orders    " +
                              "        where order_id = orders.id         " +
                              "        and orders.name = 'Dave Thomas'     ")
```

В результирующем множестве объектов модели будут доступны только те свойства, которые были возвращены в результате выполнения запроса. Определить свойства, доступные в объектах модели, можно с помощью методов `attributes()`, `attribute_names()` и `attribute_present?()`. Первый метод вернет хэш свойств, состоящий из пар имя-значение, второй метод вернет массив имен, а третий метод вернет `true`, если названное свойство доступно в этом объекте модели.

rails40/e1/ar/find_examples.rb

```
orders = Order.find_by_sql("select name, pay_type from orders")
first = orders[0]
p first.attributes
p first.attribute_names
p first.attribute_present?("address")
```

Выполнение этого кода даст следующие результаты:

```
{"name"=>"Dave Thomas", "pay_type"=>"check"}
["name", "pay_type"]
false
```

Метод `find_by_sql()` также может использоваться для создания объектов модели, содержащих данные, извлеченные из столбцов. Если для задания имен в результирующем множестве воспользоваться SQL-синтаксисом `as xxx`, эти имена будут задействованы в качестве имен свойств.

rails40/e1/ar/find_examples.rb

```
items = LineItem.find_by_sql ("                  select *,                        " +
                              "       products.price as unit_price,               " +
                              "   quantity*products.price as total_price,         " +
                              "            products.title as title                " +
                              "            from line_items, products              " +
                              "   where line_items.product_id = products.id       ")
li = items[0]
puts "#{li.title}: #{li.quantity}x#{li.unit_price} => #{li.total_price}"
```

Как и в случае применения условий, методу `find_by_sql` можно передать массив, содержащий в качестве первого элемента строку с заменяемыми элементами. Остальная часть массива должна быть либо хэшем, либо перечнем значений для подстановки.

```
Order.find_by_sql(["select * from orders where amount > ?",
                   params[:amount]])
```

В прежние времена при работе с Rails специалисты часто прибегали к использованию метода `find_by_sql()`. Сейчас те аргументы, которые были добавлены к основному методу `find()`, позволяют обойтись без обращения к этому низкоуровневому методу.

Перезагрузка данных

В тех приложениях, в которых база данных может быть потенциально доступна сразу нескольким процессам (или нескольким приложениям), всегда есть возможность того, что извлеченные объекты модели станут устаревшими — кто-нибудь мог записать в базу данных более свежую копию.

В некоторой степени этот вопрос относится к поддержке транзакций (которые будут рассмотрены в разделе 19.5 «Транзакции»). Но иногда все же случается, что вам требуется освежить объекты модели самостоятельно. Active Record облегчает эту задачу — для этого нужно лишь вызвать принадлежащий этому компоненту метод `reload()`, и свойства объекта будут заново получены из базы данных:

```
stock = Market.find_by(ticker: "RUBY")
loop do
    puts "Price = #{stock.price}"
    sleep 60
    stock.reload
end
```

На практике, кроме контекста блочных тестов, метод `reload()`используется крайне редко.

Обновление существующих строк

После столь продолжительного разговора о поисковых методах, наверное, приятно будет узнать, что про обновление записей в Active Record слишком долго говорить не придется.

Если имеется объект Active Record (скажем, отображающий строку из таблицы `orders`), его можно записать в базу данных, вызвав его же метод `save()`. Если этот объект ранее был прочитан из базы данных, его сохранение приведет к обновлению существующей строки, в противном случае при сохранении будет вставлена новая строка.

Если происходит обновление существующей строки, для сопоставления с объектом, находящимся в памяти, Active Record воспользуется ее же столбцом первичного ключа. Свойства, содержащиеся в объекте Active Record, определяют обновляемые столбцы: столбец базы данных будет обновлен только в том случае, если его значение было изменено. В следующем примере в таблице базы данных могут быть обновлены все значения строки заказа 123:

```
table:
order = Order.find(123)
order.name = "Fred"
order.save
```

Но в следующем примере объект Active Record содержит только такие свойства, как `id`, `name` и `paytype`, поэтому при его сохранении могут быть обновлены только соответствующие им столбцы. (Имейте в виду: если вы намерены сохранить строку, извлеченную с помощью метода `find_by_sql()`, нужно обязательно задействовать столбец `id`.)

```
orders = Order.find_by_sql("select id, name, pay_type from orders where id=123")
first = orders[0]
first.name = "Wilma"
first.save
```

ДЭВИД ГОВОРИТ: РАЗВЕ МОЖНО ПРИМЕНЕНИЕ SQL СЧИТАТЬ ЭТИЧНЫМ?

В ту пору, когда разработчики впервые заключили реляционные базы данных в объектно-ориентированную среду, они оговорили ту глубину, до которой нужно было доводить степень абстрагирования. Некоторые специалисты по объектно-реляционному отображению стремились полностью исключить использование SQL, надеясь достичь объектно-ориентированного подхода в чистом виде, вызывая все запросы исключительно в объектно-ориентированной среде.

Разработчики Active Record по этому пути не пошли. Это средство было разработано на основе понятия, что SQL, несмотря на его многословность даже в самом элементарном применении, не может быть ни изгоем, ни нарушителем чистоты подхода. Они сосредоточились на том, чтобы исключить необходимость многословия в самых обычных случаях (самостоятельный набор оператора insert с десятью атрибутами может утомить любого программиста), но сохранить при этом выразительность при составлении сложных запросов — ведь SQL был создан для изящного решения именно таких задач.

> Поэтому у вас не должно возникать чувство вины, когда метод find_by_sql() используется с целью избавления от узких мест, снижающих производительность, или с целью составления сложных запросов. Чтобы работа шла продуктивнее и доставляла удовольствие, следует начинать с использования объектно-ориентированного интерфейса, а затем, при крайней необходимости, нужно спуститься с небес и стать ближе к «железу».

В дополнение к методу `save()` Active Record позволяет изменять значения свойств и сохранять объект модели с помощью одного вызова метода `update()`:

```
order = Order.find(321)
order.update(name: "Barney", email: "barney@bedrock.com")
```

Чаще всего метод `update()` используется в действиях контроллера, когда данные, полученные из формы, объединяются с данными существующей строки базы данных:

```
def save_after_edit
    order = Order.find(params[:id])
    if order.update(order_params)
        redirect_to action: :index
    else
        render action: :edit
    end
end
```

Функции чтения и обновления строки можно объединить, используя методы класса `update()` и `update_all()`. Методу `update()` передается аргумент `id` и набор свойств. Он извлекает соответствующую строку, обновляет переданные ему свойства, сохраняет результат в базе данных и возвращает объект модели.

```
order = Order.update(12, name: "Barney", email: "barney@bedrock.com")
```

Методу `update()` можно передать массив идентификаторов (`id`) и массив, состоящий из хэшей, содержащих значения свойств, и он обновит в базе данных все соответствующие строки, вернув массив объектов модели.

И наконец, принадлежащий классу метод `update_all()` позволяет определить операторы `set` и `where` SQL-инструкции `update`. Например, в следующем примере на 10% увеличиваются цены всех товаров, в названии которых присутствует слово «Java»:

```
result = Product.update_all("price = 1.1*price", "title like '%Java%'")
```

Значение, возвращаемое методом `update_all()`, зависит от адаптера базы данных — большинство адаптеров (но только не адаптер Oracle) возвращают количество строк, измененных в базе данных.

save, save!, create и create!

Оказывается, у методов `save` и `create` есть две версии, различающиеся способом сообщения об ошибках:

Метод `save` возвращает `true`, если запись была сохранена, в противном случае он возвращает `nil`.

Метод `save!` возвращает `true`, если сохранение прошло успешно, в противном случае он выдает исключение.

Метод `create` возвращает объект Active Record независимо от того, был ли он успешно сохранен. Чтобы определить, записались ли данные, объект нужно подвергнуть проверке на наличие ошибок приемлемости данных.

Метод `create!` в случае успешной работы возвращает объект, в противном случае он выдает исключение.

Рассмотрим все это более подробно.

Старый добрый `save()` возвращает `true`, если объект модели не содержит ошибок и может быть сохранен:

```
if order.save
    # все в порядке
else
    # объект содержит ошибки
end
```

Вам решать, стоит ли проверять возвращаемое методом `save()` значение, чтобы определить, оправдал ли он ваши надежды. Смысл подобной снисходительности, заложенной в Active Record, заключается в предположении, что метод `save()` вызывается при выполнении метода, являющегося действием контроллера, и что код представления вернет все сообщения об ошибках конечному пользователю. Именно так во многих приложениях и происходит.

Если же объект модели требуется сохранить, обеспечив программную обработку всех ошибок, следует воспользоваться методом `save!()`. Если объект не может быть сохранен, этот метод выдаст исключение `RecordInvalid`:

```
begin
    order.save!
    rescue RecordInvalid => error
    # объект содержит ошибки
end
```

Удаление строк

Active Record поддерживает два способа удаления строк. Для начала есть два метода, определенные на уровне класса, `delete()` и `delete_all()`, которые работают непосредственно с базой данных. Метод `delete()` получает одиночный идентификатор (`id`) или массив идентификаторов и удаляет соответствующую строку или строки в используемой базе данных. Метод `delete_all()` удаляет строки, соответствующие заданным условиям (или все строки, если условия не определены). Возвращаемые значения при вызове обоих методов зависят от адаптеров, но обычно они содержат количество обработанных строк. Если строки до вызова функции не существует, исключение не выдается.

```
Order.delete(123)
User.delete([2,3,4,5])
Product.delete_all(["price > ?", @expensive_price])
```

Второй способ удаления строк, предоставляемый Active Record, основан на применении различных методов `destroy`. Все эти методы работают с объектами модели Active Record.

Принадлежащий экземпляру объекта метод `destroy()` удаляет из базы данных строку, соответствующую конкретному объекту модели. Затем содержимое модели «замораживается», предотвращая последующие изменения свойств.

```
order = Order.find_by(name: "Dave")
order.destroy
# ... теперь заказ "заморожен"
```

На уровне класса определены два метода уничтожения: `destroy()` (которому передается идентификатор (`id`) или массив идентификаторов) и `destroy_all()` (которому передается условие). Оба они считывают соответствующие строки таблицы базы данных в объекты модели и вызывают принадлежащий экземплярам этих объектов метод `destroy()`. Ни один из этих методов ничего стоящего не возвращает.

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

Но зачем нужны оба метода класса, и `delete()`, и `destroy()`? Методы `delete()` обходят различные функции проверок и обратные вызовы, присущие Active Record, а методы `destroy()` обеспечивают их безусловный вызов. Вообще-то, если нужно, чтобы база данных не противоречила бизнес-правилам, определенным в классах модели, лучше использовать методы `destroy`.

Функции проверок были рассмотрены в главе 7, в разделе «Задача Б: проверка приемлемости данных и блочное тестирование». А обратные вызовы будут рассмотрены далее.

19.4. Участие в процессе мониторинга

Active Record управляет жизненным циклом объектов модели — создает их, следит за ними при изменениях, сохраняет и обновляет их и с сожалением наблюдает за их уничтожением. Используя обратные вызовы, Active Record позволяет нашему коду принимать активное участие в этом процессе мониторинга. Мы можем создавать код, который будет вызван при любом значимом событии в жизни объекта. Обратные вызовы позволяют проводить сложные проверки, отображать значения столбцов, когда они передаются в базу данных или извлекаются из нее, и даже предотвращать завершение конкретных операций.

В Active Record определены шестнадцать обратных вызовов. Четырнадцать из них составляют пары «до-после» и охватывают некоторые операции над объектами Active Record. Например, обратный вызов `before_destroy` будет осуществлен непосредственно перед вызовом метода `destroy()`, а обратный вызов

`after_destroy` — после вызова метода `destroy()`. Исключение составляют два обратных вызова: `after_find` и `after_initialize`, у которых нет соответствующих им обратных вызовов вида `before_`*xxx*. (Как мы увидим далее, у этих двух обратных вызовов есть еще и другие отличия.)

На рис. 19.4 показано, как Rails шестнадцатью парными обратными вызовами охватывает базовые операции по созданию, обновлению и уничтожению объектов модели. Как ни удивительно, у обратных вызовов `before_` и `after_validation` нет четко определенного места.

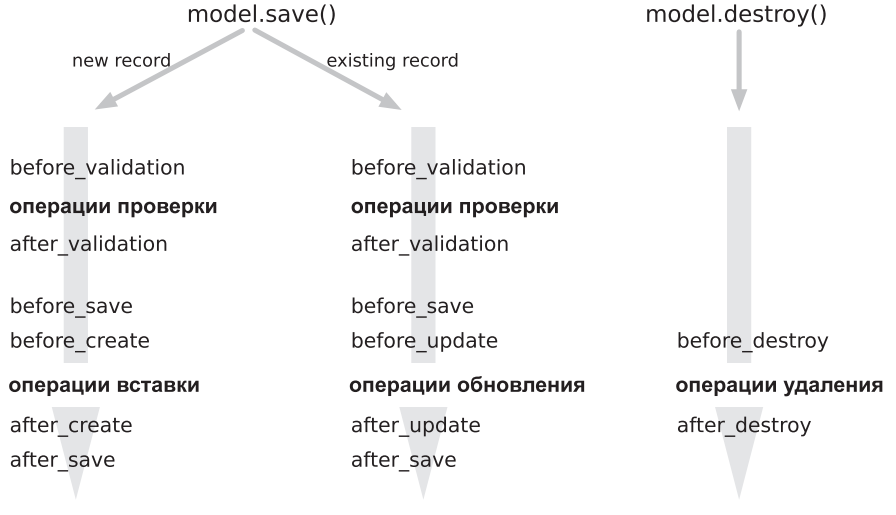


Рис. 19.4. Последовательность обратных вызовов Active Record

Обратные вызовы `before_validation` и `after_validation` также допускают использование параметра `on::create` (при создании) или параметра `on::update` (при обновлении), наличие которых приводит к тому, что соответствующий обратный вызов осуществляется только при выбранной этими аргументами операции.

В дополнение к этим шестнадцати вызовам обратный вызов `after_find` осуществляется после любой операции поиска, а обратный вызов `after_initialize` осуществляется после создания объекта модели Active Record.

Чтобы ваш код выполнялся при обратном вызове, нужно создать обработчик и связать его с соответствующим обратным вызовом.

Есть два основных способа реализации функций обратного вызова.

Более предпочтительный способ определения функции обратного вызова заключается в объявлении обработчиков. Обработчик может быть либо методом, либо блоком. Связь обработчика с конкретным событием осуществляется путем использования метода класса, названного по имени события. Чтобы связать метод, его нужно объявить закрытым (`private`) или защищенным (`protected`) и указать его имя в виде обозначения при объявлении обработчика. Для указания блока

нужно просто добавить этот блок после объявления. В качестве аргумента этот блок получает объект модели.

```
class Order < ActiveRecord::Base
    before_validation :normalize_credit_card_number
    after_create do |order|
        logger.info "Заказ #{order.id} создан"
    end
    protected
    def normalize_credit_card_number
        self.cc_number.gsub!(/[-\s]/, '')
    end
end
```

Для одного и того же обратного вызова можно определить сразу несколько обработчиков. Как правило, они будут вызваны в порядке объявления, если только обработчик не вернет значение `false` (и это должно быть настоящее значение `false`), в таком случае цепочка обработчиков обратного вызова будет разорвана на ранней стадии.

Кроме этого, вы можете определить методы экземпляра для обратного вызова, применяя объекты обратных вызовов, встраиваемые методы (через Proc-объект) или встраиваемые вычисляемые методы (eval metods), используя строковое значение. Более подробно об этом можно узнать из интерактивной документации[1].

Группировка взаимосвязанных обратных вызовов

Если у вас имеется группа взаимосвязанных обратных вызовов, может быть удобно сгруппировать их в отдельный класс обработчиков. Эти обработчики могут совместно использоваться сразу несколькими моделями. Класс обработчиков является обыкновенным классом, в котором определяются методы обратных вызовов (`before_save()`, `after_create()` и т. д.). Исходные файлы для этих классов обработчиков нужно создавать в каталоге app/models.

В объекте модели, использующем обработчик, нужно создать экземпляр этого класса обработчика для различных объявлений обратных вызовов.

Разобраться в данном вопросе помогут два примера.

Если наше приложение использует в нескольких местах кредитные карты, нам может понадобиться использовать наш общий метод `normalize_credit_card_number()` в нескольких моделях. Для этого мы выделим метод в его собственный класс и назовем его по имени события, которое он должен обрабатывать. Этот метод будет получать один аргумент — объект модели, генерирующий обратный вызов.

[1] http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html#label Types+of+callbacks

```
class CreditCardCallbacks

    # Приведение номера кредитной карты к общему виду
    def before_validation(model)
        model.cc_number.gsub!(/[-\s]/, '')
    end
end
```

Теперь осуществление этого общего обратного вызова можно организовать в наших классах моделей:

```
class Order < ActiveRecord::Base
    before_validation CreditCardCallbacks.new
    # ...
end

class Subscription < ActiveRecord::Base
    before_validation CreditCardCallbacks.new
    # ...
end
```

В данном примере в классе обработчика предполагается, что номер кредитной карты хранится в свойстве модели по имени `cc_number`; и свойство с таким же именем имеется в классах `Order` и `Subscription`. Но мы можем расширить границы применения, сделав класс обработчика менее зависимым от деталей реализации класса, в котором он используется.

Например, мы можем создать общий обработчик кодирования и раскодирования. Он может использоваться для кодирования имен полей перед их сохранением в базе данных и их раскодирования при считывании строки. Этот обработчик обратного вызова можно включить в любую модель, нуждающуюся в такой функции.

Обработчик должен зашифровать заданный набор свойств модели перед тем, как ее данные будут записаны в базу данных. Поскольку в модели нужно использовать обычную текстовую версию этих свойств, обработчик настроен на их раскодирование после того, как запись будет завершена. Раскодирование данных также понадобится при считывании строки базы данных в объект модели. Эти требования означают, что обработке необходимо подвергнуть события `before_save`, `after_save` и `after_find`. Поскольку раскодировать строку базы данных нужно будет как после сохранения, так и после поиска новой строки, код можно сохранить, задав для имени `after_find()` псевдоним `after_save()` — у одного и того же метода будут два имени.

rails40/e1/ar/encrypt.rb

```
class Encrypter
    # Сюда передается список свойств, которые должны
    # быть сохранены в базе данных в закодированном виде
    def initialize(attrs_to_manage)
        @attrs_to_manage = attrs_to_manage
    End
```

```
    # Поля кодируются перед сохранением или обновлением с использованием
    # NSA и DHS на основе шифра смещения
    def before_save(model)
        @attrs_to_manage.each do |field|
            model[field].tr!("a-z", "b-za")
        end
    end

    # Их раскодирование после сохранения
    def after_save(model)
        @attrs_to_manage.each do |field|
            model[field].tr!("b-za", "a-z")
        end
    end

    # Выполнение этого же действия после нахождения существующей записи
    alias_method :after_find, :after_save
end
```

В этом примере используется очень простое кодирование, поэтому перед использованием этого класса в реальном приложении его нужно будет заменить чем-то более серьезным.

Теперь мы можем сделать так, чтобы методы класса `Encrypter` вызывались из нашей модели заказов:

```
require "encrypter"

class Order < ActiveRecord::Base
    encrypter = Encrypter.new([:name, :email])

    before_save encrypter
    after_save encrypter
    after_find encrypter

protected
    def after_find
    end
end
```

Мы создали новый объект `Encrypter` и привязали его к событиям `before_save`, `after_save` и `after_find`. Таким образом, непосредственно перед сохранением заказа в объекте кодировщика будет вызван метод `before_save()` и т. д.

А зачем тогда мы определили пустой метод `after_find()`? Помните, мы говорили, что из соображений производительности `after_find` и `after_initialize` обрабатываются особым образом. Одним из последствий этого особого толкования является то, что Active Record не будет знать, что нужно вызвать обработчик `after_find`, пока не увидит реально существующий метод `after_find()` в классе модели. Поэтому чтобы обработать событие `after_find`, нам и нужно определить пустой заместитель метода `after_find()`.

Все это, конечно, хорошо, но каждый класс модели, желающий использовать наш кодирующий обработчик, должен включать те же восемь строк кода, которые

мы уже использовали в классе `Order`. Желание улучшить код ведет нас к определению вспомогательного метода, который проделает всю работу, чтобы сделать этот класс доступным для всех моделей Active Record. Для этого мы добавим его к классу `ActiveRecord::Base`:

rails40/e1/ar/encrypt.rb

```
class ActiveRecord::Base
    def self.encrypt(*attr_names)
        encrypter = Encrypter.new(attr_names)
        before_save encrypter
        after_save encrypter
        after_find encrypter
        define_method(:after_find) { }
    end
end
```

Теперь с его помощью мы можем добавить функцию кодирования к любым свойствам классов модели, используя всего один вызов.

```
class Order < ActiveRecord::Base
    encrypt(:name, :email)
end
```

Проверить этот код в действии можно с помощью простой демонстрационной программы:

```
o = Order.new
o.name = "Dave Thomas"
o.address = "123 The Street"
o.email = "dave@example.com"
o.save
puts o.name

o = Order.find(o.id)
puts o.name
```

В окне командной строки мы увидим имя нашего клиента (в виде обычного текста) в объекте модели:

```
ar> ruby encrypt.rb
Dave Thomas
Dave Thomas
```

А вот в базе данных имя и адрес электронной почты скрыты нашей стойкой кодировкой:

```
depot> sqlite3 -line db/development.sqlite3 "select * from orders"
         id  = 1
    user_id  =
       name  = Dbwf Tipnbt
    address  = 123 The Street
      email  = ebwf@fybnqmf.dpn
```

Обратные вызовы — прекрасная технология, но иногда она может привести к тому, что класс модели станет брать на себя обязанности, несвойственные самой природе этой модели. Например, в разделе 19.4 «Участие в процессе мониторинга», мы создали обратный вызов, который генерирует регистрационное сообщение при создании заказа. На самом деле эта функция обратного вызова не является частью базового класса `Order` — мы поместили ее туда, потому что в нем этот обратный вызов выполняется.

При умеренном использовании такой подход не вызывает особых проблем. Если же выявится наличие повторяющегося кода, нужно рассмотреть возможность использования вместо него таких структур, как концерны[1] (concern).

19.5. Транзакции

Транзакции баз данных сводят в группу ряд изменений таким образом, что все эти изменения либо применяются, либо отклоняются. Классический пример, доказывающий необходимость транзакций (из приводимых в документации по Active Record), — это перевод денег с одного банковского счета на другой. Основная логика довольно проста:

```
account1.deposit(100)          # зачисление на счет
account2.withdraw(100)         # снятие со счета
```

Но здесь следует проявить особую осмотрительность. Что, если зачисление на счет состоится, а снятие средств со счета по каким-нибудь причинам потерпит неудачу (возможно, покупатель превысил лимит кредитования)? Мы зачислили на баланс первого счета (`account1`) 100 долларов без соответствующего снятия средств со второго счета (`account2`). В результате мы создадим 100 долларов буквально из ничего.

Спасти нас смогут только транзакции. Транзакция иногда похожа на трех мушкетеров с их девизом «Один за всех и все за одного». В пределах транзакции либо будет успешно выполнена каждая SQL-инструкция, либо все они не произведут никакого эффекта. Иначе говоря, если какая-либо из инструкций потерпит неудачу, то вся транзакция не окажет на базу данных никакого влияния.

В Active Record для выполнения блока в контексте конкретной транзакции базы данных используется метод `transaction()`. В конце блока осуществляется передача транзакции и обновление базы данных, если только внутри самого блока не возникнет исключительная ситуация — тогда все изменения будут отменены, и база данных останется в прежнем состоянии. Поскольку транзакции осуществляются в контексте подключения к базе данных, нам нужно вызывать их с классом Active Record в качестве получателя.

Соответственно, мы можем создать следующий код:

```
Account.transaction do
    account1.deposit(100)
```

[1] http://37signals.com/svn/posts/3372-put-chubby-models-on-a-diet-with-concerns

```
    account2.withdraw(100)
end
```

Давайте поэкспериментируем с транзакциями. Начнем с создания новой таблицы базы данных. (Убедитесь в том, что ваша база данных поддерживает транзакции, иначе этот код работать не будет.)

rails40/e1/ar/transactions.rb

```
create_table :accounts, force: true do |t|
    t.string :number
    t.decimal :balance, precision: 10, scale: 2, default: 0
end
```

Затем мы определим простой класс банковского счета. В этом классе будут определены методы экземпляров для зачисления и снятия денежных средств. В нем также будут предоставлены некоторые основные виды проверок — для этого конкретного типа счета баланс не может быть отрицательным.

rails40/e1/ar/transactions.rb

```
class Account < ActiveRecord::Base
    validates :balance, numericality: {greater_than_or_equal_to: 0}
    def withdraw(amount)
        adjust_balance_and_save!(-amount)
    end

    def deposit(amount)
        adjust_balance_and_save!(amount)
    end

    private
    def adjust_balance_and_save!(amount)
        self.balance += amount
        save!
    end
end
```

Рассмотрим вспомогательный метод `adjust_balance_and_save!()`. В первой строке происходит простое обновление поля баланса. Затем метод вызывает `save!` для сохранения данных модели. (Следует помнить, что `save!` выдает исключение, если объект не может быть сохранен, — мы используем исключение, чтобы сообщить транзакции о неблагоприятном развитии событий.)

А теперь напишем довольно простой код для перевода денег с одного счета на другой:

rails40/e1/ar/transactions.rb

```
peter = Account.create(balance: 100, number: "12345")
paul = Account.create(balance: 200, number: "54321")

Account.transaction do
    paul.deposit(10)
    peter.withdraw(10)
end
```

Мы проверяем состояние базы данных и убеждаемся в том, что перевод состоялся:

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
         id  = 1
     number  = 12345
    balance  = 90
         id  = 2
     number  = 54321
    balance  = 210
```

Теперь создадим совершенно иную ситуацию. Если провести эту операцию повторно, но на этот раз попытаться перевести 350 долларов, мы превысим лимит средств, имеющихся на счете клиента по имени Peter, а это запрещено правилами проверки. Давайте испытаем это на практике:

rails40/e1/ar/transactions.rb

```
peter = Account.create(balance: 100, number: "12345")
paul = Account.create(balance: 200, number: "54321")
```

rails40/e1/ar/transactions.rb

```
Account.transaction do
    paul.deposit(350)
    peter.withdraw(350)
end
```

Когда код будет запущен на выполнение, создастся исключительная ситуация, сообщение о которой отобразится в окне командной строки:

```
.../validations.rb:736:in `save!': Validation failed: Balance is negative
from transactions.rb:46:in `adjust_balance_and_save'
: : :
from transactions.rb:80
```

Изучение базы данных покажет, что она не претерпела никаких изменений:

```
depot> sqlite3 -line db/development.sqlite3 "select * from accounts"
         id  = 1
     number  = 12345
    balance  = 100
         id  = 2
     number  = 54321
    balance  = 200
```

Однако здесь вас подстерегает ловушка. Базу данных от нарушения целостности транзакция защитила, а что при этом произошло с нашими объектами модели? Чтобы увидеть все, что с ними произошло, нам нужно перехватить исключение и позволить программе продолжить выполнение:

rails40/e1/ar/transactions.rb

```
peter = Account.create(balance: 100, number: "12345")
paul = Account.create(balance: 200, number: "54321")
```

rails40/e1/ar/transactions.rb

```
begin
    Account.transaction do
        paul.deposit(350)
        peter.withdraw(350)
    end
rescue
    puts "Перевод прекращен"
end

puts "У клиента Paul имеется #{paul.balance}"
puts "У клиента Peter имеется #{peter.balance}"
```

Перед нами предстанет удивительная картина:

```
Перевод прекращен
У клиента Paul имеется 550.0
У клиента Peter имеется -250.0
```

Хотя наша база данных не подверглась изменениям, объекты модели все же были обновлены. Причина в том, что модуль Active Record не отслеживал первоначальное и конечное состояние различных объектов — фактически он и не мог этого сделать, поскольку в его распоряжении не было простого способа получения сведений о том, какие модели принимали участие в транзакциях.

Встроенные транзакции

Когда в разделе «Определение связей в моделях» рассматривались родительские и дочерние таблицы, мы сказали, что Active Record при сохранении родительских строк обязательно сохраняет все зависящие от них дочерние строки. Это влечет за собой выполнение нескольких SQL-инструкций (одной для сохранения родительской информации и по одной для каждого измененного или нового дочернего элемента данных). Понятно, что это изменение должно быть атомарным, но до сих пор для сохранения взаимосвязанных объектов мы транзакциями не пользовались. Неужели мы допустили оплошность?

К счастью, нет. Active Record ведет себя вполне разумно, вкладывая все обновления и вставки, связанные с конкретным методом `save()` (а при удалении — с методом `destroy()`), в транзакции; либо все они будут успешно выполнены, либо все эти данные постоянной прописки в базе данных не получат. А задавать транзакции в явном виде вам придется лишь при самостоятельном управлении несколькими SQL-инструкциями.

Вообще-то транзакции — вещь еще более тонкая. Они проявляют так называемые ACID-свойства: они неразделимы (Atomic), обеспечивают согласованность (Consistency), не зависят ни от чего другого (Isolation) и дают устойчивые результаты (Durable) (эти свойства проявляются, когда транзакция уже передана). Если вы хотите придать жизнестойкость вашим приложениям, использующим базы данных, стоит поискать хорошую книгу по базам данных и дойти в ней до описания транзакций.

Наши достижения

Мы изучили важные структуры данных и соглашения об именах для таблиц, классов, столбцов, свойств, идентификаторов и связей. Мы увидели, как нужно создавать, читать, обновлять и удалять эти данные. И наконец, теперь мы понимаем, как можно использовать транзакции и обратные вызовы для предотвращения противоречивых изменений.

Все это в сочетании с проверками приемлемости данных, рассмотренными в главе 7 «Задача Б: Проверка приемлемости данных и блочное тестирование», охватывает основные элементы Active Record, о которых необходимо знать каждому Rails-программисту. Если возникнут особые потребности в получении сведений, не охваченных в данной главе, обратитесь к руководствам по Rails, создание которых рассматривалось в главе 18, в разделе «Место для документации».

Следующей важной рассматриваемой подсистемой будет модуль Action Pack, охватывающий такие составляющие Rails, как представление и контроллер.

20 Action Dispatch и Action Controller

Основные темы:

- передача репрезентативного состояния — Representational State Transfer (REST);
- определение порядка маршрутизации запросов к контроллерам;
- выбор формата отображения данных;
- тестирование маршрутов;
- среда окружения контроллера;
- визуализация и перенаправление;
- сессии, флэш и обратные вызовы.

В основе Rails-приложений лежит Action Pack. Он состоит из трех Ruby-модулей: Action Dispatch, Action Controller и Action View. Action Dispatch занимается маршрутизацией запросов к контроллерам. Action Controller превращает запросы в ответы.

Action View используется Action Controller для форматирования этих ответов. Если приводить конкретный пример, то в приложении Depot мы направляли адресацию на корневой каталог сайта (`/`) методу `index()` класса `StoreController`. В завершение выполнения этого метода происходили отправка и визуализация шаблона, находящегося в файле app/views/store/index.html.erb. Каждое из этих действий управлялось модулями, составляющими Action Pack.

Совместная работа этих трех подмодулей предоставляет поддержку обработки входящих запросов и генерирует исходящие ответы. В этой главе мы рассмотрим Action Dispatch и Action Controller, а в следующей главе — Action View.

При изучении Active Record мы увидели, что этот модуль может использоваться как самостоятельная библиотека, позволяющая использовать Active Record в качестве части несетевого Ruby-приложения. В этом плане Action Pack отличается от Active Record. Его, конечно, можно использовать непосредственно в качестве среды, но вам вряд ли захочется это делать. Лучше воспользоваться его тесной интеграцией с другими модулями, предоставляемой Rails. Такие компоненты, как Action Controller, Action View и Active Record, занимаются обработкой запросов, а среда окружения Rails связывает их в единое (и простое в использовании) целое. Поэтому мы даем описание Action Controller в контексте Rails. Давайте начнем с изучения того, как Rails-приложения обрабатывают запросы. Затем мы углубимся в подробности маршрутизации и обработки URL-адресов. После этого мы рассмотрим вопрос создания кода контроллера. И наконец, мы рассмотрим сессии, флэш и обратные вызовы.

20.1. Направление запросов контроллерам

В самом простом представлении веб-приложение принимает от браузера входящий запрос, обрабатывает его и отправляет ответ.

И первое, о чем хочется спросить, как приложение узнает, что ему нужно делать с входящим запросом? Интернет-магазин будет получать запросы на вывод каталога товаров, на добавление товаров в корзину, на создание заказа и т. д. Как он направляет эти запросы соответствующему коду?

Оказывается, Rails предоставляет два способа определения порядка маршрутизации запроса: подробный, используемый при необходимости, и удобный, который обычно используется везде, где только можно.

Подробный способ позволяет определять непосредственное отображение URL-адресов на действия на основе шаблона соответствия, требований и условий. Удобный способ позволяет определять маршруты на основе таких ресурсов, как определенная вами модель. И, поскольку удобный способ основан на подробном способе, вы можете произвольно смешивать и подгонять друг под друга оба способа.

В обоих случаях Rails кодирует информацию в URL-адресе запроса и использует подсистему под названием Action Dispatch для определения того, что должно быть сделано с этим запросом. Реальный процесс очень гибок, и по его завершении Rails определяет имя контроллера, обрабатывающего данный конкретный запрос, а также перечень любых других параметров запроса. В ходе этого процесса либо один из этих дополнительных параметров, либо сам HTTP-метод используется для определения *действия*, которое должно быть вызвано в заданном контроллере.

Маршрутизация Rails поддерживает отображение URL-адресов на действия на основе содержимого URL-адреса и на основе HTTP-метода, использованного для выдачи запроса. Мы уже видели, как это делается от URL к URL с использованием безымянных и поименованных маршрутов. Rails также поддерживает высокоуровневый метод создания групп родственных маршрутов. Чтобы понять, зачем это сделано, нам нужно немного отвлечься на изучение мира передачи репрезентативного состояния — Representational State Transfer (REST)

REST: передача репрезентативного состояния

Идеи, положенные в основу REST, были сформулированы в главе 5 докторской диссертации Роя Филдинга (Roy Fielding) в 2000 году[1]. В подходе, использованном в REST, серверы поддерживают связь с клиентами, используя подключения без указания их статуса: вся информация о состоянии взаимодействия между двумя системами закодирована внутри запросов и ответов, которыми они обмениваются. Долгосрочное состояние хранится на сервере в виде набора идентифицируемых ресурсов.

Клиенты получают доступ к этим ресурсам, используя четкий (и строго ограниченный) набор идентификаторов ресурсов (в нашем контексте — URL). REST отличает содержимое этих ресурсов от представления этого содержимого. REST спроектирован для поддержки высокой масштабируемости при обработке данных, а также для удерживания архитектуры приложения от разрушения естественных связей.

В нашем описании слишком много абстрактных понятий. А в чем же заключается практическое значение REST?

Во-первых, формальности подхода REST предполагают, что проектировщики сети знают, когда и где они могут кэшировать ответы на запросы. Это способствует снижению загруженности сети, повышая ее производительность и устойчивость наряду с уменьшением времени отклика.

Во-вторых, налагаемые REST ограничения могут упростить создание (и сопровождение) приложений. Приложения, использующие REST, не беспокоятся о реализации служб удаленного доступа. Вместо этого они предоставляют стандартное (и простое) взаимодействие с набором ресурсов. Ваше приложение предоставляет способ регистрации, создания, редактирования и удаления каждого ресурса, а ваши клиенты делают все остальное.

Сделаем изложение более конкретным. В REST, чтобы работать с богатым набором существительных, используется простой набор глаголов. Если используется HTTP, глаголы соответствуют методам HTTP (обычно GET, PUT, PUTCH, POST и DELETE). В качестве существительных выступают ресурсы нашего приложения. Мы указываем на эти ресурсы, используя URL.

Созданное нами приложение Depot содержит набор товаров. Здесь подразумевается наличие ресурсов двух видов. Прежде всего, это отдельные товары. Каждый из них учреждает некий ресурс. Также существует и второй ресурс — коллекция товаров.

Чтобы получить перечень всех товаров, мы можем послать HTTP-запрос GET к этой коллекции, указав в пути фрагмент `/products`. Чтобы получить содержимое отдельного ресурса, его нужно идентифицировать. Способ, свойственный Rails, будет заключаться в предоставлении значения его первичного ключа (который и будет служить идентификатором). Для этого нужно опять послать GET-запрос, на сей раз по URL `/products/1`.

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Для создания нового товара в нашей коллекции мы воспользуемся HTTP-запросом POST, направленным по пути `/products`, отправляя в нем данные, в которых содержатся сведения о добавляемом товаре. Да, это тот же самый путь, который использовался нами для получения перечня товаров: если вы посылаете по этому пути GET-запрос, в ответ получаете перечень, а если туда посылается POST-запрос, к коллекции добавляется новый товар.

Сделаем еще один шаг. Мы уже поняли, что послав GET-запрос по пути `/products/1`, можно получить сведения о товаре. Для обновления сведений о товаре нужно послать HTTP-запрос PUT по тому же самому URL-адресу. А чтобы удалить эти сведения, можно послать HTTP-запрос DELETE, используя тот же самый URL.

Пойдем дальше. Возможно, наша система также отслеживает пользователей. Тогда в нашем распоряжении опять есть набор ресурсов, и REST предписывает использовать аналогичный набор глаголов (GET, POST, PUTCH, PUT и DELETE) и посылать запросы по похожему набору URL (/users, /user/1, ...).

Вот теперь мы понимаем, в чем отчасти состоит мощь тех ограничений, которые накладываются REST. Мы уже знакомы с тем, что Rails заставляет придерживаться вполне определенной структуры наших приложений. Философия REST также предписывает нам структурировать интерфейс к нашим приложениям. Все как-то сразу упростилось.

В Rails имеется непосредственная поддержка для этого типа интерфейса; она добавляет некую разновидность макромаршрутов, называемых *ресурсами*. Давайте посмотрим, как может выглядеть файл config/routes.rb, используемый в главе 6, в разделе «Создание Rails-приложения».

```
Depot::Application.routes.draw do
    resources :products
end
```

Строка `resources` привела к добавлению к нашему приложению семи новых маршрутов. Попутно было выдвинуто предположение, что в приложении будет контроллер под названием `ProductsController`, содержащий семь действий с заданными именами.

Вы можете посмотреть на сгенерированные для нас маршруты. Это можно сделать, воспользовавшись удобной командой `rake routes`:

```
Prefix Verb   URI Pattern
              Controller#Action
products      GET        /products(.:format)
                    {:action=>"index", :controller=>"products"}
              POST       /products(.:format)
                    {:action=>"create", :controller=>"products"}
new_product   GET        /products/new(.:format)
                    {:action=>"new", :controller=>"products"}
edit_product  GET        /products/:id/edit(.:format)
                    {:action=>"edit", :controller=>"products"}
product       GET        /products/:id(.:format)
                    {:action=>"show", :controller=>"products"}
```

```
PATCH      /products/:id(.:format)
     {:action=>"update", :controller=>"products"}
DELETE     /products/:id(.:format)
     {:action=>"destroy", :controller=>"products"}
```

Все определенные маршруты разбиты по столбцам. На вашем экране эти строки вообще-то должны умещаться, но на этой странице каждый маршрут, чтобы он поместился, пришлось разбивать на две строки. Столбцами (которые здесь не указываются) являются имя маршрута, HTTP-метод, путь маршрута и (в отдельной строке на этой странице) требования к маршруту.

Поля в круглых скобках являются необязательной частью пути. Имена полей, предваряемые двоеточием, дают название переменным, в которые для дальнейшей обработки контроллером помещается сопоставляемая часть пути.

Рассмотрим теперь семь действий контроллера, к которым ведут эти маршруты. Несмотря на то что мы создали маршруты для управления товарами в нашем приложении, расширим тематику и поговорим о ресурсах — в конце концов, те же семь методов потребуются для всех маршрутов, создаваемых на основе ресурсов.

`index`
: Возвращает перечень ресурсов.

`create`
: Создает новый ресурс из тех данных, которые содержатся в POST-запросе, добавляя этот ресурс к коллекции.

`new`
: Создает новый ресурс и передает его клиенту. Этот ресурс не будет сохранен на сервере. Можно считать, что действие `new` создает пустую форму, передаваемую клиенту для заполнения.

`show`
: Возвращает содержимое ресурса, идентифицированного выражением `params[:id]`.

`update`
: Обновляет содержимое ресурса, идентифицированного выражением `params[:id]`, используя данные, связанные с запросом.

`edit`
: Возвращает содержимое ресурса, идентифицированного выражением `params[:id]`, в форму, приспособленную для редактирования этого содержимого.

`destroy`
: Удаляет ресурс, идентифицированный выражением `params[:id]`.

Нетрудно заметить, что эти семь действий включают четыре базовых операции для создания, чтения, обновления и удаления ресурса — CRUD (create, read, update, and delete). Среди них также имеется действие для вывода перечня ресурсов и два дополнительных действия, которые возвращают клиенту новый и существующий ресурсы в форму, предназначенную для редактирования.

Если по каким-нибудь причинам вам не нужны или нежелательны все семь действий, вы можете ограничить создаваемые действия с помощью параметров `:only` или `:except`, применяя их к своим ресурсам:

```
resources :comments, except: [:update, :destroy]
```

Некоторые маршруты называют маршрутами, позволяющими использовать такие вспомогательные функции, как `products_url` и `edit_product_url(id:1)`.

Заметьте, что каждый маршрут определен с необязательным спецификатором формата. Более подробно форматы будут рассмотрены в разделе «Выбор формата представления данных».

Давайте взглянем на код контроллера:

rails40/depot_a/app/controllers/products_controller.rb

```
class ProductsController < ApplicationController
    before_action :set_product, only: [:show, :edit, :update, :destroy]
    # GET /products
    # GET /products.json
    def index
        @products = Product.all
    end

    # GET /products/1
    # GET /products/1.json
    def show
    end

    # GET /products/new
    def new
        @product = Product.new
    end

    # GET /products/1/edit
    def edit
    end

    # POST /products
    # POST /products.json
    def create
        @product = Product.new(product_params)

        respond_to do |format|
            if @product.save
                format.html { redirect_to @product,
                    notice: 'Product was successfully created.' }
                format.json { render action: 'show', status: :created,
                    location: @product }
            else
                format.html { render action: 'new' }
                format.json { render json: @product.errors,
                    status: :unprocessable_entity }
            end
        end
    end
```

```
    # PATCH/PUT /products/1
    # PATCH/PUT /products/1.json
    def update
        respond_to do |format|
            if @product.update(product_params)
                format.html { redirect_to @product,
                    notice: 'Product was successfully updated.' }
                format.json { head :no_content  }
            else
                format.html { render action: 'edit' }
                format.json { render json: @product.errors,
                    status: :unprocessable_entity }
            end
        end
    end

    # DELETE /products/1
    # DELETE /products/1.json
    def destroy
        @product.destroy

        respond_to do |format|
            format.html { redirect_to products_url }
            format.json { head :no_content  }
        end
    end

    private
      # Use callbacks to share common setup or constraints between actions.
      def set_product
        @product = Product.find(params[:id])
      end

      # Never trust parameters from the scary internet, only allow the white
      # list through.
      def product_params
        params.require(:product).permit(:title, :description, :image_url, :price)
      end
end
```

Обратите внимание на то, что у нас имеется одно действие для каждого из действий RESTful. Комментарий перед каждым из них показывает формат вызывающего URL-адреса.

Также обратите внимание на то, что многие из действий содержат блок `respond_to()`. Как уже было показано в главе 11 «Задача E: добавление AJAX», Rails использует его для определения типа содержимого, отправляемого в качестве ответа. Генератор временных платформ автоматически создает код, который, соответственно, ответит на запросы, посылая содержимое в HTML- или JSON-формате. Совсем скоро мы посмотрим на все это на практике.

Представления, создаваемые генератором, довольно просты. Единственная хитрость в них понадобилась при использовании правильного HTTP-метода для отправки запросов серверу. К примеру, отображение для действия `index` выглядит следующим образом:

rails40/depot_a/app/views/products/index.html.erb

```
<h1>Listing products</h1>

<table>
<% @products.each do |product| %>
    <tr class="<%= cycle('list_line_odd', 'list_line_even') %>">

        <td>
            <%= image_tag(product.image_url, class: 'list_image') %>
        </td>

        <td class="list_description">
            <dl>
                <dt><%= product.title %></dt>
                <dd><%= truncate(strip_tags(product.description),
                    length: 80) %></dd>
            </dl>
        </td>

        <td class="list_actions">
            <%= link_to 'Show', product %><br/>
            <%= link_to 'Edit', edit_product_path(product) %><br/>
            <%= link_to 'Destroy', product, method: :delete,
                        data: { confirm: 'Are you sure?' } %>
        </td>
    </tr>
<% end %>
</table>

<br />

<%= link_to 'New product', new_product_path %>
```

Обе ссылки на действия по редактированию товара и добавлению нового товара должны использовать обычные GET-методы, поэтому стандартные `link_to` будут работать неплохо. Однако запрос на удаление товара должен выдавать HTTP-метод DELETE, поэтому вызов `link_to` включает параметр `method: :delete`.

Добавление дополнительных действий

Ресурсы Rails предоставляют вам исходный набор действий, но не вынуждают вас на этом останавливаться. В разделе 12.2 «Шаг Ж2: применение Atom-канала» мы добавили интерфейс, позволяющий извлекать список тех, кто приобрел тот или иной товар. Чтобы сделать это с использованием Rails, мы воспользуемся расширением к вызову ресурса:

```
Depot::Application.routes.draw do
    resources :products do
        get :who_bought, on: :member
    end
end
```

Применяемый синтаксист прост и понятен. Он говорит: «Нам нужно добавить новое действие по имени `who_bought`, вызываемое посредством HTTP-запроса GET. Это действие применяется к каждому элементу коллекции товаров».

Если вместо указания параметра `:member` будет использован параметр `:collection`, маршрут будет применяться к коллекции в целом. Это часто используется для определения области видимости — например, у вас может быть коллекция товаров для полной распродажи или товаров, снятых с продажи.

Вложенные ресурсы

Часто бывает так, что в самих наших ресурсах находится дополнительная коллекция ресурсов. Например, у нас может появиться потребность предложить людям оставлять рецензии на наши товары. Каждая рецензия будет ресурсом, и коллекция рецензий будет связана с каждым ресурсом товара.

Rails предоставляет для таких ситуаций удобный и интуитивно понятный способ объявления маршрутов:

```
resources :products do
    resources :reviews
end
```

Этот код определяет набор маршрутов верхнего уровня для товаров и дополнительно создает набор подмаршрутов для рецензий. Поскольку ресурсы рецензий появляются внутри блока товаров, они могут быть уточнены с помощью ресурса товара. Это означает, что путь к рецензии должен всегда иметь в качестве префикса путь к конкретному товару. Чтобы извлечь рецензию с идентификатором 4 для товара с идентификатором 99, будет использоваться путь `/products/99/reviews/4`.

Поименованным маршрутом для `/products/:product_id/reviews/:id` является `product_review`, а не просто `review`. Такой порядок присвоения имени отражает вложенность этих ресурсов.

Как обычно, вы можете увидеть полный набор маршрутов, сгенерированный благодаря нашей настройке конфигурации, воспользовавшись командой `rake routes`.

Концерны маршрутизации

До сих пор мы имели дело с относительно небольшим набором ресурсов. На более крупных системах могут быть такие типы объектов, которым свойственны рецензии или к которым имеет смысл применить действие `who_bought` («кто купил»). Вместо повторения соответствующих инструкций для каждого ресурса стоит присмотреться к перестройке маршрутов с помощью концернов (concern), позволяющих выделить общее поведение.

```
concern :reviewable do
   resources :reviews
end
```

```
resources :products, concern: :reviewable
resources :users, concern: :reviewable
```

Показанное определение ресурса `products` эквивалентно такому же определению из предыдущего раздела.

Неглубокое вложение маршрута

Временами вложенные ресурсы могут приводить к созданию громоздких URL-адресов. Решение этой проблемы заключается в использовании неглубокого вложения маршрута (shallow route nesting):

```
resources :products, shallow: true do
    resources :reviews
end
```

Это позволит распознать следующие маршруты:

```
/products/1            => product_path(1)
/products/1/reviews    => product_reviews_index_path(1)
/reviews/2            => reviews_path(2)
```

Чтобы увидеть полное отображение, воспользуйтесь командой `rake routes`.

Выбор формата представления данных

Одной из задач архитектуры REST является отделение данных от их представления. Если путь /products используется людьми для извлечения каких-нибудь товаров, они увидят красочно оформленное HTML-содержимое. Если тот же URL будет присутствовать в запросе какого-нибудь приложения, оно может выбрать получение результатов в удобном для кода формате (YAML, JSON или, возможно, XML).

Мы уже видели, как Rails может использовать HTTP-заголовок `Accept` в контроллере, в блоке `respond_to`. Тем не менее установить заголовок `Accept` удается не всегда (а иногда и вовсе не удается). Чтобы справиться с поставленной задачей, Rails предоставляет возможность передать формат желаемого ответа в виде составной части URL. Как вы уже видели, Rails делает это, включая в определение маршрута поле под названием `:format`. Для этого нужно установить значение параметра `:format` на расширение файла того MIME-типа, в котором требуется вернуть данные.

```
GET        /products(.:format)
        {:action=>"index", :controller=>"products"}
```

Поскольку точка (знак препинания) является в определениях маршрута разделительным символом, то `:format` уже рассматривается в качестве другого поля. Присвоение этому полю значения по умолчанию `nil` делает его необязательным.

После этого мы сможем использовать в нашем контроллере блок `respond_to()` для выбора типа ответа в зависимости от формата запроса:

```
def show
    respond_to do |format|
        format.html
        format.xml { render xml: @product.to_xml }
        format.yaml { render text: @product.to_yaml }
    end
end
```

Благодаря этому запрос к /store/show/1 или /store/show/1.html вернет содержимое в виде HTML, а запрос к /store/show/1.xml вернет XML-содержимое, так же как запрос к /store/show/1.yaml вернет содержимое YAML. Формат можно также передать в качестве параметра HTTP-запроса:

```
GET HTTP://pragprog.com/store/show/123?format=xml
```

В маршрутах, определяемых с помощью `resources`, это свойство включено по умолчанию.

Несмотря на привлекательность идеи об одном контроллере, готовящем ответы с различными типами содержимого, на самом деле оказывается, что не все так просто. В частности, затрудняется обработка ошибок. Несмотря на все удобства переадресации пользователя, в случае ошибки на форму, в которой демонстрируется красиво подсвеченное сообщение, когда дело доходит до XML, нужно принимать какую-то иную стратегию. Перед тем как решиться на сведение всей обработки в одном контроллере, нужно тщательно продумать архитектуру приложения.

Rails упрощает разработку приложения, которое полагается на маршрутизацию на основе ресурсов. Многие утверждают, что она существенно упростила создание программного кода их приложений. Но такая маршрутизация подходит не во всех случаях. Если вам не удается найти способ, заставляющий ее работать, не нужно чувствовать себя обязанным непременно этого добиться. Всегда можно найти и настроить смешанный вариант. Часть контроллеров может основываться на использовании ресурсов, а часть может основываться на действиях. Некоторые контроллеры могут даже быть основанными на использовании ресурсов и иметь несколько дополнительных действий.

20.2. Обработка запросов

В предыдущем разделе мы выяснили, как Action Dispatch направляет входящий запрос к предназначенному для него коду приложения. Теперь давайте посмотрим, что делается внутри этого кода.

Методы действий

Когда объект контроллера обрабатывает запрос, он ищет общедоступный метод экземпляра, имя которого совпадает с именем действия во входящем запросе. Найденный метод запускается на выполнение. Если метод не найден, контроллер выполняет метод `method_missing()`, передавая ему в качестве первого аргумента

имя действия, а в качестве второго — пустой список аргументов. Если ни один из методов не может быть вызван, контроллер ищет шаблон, названный по имени текущего контроллера и действия. Если шаблон будет найден, он отправляется клиенту напрямую. Если ничего из вышеперечисленного не произойдет, будет сгенерирована ошибка, связанная с неизвестным действием, — `AbstractController::ActionNotFound`.

Среда окружения контроллера

Контроллер устанавливает для действий среду окружения (и распространяет ее на вызываемые ими представления).

Многие из этих методов предоставляют непосредственный доступ к информации, содержащейся в URL или в запросе.

`action_name`
: Имя текущего выполняемого действия.

`cookies`
: Связанные с запросом cookie-файлы. Установленные в этом объекте значения сохраняют cookie-файлы на браузере при отправке ответа. Поддержка сессий в Rails основана на использовании cookie-файлов. Сессии будут рассмотрены в разделе «Сессии Rails».

`headers`
: Хэш HTTP-заголовков, которые будут использованы в ответе. По умолчанию заголовок `Cache-Control` установлен в `no-cache`. Для приложений специального назначения может потребоваться установить заголовки `Content-Type`. Учтите, что в заголовках невозможно напрямую устанавливать значения cookie — для этого следует использовать cookie API.

`params`
: Объект, похожий на хэш, в котором содержатся параметры запроса (наряду с псевдопараметрами, сгенерированными в процессе маршрутизации). Его сходство с хэшем состоит в том, что он допускает указание элементов с использованием либо обозначения, либо строки — выражения `params[:id]` и `params['id']` возвращают одно и то же значение. Для Rails-приложений более характерна форма, использующая обозначения.

`request`
: Объект входящего запроса. Он включает в себя следующие свойства:

  - `request_method` возвращает метод запроса, являющийся одним из следующих значений: `:delete`, `:get`, `:head`, `:post` или `:put`;
  - `method` возвращает то же самое значение, что и `request_method`, за исключением `:head`, которое возвращается как `:get`, потому что с точки зрения приложения эти два значения считаются функциональными эквивалентами;
  - `delete?`, `get?`, `head?`, `post?` и `put?` возвращают `true` или `false` в зависимости от метода запроса;

- `xml_http_request?` и `xhr?` возвращают `true`, если этот запрос был выдан одним из вспомогательных методов AJAX. Учтите, что этот параметр зависит от параметра `method`;
- `url()` возвращает полный URL, использованный для запроса;
- `protocol()`, `host()`, `port()`, `path()` и `query_string()` возвращают компоненты URL, использованного для запроса, на основе следующего шаблона: `protocol://host:port/path?query_string`;
- `domain()` возвращает последние два компонента доменного имени запроса;
- `host_with_port()` возвращает строку `host:port` запроса;
- `port_string()` возвращает значение строки `:port` запроса, если порт не является портом по умолчанию (80 для HTTP, 443 для HTTPS);
- `ssl?()` возвращает `true`, если это SSL-запрос, то есть если запрос был сделан с применением протокола HTTPS;
- `remote_ip()` возвращает в виде строки удаленный IP-адрес. Строка может иметь более одного адреса, если клиент использует прокси-сервер;
- `env()` возвращает среду окружения запроса. Его можно использовать для доступа к значениям, установленным браузером, как в следующем примере:

```
request.env['HTTP_ACCEPT_LANGUAGE']
```

- `accepts()`, возвращает массив объектов `Mime::Type`, представляющий MIME-типы заголовка `Accept`.
- `format()` возвращает значение, вычисляемое на основе значения заголовка `Accept`, с `Mime::HTML` в качестве альтернативного варианта.
- `content_type()` возвращает MIME-тип запроса. Может пригодиться при `put`- и `post`-запросах.
- `headers()` возвращает полный набор HTTP-заголовков.
- `body()` возвращает тело запроса в виде потока ввода-вывода.
- `content_length()` возвращает количество байт, подразумеваемых в качестве тела запроса.

  Для предоставления большей части этих функциональных возможностей Rails использует gem-пакет по имени `Rack`. Все подробности приведены в документации по `Rack::Request`.

`response`
: Объект ответа, сформированный в процессе обработки запроса. Обычно Rails все управление этим объектом берет на себя. При рассмотрении обратных вызовов в разделе «Обратные вызовы» мы увидим, что иногда для специализированной обработки придется вмешиваться во внутреннюю организацию процесса.

`session`
: Объект, похожий на хэш, представляющий данные текущей сессии. Он будет рассмотрен в разделе «Сессии Rails».

Дополнительно через Action Pack будет доступен объект `logger`.

Формирование ответа пользователю

Частью работы контроллера является формирование ответа пользователю. Для этого существуют четыре основных способа.

- Наиболее распространенный способ заключается в визуализации шаблона. В понятиях MVC-парадигмы шаблон является представлением, которое получает информацию, предоставляемую контроллером, и использует ее для генерации ответа, отправляемого браузеру.
- Контроллер может вернуть строку непосредственно браузеру, минуя представление. Этот способ используется крайне редко, но он может применяться для отправки уведомлений об ошибках.
- Контроллер может вообще ничего не возвращать браузеру. Иногда этот способ используется в ответе на AJAX-запросы. Но в любом случае контроллер возвращает набор HTTP-заголовков, поскольку тот или иной ответ все равно ожидается.
- Контроллер может отправить клиенту какие-нибудь другие данные (отличающиеся от HTML). Обычно это относится к какой-нибудь загрузке (возможно, PDF-документа или содержимого файла).

Контроллер всегда готовит пользователю по одному ответу на каждый запрос. Это означает, что при обработке любого запроса должен быть всего один вызов метода `render()`, `redirect_to()` или `send_`*`xxx`*`()`. (При повторном вызове визуализации выдается исключение `DoubleRenderError`.)

Поскольку ответ у контроллера может быть только один, в заключительной фазе обработки запроса контроллер проверяет наличие сгенерированного ответа. Если ответ отсутствует, контроллер ищет шаблон, названный по имени контроллера и действия, и отправляет его автоматически. Это наиболее распространенный способ отправки. Вы, наверное, заметили, что в большинстве действий, совершаемых в учебном приложении интернет-магазина, мы никогда ничего не отправляли в явном виде. Вместо этого наш метод действий обеспечивал содержимое для представления и завершал свою работу. Контроллер замечал, что ничего отправлено не было, и автоматически вызывал соответствующий шаблон.

У вас может быть несколько шаблонов с одинаковыми именами, но с разными расширениями имен (например, .html.erb, .xml.builder и .js.coffee). Если в запросе на визуализацию расширение не указано, Rails предполагает, что нужен шаблон html.erb.

Визуализация шаблонов

Шаблон представляет собой файл, в котором определяется содержимое ответа для нашего приложения. Rails поддерживает на выходе три формата шаблонов: *erb*, являющийся встроенным Ruby-кодом (обычно вместе с HTML); *builder*, являющийся средством создания XML-содержимого с более широким использованием

программного кода; и *RJS*, который генерирует код JavaScript. Мы будем рассматривать содержимое файлов этих шаблонов, начиная с раздела 21.1 «Использование шаблонов».

По соглашению, шаблон для действия `action` контроллера `controller` будет находиться в файле app/views/controller/action.*type*.*xxx* (где *type* — тип файла, например html, atom или js, а *xxx* — одно из трех расширений: erb, rxml, coffee или scss). Часть полного имени app/views используется по умолчанию. Она может быть заменена для всего приложения путем следующей настройки:

```
ActionController.prepend_view_path путь_к_каталогу
```

Метод `render()` является в Rails основой для всех визуализаций. Он получает хэш аргументов, сообщающий ему, что и как нужно вывести.

У вас может появиться соблазн вставить в ваши контроллеры следующий код:

```
# НЕ ДЕЛАЙТЕ ЭТОГО!
def update
    @user = User.find(params[:id])
    if @user.update(user_params)
       render action: show
    end
    render template: "fix_user_errors"
end
```

Казалось бы, при вызове метода `render` (и `redirect_to`) должно происходить прерывание выполняемого действия. Но этого не происходит. Код, приведенный выше в том случае, если будет выполнено условие `update`, выдаст ошибку (поскольку тогда вызов метода `render` будет фигурировать дважды).

Давайте посмотрим на настройки визуализации, используемые в контроллере в следующих примерах (отдельно визуализация будет рассмотрена в представлении, рассматриваемом в главе 21, начиная с раздела «Шаблоны фрагментов страниц»).

`render()`

Без значимых аргументов метод `render()` выводит шаблон по умолчанию для текущего контроллера и действия. Следующий код приведет к визуализации шаблона app/views/blog/index.html.erb:

```
class BlogController < ApplicationController
    def index
        render
    end
end
```

Следующий код сделает то же самое (по умолчанию на контроллер возлагается вызов метода `render`, если этого не сделает действие):

```
class BlogController < ApplicationController
    def index
    end
end
```

То же самое будет сделано и следующим кодом (поскольку, если не определено никакого метода действия, контроллер вызовет шаблон напрямую):

```
class BlogController < ApplicationController
end
```

`render(text: строка)`

Отправляет клиенту заданную строку. При этом не выполняется никакой интерпретации шаблона и дезактивации HTML-кода.

```
class HappyController < ApplicationController
    def index
        render(:text => "Всем привет!")
    end
end
```

`render(inline: строка, [ type: "erb"|"builder"|"coffee"|"scss" ], [ locals: хэш] )`

Интерпретирует *строку* как источник шаблона заданного типа, отправляющего результаты клиенту. Для установки значений локальных переменных шаблона можно воспользоваться хэшем `locals:`.

Следующий код добавляет к контроллеру метод `method_missing()`, если приложение запущено в режиме разработки. Если контроллер вызван с неправильным действием, будет визуализирован встраиваемый шаблон для отображения имени действия и отформатированной версии параметров запроса.

```
class SomeController < ApplicationController
          if RAILS_ENV == "development"
              def method_missing(name, *args)
                  render(inline: %{
                      <h2>Неизвестное действие: #{name}</h2>
                      Параметры запроса:<br/>
                      <%= debug(params) %> })
                  end
              end
          end
```

`render(action: имя_действия)`

Визуализирует шаблон для заданного действия контроллера. Иногда форма `action:` метода `render()` используется, когда нужно воспользоваться перенаправлениями, — о том, почему этого делать не стоит, мы поговорим в разделе «Перенаправления».

```
def display_cart
    if @cart.empty?
        render(action: :index)
    else
        # ...
    end
end
```

Учтите, что вызов `render(action:...)` не приводит к вызову метода действия; в результате происходит обыкновенное отображение шаблона. Если шаблону нужны переменные экземпляра, значения им должны быть присвоены тем методом, который вызвал метод `render()`.

Поскольку новички здесь часто ошибаются, следует повторить: вызов `render(action:...)` не приводит к вызову метода действия — он всего лишь визуализирует шаблон, использующийся этим действием по умолчанию.

`render(template: имя, [:locals => хэш] )`
Визуализирует шаблон и осуществляет отправку клиенту результирующего текста. Значение `template:` должно содержать в новом имени как контроллер, так и действие, разделенные прямым слэшем. Следующий код выведет шаблон app/views/blog/short_list:

```
class BlogController < ApplicationController
    def index
        render(template: "blog/short_list")
    end
end
```

`render(file: путь)`
Визуализирует представление, которое может быть полностью вне вашего приложения (возможно, из-за того, что оно совместно используется с другим Rails-приложением). По умолчанию файл визуализируется без использования текущего макета. Эту установку можно отменить заданием параметра `layout: true`.

`render(partial: имя, …)`
Визуализирует парциальный шаблон. Более подробно парциальные шаблоны будут рассмотрены в главе 21, в разделе «Шаблоны фрагментов страниц».

`render(nothing: true)`
Ничего не возвращает, отправляет браузеру пустышку.

`render(xml: материал)`
Выводит *материал* в виде текста, вынуждая тип содержимого иметь значение `application/xml`.

`render(json: материал, [:callback => хэш] )`
Выводит *материал* в виде JSON, вынуждая тип содержимого иметь значение application/json. Указание аргумента `callback:` приведет к тому, что результат будет заключен в вызов названной функции обратного вызова.

`render(:update) do |page| ... end`
Выводит блок как RJS-шаблон, передавая в него объект страницы.

```
render(:update) do |page|
    page[:cart].replace_html partial: 'cart', object: @cart
    page[:cart].visual_effect :blind_down if @cart.total_items == 1
end
```

Все формы метода `render()` принимают дополнительные аргументы `:status`, `:layout` и `:content_type`. Аргумент `:status` предоставляет значение, используемое в заголовке статуса в HTTP-ответе. Его значение по умолчанию равно `"200 OK"`. Не нужно для перенаправлений использовать метод `render()` со статусом 3*xx* — в Rails для этого есть метод `redirect()`.

Аргумент `:layout` определяет, должен ли результат визуализации заключаться в макет. (Впервые макеты нам повстречались в разделе 8.2 «Шаг B2: добавление макета страницы». Более подробно макеты будут рассмотрены начиная с раздела 21.6 «Сокращение объемов поддержки с помощью макетов и парциалов».) Если аргумент имеет значение `false`, макет применяться не будет. Если аргумент имеет значение `nil` или `true`, макет будет применяться, только если имеется макет, связанный с текущим действием. Если значением аргумента `:layout` служит строка, она будет взята в качестве имени макета, используемого при визуализации. Если действует аргумент `:nothing`, макет не применяется ни при каких условиях.

Аргумент `:content_type` позволяет определить значение, которое будет передано браузеру в HTTP-заголовке `Content-Type`.

Иногда полезно иметь возможность перехватывать и помещать в строку все, что будет отправлено браузеру. Метод `render_to_string()` получает те же аргументы, что и метод `render()`, но возвращает результаты вывода в виде строки — вывод не сохраняется в объекте ответа и поэтому не будет отправлен пользователю до тех пор, пока не будут предприняты какие-нибудь дополнительные действия. Вызов метода `render_to_string` не считается реальной визуализацией. Настоящий метод `render` можно вызвать позже, не получив при этом ошибку `DoubleRender`.

Отправка файлов и других данных

Мы уже рассматривали, как в контроллере осуществляется визуализация шаблонов и отправка строк. Третий тип ответа заключается в отправке клиенту каких-либо данных (обычно, но не обязательно, — содержимого файлов).

send_data

Отправляет клиенту строку, содержащую двоичные данные.

```
send_data(данные, аргументы…)
```

Отправляет клиенту поток данных. Обычно браузер использует комбинацию из сведений о типе содержимого и его размещении, которые устанавливаются в аргументах, чтобы определить, как нужно распорядиться этими данными.

```
def sales_graph
  png_data = Sales.plot_for(Date.today.month)
  send_data(png_data, type: "image/png", disposition: "inline")
end
```

Аргументы:

`:disposition`	строка	Подсказывает браузеру, что файл должен быть отображен в виде вставки (вариант `inline`) или загружен и сохранен (вариант `attachment`, использующийся по умолчанию).
`:filename`	строка	Подсказывает браузеру имя файла, используемое по умолчанию при сохранении этих данных.
`:status`	строка	Код статуса (по умолчанию — `"200 OK"`).
`:type`	строка	Тип содержимого (по умолчанию `application/octet-stream`).
`:url_based_filename`	булево значение	Если имеет значение `true`, а аргумент `:filename` не установлен, эта настройка мешает Rails предоставлять базовое имя файла в заголовке `Content-Disposition`. Указание базового имени файла необходимо, чтобы заставить некоторые браузеры корректно обрабатывать локализованные имена файлов.

send_file

Отправляет клиенту содержимое файла.

`send_file(`*`путь, аргументы…`*`)`

Отправляет клиенту указанный файл. Метод устанавливает заголовки `Content-Length`, `Content-Type`, `Content-Disposition` и `Content-Transfer-Encoding`.

Аргументы:

`:buffer_size`	число	Объем данных, отправляемых на браузер в каждой записи, если разрешена потоковая отправка (аргумент `:stream` имеет значение `true`).
`:disposition`	строка	Подсказывает браузеру, что файл должен быть отображен в виде вставки (вариант `inline`) или загружен и сохранен (вариант `attachment`, использующийся по умолчанию).
`:filename`	строка	Подсказывает браузеру исходное имя файла, используемое при его сохранении. Если параметр не установлен, по умолчанию используется имя, извлеченное из пути к файлу.
`:status`	строка	Код статуса (по умолчанию — `"200 OK"`).
`:stream`	true или false	Если аргумент имеет значение `false`, весь файл целиком считывается в память сервера и отправляется клиенту. В противном случае файл читается и записывается у клиента по частям, размер которых установлен значением аргумента `:buffer_size`.

`:type`	строка	Тип содержимого (по умолчанию — `application/octet-stream`).

Дополнительные заголовки для обоих методов `send_` можно установить, используя свойство контроллера `headers`.

```
def send_secret_file
    send_file("/files/secret_list")
    headers["Content-Description"] = "Top secret"
end
```

Как выкладывать файлы на удаленный компьютер, будет показано в разделе 21.4 «Выкладывание файлов для Rails-приложений».

Перенаправления

HTTP-перенаправление отправляется клиенту с сервера в ответ на запрос. Клиенту как бы говорится: «Я обработал этот запрос, и вы должны пойти сюда, чтобы увидеть результаты». Перенаправление в качестве ответа включает в себя URL, который клиент должен попытаться затем применить, а также информацию о статусе, в которой сообщается о постоянном (код статуса 301) или временном (код статуса 307) характере этого перенаправления. Перенаправления иногда используются в случае реорганизации веб-страниц; клиент, обращающийся к странице по старому адресу, получает ссылку на ее новое размещение. Чаще всего Rails-приложения используют перенаправления для передачи обработки запроса какому-нибудь другому действию.

Перенаправления обрабатываются браузерами закулисно. Обычно единственным признаком перенаправления является небольшая задержка и изменение URL просматриваемой страницы по сравнению с тем адресом, с которого посылался запрос. Следует обратить внимание на последнее обстоятельство — что касается поведения браузера, перенаправление с сервера практически ничем не отличается от ввода конечным пользователем нового целевого URL вручную.

Перенаправление играет важную роль при создании послушных веб-приложений. Взглянем на простое блог-приложение, поддерживающее отправку комментариев. После того как пользователь отправит свой комментарий, наше приложение должно отобразить статью заново, предположительно с новым комментарием в конце.

Может появиться соблазн решить задачу с помощью следующей логики:

```
class BlogController
    def display
        @article = Article.find(params[:id])
    end

    def add_comment
        @article = Article.find(params[:id])
        comment = Comment.new(params[:comment])
        @article.comments << comment
```

```
        if @article.save
            flash[:note] = "Спасибо за ценный комментарий"
        else
            flash[:note] = "От Вашего комментария нам пришлось отказаться"
        end
        # НЕ ДЕЛАЙТЕ ЭТОГО!
        render(action: 'display')
    end
end
```

Несомненно, предполагалось отобразить статью после отправки комментария. Поэтому разработчик в завершение кода метода `add_comment()` вызвал метод `render(action: 'display')`, которым визуализировал представление `display`, показывающее обновленную статью конечному пользователю. Но взглянем на все это с позиции браузера.

Он отправляет URL, оканчивающийся фрагментом blog/add_comment, и получает назад отображение каталога статей. Но браузер считает, что текущий URL по-прежнему оканчивается на blog/add_comment. А значит, если пользователь щелкнет на кнопке Refresh (Обновить) (возможно, чтобы посмотреть, не послал ли кто-нибудь другой еще один комментарий), URL с окончанием `add_comment` снова будет послан приложению. Пользователь хотел обновить информацию на дисплее, а приложение увидит запрос на добавление еще одного комментария. Этот непреднамеренный двойной ввод в блог-приложении будет воспринят негативно. А в интернет-магазине он может обойтись весьма дорого.

В подобных обстоятельствах правильный способ отображения добавленного комментария в каталоге статей заключается в перенаправлении браузера на действие `display`. Для этого применяется Rails-метод `redirect_to()`. Если после этого пользователь щелкнет на кнопке Refresh (Обновить), он просто заново вызовет действие `display` и еще один комментарий добавлен не будет.

```
def add_comment
    @article = Article.find(params[:id])
    comment = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
        flash[:note] = "Спасибо за ценный комментарий"
    else
        flash[:note] = "От Вашего комментария нам пришлось отказаться"
    end
▶   redirect_to(action: 'display')
end
```

В Rails используется простой, но достаточно эффективный механизм перенаправлений. Он может перенаправить на действие в заданном контроллере (с передачей аргументов) на другой URL (на том же или на другом сервере) или на предыдущую страницу. Рассмотрим по очереди все три формы.

`redirect_to(action: ..., `*`аргументы`*`...)`

Отправляет на браузер временное перенаправление, основанное на значениях хэша `аргументы`. Целевой URL генерируется с использованием метода

`url_for()`, поэтому форме метода `redirect_to()` присущи все премудрости кода маршрутизации Rails, выполняемого вкупе с ним.

`redirect_to(`*путь*`)`
Осуществляется перенаправление по заданному пути. Если путь не начинается с протокола (такого как http://), вначале будут подставлены протокол и порт текущего запроса. Этот метод не осуществляет каких-либо перезаписей URL, поэтому он не должен использоваться для создания путей, предназначенных для связи с действиями в приложении (если только путь не будет сгенерирован с использованием метода `url_for` или поименованного генератора маршрутов URL).

```
def save
    order = Order.new(params[:order])
    if order.save
        redirect_to action: "display"
    else
        session[:error_count] ||= 0
        session[:error_count] += 1
        if session[:error_count] < 4
            self.notice = "Пожалуйста, попробуйте еще раз"
        else
            # Безнадежно - пользователь явно ничего не понимает
            redirect_to("/help/order_entry.html")
        end
    end
end
```

`redirect_to(:back)`
Осуществляет перенаправление на URL, заданный заголовком `HTTP_REFERER` текущего запроса.

```
def save_details
    unless params[:are_you_sure] == 'Y'
        redirect_to(:back)
    else
        ...
    end
end
```

По умолчанию статус всех перенаправлений выставляется как временный (они могут оказать влияние лишь на текущий запрос). При перенаправлении на URL вам предоставляется возможность сделать его постоянным. В этом случае нужно установить соответствующий статус в заголовке ответа:

```
headers["Status"] = "301 Moved Permanently"
redirect_to("http://my.new.home")
```

Поскольку методы перенаправления отправляют ответы браузеру, к ним применяются те же правила, что и к методам визуализации, — их можно использовать только по одному на каждый запрос.

До сих пор мы рассматривали запросы и ответы обособленно. Rails также предоставляет ряд механизмов, расширяющих диапазон действия запросов.

20.3. Объекты и операции, расширяющие диапазон действия запросов

Хотя основной объем состояния данных, сохраняемый от запроса к запросу, принадлежит базе данных и доступ к нему осуществляется через Active Record, некоторые другие небольшие объемы состояния данных имеют другую продолжительность жизни и должны управляться по-другому. Хотя в приложении Depot сама корзина хранилась в базе данных, сведения о том, какая из корзин является текущей, управлялись с помощью сессий. Для передачи простых сообщений вроде «Последний пользователь не может быть удален» следующему запросу после перенаправления были использованы флэш-уведомления. А для извлечения локальных данных из самих URL были использованы обратные вызовы.

В данном разделе мы по очереди рассмотрим каждый из этих механизмов.

Сессии Rails

Сессия Rails представляет собой структуру, похожую на хэш, которая продолжает существовать от запроса к запросу. В отличие от простых cookie-файлов, сессии могут содержать любые объекты (если только объекты могут подвергаться маршализации), которые превращают их в идеальное средство для хранения в веб-приложениях информации о состоянии. Например, в нашем интернет-магазине мы использовали сессию для хранения объектов корзины покупателя между запросами. Объект `Cart` может использоваться в нашем приложении точно так же, как и любой другой объект. Но Rails все обставляет таким образом, что корзина сохраняется в конце обработки каждого запроса, и, что более важно, нужная корзина для входящего запроса восстанавливается, как только Rails приступает к его обработке. Используя сессии, мы можем имитировать ситуацию, при которой наше приложение остается между запросами в курсе происходящего.

И это наводит на интересный вопрос: где конкретно эти данные находятся между запросами? Сервер может выбрать вариант их отправки обратно клиенту в виде cookie-файла. Rails 4 делает это по умолчанию. Это накладывает ограничения на размер данных и увеличивает объем передаваемой по сети информации, но означает, что серверу приходится меньше заниматься управлением и очисткой. Учтите, что содержимое (по умолчанию) зашифровано, значит, пользователи не могут его ни увидеть, ни подделать.

Другим вариантом является хранение данных на сервере. Оно состоит из двух частей. Во-первых, Rails нужно отслеживать сессии. Она выполняет эту задачу путем создания (по умолчанию) 32-разрядного шестнадцатеричного символьного ключа (предоставляя в ваше распоряжение 16^{32} комбинаций). Этот ключ называется идентификатором сессии и имеет случайные значения. Rails обеспечивает

хранение этого идентификатора сессии в cookie-файле (с ключом `_session_id`) на пользовательском браузере. С приходом в приложение очередного запроса от этого браузера Rails может восстановить идентификатор сессии.

Во-вторых, Rails содержит на сервере постоянное хранилище для данных сессий, индексированных по их идентификаторам. При получении запроса Rails просматривает хранилище данных, используя идентификатор сессии. Найденные там данные представлены в виде преобразованного в последовательную форму объекта Ruby. Результат их восстановления и сохранения помещается в свойство контроллера `session`, а в нем данные становятся доступны для всего кода приложения. Приложение может сколько угодно дополнять или изменять эти данные. Завершая обработку каждого запроса, Rails записывает данные сессии обратно в хранилище данных. Они там находятся, пока от этого же браузера не будет получен следующий запрос.

А что можно хранить в сессиях? С учетом некоторых ограничений и предостережений — все, что угодно.

- В отношении типов объектов, которые можно хранить в сессии, существует ряд ограничений. Конкретные особенности зависят от избранного механизма хранения (который мы вскоре рассмотрим). В общем случае объекты в сессии должны подходить для маршализации, то есть для преобразования в последовательную форму (при использовании Marshal-функций Ruby). Это, к примеру, означает, что объекты ввода-вывода в сессиях хранить невозможно.
- Если в сессии хранятся какие-либо объекты, относящиеся к модели Rails, для них следует добавить объявления модели. Это заставляет Rails предварительно загружать класс модели, чтобы эти объявления становились доступными на тот момент, когда Ruby приступает к преобразованию объекта из последовательной формы, в которой он находится в хранилище сессии. Если использование сессии ограничено только одним контроллером, это объявление должно помещаться в верхней части его кода.

```
class BlogController < ApplicationController
    model :user_preferences
    # . . .
```

Но чтобы сессия могла читаться другим контроллером (что вполне вероятно для приложений с несколькими контроллерами), придется добавить объявление к файлу application_controller.rb в каталоге app/controllers.

ДЭВИД ГОВОРИТ: ЧУДЕСА СЕССИЙ, ОСНОВАННЫХ НА ИСПОЛЬЗОВАНИИ COOKIE-ФАЙЛОВ

Порядок хранения сессии Rails, заданный по умолчанию, при первой оценке похож на какую-то безумную затею. Вы что, на самом деле собираетесь хранить значения у клиента?! А что, если я захочу хранить в сессии коды запуска ракет с ядерными боеголовками и у меня не может быть клиента, фактически знающего об этом?

Да, место хранения, заданное по умолчанию, не подходит для тех секретов, которые нужно утаить от клиента. Но на самом деле существует важное ограничение, позволяющее вам из-

бежать опасности хранения в сессии сложных объектов, которые могут устареть. И бумажный дракон кодов запуска ракет с ядерными боеголовками никогда не станет реальностью, из-за которой стоило бы беспокоиться.

Имеется в виду ограничение по размеру. Cookie-файлы могут иметь длину не более 4 Кбайт, поэтому их невозможно забить всякой чепухой. Лучше всего они подходят для хранения ссылок, вроде идентификатора корзины — cart_id, но никак не самой корзины.

Если клиент реально способен изменить данные сессии, нужно позаботиться о защитном ключе. Вам нужно убедиться в неприкосновенности помещенных в данные сессии значений. Нехорошо, если клиент сможет изменить имеющееся у него значение cart_id с 5 на 8 и получить чью-нибудь корзину. К счастью, Rails защищает вас от именно такого случая, подписывая сессию и выдавая исключение, предупреждая о вмешательстве, если подпись не является подлинной.

Извлекаемые из этого преимущества заключаются в отсутствии нагрузки на базу данных по извлечению и сохранению данных сессии при каждом запросе и в отсутствии обязанностей по очистке памяти от ненужных данных. Если данные вашей сессии хранить в файловой системе или в базе данных, вам придется позаботиться о том, как избавляться от данных прошедших сессий, испытывая ненужные трудности. Никто не любит заниматься очисткой. А сессии, основанные на использовании cookie-файлов, знают, как за собой прибираться. И как же после этого они могут не понравиться?

- В данных сессии вряд ли захочется хранить какие-то крупные объекты — их лучше поместить в базу данных, а из сессии на них просто ссылаться. Это особенно актуально для сессий, основанных на использовании cookie-файлов, где действует ограничение в 4 Кбайт.
- Скорее всего, в данных сессии также не захочется хранить часто изменяющиеся объекты. К примеру, может появиться желание хранить в сессии количество статей блога для повышения производительности системы. Но если эту затею реализовать, число не сможет обновляться, когда статью добавит какой-то другой пользователь.

  Существует соблазн хранить в данных сессии объекты, представляющие текущего подключившегося пользователя. Но если в приложении нужно предусмотреть лишение пользователей их полномочий, эту идею вряд ли стоит признать благоразумной. Даже если пользователь будет недоступен в базе данных, его действующий статус будет по-прежнему отображаться в данных сессии.

  Данные, подверженные частым изменениям, лучше хранить в базе данных и ссылаться на них из сессии.
- Наверное, вам также не захочется хранить только лишь в данных сессии какую-нибудь важную информацию. Например, если ваше приложение в одном из запросов генерирует номер подтверждения заказа и помещает его в данные сессии, чтобы он мог быть сохранен в базе данных при обработке следующего запроса, вы рискуете утратить этот номер, если пользователь удалит cookie со своего браузера. Важную информацию следует хранить в базе данных.

Существует еще одно, весьма существенное предостережение. Если вы храните объект в данных сессии, при следующем возвращении к тому же браузеру ваше

приложение не станет этот объект извлекать. Однако если к тому времени приложение будет обновлено, объект, сохраненный в данных сессии, возможно, утратит согласованность с определением класса объекта в вашем приложении, и приложение при обработке запроса выдаст сбой. В таком случае есть три варианта. Один заключается в хранении объекта в базе данных с использованием обычной модели и сохранением в данных сессии только лишь идентификатора строки. Объекты модели гораздо меньше восприимчивы к изменениям схемы данных, чем к изменениям в принадлежащей Ruby библиотеке маршализации. Второй вариант состоит в самостоятельном удалении всех данных сессии, хранящихся на сервере, как только будут изменены определения класса, хранимые в этих данных.

Третий вариант немного сложнее. Если к ключам сессии добавить номер версии и изменять этот номер при каждом обновлении хранящихся данных, всегда будут загружаться только те данные, которые соответствуют текущей версии приложения. Возможно, вам удастся создать новые версии классов, чьи объекты сохранены в данных сессии, и использовать соответствующие классы в зависимости от ключей сессии, связанных с каждым запросом. Последняя идея может потребовать немалых усилий, поэтому вам нужно решить, стоит ли овчинка выделки.

Поскольку хранилище сессии похоже на хэш, в нем может храниться множество объектов, каждый со своим собственным ключом.

Не нужно также отключать сессии для конкретных действий. Поскольку данные сессий загружаются только в случае крайней необходимости, нужно просто не ссылаться на сессию в том действии, в котором она вам не нужна.

Хранение данных сессии

Для хранения данных сессии в Rails используется множество вариантов. У каждого из них есть свои сильные и слабые стороны. Мы начнем с перечня этих вариантов, а в завершение проведем их сравнение.

Механизм хранения данных сессии определяется в свойстве `session_store` в `ActiveRecord::Base` — в этом свойстве устанавливается указатель на класс, осуществляющий стратегию хранения. Этот класс должен быть определен в модуле `ActiveSupport::Cache::Store`. Для названий стратегий хранения данных сессии применяются обозначения, которые конвертируются в класс имен `CamelCase`.

`session_store = :cookie_store`
: Это механизм хранения данных сессии, используемый Rails, начиная с версии 2.0, по умолчанию. Этот формат представляет объекты в их маршализованной форме, что позволяет хранить в сессиях любые данные, которые могут быть преобразованы в последовательную форму, но объемом не более 4 Кбайт. Именно этот вариант используется в приложении Depot.

`session_store = :active_record_store`
: Используя gem-пакет `activerecord-session_store`[1], вы можете хранить данные сессии в базе данных своего приложения с помощью `ActiveRecordStore`.

[1] https://github.com/rails/activerecord-session_store#installation

`session_store = :drb_store`

DRb является протоколом, позволяющим Ruby осуществлять процессы совместного использования объектов по сетевому подключению. Используя администратор базы данных DRbStore, Rails хранит данные сессии на DRb-сервере (который управляется вне веб-приложения).

Множество экземпляров приложения, потенциально выполняемых на распределенных серверах, могут иметь доступ к одному и тому же хранилищу DRb. Для преобразования объектов в последовательную форму DRb использует функцию `Marshal`.

`session_store = :mem_cache_store`

`memcached` — это находящаяся в свободном доступе распределенная система кэширования объектов, поддерживаемая Dormando[1]. Система `memcached` сложнее в использовании по сравнению с другими альтернативными системами и может представлять интерес только в том случае, если вы ее уже применяете на своем веб-сайте в каких-то других целях.

`session_store = :memory_store`

Этот вариант предписывает локальное хранение данных сессии в памяти приложения. Поскольку преобразование данных в последовательную форму не применяется, в хранящейся в локальной памяти сессии могут размещаться любые объекты. Скоро мы увидим, что для Rails-приложений эта идея, в общем-то, вряд ли подходит.

`session_store = :file_store`

Данные сессии хранятся в простых файлах. Поскольку содержимое этих файлов должно состоять из строк, для Rails-приложений этот механизм, скорее всего, бесполезен. Он поддерживает дополнительные конфигурационные параметры `:prefix`, `:suffix` и `:tmpdir`.

Сравнение вариантов хранения данных сессии

И что же в конце концов следует выбрать из всех этих вариантов именно для вашего приложения? Как всегда, ответ будет один: «Все зависит от обстоятельств».

Если добиваться высокой производительности, существует ряд абсолютных критериев выбора, но у каждого ведь свои собственные обстоятельства, из которых следует исходить. На то, как взаимодействуют все компоненты хранения данных сессии, будет влиять состав вашего оборудования, сетевые задержки, подбор баз данных и, возможно, даже погодные условия. По нашему мнению, лучше всего начать с простейшего работоспособного решения, а затем отслеживать параметры его работы. Если оно приведет к замедлению работы приложения, то прежде чем отказываться от него, нужно найти причину.

Если у вашего сайта большой объем посещений, старайтесь поддерживать небольшой размер данных сессии, и тогда вам вполне подойдет механизм `cookie_store`.

[1] http://memcached.org/

Если исключить хранение в памяти как слишком упрощенное, хранение в файлах как слишком ограничивающее и `memcached` как слишком сложное, варианты, относящиеся к серверной стороне, сводятся к CookieStore, Active Record и к хранилищу на основе DRb-протокола. Если вам нужно хранить в сессиях объем данных, превышающий возможности cookie-файлов, мы рекомендуем начать с решения, использующего Active Record. Если по мере расширения вашего приложения вы поймете, что этот способ хранения становится узким местом, можно будет перейти на решение, основанное на использовании DRb-протокола.

Регулирование срока существования сессий и очистка хранилища данных

Для всех решений, основанных на хранении данных сессии на сервере, характерна одна общая проблема. Каждая новая сессия добавляет к хранилищу сессий свои данные. В конце концов, возникнет потребность в наведении порядка, иначе можно исчерпать все серверные ресурсы.

Есть и еще одна причина для очистки сессий. Многим приложениям совсем не нужно, чтобы сессия длилась до бесконечности. Как только пользователь подключился с определенного браузера, приложению может потребоваться ввести правило, согласно которому пользователь остается подключенным только при проявлении активности. Когда он отключается или проходит какое-то определенное время после того, как он в последний раз обращался к приложению, его сессия должна быть прервана.

Иногда этого эффекта можно достичь, назначая срок существования cookie, в которых содержится идентификатор сессии. Но этим можно вызвать недовольство конечных пользователей. Хуже того, синхронизировать время существования cookie на браузере с наведением порядка с данными сессии на сервере очень трудно.

Поэтому мы предлагаем регулировать срок существования сессий простым удалением с сервера всех относящихся к ним данных. Если в очередном запросе, поступившем с браузера, будет содержаться идентификатор сессии, указывающий на ранее удаленные данные, приложение не получит никаких данных сессии, поскольку такой сессии уже не существует.

Осуществление этого способа регулирования срока существования сессий зависит от используемого механизма хранения данных.

Для хранилища данных сессий на основе Active Record следует использовать столбец `updated_at` таблицы `sessions`. Вы можете удалить все сессии, не обновлявшиеся в течение последнего часа (игнорируя переходы на летнее время и обратно), воспользовавшись при решении задачи очистки следующим SQL-кодом:

```
delete from sessions
    where now() - updated_at > 3600;
```

Для решений, основанных на применении DRb-протокола, вопросы времени существования сессии регулируются в процессе внутренней работы DRb-сервера. Возможно, у вас возникнет желание записывать метки времени наряду с записями

в хэше данных сессии. Вы можете запустить отдельный поток (или даже отдельный процесс), который периодически удалял бы записи из этого хэша.

Во всяком случае, приложение может помочь этому процессу, если для удаления ненужных сессий будет вызван метод `reset_session()` (к примеру, после того, как пользователь отключится).

Флэш — связь между действиями

Когда для передачи управления другому действию используется метод `redirect_to()`, браузер генерирует отдельный запрос на вызов этого действия. Этот запрос будет обрабатываться в приложении вновь созданным экземпляром объекта контроллера. При этом переменные экземпляра, которым были присвоены значения в исходном действии, станут недоступны коду второго действия, вызванного при перенаправлении. Но порой возникает потребность связать эти два экземпляра объекта контроллера. Осуществить ее позволит механизм под названием *флэш*.

Флэш — это временный электронный блокнот для значений. Он имеет структуру, похожую на хэш, и хранится в данных сессии, поэтому в нем можно хранить значения, связанные с ключами, которые затем можно извлечь. У него есть одна характерная особенность. По умолчанию данные, сохраненные во флэше в процессе обработки какого-нибудь запроса, будут доступны лишь при обработке того запроса, который следует непосредственно за ними. Как только этот второй запрос будет обработан, значения из флэша будут удалены.

Наверное, чаще всего флэш используется для передачи сообщения об ошибке и информационной строки от одного действия к следующему. Идея состоит в том, что первое действие замечает какие-то условия, создает сообщение, в котором эти условия описываются, и осуществляет перенаправление на какое-то особое действие. Благодаря тому, что сообщение сохраняется во флэше, второе действие получает доступ к тексту этого сообщения и использует его в представлении.

Иногда флэш удобно использовать для передачи в шаблон сообщения из текущего действия. Например, нашему методу `display()` потребуется вывести приветственное сообщение, если нет какого-нибудь другого, неотложного уведомления. Это сообщение не нужно передавать следующему действию — оно используется только в текущем запросе. Осуществить задуманное поможет метод `flash.now`, который обновляет флэш, не добавляя его к данным сессии.

Если метод `flash.now` создает во флэше временную запись, то метод `flash.keep` осуществляет прямо противоположную функцию, создавая запись в текущем флэше, которая останется до следующего цикла запроса.

Если метод `flash.keep` оставить без параметров, будет сохранено все флэш-содержимое.

Во флэше могут храниться не только текстовые сообщения — их можно использовать для передачи от действия к действию любого вида информации. Понятно, что для хранения долгосрочной информации лучше воспользоваться сессией (возможно, в сочетании с базой данных), но для передачи параметров от одного запроса к другому флэш остается вне конкуренции.

Поскольку данные флэша хранятся в сессии, здесь применимы все соответствующие правила. В частности, любой объект должен быть пригоден к преобразованию в последовательную форму. Мы настоятельно рекомендуем передавать во флэш только простые объекты.

Обратные вызовы

Обратные вызовы позволяют создавать код контроллера, в котором заключается обработка, осуществляемая действиями, — создав однажды какой-нибудь фрагмент кода, можно вызывать его до или после любого количества действий, имеющихся в контроллере (или в подклассе контроллеров). Оказывается, это очень эффективное свойство. Используя обратные вызовы, можно применять схемы по определению личности пользователя, по подключению пользователей, по сжатию ответов и даже по их настройке под определенного пользователя.

Rails поддерживает три вида обратных вызовов: *«до»*, *«после»*, а также *«до и после»*. Обратные вызовы выполняются непосредственно перед и (или) после действий. В зависимости от того, как их определить, обратные вызовы либо запускаются как методы внутри контроллера, либо им при запуске передается объект контроллера. При любом способе они наряду с другими свойствами контроллера получают доступ к деталям объектов запроса и ответа, а также к другим свойствам контроллера.

Обратные вызовы «до» и «после»

Как следует из их названий, обратные вызовы «до» и «после» вызываются до или после действия. Для каждого контроллера Rails обслуживает две последовательности обратных вызовов. Когда контроллер собирается выполнить какое-нибудь действие, он задействует все обратные вызовы, принадлежащие последовательности «до». А перед запуском последовательности обратных вызовов «после» он выполняет действие.

Обратные вызовы могут быть пассивными и отслеживать работу контроллера. Также они могут принимать более активное участие в обработке запроса. Если обратный вызов «до» возвращает значение `false`, работа последовательности обратных вызовов прерывается, и действие не запускается. Обратный вызов также может визуализировать вывод или перенаправить запросы, в таком случае исходное действие уже не вызывается.

Мы уже рассматривали использование обратных вызовов для авторизации на примере управления административной областью нашего интернет-магазина в разделе 14.3 «Шаг ИЗ: ограничение доступа». Там был определен метод авторизации, перенаправляющий на страницу входа в административную область, если в текущей сессии не было авторизованного пользователя. Затем мы сделали этот метод обратным вызовом «до» для всех действий в контроллере администрирования.

Объявления обратных вызовов также допускают использование блоков и имен классов. Если указывается какой-нибудь блок, он должен быть вызван с использованием в качестве параметра текущего контроллера. Если обратному вызову

передается класс, будет вызван его метод класса `filter()`, а в качестве его аргумента будет использован контроллер.

По умолчанию обратные вызовы применяются ко всем действиям, имеющимся в контроллере (и ко всем подклассам этого контроллера). Это положение можно изменить, применив параметр `:only` или `:except`. Параметру `:only` передается одно и более действий, с которыми осуществляются обратные вызовы. В параметре `:except` перечисляются действия, с которыми не осуществляются обратные вызовы.

Объявления `before_action` и `after_action` присоединяются к последовательности обратных вызовов контроллера. Чтобы поставить обратные вызовы впереди этой последовательности, следует воспользоваться вариантами `prepend_before_action()` и `prepend_after_action()`.

Обратные вызовы «после» могут использоваться для модификации исходящих ответов, с изменением при необходимости их заголовков и содержимого. В некоторых приложениях эта технология используется для осуществления глобальных замен в содержимом, сгенерированном шаблонами контроллера (например, для замены в теле ответа строки `<customer/>` именем клиента). Другим применением обратного вызова может стать сжатие ответа, если пользовательский браузер его поддерживает.

Обратные вызовы «до» и «после» охватывают выполнение действий. Охватывающие обратные вызовы можно создавать двумя различными способами. В первом из них обратный вызов представляет собой единый фрагмент кода. Этот код вызывается перед выполняемым действием. Если в коде обратного вызова выполняется инструкция `yield`, действие выполняется. Когда действие будет выполнено, код обратного вызова продолжит выполнение.

Таким образом, код перед вызовом `yield` подобен обратному вызову «до», а код, следующий за вызовом `yield`, является обратным вызовом «после». Если код обратного вызова не вызывает `yield`, действие не выполняется — эта ситуация аналогична возвращению значения `false` обратным вызовом «до».

Польза от применения охватывающих обратных вызовов в том, что они могут сохранять свое содержимое до и после вызова действия.

Объявлению `around_action` кроме имени метода можно передать блок кода или класс обратного вызова.

Если в качестве обратного вызова используется блок, ему нужно передать два параметра: объект контроллера и доверенность на действие. Для вызова исходного действия ко второму параметру применяется метод `call()`.

Вторая форма позволяет передавать в качестве обратного вызова объект. Этот объект должен выполнить метод под названием `filter()`. Этому методу будет передан объект контроллера. Для вызова действия в нем применяется инструкция `yield`.

Охватывающие обратные вызовы, так же как и обратные вызовы «до» и «после», допускают использование параметров `:only` и `:except`.

По умолчанию охватывающие обратные вызовы добавляются к последовательности обратных вызовов по-другому: первый добавленный охватывающий обратный вызов выполняется первым. Охватывающие обратные вызовы, добавленные позже, будут вложены в уже существующие охватывающие обратные вызовы.

Наследование обратных вызовов

Если подкласс представляет собой контроллер, содержащий обратные вызовы, то эти обратные вызовы в дочернем объекте будут запускаться так же, как и в родительском. Но обратные вызовы, определенные в дочернем объекте, на родительский влиять не будут.

Если какой-то конкретный обратный вызов в дочернем объекте контроллера запускать нежелательно, можно заменить обработку по умолчанию, применив объявления `skip_before_action` и `skip_after_action`. Эти объявления допускают применение параметров `:only` и `:except`.

Чтобы пропустить любой обратный вызов («до», «после» или охватывающий), можно использовать объявление `skip_action`. Но оно работает только для тех обратных вызовов, которые были определены как имя метода (в виде обозначения).

Объявление `skip_before_action` мы уже использовали в разделе 14.3, «Шаг И3: ограничение доступа».

Наши достижения

Мы изучили порядок взаимодействия модулей Action Dispatch и Action Controller, позволяющего нашему серверу отвечать на запросы. Важность этого взаимодействия трудно переоценить. Практически в каждом приложении это главное место, где выражен творческий потенциал вашего приложения. Active Record и Action View являются довольно-таки пассивными компонентами, а все действия разворачиваются в наших маршрутизаторах и в наших контроллерах.

Эта глава начиналась с рассмотрения концепции REST, влияющей на способ подхода Rails к маршрутизации запросов. Мы увидели, как сначала предоставляются семь базовых действий и как к ним можно добавить дополнительные действия. Мы также увидели, как выбирается способ представления данных (например, JSON или XML). И мы рассмотрели вопрос тестирования маршрутов.

Затем мы рассмотрели среду окружения, которую Action Controller предоставляет вашим действиям, а также методы, которые этот модуль предоставляет для визуализации и перенаправления. И в завершение мы рассмотрели сессии, флэш и обратные вызовы, механизмы, каждый из которых можно использовать в контроллерах вашего приложения.

Попутно было показано, как каждая из этих концепций была использована в приложении Depot. Теперь, после того как вы увидели все это в практическом применении и получили сведения о теоретических основах каждого из механизмов, порядок сочетания и использования всех этих концепций ограничивается только вашей собственной творческой фантазией.

В следующей главе будет рассмотрен еще один компонент Action Pack, а именно Action View, который управляет визуализацией результатов.

21 Action View

Основные темы:

- шаблоны;
- формы: поля и выкладываемые файлы;
- помощники;
- макеты и парциалы.

Мы уже знаем, как компонент, осуществляющий маршрутизацию, определяет, какой контроллер следует использовать и как этот контроллер выбирает действие. Мы также знаем, как контроллер и действие решают между собой, что показывать пользователю. Обычно визуализация производится в конце действия, и в этом процессе, как правило, участвует шаблон. Именно этой последней стадии и будет посвящена данная глава. Модуль `ActionView` содержит все функциональные возможности, необходимые для визуализации шаблонов, для чего пользователю чаше всего возвращается сгенерированный в формате HTML, XML или JavaScript. Как следует из самого названия, Action View является той частью нашей MVC-трилогии, которая осуществляет представление.

Эта глава начнется с рассмотрения шаблонов, для которых Rails предоставляет целый диапазон вариантов. Затем будет рассмотрен ряд различных способов предоставления пользовательского ввода: формы, выкладывание файлов и ссылки. Завершится глава рассмотрением ряда способов сокращения объемов поддержки приложения путем использования помощников, макетов и парциалов.

21.1. Использование шаблонов

При создании представления создается шаблон: некая заготовка, которая будет расширена для генерации конечного результата. Чтобы понять, как работают шаблоны, нужно рассмотреть три вопроса.

- Где находятся шаблоны?
- В какой среде окружения они работают?
- Что происходит внутри шаблонов?

Где находятся шаблоны

Метод `render()` предполагает, что шаблоны находятся в каталоге app/views текущего приложения. По соглашению, внутри этого каталога должен быть отдельный подкаталог для представлений, относящихся к каждому контроллеру. К примеру, наше приложение Depot использует контроллеры `products` и `store`. В результате у нас имеются шаблоны в каталогах app/views/products и app/views/store. Как правило, каждый каталог содержит шаблоны, названные по именам действий соответствующего контроллера.

Имена шаблонов могут и не соответствовать именам действий. Такие шаблоны нужно выводить из контроллера, используя вызовы следующего вида:

```
render(action:    'имя_несуществующего_действия' )
render(template:  'контроллер/имя' )
render(file:      'каталог/шаблон' )
```

Последний пример позволяет хранить шаблоны в любом месте файловой системы. Эта форма вызова может пригодиться для тех шаблонов, которые предназначены для совместного использования несколькими приложениями.

Среда окружения шаблонов

В шаблонах содержится смесь из статического текста и кода, который придает динамичность содержимому шаблона. Этот код работает в той среде окружения, которая дает ему доступ к информации, созданной контроллером.

- Все имеющиеся в контроллере переменные экземпляра будут также доступны и в шаблоне — это позволит действиям передавать содержащиеся в них данные шаблонам.
- В качестве методов-аксессоров в представлении доступны такие объекты контроллера, как `flash`, `headers`, `logger`, `params`, `request`, `response` и `session`. Кроме `flash`, код представления, наверное, не может их использовать напрямую, поскольку обязанности по их обработке должны оставаться за контроллером. Тем не менее реальная польза от данного обстоятельства проявляется при отладке. Например, в следующем html.erb-шаблоне для ото-

бражения содержимого сессии, детализации параметров и текущего ответа используется метод debug():

```
<h4>Сессия</h4> <%= debug(session) %>
<h4>Параметры</h4> <%= debug(params) %>
<h4>Ответ</h4> <%= debug(response) %>
```

- Объект текущего контроллера доступен через свойство под названием `controller`, что позволяет шаблону вызывать любой общедоступный метод контроллера (включая и методы, содержащиеся в `ActionController::Base`).
- Путь к основному каталогу шаблонов хранится в свойстве `base_path`.

Что происходит внутри шаблона

Изначально Rails поддерживает четыре типа шаблонов.

- Builder-шаблоны, использующие библиотеку Builder для создания XML-ответов. Дополнительные сведения о библиотеке Builder будут даны в разделе 24.1 «Создание XML с помощью Builder».
- CoffeeScript-шаблоны, создающие код JavaScript, способный менять как способ представления, так и поведение содержимого в вашем браузере.
- ERB-шаблоны, представляющие собой смесь содержимого и встроенного кода Ruby. Обычно они используются для генерации HTML-страниц. Более подробно ERB-технология будет рассмотрена в разделе 24.2 «Создание HTML с помощью ERB».
- SCSS-шаблоны, создающие таблицы стилей CSS для управления способом представления создаваемого вами содержимого в браузере.

Конечно же, чаще всего вы будете использовать ERb-шаблоны, которые уже фактически широко использовались при разработке приложения Depot.

Пока в этой главе мы обращали основное внимание на создании вывода. А в главе 20 «Action Dispatch и Action Controller», основное внимание уделялось обработке ввода. В хорошо продуманных приложениях эти две составляющие тесно связаны: создаваемый нами вывод содержит формы, ссылки и кнопки, приводящие конечного пользователя к созданию следующего набора входящей информации. Теперь уже для вас, наверное, не будет неожиданностью, что Rails предоставляет существенный объем вспомогательных средств и в данной сфере.

21.2. Создание форм

HTML предоставляет ряд элементов, свойств и значений свойств, управляющих сбором вводимых данных. Конечно, можно было бы вручную набрать код вашей формы непосредственно в шаблоне, но в этом нет никакой необходимости.

В этом разделе мы рассмотрим ряд предоставляемых Rails помощников, которые содействуют этому процессу. В разделе 21.5 «Использование помощников», будет показан порядок создания своих собственных помощников.

Для сбора данных в формах HTML предоставляет несколько способов. Несколько самых распространенных средств показаны на рис. 21.1. Учтите, что сама форма не представляет собой какой-либо пример типичного использования; чаще всего для сбора данных вами будет использоваться только поднабор этих методов.

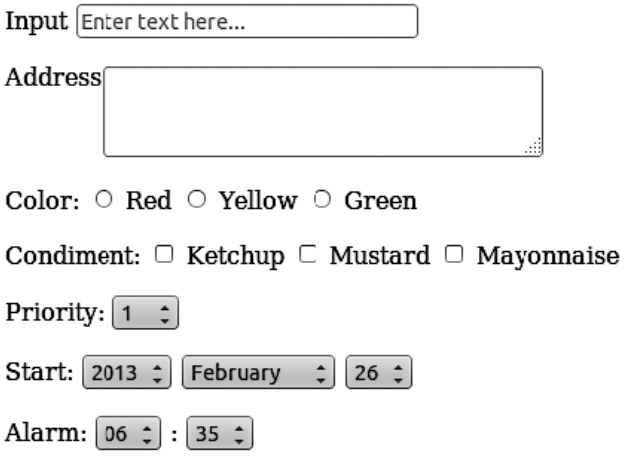


Рис. 21.1. Некоторые самые распространенные способы ввода данных в формы

Давайте посмотрим на шаблон, который использовался для создания этой формы:

rails40/views/app/views/form/input.html.erb

```
Строка
1    <%= form_for(:model) do |form| %>
-    <p>
-        <%= form.label :input %>
-        <%= form.text_field :input, :placeholder => 'Enter text here...' %>
5    </p>
-    <p>
-        <%= form.label :address, :style => 'float: left' %>
-        <%= form.text_area :address, :rows => 3, :cols => 40 %>
-    </p>
10    <p>
-        <%= form.label :color %>:
-        <%= form.radio_button :color, 'red' %>
-        <%= form.label :red %>
-        <%= form.radio_button :color, 'yellow' %>
15        <%= form.label :yellow %>
-        <%= form.radio_button :color, 'green' %>
-        <%= form.label :green %>
-    </p>
-
20    <p>
-        <%= form.label 'condiment' %>:
-        <%= form.check_box :ketchup %>
```

```
-        <%= form.label :ketchup %>
-        <%= form.check_box :mustard %>
25        <%= form.label :mustard %>
-        <%= form.check_box :mayonnaise %>
-        <%= form.label :mayonnaise %>
-    </p>
-
30    <p>
-        <%= form.label :priority %>:
-        <%= form.select :priority, (1..10) %>
-    </p>
-
35    <p>
-        <%= form.label :start %>:
-        <%= form.date_select :start %>
-    </p>
-
40    <p>
-        <%= form.label :alarm %>:
-        <%= form.time_select :alarm %>
-    </p>
-    <% end %>
```

В данном шаблоне вы видите несколько надписей, одна из которых представлена в строке 3. Надписи используются для связи текста с полем ввода указанного свойства. Текст надписи будет по умолчанию использован для имени свойства, пока это имя не будет указано в явном виде.

Для комплектования однострочного и многострочного полей ввода используются помощники `text_field()` и `text_area()` (соответственно в строках 4 и 8). Можно указать некий заполнитель, который будет показан внутри поля до тех пор, пока пользователь не предоставит свое значение. Эта функция поддерживается не всеми браузерами, и те из них, которые ее не поддерживают, просто показывают пустое место. Поскольку это будет существенным ухудшением, вам не нужно сводить свою конструкцию к наименьшему общему знаменателю — воспользуйтесь этим свойством, поскольку те, кто сможет его увидеть, сразу же извлекут из него пользу.

Заполнители — одно из многих небольших средств из серии «подготовка и завершение», предоставляемых с HTML5. Как и прежде, Rails готова к этому, даже если браузеры ваших пользователей еще не готовы. Для запроса определенного типа вводимой информации вы можете воспользоваться помощниками `search_field()`, `telephone_field()`, `url_field()`, `email_field()`, `number_field()` и `range_field()`. Браузеры распоряжаются этой информацией по-разному. Некоторые могут показать поле немного по-другому, чтобы понятнее обозначить его функцию. Например, браузер Safari на Mac покажет поля поиска с закругленными углами и вставит небольшой крестик для очистки поля после начала ввода данных. Некоторые браузеры могут предоставить дополнительную проверку приемлемости данных. Например, Opera проверит поле URL до его отправки. А iPad при вводе адреса электронной почты даже настроит виртуальную экранную клавиатуру, чтобы предоставить готовый доступ к таким символам, как знак `@`.

Хотя поддержка этих функций зависит от применяемого браузера, те браузеры, которые не предоставляют дополнительную поддержку этих функций, просто показывают обычное, неприукрашенное поле ввода. И снова ожиданием ничего не приобретешь. Если есть поле для ввода адреса электронной почты, не пользуйтесь простым помощником `text_field()`, не стойте на месте и приступайте к использованию помощника `email_field()` прямо сейчас.

В строках 12, 22 и 32 показаны три разных способа предоставления ограниченного набора вариантов. Хотя их отображение от браузера к браузеру может немного различаться, все эти подходы хорошо поддерживаются всеми браузерами. Особой гибкостью отличается метод `select()`, ему может быть передано простое перечисление (`Enumeration`), как показано здесь, массив, состоящий из пар имя-значение или хэш. Создавать такие перечни из различных источников, включая базы данных, способны сразу несколько помощников по созданию используемых в формах элементов выбора[1].

И наконец, в строках 37 и 42 показаны, соответственно, запросы на ввод даты и времени. Как вы теперь уже можете ожидать, Rails и здесь предоставляет множество вариантов[2]. В этом примере не показаны помощники `hidden_field()` и `password_field()`. Скрытые (hidden) поля вообще не отображаются, но их значение передается на сервер. Этим можно воспользоваться как альтернативой хранению временных данных в сессиях, позволив данным из одного запроса передаваться в следующий запрос. Поля пароля (password) отображаются, но вводимый в них текст скрывается.

Это более чем достаточный начальный набор для большинства потребностей. Как только у вас появятся какие-нибудь дополнительные потребности, скорее всего, найдется доступный помощник или gem-пакет. Для начала их можно поискать в Rails Guides[3].

А теперь давайте посмотрим, как обрабатываются отправленные данные формы.

21.3. Обработка форм

На рис. 21.2 показано, как различные свойства модели переходят через контроллер к представлению, попадают на HTML-страницу и возвращаются обратно в модель. В объекте модели есть такие свойства, как `name` (имя), `country` (страна) и `password` (пароль). В шаблоне используются вспомогательные методы для создания HTML-формы, позволяющей пользователю редактировать данные модели. Обратите внимание на то, как называются поля формы. К примеру, свойство `country` отображается в HTML-поле ввода, названном `user[country]`.

Когда пользователь отправляет форму, приложению возвращаются необработанные POST-данные. Rails извлекает поля из формы и создает хэш `params`.

---

[1] http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html
[2] http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html
[3] http://guides.rubyonrails.org/form_helpers.html

Простые значения (например, поле идентификатора, извлеченное из части маршрута, следующей за указанием на действие по обработке формы) сохраняются непосредственно в хэше. Но если параметр содержит скобки, Rails предполагает, что он является частью более сложных по структуре данных, и приспосабливает хэш под их хранение. Внутри такого хэша строка, находившаяся в скобках, используется в качестве ключа. Этот процесс может повторяться, если в имени параметра имеется несколько наборов скобок.

Параметры формы	Хэш params
`id=123`	`{ id: "123" }`
`user[name]=Dave`	`{ user: { name: "Dave" }}`
`user[address][city]=Wien`	`{ user: { address: { city: "Wien" }}}`

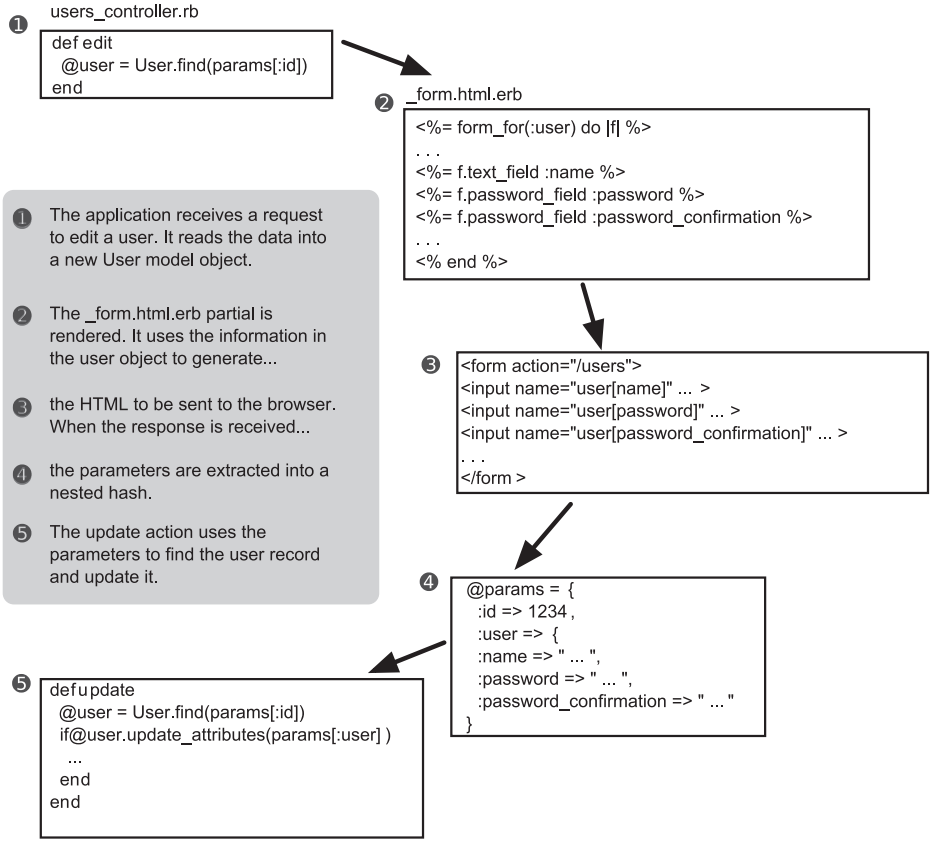


Рис. 21.2. Совместная работа моделей, контроллеров и представлений

В завершающей части интегрированного целого объекты модели могут получать из хэша новые значения свойств, что позволяет нам построить следующее выражение:

```
user.update(user_params)
```

Но интеграция Rails идет еще глубже. Если посмотреть на файл с расширением .html.erb на рис. 21.2, можно увидеть, что для создания кода HTML-формы шаблон использует набор вспомогательных методов `form_for()` и `text_field()`.

Перед тем как идти дальше, стоит заметить, что хэш `params` может использоваться для более сложных данных, чем простой текст. На сервер можно выкладывать целые файлы. Давайте рассмотрим этот вопрос.

21.4. Выкладывание файлов для Rails-приложений

Ваше приложение может позволить пользователям выкладывать файлы на сервер. К примеру, система уведомлений об ошибках может позволить пользователям прикреплять файлы регистрационного журнала и примеры кода к уведомлению о возникшей проблеме или блог-приложение может позволить пользователям выкладывать небольшие изображения, появляющиеся вслед за их статьями.

При использовании HTTP-протокола файлы выкладываются в виде POST-сообщения в формате `multipart/form-data`. Как следует из названия формата, этот тип сообщения генерируется формой. Внутри этой формы используется один или несколько тегов `<input>` с атрибутом `type="file"`. Когда этот тег отображается браузером, пользователь получает возможность выбрать файл по его имени. Когда впоследствии форма будет отправлена, файл или файлы будут отправлены на сервер вместе с другими данными формы.

Чтобы проиллюстрировать процесс выкладывания файлов, мы покажем код, позволяющий пользователю выложить изображение и отобразить его рядом с комментарием. Но для этого сначала нужно создать таблицу `pictures`, в которой будут храниться данные:

rails40/e1/views/db/migrate/20121130000004_create_pictures.rb

```
class CreatePictures < ActiveRecord::Migration
    def change
        create_table :pictures do |t|
            t.string :comment
            t.string :name
            t.string :content_type
            # При использовании MySQL блог по умолчанию ограничен размером
                # 64 Кбайт, поэтому, чтобы его расширить, нужно воспользоваться
                # конкретным указанием размера
            t.binary :data, :limit => 1.megabyte
        end
    end
end
```

Для демонстрации процесса мы создадим специально придуманный контроллер выкладки. Действие `get` выполняет самую обычную задачу, оно просто создает новый объект `picture` и выводит форму:

rails40/e1/views/app/controllers/upload_controller.rb

```
class UploadController < ApplicationController
    def get
        @picture = Picture.new
    end
    # . . .
    private
      # Не доверяйте параметрам из этого жуткого Интернета,
      # позволяйте пропускать через него только пустой список.
      def picture_params
        params.require(:picture).permit(:comment, :uploaded_picture)
      end
end
```

Шаблон `get` содержит форму, которая выкладывает изображение (вместе с комментарием). Обратите внимание на то, как мы заменяем тип кодирования, позволяя данным быть отправленными назад вместе с ответом:

rails40/e1/views/app/views/upload/get.html.erb

```
<% form_for(:picture,
            url: {action: 'save'},
            html: {multipart: true}) do |form| %>

    Комментарий:             <%= form.text_field("comment") %><br/>
    Выложите изображение:    <%= form.file_field("uploaded_picture") %><br/>

    <%= submit_tag("Выложить файл") %>
<% end %>
```

В форме есть еще одна особенность. Изображение выкладывается в свойство под названием `uploaded_picture`. Но в таблице базы данных нет столбца с таким названием. Значит, в модели должно произойти нечто магическое.

rails40/e1/views/app/models/picture.rb

```
class Picture < ActiveRecord::Base

    validates_format_of :content_type,
                        with: /^image/,
                        message: "должно быть изображение"

    def uploaded_picture=(picture_field)
        self.name = base_part_of(picture_field.original_filename)
        self.content_type = picture_field.content_type.chomp
        self.data = picture_field.read
    end

    def base_part_of(file_name)
        File.basename(file_name).gsub(/[^\w._-]/, '')
    end
end
```

Для получения файла, выложенного формой, мы определяем метод-аксессор по имени `uploaded_picture=()`. Объект, возвращенный формой, представляет собой весьма любопытный гибрид. Он подобен файлу, поэтому его содержимое можно прочитать, используя метод `read()` — именно так мы помещаем данные изображения в столбец `data`. У него также есть свойства `content_type` и `original_filename`, которые позволяют нам добраться до метаданных выкладываемого файла. Методы-аксессоры собирают все это по частям, выдавая в результате единый объект, сохраняемый в базе данных в виде отдельных свойств.

Следует заметить, что мы также добавили простую проверку на то, что тип содержимого имеет форму `image/xxx`, чтобы воспрепятствовать чьим-либо намерениям выложить код JavaScript.

Действие `save`, имеющееся в контроллере, не содержит ничего необычного:

rails40/e1/views/app/controllers/upload_controller.rb

```
def save
    @picture = Picture.find(params[:id])
    if @picture.save
        redirect_to(action: 'show', id: @picture.id)
    else
        render(action: :get)
    end
end
```

А как теперь отобразить изображение, попавшее в базу данных? Один из способов предусматривает присвоение ему собственного URL и помещение ссылки на этот URL в тег `image`. Например, чтобы вернуть изображение под номером 123, мы можем воспользоваться URL upload/picture/123. Для возвращения изображения браузеру будет использован метод `send_data()`. Обратите внимание на то, как мы устанавливаем тип содержимого и имя файла, позволяя браузеру интерпретировать данные и снабжать изображение именем по умолчанию на тот случай, если пользователь решит его сохранить:

rails40/e1/views/app/controllers/upload_controller.rb

```
def picture
    @picture = Picture.find(params[:id])
    send_data(@picture.data,
            filename: @picture.name,
            type: @picture.content_type,
            disposition: "inline")
end
```

В заключение мы можем создать действие `show`, отображающее комментарий и изображение. Это действие просто загружает объект модели `picture`:

rails40/e1/views/app/controllers/upload_controller.rb

```
def show
    @picture = Picture.find(params[:id])
end
```

В шаблоне тег `image` содержит ссылку на действие, которое возвращает содержимое изображения. На рис. 21.3 действия `get` и `show` показаны во всей своей красе:

rails40/e1/views/app/views/upload/show.html.erb

```
<h3><%= @picture.comment %></h3>

<img src="<%= url_for(:action => 'picture', :id => @picture.id) %>"/>
```

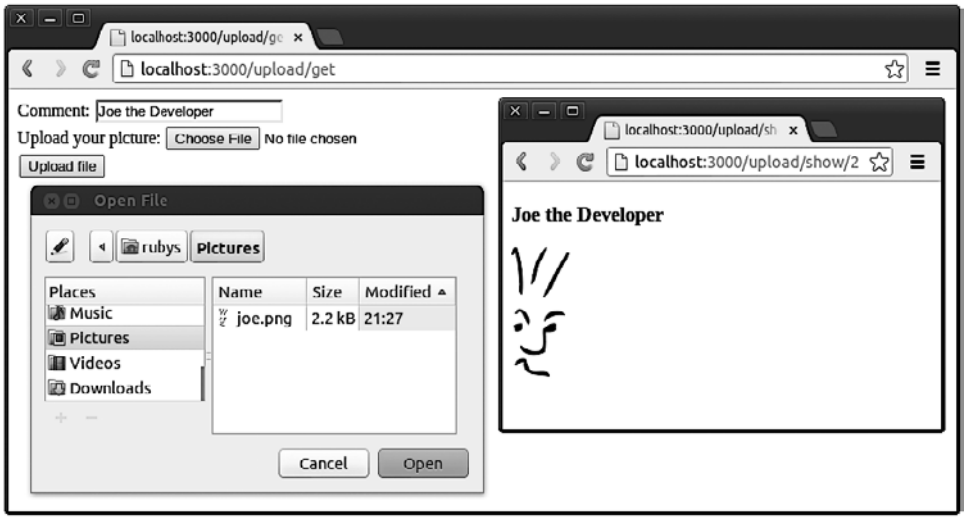


Рис. 21.3. Выкладывание файла

Если вам нужен более простой способ работы с выложенными и сохраненными изображениями, обратите внимание на дополнительный модуль Paperclip[1] (разработки thoughtbot) или attachment_fu[2] (разработки Рика Олсона (Rick Olson)). Создайте таблицу базы данных, включающую заданный набор столбцов (задокументированный на сайте Рика), и дополнительный модуль автоматически справится как с выложенными данными, так и с относящимися к этому метаданными. В отличие от предыдущего подхода, он может хранить выложенные данные либо в вашей файловой системе, либо в таблице базы данных.

Формы и выкладывание данных на сервер — это всего лишь два примера использования предоставляемых Rails помощников. Далее будет показано, как можно предоставлять свои собственные помощники, и будет представлен ряд других помощников, поставляемых с Rails.

[1] https://github.com/thoughtbot/paperclip#readme
[2] https://github.com/technoweenie/attachment_fu

21.5. Использование помощников

Ранее мы говорили, что код в шаблоне вполне допустим. Теперь мы собираемся несколько скорректировать это утверждение. В шаблонах допустимо лишь *весьма незначительное* присутствие программного кода, придающего им динамичность. А его излишки свидетельствуют о плохом стиле разработки.

Подтверждением этому служат три основные причины. Во-первых, чем больше кода вы будете размещать в представлении вашего приложения, тем чаще станете нарушать дисциплину и добавлять в шаблон код, функционально свойственный самому приложению. А это заведомо неверный подход: чтобы обеспечить общедоступность всех основных частей приложения, вы должны определить их на уровне контроллера и модели. Преимущества станут очевидны при добавлении новых способов отображения приложения.

Во-вторых, html.erb в основном состоит из HTML. Значит, при редактировании шаблона вы редактируете HTML-файл. Если вы в состоянии позволить себе при создании макетов сотрудничество с профессиональными дизайнерами, им захочется иметь дело только с HTML. Наличие в нем большого количества Ruby-кода может затруднить их работу.

И последняя причина состоит в том, что этот встроенный в представление код трудно протестировать, а код, выделенный во вспомогательные модули, может быть изолирован и протестирован в виде отдельных блоков.

Rails предоставляет превосходное компромиссное решение в виде помощников. *Помощник* — это простой модуль, в котором содержатся методы, содействующие работе представления.

Методы-помощники имеют направленность на формирование выходной информации. Они предназначены для генерации кода HTML (XML, JavaScript) и расширения свойств шаблонов.

Ваши собственные помощники

По умолчанию каждый контроллер получает свой собственный вспомогательный модуль. Кроме этого есть еще помощник, доступный всему приложению, который находится в файле application_helper.rb. Наверное, вас уже не удивит, что Rails делает соответствующие предположения, помогающие связывать помощники с контроллером и его представлениями. Хотя все помощники представления доступны всем контроллерам, зачастую считается хорошей практикой придать им некую организованность. Помощники, относящиеся только к представлениям, связанным с `ProductController`, лучше поместить в модуль помощников с названием `ProductHelper` в файле product_helper.rb, находящийся в каталоге app/helpers. Запоминать все эти подробности не нужно — сценарий `rails generate controller` создает заготовку модуля помощника автоматически.

В разделе 11.4 «Шаг E4: предотвращение отображения пустой корзины», мы создали такой метод-помощник по имени `hidden_div_if()`, который позволил нам при определенных условиях скрыть корзину. Такую же технологию мы можем

использовать, чтобы немного очистить макет приложения. В настоящее время у нас есть следующий код:

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
```

Давайте переместим код, вырабатывающий заголовок страницы, в контроллер магазина. Поскольку мы находимся в этом контроллере, отредактируем файл store_helper.rb в каталоге app/helpers.

```
module StoreHelper
    def page_title
        @page_title || "Pragmatic Store"
    end
end
```

Теперь в коде представления просто вызывается метод-помощник:

```
<h3><%= page_title %></h3>
```

(Можно исключить и другие повторения, перемещая весь заголовок в отдельный парциальный шаблон, используемый совместно несколькими контроллерными представлениями, но мы не станем касаться парциалов, пока не дойдем до раздела «Шаблоны фрагментов страниц».)

Помощники, предназначенные для форматирования и ссылок

Rails поставляется вместе с пакетом встроенных методов-помощников, доступных всем представлениям. В этом разделе мы коснемся только самых заметных из них, но, возможно, вам захочется просмотреть Action View RDoc, где приведено описание многих специализированных методов, обладающих широкими функциональными возможностями.

Помощники, предназначенные для форматирования

Этот набор методов-помощников работает с датами, числами и текстом.

```
<%= distance_of_time_in_words(Time.now, Time.local(2013, 12, 25)) %>
```
4 months

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, include_
seconds: false) %>
```
1 minute

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, include_
seconds: true) %>
```
Half a minute

```
<%= time_ago_in_words(Time.local(2012, 12, 25)) %>
```
7 months

```
<%= number_to_currency(123.45) %>
```
$123.45

```
<%= number_to_currency(234.56, unit: "CAN$", precision: 0) %>
```
CAN$235
```
<%= number_to_human_size(123_456) %>
```
120.6 KB
```
<%= number_to_percentage(66.66666) %>
```
66.667%
```
<%= number_to_percentage(66.66666, precision: 1) %>
```
66.7%
```
<%= number_to_phone(2125551212) %>
```
212-555-1212
```
<%= number_to_phone(2125551212, area_code: true, delimiter: " ")
%>
```
(212) 555 1212
```
<%= number_with_delimiter(12345678) %>
```
12,345,678
```
<%= number_with_delimiter(12345678, delimiter: "_") %>
```
12_345_678
```
<%= number_with_precision(50.0/3, precision: 2) %>
```
16.67

Метод `debug()` выводит свой параметр, используя формат YAML и нейтрализуя результат, чтобы он мог быть отображен на HTML-странице. Он может пригодиться при попытке взглянуть на значения объектов модели или на параметры запроса.

```
<%= debug(params) %>
```

```
--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test
```

Еще один набор помощников работает с текстом. В нем имеются методы усечения строк и выделения слов в строке.

```
<%= simple_format(@trees) %>
```
Форматирует строку, соблюдая разбиение на строки и абзацы. Ему можно передать обычный текст стихотворения Джойса Килмера (Joyce Kilmer) «Trees», и он добавит к нему HTML-теги для получения следующего формата:

 I think that I shall never see
A poem lovely as a tree. A tree whose hungry mouth is prest
Against the sweet earth's flowing breast;

```
<%= excerpt(@trees, "lovely", 8) %>
```
...A poem lovely as a tre...

```
<%= highlight(@trees, "tree") %>
```
I think that I shall never see A poem lovely as a tree. A tree whose hungry mouth is prest Against the sweet earth's flowing breast;

```
<%= truncate(@trees, length: 20) %>
```
I think that I sh...

В нем также имеется метод для придания существительным формы множественного числа.

```
<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>
```
1 person but 2 people (1 человек, но 2 человека)

Если вам захочется последовать примеру модных веб-сайтов и автоматически выделить гиперссылки на адреса URL и электронной почты, есть помощники, позволяющие это сделать. Существуют и другие помощники, отделяющие гиперссылки от текста.

Ранее, при выполнении шага A2, мы видели, как помощник `cycle()` может использоваться для возвращения при каждом вызове следующих друг за другом значений из последовательности, при необходимости повторяя последовательность. Он часто используется для создания альтернативных стилей для строк в таблице или списке. Также доступны методы `current_cycle()` и `reset_cycle()`.

В заключение следует отметить, что если вы создаете нечто вроде веб-сайта для блогов или позволяете пользователям добавлять комментарии к описаниям товаров в вашем магазине, вы можете предложить им возможность создавать текст в форматах Markdown (BlueCloth)[1] или Textile (RedCloth)[2]. Это простые средства форматирования, которые получают текст с очень простой, легкочитаемой разметкой, и конвертируют его в HTML.

Создание ссылок на другие страницы и ресурсы

Модули `ActionView::Helpers::AssetTagHelper` и `ActionView::Helpers::UrlHelper` содержат ряд методов, позволяющих ссылаться на ресурсы, являющиеся внешними по отношению к текущему шаблону. Чаще всего используется метод `link_to()`, создающий гиперссылку на другое действие вашего приложения:

```
<%= link_to "Добавить комментарий ", new_comments_path %>
```

Первым аргументом метода `link_to()` является текст, отображаемый для ссылки. Следующий аргумент является строкой или хэшем, определяющим цель ссылки.

Необязательный третий аргумент предоставляет HTML-атрибуты для создаваемой ссылки:

```
<%= link_to "Удалить", product_path(@product),
    { class: "dangerous", method: 'delete' }
%>
```

[1] https://github.com/rtomayko/rdiscount

[2] http://redcloth.org/

Этот третий аргумент также поддерживает два необязательных спецификатора, изменяющих поведение ссылки. Каждый из них требует включения на браузере JavaScript.

Спецификатор `:method` дает возможность применить трюк, позволяющий ссылке выглядеть для приложения так, как будто запрос был создан методом POST, PUT, PATCH или DELETE, а не обычным GET-методом. Это достигается созданием фрагмента JavaScript-кода, который отправляет запрос, когда происходит щелчок на ссылке — если JavaScript в браузере отключен, будет сгенерирован GET-запрос.

Параметр `:data` позволяет установить пользовательские атрибуты данных. Одним из наиболее часто используемых спецификаторов является `:confirm`, который получает краткое сообщение. Если он присутствует в аргументе, ненавязчивый драйвер JavaScript выводит сообщение и получает пользовательское подтверждение, прежде чем проследовать по ссылке.

```
<%= link_to "Delete", product_path(@product),
                      method: :delete,
                      data: { confirm: 'Вы уверены?' }
%>
```

Метод `button_to()` работает точно так же, как и `link_to()`, но создает не простую гиперссылку, а кнопку в отдельной форме. Этот метод предпочтителен для ссылки на действия, имеющие побочные эффекты. Но, поскольку эти кнопки находятся в своих собственных формах, на них накладывается ряд ограничений: они не могут появляться внутри строки и не могут появляться внутри других форм.

В Rails имеются условные методы создания ссылок, генерирующие гиперссылки при соблюдении определенных условий, а в противном случае возвращающие лишь текст ссылки. Методы `link_to_if()` и `link_to_unless()` получают аргумент условия, за которым следуют аргументы, обычные для метода `link_to()`. Если условие соблюдается — оно возвращает `true` (для метода `link_to_if`), а если не соблюдается — возвращает `false` (для метода `link_to_unless`), при этом создается обычная гиперссылка, использующая остальные аргументы. Если этого не происходит, имя гиперссылки добавляется в качестве обычного текста, а сама гиперссылка не создается.

Помощник `link_to_unless_current()` создает меню на боковых панелях, где имя текущей страницы показано в виде обычного текста, а другие записи являются гиперссылками:

```
<ul>
<% %w{ создать перечислить изменить сохранить выйти }.each do |action| %>
    <li>
        <%= link_to_unless_current(action.capitalize, action: action) %>
    </li>
<% end %>
</ul>
```

Помощнику `link_to_unless_current()` может также передаваться блок, вычисляемый, только если текущее действие является данным действием, эффективно предоставляющий альтернативу ссылке. Существует также метод-помощник

current_page(), который просто проверяет, был ли текущий запрошенный URI сгенерирован данными элементами.

Как и метод url_for(), link_to(), его производные также поддерживают абсолютные URL-адреса:

```
<%= link_to("Справка", "http://my.site/help/index.html") %>
```

Помощник image_tag() создает теги <img>. Размер изображения определяется необязательным аргументом :size (имеющим форму *ширинаxвысота*) или отдельными аргументами width и height:

```
<%= image_tag("/assets/dave.png", class: "bevel", size: "80x120") %>
<%= image_tag("/assets/andy.png", class: "bevel",
                width: "80", height: "120") %>
```

Если не будет указан необязательный аргумент :alt, Rails его синтезирует на основе имени файла изображения. Если путь к изображению не начинается с символа /, Rails предположит, что это изображение находится в каталоге /images.

Изображения можно превращать в ссылки, объединяя методы link_to() и image_tag():

```
<%= link_to(image_tag("delete.png", size: "50x22"),
            product_path(@product),
            data: { confirm: "Вы уверены?" },
            method: :delete)
%>
```

Помощник mail_to() создает гиперссылку :mailto, по щелчку на которой обычно загружается используемое клиентом приложение электронной почты. Оно получает адрес электронной почты, имя ссылки и набор HTML-аргументов. Внутри этих аргументов для задания исходных значений соответствующим полям сообщения электронной почты можно также использовать спецификаторы :bcc, :cc, :body и :subject.

И наконец, волшебный аргумент encode: "javascript" использует JavaScript на стороне клиента для того, чтобы скрыть сгенерированную ссылку, затрудняя поисковым роботам съем адреса электронной почты с вашего веб-сайта. К сожалению, это также означает, что ваши пользователи не увидят ссылку на адрес электронной почты, если на их браузерах отключен JavaScript.

```
<%= mail_to("support@pragprog.com", "Contact Support",
            subject: "Запрос помощи от #{@user.name}",
            encode: "javascript") %>
```

В качестве более слабого средства запутывания можно воспользоваться спецификаторами :replace_at и :replace_dot, заменяющими символ «собаки» и точки в отображаемом имени другими строковыми значениями. Но сборщиков адресов это вряд ли сможет запутать.

Модуль AssetTagHelper также включает методы-помощники, облегчающие создание ссылок на таблицы стилей и код JavaScript и создание обнаруживаемых в автоматическом режиме ссылок на Atom-каналы. Мы создали такие ссылки

в макете приложения Depot, где в заголовке использовали методы `stylesheet_link_tag()` и `javascript_link_tag()`:

rails40/depot_r/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "application", media: "all",
     "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
```

Метод `javascript_include_tag()` получает список имен JavaScript-файлов (предполагается, что они находятся в каталоге assets/javascripts) и создает HTML-код для их загрузки в страницу. Кроме аргумента `all:` метод `javascript_include_tag` допускает в качестве аргумента значение `:defaults`, которое действует как средство сокращенного вызова и заставляет Rails загрузить `jQuery.js`.

RSS- или Atom-ссылка является полем заголовка, указывающим на URL-адрес в нашем приложении.

При обращении к этому URL приложение должно вернуть соответствующий RSS или Atom XML-код:

```
<html>
<head>
    <%= auto_discovery_link_tag(:atom, products_url(format: 'atom')) %>
</head>
. . .
```

И наконец, в модуле `JavaScriptHelper` определяются несколько помощников для работы с JavaScript. Они создают фрагменты кода JavaScript, запускаемые в браузере для создания специальных эффектов и для получения динамического взаимодействия с нашим приложением.

По умолчанию считается, что такие ресурсы, как изображения и таблицы стилей, находятся в каталогах images и stylesheets, являющихся подкаталогами принадлежащего приложению каталога assets. Если путь, переданный методу тега ресурса, начинается с косой черты (слэша), он считается абсолютным и может обходиться без префикса. Иногда есть смысл переместить это статическое содержимое на отдельный компьютер или в другое место на текущем компьютере. Это делается с помощью настройки переменной конфигурации `asset_host`:

```
config.action_controller.asset_host = "http://media.my.url/assets"
```

Хотя и этот перечень помощников может показаться вполне достаточным, Rails предоставляет намного большее их количество: новые помощники появляются с каждым выпуском, а часть из них выводятся из употребления или перемещаются в дополнительные модули, где они могут совершенствоваться иными темпами, чем сама Rails. Чтобы увидеть, какие еще полезные средства припасла для вас

Rails, сейчас самое время изучить интерактивную документацию, созданную при изучении раздела «Место для документации».

21.6. Сокращение объемов поддержки приложения с помощью макетов и парциалов

До сих пор мы рассматривали в этой главе шаблоны, представленные отдельными фрагментами кода и HTML-содержимого. Но одна из ведущих идей, положенная в основу Rails, заключается в соблюдении принципа DRY и исключении потребностей в повторениях. Тем не менее на среднем веб-сайте имеется множество продублированных фрагментов.

- На многих страницах используются одни и те же заголовки, окончания и боковые панели.
- На многих страницах могут содержаться одни и те же фрагменты выводимого HTML-содержимого (на блог-сайтах, к примеру, может быть много мест, в которых отображается одна и та же статья).
- Во многих местах могут появляться одни и те же функциональные решения. У многих веб-сайтов есть стандартные поисковые компоненты или компоненты опроса, появляющиеся чуть ли не на каждой боковой панели сайта.

Чтобы исключить повторения в этих трех ситуациях, в Rails предусмотрены макеты и парциалы.

Макеты

Rails позволяет выводить страницы, которые вложены в другие выводимые страницы. Как правило, это свойство используется для размещения содержимого, посылаемого действием, внутри стандартного страничного обрамления, предназначенного для всего сайта (заголовка, нижней части и боковой панели). Фактически, если для создания приложений на основе временных платформ вы уже использовали сценарий `generate`, значит, такими макетами вы уже пользовались с самого начала.

Когда Rails принимает запрос на вывод шаблона из контроллера, на самом деле выводятся два шаблона. Вполне очевидно, что одним из них будет запрошенный шаблон (или шаблон, названный по имени действия и выводимый по умолчанию, если явный запрос на вывод отсутствовал). Но кроме этого Rails пытается также найти и вывести шаблон макета (к поиску макета мы вскоре вернемся). Если макет будет найден, Rails вставляет вывод, относящийся к действию в HTML, произведенный макетом.

Рассмотрим шаблон макета:

```
<html>
    <head>
        <title>Form: <%= controller.action_name %></title>
        <%= stylesheet_link_tag 'scaffold' %>
    </head>
    <body>

        <%= yield :layout %>

    </body>
</html>
```

В макете определяется стандартная HTML-страница с разделами `head` и `body`. В качестве заголовка страницы в макете используется имя текущего действия, и в нее включается CSS-файл. В теле присутствует вызов метода `yield`. Именно здесь и происходит волшебство. Когда выставляется шаблон для действия, Rails сохраняет его содержимое с меткой `:layout`. А вызов `yield` внутри шаблона макета извлекает этот текст. Фактически при визуализации `:layout` является контентом по умолчанию, поэтому вместо `yield :layout` можно написать просто `yield`. Мы предпочитаем более явную версию.

Если в шаблоне my_action.html.erb содержится следующее:

```
<h1><%= @msg %></h1>
```

то браузер «увидит» следующий код HTML:

```
<html>
    <head>
        <title>Form: my_action</title>
        <link href="/stylesheets/scaffold.css" media="screen"
            rel="Stylesheet" type="text/css" />
    </head>
    <body>

        <h1>Hello, World!</h1>

    </body>
</html>
```

Размещение файлов макета

Наверное, для вас уже не будет неожиданностью, что Rails позаботилась и о том, чтобы предоставить по умолчанию места для файлов макета, но, если нужно, их можно изменить.

Макеты определяются для контроллеров. Если текущий запрос обрабатывается контроллером по имени `store`, Rails по умолчанию будет искать макет под названием `store` (для которого обычно используется расширение имени файла .html.erb или .xml.builder) в каталоге app/views/layouts. Если в каталоге layouts создается макет

по имени application, он будет применяться для всех контроллеров, у которых нет своих собственных макетов.

Это положение можно изменить, используя внутри контроллера объявление `layout`. В простейшем варианте объявлению передается строковое значение с именем макета. Следующее объявление превратит шаблон в файле standard.html.erb или standard.xml.builder в макет для всех действий в контроллере `store`. Rails будет искать файл макета в каталоге app/views/layouts.

```
class StoreController < ApplicationController

    layout "standard"

    # ...
end
```

Вы можете определить, к каким действиям будет применяться макет, используя спецификаторы `:only` и `:except`:

```
class StoreController < ApplicationController

    layout "standard", except: [ :rss, :atom ]

    # ...
end
```

Если определить для объявления `layout` макета значение `nil`, макеты для контроллера будут отключены.

Бывают ситуации, в которых требуется изменить внешний вид набора страниц при работе приложения. К примеру, на блог-сайте может быть предложен другой внешний вид бокового меню, если пользователь пройдет регистрацию, или на веб-сайте интернет-магазина страницы могут выглядеть по-другому, если сайт выведен на обслуживание. В Rails такие потребности реализованы с помощью динамических макетов.

Если в качестве параметра объявления `layout` используется обозначение, оно должно быть именем метода экземпляра, который возвращает имя используемого макета:

```
class StoreController < ApplicationController

    layout :determine_layout
    # ...
    private

    def determine_layout
        if Store.is_closed?
            "store_down"
        else
            "standard"
        end
    end
end
```

Подклассы контроллера будут использовать родительский макет до тех пор, пока он не будет заменен с помощью объявления `layout`. И наконец, передав в отдельных действиях методу `render` спецификатор `:layout`, можно выбрать визуализацию с использованием особого макета (или вообще без использования какого-либо макета):

```
def rss
    render(layout: false) # never use a layout
end
def checkout
    render(layout: "layouts/simple")
end
```

Передача данных в макеты

Макетам доступны те же самые данные, что и обычным шаблонам. Дополнительно макету будут доступны любые переменные экземпляра, определенные в обычном шаблоне (поскольку такой шаблон выставляется перед тем, как активируется макет). Это обстоятельство может быть использовано для передачи в макет параметров заголовков или меню. К примеру, в макете может быть следующее содержимое:

```
<html>
    <head>
        <title><%= @title %></title>
        <%= stylesheet_link_tag 'scaffold' %>
    </head>
    <body>
        <h1><%= @title %></h1>
        <%= yield :layout %>
    </body>
</html>
```

В отдельном шаблоне заголовок может быть установлен путем присвоения значения переменной `@title`:

```
<% @title = "Моя замечательная жизнь" %>
<p>
    Дорогой дневник:
</p>

<p>
    Вчера на обед была пицца. Она была превосходна.
</p>
```

Фактически можно пойти еще дальше. Тот же механизм, который позволяет использовать `yield :layout` для внедрения интерпретации шаблона в макет, также позволяет создавать произвольное содержимое в шаблоне, который затем может быть внедрен в любой другой шаблон.

Например, различные шаблоны могут нуждаться в добавлении собственных, только им присущих элементов к стандартной боковой панели страницы. Для определения содержимого в таких шаблонах мы воспользуемся механизмом `content_for`, а затем в другом шаблоне, используя метод `yield`, внедрим это содержимое в боковую панель.

В каждом обычном шаблоне `content_for` применяется для присвоения имени содержимому, представленному внутри блока. Это содержимое будет сохранено в Rails и не попадет в вывод, генерируемый шаблоном.

```
        <h1>Обычный шаблон</h1>

<% content_for(:sidebar) do %>
        <ul>
            <li>этот текст будет выставлен</li>
            <li>и сохранен для дальнейшего использования</li>
            <li>он может содержать <%= "dynamic" %> (динамическое) содержимое</li>
        </ul>
    <% end %>

    <p>
        Это обычное содержимое, которое появится на
        странице, выведенной этим шаблоном.
    </p>
```

Затем в макете для включения этого блока в боковую панель страницы используется выражение `yield :sidebar`:

```
<!DOCTYPE .... >
<html>
    <body>
        <div class="sidebar">
            <p>
                Обычное содержимое боковой панели
            </p>

            <div class="page-specific-sidebar">
▶               <%= yield :sidebar %>
            </div>
        </div>
    </body>
</html>
```

Такая же технология может использоваться для добавления специфических для данной страницы функций JavaScript в раздел `<head>`, принадлежащий макету, чтобы создавать специализированные панели меню и т. д.

Шаблоны фрагментов страниц

Веб-приложения обычно отображают информацию об одном и том же объекте или объектах приложения на нескольких страницах. Корзина покупателя может отображать отдельную товарную позицию заказа на своей собственной странице

и показывать ее же на итоговой странице заказа. Блог-приложение может отображать содержимое статьи на главной странице каталога, а потом еще и в верхней части страницы, предназначенной для добавления комментариев. Как правило, это привело бы к копированию фрагментов кода между различными страницами шаблона.

Но в Rails подобное дублирование исключается благодаря применению шаблонов фрагментов страниц или *парциальных шаблонов* (которые чаще всего называют просто *парциалами*). Парциал можно рассматривать как своеобразную подпрограмму: он вызывается из другого шаблона один или несколько раз, возможно, с передачей ему объектов для вывода в качестве параметров. После завершения интерпретации парциального шаблона управление будет возвращено вызвавшему его шаблону.

По внутреннему содержанию парциал похож на любой другой шаблон. А внешне они несколько различаются. Имя файла, в котором содержится код парциала, должно начинаться с символа подчеркивания, чтобы источник шаблонного фрагмента отличался от своих более сложных собратьев.

Например, парциал для визуализации записи блога может быть сохранен в файле _article.html.erb в обычном каталоге представлений app/views/blog:

```
<div class="article">
    <div class="articleheader">
        <h3><%= article.title %></h3>
    </div>
    <div class="articlebody">
        <%= article.body %>
    </div>
</div>
```

Для вызова парциала в других шаблонах используется метод `render(partial:)`:

```
<%= render(partial: "article", object: @an_article) %>
<h3>Добавление комментария</h3>
. . .
```

Аргумент `:partial` метода `render()` является именем вызываемого шаблона (но без лидирующего знака подчеркивания). Это имя одновременно должно отвечать правилам присваивания имен файлам и быть правильным Ruby-идентификатором (поэтому для парциалов не подойдут имена вроде `a-b` и `20042501`). Аргумент `:object` идентифицирует объект, передаваемый парциалу. Этот объект будет доступен внутри шаблона через локальную переменную с таким же именем, как и у самого шаблона. В данном примере шаблону будет передан объект `@an_article`, и шаблон может обращаться к нему, используя локальную переменную `article`. Поэтому мы можем использовать в парциале такие выражения, как `article.title`.

В шаблоне можно установить дополнительные локальные переменные, передав методу `render` аргумент `:locals`. Этому аргументу передается хэш, записи которого представляют собой имена и значения устанавливаемых локальных переменных:

```
render(partial: 'article',
        object: @an_article,
        locals: { authorized_by: session[:user_name],
                  from_ip: request.remote_ip })
```

Парциалы и коллекции

Приложениям обычно нужно вывести коллекции отформатированных записей. В блоге может выводиться подборка статей, у каждой из которых будет текст, автор, дата и т. д. Магазин может выводить записи каталога, у каждой из которых может быть изображение, описание и цена.

В методе `render()` в сочетании с аргументом `:partial` может быть использован аргумент `:collection`. Аргумент `:partial` позволяет использовать парциал для определения формата отдельной записи, а аргумент `:collection` применяет этот шаблон к каждому представителю коллекции. Для отображения списка объектов модели статьи (`article`) с использованием ранее определенного нами парциала _article.html.erb можно воспользоваться следующим кодом:

```
<%= render(partial: "article", collection: @article_list) %>
```

Внутри парциала локальной переменной `article` будет присвоено значение текущей статьи из коллекции, поскольку переменная носит имя шаблона. Вдобавок переменной `article_counter` будет присвоено значение индекса текущей статьи в коллекции.

Необязательный аргумент `:spacer_template` позволяет определить шаблон, который будет выведен между каждым элементом коллекции. К примеру, представление может содержать следующий код:

rails40/e1/views/app/views/partial/_list.html.erb

```
<%= render(partial: "animal",
           collection: %w{ муравей пчела кошка собака лось },
           spacer_template: "spacer")
%>
```

В нем используется парциал _animal.html.erb для вывода названия каждого животного из представленного списка со вставкой фрагмента _spacer.html.erb между этими названиями. Если парциал _animal.html.erb содержит код:

rails40/e1/views/app/views/partial/_animal.html.erb

```
<p>Животное: <%= animal %></p>
```

а _spacer .html.erb содержит код:

rails40/e1/views/app/views/partial/_spacer.html.erb

```
<hr />
```

то ваши пользователи увидят список названий животных с горизонтальной линией между ними.

Общие шаблоны

Если в вызове метода `render` аргумент `:partial` содержит только имя, Rails предполагает, что нужный парциал находится в каталоге представления текущего контроллера. Но если имя содержит один или более символов прямого слэша (`/`), Rails предполагает, что та часть, которая следует до последнего слэша, является именем каталога, а остальная часть — это имя парциала. Предполагается, что каталог, в свою очередь, находится в каталоге app/views. Это упрощает совместное использование парциалов несколькими контроллерами.

По соглашению, действующему для Rails-приложений, общие парциалы хранятся в подкаталоге app/views, названном shared. Они могут быть выведены с использованием следующего кода:

```
<%= render("shared/header", locals: {title: @article.title}) %>
<%= render(partial: "shared/post", object: @article) %>
. . .
```

В предыдущем примере объект `@article` внутри шаблона будет назначен локальной переменной `post`.

Парциалы с макетами

Парциалы могут выводиться с макетом, и вы можете применить макет к блоку внутри любого шаблона:

```
<%= render partial: "user", layout: "administrator" %>

<%= render layout: "administrator" do %>
    # ...
<% end %>
```

Парциальные макеты должны находиться непосредственно в каталоге app/views, связанном с контроллером, и иметь общепринятый префикс в виде знака подчеркивания, например app/views/users/_administrator.html.erb.

Парциалы и контроллеры

Парциалы используются не только шаблонами представления, но и контроллерами.

Парциалы дают контроллерам возможность генерировать фрагменты из страницы, используя такие же парциальные шаблоны, что и в самом представлении. Это особенно важно при использовании поддержки AJAX для частичного обновления страницы из контроллера — использование парциала гарантирует вам, что форматирование, примененное к обновляемой строке таблицы или товарной позиции, будет совместимо с тем, что использовалось первоначально для генерации их собратьев.

Парциалы и макеты вместе взятые предоставляют эффективный способ обеспечения обслуживаемости той части вашего приложения, которая касается пользовательского интерфейса. Но обслуживаемость — это только часть истории: также очень важно, что их применение является способом добиться от приложения хорошей работы.

Наши достижения

Представления являются открытым лицом Rails-приложений, и мы увидели, что Rails предоставляет широкую поддержку всего необходимого для создания надежных и легко обслуживаемых интерфейсов для пользователя и для прикладного программирования.

Сначала были рассмотрены шаблоны. Rails предоставляет встроенную поддержку для четырех типов: ERB, Builder, CoffeeScript и SCSS. Шаблоны облегчают нам предоставление ответов в форматах HTML, XML, CSS и JavaScript на любой запрос. Добавление еще одного варианта будет рассмотрено в разделе 25.2 «Придание красоты нашей разметке с помощью Haml».

Мы погружались в формы, являющиеся первичными средствами, с помощью которых пользователи взаимодействуют с вашим приложением. Наряду с этим мы рассмотрели вопрос выкладывания файлов на сервер.

Мы продолжили рассмотрение методов-помощников, позволивших нам вынести сложную логику приложения за пределы наших представлений, чтобы можно было сконцентрироваться на чисто представительских аспектах. Мы исследовали ряд помощников, предоставляемых Rails: от тех, что выполняют простое форматирование, и до тех, что создают гипертекстовые ссылки, являющиеся конечным способом взаимодействия пользователей с HTML-страницами.

Мы завершили наш тур по Action View рассмотрением двух родственных способов «вынесения за скобки» больших порций содержимого для их повторного использования. Мы воспользовались макетами, чтобы вынести за скобки самые внешние слои представления и обеспечить тем самым однообразный внешний вид приложения. И мы воспользовались парциалами, чтобы вынести за скобки общие внутренние компоненты, такие как отдельная форма или таблица.

Все это касалось вопросов возможного доступа к нашему Rail-приложению пользователя с браузером. Далее нам предстоит рассмотреть вопросы определения и обслуживания схемы базы, которую наше приложение будет использовать для хранения данных.

22 Миграции

Основные темы:

- присваивание имен файлам миграции;
- переименование столбцов;
- создание и переименование таблиц;
- определение индексов и ключей;
- использование исходного кода SQL.

Rails поддерживает гибкий, пошаговый стиль разработки. Но не стоит ожидать, что все получится с первого раза. Просветление наступает в процессе работы, при создании тестов и общении с заказчиками.

Поэтому для работы нам нужен ряд вспомогательных действий. Мы пишем тесты, помогающие разрабатывать интерфейсы и выстраивать систему безопасного внесения различных изменений, и используем систему управления версиями для хранения исходных файлов приложения, позволяющую осуществлять отмену в случае ошибочных действий и отслеживать повседневные изменения.

Но есть такая изменяемая область приложения, которой невозможно управлять напрямую, используя систему управления версиями. По мере разработки Rails-приложения схема, используемая базой данных, подвергается постоянным изменениям: мы добавляем в нее таблицы, изменяем имена столбцов и т. д. Изменения, вносимые в базу данных, взаимосвязаны с изменениями кода приложения.

При использовании Rails каждый из этих шагов становится возможным благодаря миграциям. Все это вы уже видели в действии при разработке приложения Depot, когда создавали первую таблицу `products` в главе 6, в разделе «Генерирование временной платформы», и когда выполняли такие задачи, как добавление количества к таблице `line_items` в разделе 10.1, «Шаг Д1: создание

усовершенствованной корзины». Теперь настало время глубже разобраться с тем, как работают миграции и что еще с ними можно делать.

22.1. Создание и запуск миграций

Миграции — это простые файлы, находящиеся в каталоге db/migrate, в которых содержится код Ruby. По умолчанию имя каждого файла миграции начинается с нескольких цифр (обычно их четырнадцать) и знака подчеркивания. Эти цифры являются для миграций ключевыми, поскольку они определяют порядок, в котором миграции применяются, — для каждой миграции они являются индивидуальным номером версии.

Сам по себе номер версии является меткой создания миграции в формате всемирного времени (UTC). Входящие в эту метку числа включают в себя четыре цифры года, за которыми следуют цифры для месяца, дня, часа, минуты и секунды, основанные на среднем солнечном времени Королевской обсерватории в лондонском предместье Гринвич. Поскольку миграции создаются относительно редко и метка времени записывается с точностью до секунды, шансы того, что двое людей получат одну и ту же метку времени, пренебрежительно малы. А преимущества от наличия меток времени, которые могут быть выстроены в обусловленном порядке, существенно перевешивают ничтожный риск получения одинаковой метки времени.

Каталог db/migrate приложения Depot имеет следующий вид:

```
depot> dir db\migrate
20121130000001_create_products.rb
20121130000002_create_carts.rb
20121130000003_create_line_items.rb
20121130000004_add_quantity_to_line_items.rb
20121130000005_combine_items_in_cart.rb
20121130000006_create_orders.rb
20121130000007_add_order_id_to_line_item.rb
20121130000008_create_users.rb
```

Хотя миграции можно создавать и самостоятельно, гораздо проще (и с меньшим риском появления ошибок) использовать генератор. При разработке приложения Depot мы видели, что существует два генератора, создающие файлы миграции.

Генератор *модели* создает миграцию, которая, в свою очередь, создает таблицу, связанную с моделью (если только не будет применен ключ `--skip-migration`). В следующем примере показано, что при создании модели под названием `discount` заодно создается и миграция под названием *yyyyMMddhhmmss*_create_discounts.rb:

```
depot> rails generate model discount
    invoke active_record
    db/migrate/20121113133549_create_discounts.rb
    create app/models/discount.rb
    invoke test_unit
    create test/models/discount_test.rb
    create test/fixtures/discounts.yml
```

Также можно создать миграцию саму по себе:

```
depot> rails generate migration add_price_column
    invoke active_record
▶   create db/migrate/20121113133814_add_price_column.rb
```

Далее, начиная с раздела «Внутреннее устройство миграции», мы увидим, какой код попадает в файлы миграций. А сейчас давайте забежим немного вперед и посмотрим, как запускаются миграции.

Запуск миграций

Миграции запускаются с помощью Rake-задачи `db:migrate`:

```
depot> rake db:migrate
```

Чтобы посмотреть, что происходит дальше, давайте заглянем поглубже в Rails.

Код миграции обслуживает таблицу под названием `schema_migrations`, которая имеется в каждой базе данных Rails. Эта таблица состоит всего лишь из одного столбца, названного `version`, и в ней будет содержаться всего одна строка на каждую успешно примененную миграцию.

При запуске команды `rake db:migrate` задача в первую очередь ищет таблицу `schema_migrations`. Если такой таблицы еще нет, она будет создана.

Затем код миграции пересмотрит все файлы, имеющиеся в каталоге db/migrate, и не станет рассматривать те из них, номер версии которых (лидирующие цифры в имени файла) уже находится в базе данных. Затем он переходит к применению оставшихся миграций, создавая для каждой строку в таблице `schema_migrations`.

Если в данный момент мы еще раз запустим эту миграцию, не последует абсолютно никакой реакции. Каждому из номеров версий файлов миграций будет соответствовать строка в базе данных, поэтому невыполненных миграций не будет.

Но если мы затем создадим новый миграционный файл, номера его версии в базе данных не будет. Это верно, даже если номер версии предшествовал номеру одной или нескольких уже примененных миграций. Такое может произойти, когда несколько пользователей используют для хранения файлов миграций систему управления версиями. Если затем запустить миграции, будет выполнен этот новый файл миграции, и только он. Это может означать, что миграции запускаются не по порядку, поэтому вам может потребоваться обеспечить независимость этих миграций и позаботиться об этом. Или же вам может потребоваться вернуть свою базу данных к предыдущему состоянию, а затем выполнить миграции по порядку.

Вы можете заставить базу данных вернуться к указанной версии, предоставив команде `rake db:migrate` аргумент `VERSION=`:

```
depot> rake db:migrate VERSION=20121130000009
```

Если заданная версия больше любой другой уже примененной миграции, эти миграции будут применены.

Если же номер версии в командной строке меньше одной или нескольких версий, перечисленных в таблице `schema_migrations`, дело примет несколько иной оборот. В таком случае Rails ищет файл миграции, чей номер соответствует последней версии в базе данных, и отменяет его действия. Этот процесс будет повторен до тех пор, пока в таблице `schema_migrations` не останется номеров, превышающих номер, указанный в командной строке. Получается, что миграции отменяются в обратном порядке, чтобы вернуть схему назад к указанной вами версии.

Вы также можете отменить применение одной или нескольких миграций:

```
depot> rake db:migrate:redo STEP=3
```

По умолчанию команда `redo` вернется на одну миграцию назад, а затем снова ее запустит. Для возвращения на несколько миграций команде нужно передать аргумент `STEP=`.

22.2. Внутреннее устройство миграции

Миграции являются подклассом Rails-класса `ActiveRecord::Migration`. При необходимости миграции могут содержать методы `up()` и `down()`:

```
class SomeMeaningfulName < ActiveRecord::Migration
    def up
        # ...
    end
    def down
        # ...
    end
end
```

Имя класса после перевода всех символов верхнего разряда в нижний разряд и установки лидирующего символа подчеркивания должно соответствовать той части имени файла, которая следует после номера версии. Например, предыдущий класс может быть найден в файле по имени 20121130000017_some_meaningful_name.rb. Никакие две миграции не могут содержать классы с одинаковыми именами.

Метод `up()` отвечает за применение миграции с целью внесения изменений в схему данных, а метод `down()`отменяет вносимые изменения. Давайте немного уточним сказанное. Рассмотрим миграцию, добавляющую столбец `e_mail` к таблице `orders`:

```
class AddEmailToOrders < ActiveRecord::Migration
    def up
        add_column :orders, :e_mail, :string
    end

    def down
        remove_column :orders, :e_mail
    end
end
```

Вы поняли, как метод `down()` отменяет то, что делает метод `up()`?

Вы также могли заметить наличие в этом коде продублированной информации. Во многих случаях Rails может найти способ автоматической отмены заданной операции. Например, противоположностью методу `add_column()`, несомненно, будет метод `remove_column()`. В таких случаях путем простого переименования `up()` в `change()` вы можете исключить надобность в методе `down()`:

```
class AddEmailToOrders < ActiveRecord::Migration
    def change
        add_column :orders, :e_mail, :string
    end
end
```

Теперь все стало намного чище, не так ли?

Типы столбцов

В третьем аргументе метода `add_column` определен тип столбца базы данных. В предыдущем примере мы определили, что столбец `e_mail` будет иметь тип `:string`. Но что это значит? В базах данных вообще-то отсутствует такой тип столбца.

Вспомним, что Rails пытается сделать приложение независимым от используемой базы данных. К примеру, если хотите, можно вести разработку, используя SQLite 3, и развертывать приложение, используя Postgres. Но различные базы данных используют для типов столбцов разные названия. Если в миграции использовать тип столбца из SQLite 3, эта миграция, возможно, не будет работать с базой данных Postgres. Поэтому в миграциях Rails применяются логические типы, что позволяет изолировать вас от системы типов, существующей в используемой базе данных. Если мы применяем миграцию к базе данных SQLite 3, то указание `:string` приведет к созданию столбца с типом данных `varchar(255)`. В Postgres та же самая миграция добавит столбец с типом данных `char varying(255)`.

В миграциях поддерживаются следующие типы данных `:binary`, `:boolean`, `:date`, `:datetime`, `:decimal`, `:float`, `:integer`, `:string`, `:text`, `:time` и `:timestamp`. Как эти типы проецируются по умолчанию на существующие в Rails адаптеры баз данных, показано в табл. 22.1 и 22.2. Используя эти таблицы, можно определить, что столбец с типом данных, объявленным как `:integer`, при осуществлении миграции получит в SQLite 3 базовый тип `integer`, а в Oracle — тип `number(38)`.

Таблица 22.1. Исходное отображение типов для адаптеров баз данных (часть1)

	db2	mysql	openbase	oracle
:binary	blob(32768)	blob	object	blob
:boolean	decimal(1)	tinyint(1)	boolean	number(1)

Окончание ▸

Таблица 22.1 (окончание)

	db2	mysql	openbase	oracle
:date	date	date	date	date
:datetime	timestamp	datetime	datetime	date
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float	number
:integer	int	int(11)	integer	number(38)
:string	varchar(255)	varchar(255)	char(4096)	varchar2(255)
:text	clob(32768)	text	text	clob
:time	time	time	time	date
:timestamp	timestamp	datetime	timestamp	date

Таблица 22.2. Исходное отображение типов для адаптеров баз данных (часть 2)

	postgresql	sqlite	sqlserver	sybase
:binary	bytea	blob	image	image
:boolean	boolean	boolean	bit	bit
:date	date	date	date	datetime
:datetime	timestamp	datetime	datetime	datetime
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float(8)	float(8)
:integer	integer	integer	int	int
:string	(прим. 1)	varchar(255)	varchar(255)	varchar(255)
:text	text	text	text	text
:time	time	datetime	time	time
:timestamp	timestamp	datetime	datetime	timestamp

Прим. 1: character varying(256)

При определении столбцов, чаще всего, можно задать до трех аргументов; для столбцов десятичных чисел требуется два дополнительных аргумента. Каждый из трех аргументов задается в виде пары *ключ*`:` *значение*. Наиболее часто применяются следующие ключи.

null: true или false

Если задано значение `false`, к столбцу добавляется ограничение, не разрешающее столбцу быть пустым, — `not null` (если база данных его поддерживает). Примечание: это не зависит от какой-либо проверки данных `presence: true`, которая может быть выполнена на уровне модели.

limit: размер
Устанавливает предельное значение размеров поля. Как правило, таким образом к определению типа столбца базы данных добавляется размер строки.

default: значение
Устанавливает для столбца значение по умолчанию. Поскольку все это выполняется базой данных, при инициализации этого значения и даже при его сохранении в новом объекте модели ничего не видно. Чтобы увидеть значение, объект нужно перезагрузить. Учтите, что значение по умолчанию вычисляется только один раз, в момент запуска миграции, поэтому следующий код установит для столбца значение по умолчанию на дату и время запуска миграции:

```
add_column :orders, :placed_at, :datetime, default: Time.now
```

При указании столбцов десятичных чисел передаются ключи `:precision` и `:scale`. Ключ `:precision` задает число сохраняемых значащих цифр, а ключ `:scale` определяет, где в этом числе будет находиться разделитель целой и дробной части (ключ `:scale` следует рассматривать как количество цифр после разделителя). Десятичное число с ключом `:precision`, имеющим значение 5, и ключом `:scale`, имеющим значение 0, может хранить числа в диапазоне от –99 999 до +99 999. А десятичное число с ключом `:precision`, имеющим значение 5, и ключом `:scale`, имеющим значение 2, может хранить числа в диапазоне от –999,99 до +999,99.

Ключи `:precision` и `:scale` являются для столбцов десятичных чисел необязательными. Но из-за несовместимости типов в различных базах данных мы настоятельно рекомендуем включать эти ключи для каждого столбца десятичных чисел.

Приведем несколько примеров определения столбцов с типами данных и ключами, используемыми в миграциях:

```
add_column :orders, :attn, :string, limit: 100
add_column :orders, :order_type, :integer
add_column :orders, :ship_class, :string, null: false, default: 'priority'
add_column :orders, :amount, :decimal, precision: 8, scale: 2
```

Переименование столбцов

При реорганизации кода, чтобы наши переменные были понятнее, мы часто изменяем их имена. Миграции Rails позволяют нам делать то же самое с именами столбцов базы данных. Например, через неделю после добавления столбца `e_mail` мы решим, что это не самое лучшее имя. В таком случае мы можем создать миграцию для переименования столбца с помощью метода `rename_column()`:

```
class RenameEmailColumn < ActiveRecord::Migration
    def change
        rename_column :orders, :e_mail, :customer_email
    end
end
```

Поскольку `rename_column()` имеет обратимый характер, отдельные методы `up()` и `down()` для его использования не требуются.

Следует заметить, что переименование не вредит существующим данным, связанным со столбцом. Также следует знать, что переименование поддерживается не всеми адаптерами.

Изменение свойств столбцов

Для изменения типа столбца или переустановки связанных с ним свойств используется метод `change_column()`. Способ применения этого метода такой же, как и у `add_column()`, только вместо нового указывается имя уже существующего столбца. Предположим, что существует столбец `order_type` типа `integer`, а нам нужно изменить его тип на `string`. Желательно, чтобы имеющиеся данные сохранились, и заказ, обозначенный цифрой `123`, превратился в заказ, обозначенный строкой `"123"`. Чуть позже мы будем использовать значения, не отображающие целочисленные значения, например `"new"` и `"existing"`.

Изменение целочисленного столбца на строковый осуществляется довольно просто:

```
def up
    change_column :orders, :order_type, :string
end
```

Но обратное преобразование без проблем не обходится. Можно впасть в искушение и написать, казалось бы, очевидную миграцию `down()`:

```
def down
    change_column :orders, :order_type, :integer
end
```

Но если наше приложение сохранило в этом столбце такие данные, как `"new"`, метод `down()` их потеряет, поскольку значение `"new"` не может быть преобразовано в целое число. Если это приемлемо, миграцию можно оставить в неизменном виде. Если же нужно создать однонаправленную, необратимую миграцию, применение метода `down` следует исключить. В таком случае Rails предоставляет специальное исключение, которое можно вставить в код:

```
class ChangeOrderTypeToString < ActiveRecord::Migration
    def up
        change_column :orders, :order_type, :string, null: false
    end
    def down
        raise ActiveRecord::IrreversibleMigration
    end
end
```

`ActiveRecord::IrreversibleMigration` также является именем исключения, выдаваемого Rails при попытке вызова метода, который не может быть автоматически реверсирован из метода `change()`.

22.3. Управление таблицами

До сих пор мы использовали миграции для работы со столбцами существующих таблиц. Теперь посмотрим, как создавать и удалять сами таблицы:

```
class CreateOrderHistories < ActiveRecord::Migration
    def change
        create_table :order_histories do |t|
            t.integer :order_id, null: false
            t.text :notes
            t.timestamps
        end
    end
end
```

Методу `create_table()` передается имя таблицы (не забудьте, что оно должно быть указано во множественном числе) и блок. (Также ему передаются некоторые необязательные аргументы, которые будут рассмотрены вскоре.) Блоку передается объект, определяющий структуру таблицы, который используется для задания столбцов таблицы.

Обычно для удаления таблицы вызывать метод `drop_table()` не нужно, поскольку метод `add_table()` является обратимым. Методу `drop_table()` передается один аргумент, являющийся именем удаляемой таблицы.

Вызовы различных методов определения таблиц могут показаться знакомыми, поскольку они идентичны вызовам использовавшегося ранее метода `add_column`, за исключением того, что им в качестве первого аргумента не передается имя таблицы, а имя самого метода является типом желаемых данных.

Обратите внимание на то, что для новой таблицы не определяется столбец `id`. Пока не указано обратное, миграции Rails автоматически добавляют первичный ключ с именем `id` ко всем создаваемым таблицам. Более подробно этот вопрос рассмотрен далее в разделе «Первичные ключи».

Метод `timestamps` создает столбцы `created_at` (время создания) и `updated_at` (время обновления) с соответствующим типом `timestamp`. Хотя добавлять эти столбцы к любой конкретной таблице не требуется, это еще один пример того, как Rails облегчает для общепринятого соглашения его простую и однообразную реализацию.

Ключи, используемые для создания таблиц

В качестве второго аргумента методу `create_table` можно передать хэш, состоящий из ключей.

Если указать ключ `force: true`, миграция перед созданием новой таблицы удалит существующую таблицу с таким же именем. Такой параметр пригодится, если нужно создать миграцию, приводящую базу данных в определенное состояние, не считаясь с полной потерей данных.

Ключ `temporary: true` приведет к созданию временной таблицы, которая будет уничтожена, когда приложение отключится от базы данных. Бессмысленность

использования миграции для подобного действия очевидна, но, как выяснится несколько позже, в других местах это действие может найти свое применение.

Ключ `options: "xxxx"` позволяет задать используемой базе данных дополнительные свойства. Они добавляются к окончанию инструкции `CREATE TABLE`, сразу же после закрывающих круглых скобок. Для SQLite 3 это вряд ли пригодится, но порой может быть использовано с другими серверами баз данных. Например, некоторые версии MySQL позволяют указать начальное значение для столбца с автоприращением по имени `id`. Мы можем передать это значение через миграцию с помощью следующего кода:

```
create_table :tickets, options: "auto_increment = 10000" do |t|
    t.text :description
    t.timestamps
end
```

Закулисно миграции сгенерируют из этого описания таблицы следующий фрагмент DDL-кода для MySQL:

```
CREATE TABLE "tickets" (
    "id" int(11) default null auto_increment primary key,
    "description" text,
    "created_at" datetime,
    "updated_at" datetime
) auto_increment = 10000;
```

При использовании ключа `:options` с MySQL нужно проявлять особое внимание. Имеющийся в Rails адаптер базы данных MySQL содержит установку по умолчанию `ENGINE=InnoDB`, которая отменяет все возможные локальные установки по умолчанию и заставляет миграции использовать процессор базы данных `InnoDB`. Но если вы используете отменяющий ключ `:options`, эти установки будут утрачены; новые таблицы будут создаваться с использованием того процессора базы данных, на который настроен по умолчанию ваш сайт. В этом случае нужно заставить систему привести поведение к стандартному, добавив к строке ключей окончание `ENGINE=InnoDB`. Возможно, вам захочется продолжить использование `InnoDB`, если вы используете MySQL, поскольку этот процессор дает вам поддержку транзакций. Транзакции могут понадобиться в вашем приложении, и они определенно нужны в ваших тестах, если по умолчанию используются тестовые стенды, рассчитанные на использование транзакций.

Переименование таблиц

Если реорганизация приводит к переименованию переменных и столбцов, неудивительно, что иногда потребуется переименовать и таблицы. На этот случай миграции поддерживают метод `rename_table()`:

```
class RenameOrderHistories < ActiveRecord::Migration
    def change
        rename_table :order_histories, :order_notes
    end
end
```

Отмена этой миграции отменяет и изменение, возвращая таблице прежнее имя.

Проблемы, связанные с методом rename_table

При переименовании таблиц с помощью миграций возникает одна довольно тонкая проблема.

Предположим, к примеру, что в миграции 4 мы создали таблицу `order_histories` и поместили в нее какие-нибудь данные:

```
def up
    create_table :order_histories do |t|
        t.integer :order_id, null: false
        t.text :notes
        t.timestamps
    end

    order = Order.find :first
    OrderHistory.create(order_id: order, notes: "test")
end
```

Затем, в миграции 7 мы переименовали таблицу `order_histories` в `order_notes`. К этому моменту мы также переименовали модель `OrderHistory` в `OrderNote`.

Теперь мы решили удалить нашу разработочную базу данных и заново применить миграции. Когда мы это делаем, миграции выдают исключение в миграции 4: наше приложение больше не содержит класс по имени `OrderHistory`, поэтому миграция не выполняется.

Тим Лукас (Tim Lucas) предложил решение этой проблемы — создавать локальные имитационные версии классов модели, которые будут нужны только самим миграциям. Например, следующая версия четвертой миграции будет работать даже в том случае, если в приложении больше не будет класса `OrderHistory`:

```
class CreateOrderHistories < ActiveRecord::Migration

    class Order < ActiveRecord::Base; end
    class OrderHistory < ActiveRecord::Base; end

    def change
        create_table :order_histories do |t|
            t.integer :order_id, null: false
            t.text :notes
            t.timestamps
        end

        order = Order.find :first
        OrderHistory.create(order: order_id, notes: "test")
    end
end
```

Этот код будет работать, пока ваши классы модели не будут нести какой-нибудь дополнительной функциональной нагрузки, которая использовалась бы в миграции. Все, что здесь создается, — это всего лишь голая структурная версия.

Определение индексов

Миграции могут (и, наверное, должны) определять индексы для таблиц. Например, к определенному моменту у вашего приложения в базе данных скопится большое количество заказов, и поиск, основанный на имени клиента, будет занимать больше времени, чем хотелось бы. Значит, настало время добавить индекс, воспользовавшись методом с соответствующим именем — `add_index()`:

```
class AddCustomerNameIndexToOrders < ActiveRecord::Migration
    def change
        add_index :orders, :name
    end
end
```

Если методу `add_index` передать необязательный аргумент `unique: true`, будет создан уникальный индекс, требующий уникальности значений в индексируемом столбце.

По умолчанию индексу будет присвоено имя вида `index_`*таблица*`_on_`*столбец*. Это положение можно отменить, воспользовавшись ключом `name: "`*некое_имя*`"`. Если при добавлении индекса используется ключ `:name`, нам также нужно указать этот ключ и при удалении индекса.

Можно создать составной индекс, то есть индекс на основе данных нескольких столбцов, передав методу `add_index` массив, состоящий из имен столбцов. В таком случае для названия индекса будет использовано имя только первого столбца.

Удаляются индексы с помощью метода `remove_index()`.

Первичные ключи

Rails предполагает, что у каждой таблицы есть числовой первичный ключ (который, как правило, называется `id`), и при этом гарантирует уникальность содержимого этого столбца для каждой новой строки, добавляемой к таблице.

Давайте все это перефразируем.

На самом деле Rails не может нормально работать, пока у каждой таблицы не будет числового первичного ключа. В отношении имени столбца она ведет себя менее критично.

Поэтому для обычного Rails-приложения мы настоятельно советуем ничего не менять и позволить Rails пользоваться ее столбцом `id`.

Если вы все же отважитесь на приключения, можете начать с использования для столбца первичного ключа другого имени (сохранив для этого столбца свойство целочисленного автоприращения). Это делается с помощью указания ключа `:primary_key` в вызове метода `create_table`:

```
create_table :tickets, primary_key: :number do |t|
    t.text :description
    t.timestamps
end
```

Этот код добавит к таблице столбец `number` и сделает его столбцом первичного ключа:

```
$ sqlite3 db/development.sqlite3 ".schema tickets"
CREATE TABLE tickets ("number" INTEGER PRIMARY KEY AUTOINCREMENT
NOT NULL, "description" text DEFAULT NULL, "created_at" datetime
DEFAULT NULL, "updated_at" datetime DEFAULT NULL);
```

Следующим шагом на пути к приключениям может стать создание первичного ключа, не имеющего целочисленного значения. Не стану скрывать, что Rails-разработчики вряд ли сочтут эту затею разумной: миграции не позволят вам это сделать (по крайней мере, напрямую).

Таблицы, не имеющие первичного ключа

Иногда могут понадобиться таблицы без первичного ключа. Чаще всего в Rails такие таблицы нужны в качестве *объединительных* — у них всего два столбца, каждый из которых является внешним ключом к другой таблице. Для создания объединительных таблиц с помощью миграций нужно сообщить Rails, что автоматическое добавление столбца `id` не требуется:

```
create_table :authors_books, id: false do |t|
    t.integer :author_id, null: false
    t.integer :book_id, null: false
end
```

В таком случае у вас может появиться желание исследовать вопрос создания одного или нескольких индексов для такой таблицы для ускорения переходов между книгами и их авторами.

22.4. Расширенное применение миграций

Большинство Rails-разработчиков используют основные возможности, предоставляемые миграциями, для создания и дальнейшей поддержки своей схемы базы данных. Но время от времени полезно воспользоваться миграциями не только для этого. В этом разделе рассматривается расширенное применение миграций.

Использование исходного кода SQL

Миграции дают вам способ манипулирования схемой, независимый от используемых баз данных. Но если в миграциях нет методов, необходимых для осуществления ваших замыслов, вам нужно спуститься вниз к тому коду, который применяется в вашей базе данных. Для этого Rails предоставляет два способа:

один — с использованием аргументов `options` для методов вроде add_column(), а второй — с использованием метода `execute()`.

При использовании `options` или `execute()` вы должны плотно привязать свою миграцию к конкретному процессору базы данных, поскольку в любом SQL, предоставляемом в этих двух местах, используется исходный синтаксис вашей базы данных.

Типичным примером в ваших собственных миграциях может послужить добавление внешнего ключа, накладывающего ограничения на дочернюю таблицу.

Это можно сделать путем добавления в наш исходный файл миграции метода, похожего на следующий:

```
def foreign_key(from_table, from_column, to_table)
    constraint_name = "fk_#{from_table}_#{to_table}"
    execute %{
        CREATE TRIGGER #{constraint_name}_insert
        BEFORE INSERT ON #{from_table}
        FOR EACH ROW BEGIN
            SELECT
                RAISE(ABORT, "constraint violation: #{constraint_name}")
            WHERE
                (SELECT id FROM #{to_table} WHERE
                    id = NEW.#{from_column}) IS NULL;
        END;
    }
    execute %{
        CREATE TRIGGER #{constraint_name}_update
        BEFORE UPDATE ON #{from_table}
        FOR EACH ROW BEGIN
            SELECT
                RAISE(ABORT, "constraint violation: #{constraint_name}")
            WHERE
                (SELECT id FROM #{to_table} WHERE
                    id = NEW.#{from_column}) IS NULL;
        END;
    }
    execute %{
        CREATE TRIGGER #{constraint_name}_delete
        BEFORE DELETE ON #{to_table}
        FOR EACH ROW BEGIN
            SELECT
                RAISE(ABORT, "constraint violation: #{constraint_name}")
            WHERE
                (SELECT id FROM #{from_table} WHERE
                    #{from_column} = OLD.id) IS NOT NULL;
        END;
    }
end
```

Внутри миграции `up()` мы можем вызвать этот новый метод, используя следующий код:

```
def up
    create_table ... do
    end
```

```
    foreign_key(:line_items, :product_id, :products)
    foreign_key(:line_items, :order_id, :orders)
end
```

Но нам может потребоваться пойти еще дальше и сделать наш метод `foreign_key()` доступным всем нашим миграциям. Для этого следует создать модуль в каталоге lib приложения и добавить к нему метод `foreign_key()`.

Но на этот раз его нужно сделать не методом класса, а простым методом экземпляра:

```
module MigrationHelpers

  def foreign_key(from_table, from_column, to_table)
    constraint_name = "fk_#{from_table}_#{to_table}"

    execute %{
      CREATE TRIGGER #{constraint_name}_insert
      BEFORE INSERT ON #{from_table}
      FOR EACH ROW BEGIN
       SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
       WHERE
        (SELECT id FROM #{to_table} WHERE id = NEW.#{from_column}) IS NULL;
      END;
    }
    execute %{
      CREATE TRIGGER #{constraint_name}_update
      BEFORE UPDATE ON #{from_table}
      FOR EACH ROW BEGIN
       SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
       WHERE
        (SELECT id FROM #{to_table} WHERE id = NEW.#{from_column}) IS NULL;
      END;
    }
    execute %{
      CREATE TRIGGER #{constraint_name}_delete
      BEFORE DELETE ON #{to_table}
      FOR EACH ROW BEGIN
       SELECT
        RAISE(ABORT, "constraint violation: #{constraint_name}")
       WHERE
       (SELECT id FROM #{from_table} WHERE #{from_column} = OLD.id) IS NOT NULL;
       END;
    }
  end
end
```

Теперь его можно добавить к любой миграции путем вставки в верхнюю часть нашего файла миграции следующих строк:

```
▶require "migration_helpers"
   class CreateLineItems < ActiveRecord::Migration
▶    extend MigrationHelpers
```

Строка `require` приносит определение модуля в код миграции, а строка `extend` добавляет методы в модуле `MigrationHelpers` в миграцию в качестве методов класса. Эту технологию можно использовать для разработки и совместного использования любого количества вспомогательных методов, предназначенных для миграции.

(И если вы желаете облегчить свою жизнь еще больше, для вас уже написан дополнительный модуль[1], автоматически добавляющий ограничения внешнего ключа.)

Специальные сообщения и подсчеты времени

Хотя это напрямую и не относится к расширенному применению миграций, в расширенных миграциях можно воспользоваться выводом своих собственных сообщений и подсчетов времени. Это делается с помощью метода `say_with_time()`:

```
def up
    say_with_time "Обновление цен..." do
        Person.all.each do |p|
            p.update_attribute :price, p.lookup_master_price
        end
    end
end
```

Метод `say_with_time()` выводит строку, переданную перед исполняемым блоком, и выводит результат подсчета затраченного времени после завершения выполнения блока.

22.5. Слабая сторона миграций

Миграции страдают от одной серьезной проблемы. Лежащие в их основе DDL-инструкции, обновляющие схему базы данных, не могут выполняться в режиме транзакций. И Rails в этом не виновата — большинство баз данных просто не поддерживает отмену созданной таблицы, изменение таблицы и другие DDL-инструкции.

Давайте взглянем на миграцию, которая пытается добавить к базе данных две таблицы:

```
class ExampleMigration < ActiveRecord::Migration
    def change
        create_table :one do ...
        end
        create_table :two do ...
        end
    end
end
```

[1] https://github.com/matthuhiggins/foreigner

При нормальном развитии событий метод `up()` добавляет таблицы, `one` и `two`, а метод `down()` их удаляет.

Но что произойдет, если возникнут проблемы с созданием второй таблицы? У нас будет база данных, содержащая таблицу `one` и не имеющая таблицы `two`. С помощью миграции мы можем справиться с любой возникшей проблемой, но в данном случае мы не сможем ее применить — если мы попытаемся это сделать, то потерпим фиаско, поскольку таблица `one` уже существует.

Можно попробовать отменить миграцию, но из этого тоже ничего не получится: поскольку сама миграция целиком не сработала, версия схемы базы данных не прошла обновление, поэтому Rails не будет осуществлять попытку ее отмены.

В такой ситуации вы могли бы оставить подобные попытки, самостоятельно изменить информацию о схеме и удалить таблицу `one`. Но, наверное, так делать не стоит. В подобных обстоятельствах мы рекомендуем просто удалить всю базу данных, создать ее заново и применить к ней существующие миграции, приводящие ее к требуемому на данный момент состоянию. Вы ничего не потеряете и будете знать, что имеете схему, выстроенную в последовательности версий.

Все эти разговоры наводят на мысль, что миграции опасно применять к базе данных в уже работающем программном продукте. Нужно ли вам их запускать? У нас нет ответа на данный вопрос. Если в вашей организации есть администраторы баз данных, это будет их обязанностью. Если все возложено на вас, нужно взвесить все риски. Но если вы решитесь на это, сначала нужно будет непременно сделать резервную копию вашей базы данных. Затем можно будет применить миграции, зайдя в каталог вашего приложения на той машине, которая выполняет роль базы данных на вашем эксплуатационном сервере, и выполнить следующую команду:

```
depot> RAILS_ENV=production rake db:migrate
```

Это один из тех случаев, когда вступают в силу замечания об ответственности, изложенные в начале этой книги. Мы не несем никакой ответственности в том случае, если вы удалите свои данные.

22.6. Манипуляции со схемой данных вне миграций

Все рассмотренные на данный момент в этой главе методы миграций доступны также в качестве методов и в объектах подключений Active Record, а следовательно, доступны в моделях, представлениях и контроллерах Rails-приложений.

Например, вы можете обнаружить, что какой-то отдельный длинный отчет составляется намного быстрее, если таблица `orders` проиндексирована по столбцу `city`. Но этот индекс при обычной ежедневной работе приложения абсолютно не нужен, а тесты показывают, что его поддержка ощутимо замедляет работу.

Давайте создадим метод, создающий индекс, запускающий блок кода, а затем удаляющий ранее созданный индекс. Он может быть закрытым (`private`) методом модели или быть реализован в какой-нибудь библиотеке.

```
def run_with_index(*columns)
    connection.add_index(:orders, *columns)
    begin
        yield
    ensure
        connection.remove_index(:orders, *columns)
    end
end
```

Имеющийся в модели метод сбора статистики может воспользоваться этим методом следующим образом:

```
def get_city_statistics
    run_with_index(:city) do
        # .. вычисление статистических данных
    end
end
```

Наши достижения

Хотя в процессе разработки приложения Depot и даже при его развертывании мы уже свободно пользовались миграциями, в данной главе мы увидели, что миграции служат основой для принципиальных и строгих подходов для управления конфигурированием схемы вашей базы данных.

Мы изучили порядок создания, переименования и удаления столбцов и таблиц; управления индексами и ключами; применения и отмены целых наборов изменений; и даже подмешивания вашего собственного специализированного кода SQL в смешанный код, и сделали все это полностью повторяемым способом.

Здесь мы рассмотрели внешние стороны Rails. В следующих нескольких главах предстоит провести более глубокое исследование. Мы собираемся показать вам, как демонтировать Rails и собрать ее снова. И первой остановкой на этом пути будет демонстрация использования избранных классов и методов Rails вне контекста веб-сервера.

Приложения, не использующие браузер

23

Основные темы:

- вызов Rails-методов;
- доступ к данным Rails-приложения;
- удаленная работа с базами данных.

В предыдущих главах основное внимание уделялось обмену данными между человеком и сервером, большей частью посредством HTML. Но не все сетевые взаимодействия нуждаются в непосредственном участии человека. В этой главе основное внимание будет уделено доступу к вашему Rails-приложению и данным из автономного сценария.

Существует множество причин, по которым вам может понадобиться доступ к частям вашего Rails-приложения не из браузера. Например, у вас может появиться желание загружать свою базу данных или периодически ее приводить в порядок с использованием фонового планировщика заданий типа cron. У вас могут быть рабочие приложения, возможно даже Rails-приложения, которым необходим непосредственный доступ к данным, находящимся в другом Rails-приложении, возможно даже на другой машине. Вам может понадобиться интерфейс командной строки — не в силу каких-то потребностей, а просто так.

Независимо от причин, Rails всегда к вашим услугам. Как вы увидите, для выполнения задуманной вами задачи у вас будет возможность привлечь Rails в минимально необходимой или в максимально необходимой степени.

Начнем с предположения, что ваше приложение находится на той же самой машине, на которой установлена среда Rails и хранятся ваши данные, а затем

перейдем к рассмотрению способов выполнения ваших задач на удаленной машине.

23.1. Автономное приложение, использующее Active Record

Прежде всего, вам захочется иметь беспрепятственный доступ к вашим данным. Вам приятно будет узнать, что вы можете всецело задействовать Active Record прямо из автономного приложения. Сначала будет показан «сложный» способ решения этого вопроса (кавычки поставлены потому, что не все так уж сложно, ведь, в конце концов, речь идет о Rails). Затем мы покажем вам легкий способ.

Начнем с автономной программы, использующей Active Record для работы с таблицей заказов в базе данных SQLite 3. После завершения поиска заказа с конкретным `id` она изменит имя покупателя и сохранит результат в базе данных, обновляя исходную строку:

```
require "active_record"

ActiveRecord::Base.establish_connection(adapter: "sqlite3",
    database: "db/development.sqlite3")

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.name = "Dave Thomas"
order.save
```

Здесь есть все необходимое — в данном случае не требуется никакой конфигурационной информации (кроме той, которая относится к подключению базы данных). Active Record определяет наши потребности на основе самой схемы базы данных, а все необходимые подробности берет на себя.

Теперь, изучив «сложный» способ, давайте посмотрим на легкий, где Rails будет обслуживать для вас подключение и загружать все ваши модели:

```
require "config/environment.rb"
order = Order.find(1)
order.name = "Dave Thomas"
order.save
```

Чтобы это работало, Ruby нужно будет найти файл config/environment.rb для того приложения, данные из которого вы хотите загрузить. Это можно сделать, указав полный путь к этому файлу в инструкции `require` или включив путь в переменную среды окружения `RUBYLIB`. Другой переменной среды окружения для ведения поиска является `RAILS_ENV` — она используется для выбора из разработочной, тестовой и рабочей среды окружения.

Поскольку мы затребовали только этот файл, мы имеем доступ примерно к той же части нашего приложения, к которой имели доступ при использовании команды `rails console` в главе 14, в разделе «Давайте все-таки оставим последнего администратора...».

Все это было сделано с помощью всего лишь одной инструкции `require`. Проще уже некуда. Но, верите вы этому или нет, временами вам будет нужен доступ только к части свойств, предоставляемых Rails вне контекста какого-нибудь Rails-приложения. Сейчас мы рассмотрим этот вопрос.

23.2. Библиотечная функция, использующая Active Support

Active Support — это набор библиотек, совместно используемый всеми Rails-компонентами. Часть из того, что там находится, предназначена для использования внутри Rails, но все это доступно для использования приложениями, не связанными с Rails. Это может представлять интерес в том случае, если вы разрабатываете Rails-приложение и в ходе этой разработки создаете набор классов или даже набор методов, которыми хотите воспользоваться в приложении, не имеющем отношения к Rails. Вы начинаете с переноса этого нужного кода в отдельный файл, а затем обнаруживаете, что этот код не запускается — не потому, что его логика каким-то образом зависит от вашего приложения, а потому, что он использует другие методы и классы, предоставляемые Rails.

Начнем с краткого обзора некоторых наиболее важных из этих библиотек, а попутно покажем, как их можно сделать доступными вашему приложению.

Расширения ядра (core-ext)

Active Support расширяет некоторые встроенные классы Ruby в весьма интересных и полезных направлениях. В этом разделе мы дадим краткий обзор наиболее популярных расширений ядра.

- *Array:* `second()`, `third()`, `fourth()`, `fifth()` и `forty_two()`. Дополняет методы `first()` и `last()`, предоставляемые самим Ruby.
- *CGI:* `escape_skipping_slashes()`. В соответствии со значением имени, отличается от метода `escape()` тем, что не занимается нейтрализацией слэшей.
- *Class:* Методы для доступа к свойствам класса, для доступа к делегированию полномочий, унаследованные методы чтения и записи, а также потомки (известные как подклассы). Их слишком много, поэтому здесь они не перечисляются, а подробности нужно искать в документации.
- *Date:* `yesterday()`, `future?()`, `next_month()` и многие, многие другие.
- *Enumerable:* `group_by()`, `sum()`, `each_with_object()`, `index_by()`, `many?()` и `exclude?()`.

- *File:* `atomic_write()` и `path()`.
- *Float:* Добавляет необязательный аргумент точности к методу `round()`.
- *Hash:* `diff()`, `deep_merge()`, `except()`, `stringify_keys()`, `symbolize_keys()`, `reverse_merge()` и `slice()`. Многие из этих методов также имеют варианты, имена которых оканчиваются восклицательным знаком.
- *Integer:* `ordinalize()`, `multiple_of?()`, `months()`, `years()`. См. также *Numeric*.
- *Kernel:* `debugger()`, `breakpoint()`, `silence_warnings()`, `enable_warnings()`.
- *Module:* Методы доступа к свойствам модуля, методы поддержки псевдонимов, делегирования, возражения, внутренние средства чтения и записи, синхронизации и родственных связей.
- *Numeric:* `bytes()`, `kilobytes()`, `megabytes()` и т. д.; `seconds()`, `minutes()`, `hours()` и т. д.
- *Object:* `blank?()`, `present?()`, `duplicable?()`, `instance_values()`, `instance_variable_names()`, `returning()` и `try()`.
- *String:* `exclude?()`, `pluralize()`, `singularize()`, `camelize()`, `titleize()`, `un-der-score()`, `dasherize()`, `demodulize()`, `parameterize()`, `tableize()`, `clas-sify()`, `humanize()`, `foreign_key()`, `constantize()`, `squish()`, `mb_chars()`, `at?()`, `from()`, `to()`, `first()`, `last()`, `to_time()`, `to_date()` и `try()`.
- *Time:* `yesterday()`, `future?()`, `advance()` и многие, многие другие.

Как видите, это довольно длинный список. Эти методы чаще всего весьма невелики по размеру, многие из них состоят всего лишь из одной строки кода. Хотя, скорее всего, вы воспользуетесь лишь небольшой частью этих методов, все они доступны для использования в вашем Rails-приложении.

Можно также убедиться в их разнообразии. Большинством из них вы, возможно, даже никогда напрямую не воспользуетесь. Но вы быстро усвоите небольшую часть этих дополнительных методов, посчитав, что они входят в язык Ruby. Хотя все эти методы задокументированы в Интернете[1], наилучшим способом их изучения зачастую становятся эксперименты с использованием консоли Rails. Вот несколько из таких методов на пробу:

- `2.years.ago`
- `[1,2,3,4].sum`
- `5.gigabytes`
- `"man".pluralize`
- `String.methods.sort`

Поскольку не существует какого-нибудь одного способа определения, какой из поднаборов на вас работает, просто знайте, что эти методы существуют,

[1] http://as.rubyonrails.org/

и просмотрите документацию, когда возникнет общая потребность в чем-нибудь особенном, поскольку разработчики Rails уже могли добавить искомый метод.

Дополнительные классы Active Support

Кроме расширения базовых объектов, предоставляемых Ruby, Active Support предоставляет массу дополнительных функциональных возможностей. Во многом, точно так же, как и с расширениями ядра, эти классы предназначены для поддержки специфических потребностей других компонентов Rails, но вы свободно можете воспользоваться этими функциями напрямую.

- *Benchmarkable:* Замеряет время выполнения блока в шаблоне и записывает результаты в регистрационный журнал.
- *Cache::Store:* Предлагает различные реализации кэш-хранилищ на основе файлов или памяти, с дополнительной синхронизацией или сжатием.
- *Callbacks:* Предоставляет средства вмешательства в жизненный цикл объекта.
- *Concern and Dependencies:* Помогает управлять зависимостями модульным способом.
- *Configurable:* Предоставляет конфигурационную переменную класса `Hash`.
- *Deprecation:* Предоставляет свойства, сообщения и средства изоляции для поддержки перевода методов в разряд нерекомендуемых.

ДЭВИД ГОВОРИТ: ПОЧЕМУ РАСШИРЕНИЕ БАЗОВЫХ КЛАССОВ НЕ ПРИВОДИТ К АПОКАЛИПСИСУ?

Настороженность, возникающая при первой встрече с таким кодом, как 5.months + 30.minutes, обычно через некоторое время переходит в легкую панику. Если каждый может запросто вмешаться в работу с целыми числами, не приведут ли все эти совершенно неподдерживаемые и наспех созданные программы к полному хаосу? Конечно, если каждый будет этим постоянно заниматься, все именно так и может произойти. Но дело обстоит несколько иначе, поэтому апокалипсис не наступит.

Не нужно видеть в Active Support коллекцию случайных расширений языка Ruby, которая открыта чуть ли не для всех, и позволяет любому добавлять к классу string свои собственные, понравившиеся ему свойства. Этот модуль следует рассматривать в качестве диалекта Ruby для универсального разговора со всеми Rails-программистами. Поскольку Active Support является неотъемлемой частью Rails, вы всегда можете положиться на то, что метод 5.months будет работать в любом Rails-приложении. Это опровергает факт наличия проблемы, связанной с существованием нескольких тысяч персональных диалектов Ruby.

Если говорить о расширениях языка, Active Support дает нам все самое лучшее из обоих миров, являясь контекстной стандартизацией.

- *Duration:* Предлагает дополнительные методы, например `ago()` и `since()`.
- *Gzip:* Предлагает удобные методы для сжатия — `compress()` и восстановления — `decompress()` строкового объекта.

- *HashWithIndifferentAccess:* Позволяет пользоваться обеими нотациями: `params[:key]` и `params['key']`.
- *I18n:* Предоставляет поддержку локализации.
- *Inflections:* Справляется с исключениями английского языка при создании множественной формы существительного.
- *JSON:* Предоставляет методы кодирования и декодирования текстового формата обмена данными JavaScript Object Notation.
- *LazyLoadHooks:* Предоставляет поддержку отложенной инициализации модулей.
- *MessageEncryptor:* Кодирует значения, которые должны быть сохранены в каком-нибудь ненадежном месте.
- *MessageVerifier:* Генерирует и проверяет подписанные сообщения (для предотвращения несанкционированного вмешательства).
- *MultiByte:* Предоставляет многобайтное кодирование (главным образом для Ruby 1.8.7).
- *Notifications:* Предлагает инструментальное оснащение API.
- *OptionMerger:* Предлагает глубокое объединение лямбда-выражений.
- *OrderedHash and OrderedOptions:* Предоставляет упорядоченный хэш (главным образом для Ruby 1.8.7).
- *Railtie:* Определяет основные объекты, от которых может зависеть вся остальная среда.
- *Rescueable:* Облегчает обработку исключений.
- *SecureRandom:* Генерирует случайные числа, подходящие для генерирования ключей сессий в cookie-файлах HTTP.
- *StringInquirer:* Предоставляет наилучший способ тестирования на равенство.
- *TestCase:* Предоставляет различные методы тестирования gem-пакетов Ruby и связанного с gem-пакетами поведения в безопасных песочницах.
- *Time и TimeWithZone:* Предлагает дополнительную поддержку для вычислений и преобразований времени.

Хотя в этой книге мы не будем исследовать все 49 (на данный момент) методов и подсчитывать то, что, к примеру, предоставляет один только метод `TimeWithZone`, предыдущий список позволит вам найти необходимые функции в руководствах и документации по API. Но в этой книге будет показано, как вы сможете использовать данные методы в ваших автономных приложениях.

```
require "active_support/time"
Time.zone = 'Eastern Time (US & Canada)'
puts Time.zone.now
```

Поэтому если вы, подобно многим другим людям, отдадите предпочтение одному или нескольким из этих расширений, вы сможете просто запросить то, что

вам нужно (например, `require "active_support/basic_object"` или `require "active_support/core_ext"`) или привлечь сразу все с помощью инструкции `require "active_support/all"`.

Использование помощников Action View

Хотя эта тема и не имеет прямого отношения к категории Active Support, она к ней все же достаточно близка. То, что применимо к Active Support, также применимо и к остальным частям Rails, хотя большинство методов, относящихся к маршрутизации, контроллерам и Action View, чаще всего занимаются обработкой активного веб-запроса.

Одним из примечательных исключений являются некоторые помощники Action View. Рассмотрим пример получения доступа к помощнику Action View из автономного приложения:

```
require "action_view"
require "action_view/helpers"
include ActionView::Helpers::DateHelper
puts distance_of_time_in_words_to_now(Time.parse("December 25"))
```

В целом здесь приходится затратить немного больше усилий, чем при получении доступа к намного более часто востребованным методам Active Support, но все равно это особого труда не составляет.

Наши достижения

Мы сняли ограничения, связанные с использованием браузера, и обращались к методам Active Support, Action View и Active Record из автономного сценария. Это позволило нам создать сценарии, которые могут запускаться из командной строки, встраиваться в существующие приложения или периодически запускаться с использованием таких средств, как cron.

И наконец, мы использовали Active Resource для снятия ограничений, вынуждавших запускать ваш сценарий на той же самой машине, на которой находится приложение. Хотя это требует немного большего объема настроек, в результате мы получаем работоспособные и безопасные средства обращения к данным, имеющимся внутри Rails-приложения.

Далее мы изучим другие отдельно устанавливаемые компоненты, которые включены в пакет при установке Rails.

24 Зависимости Rails

Основные темы:

- шаблоны XML и HTML;
- управление зависимостями приложения;
- написание сценариев для выполнения задач;
- взаимодействие с веб-сервером.

Пока мы рассматривали только основы Rails. Но еще есть многое, о чем можно рассказать. Многое, из чего складываются превосходные функциональные возможности Rails, предоставляется компонентами, на которых основана эта среда.

Эти компоненты должны быть вам знакомы, поскольку каждым из них вы уже пользовались. Atom, шаблоны HTML, `rake db:migrate`, `bundle install` и `rails server` применялись при разработке приложения Depot.

Хотя эта глава не касается вашей обычной повседневной работы и показывает, как каждый компонент может быть использован по отдельности, это не означает, что она дает исчерпывающее описание любого из этих компонентов. По правде говоря, описание каждого компонента потребовало бы отдельных небольших книг. А данная глава предназначена для того, чтобы дать вам представление о нескольких ключевых компонентах с целью заложить необходимую стартовую основу для ваших собственных исследований в выбранных вами направлениях.

Глава начнется с представления вам некоторых зависимостей, и сначала будут рассмотрены основные механизмы шаблонов, расширяющих возможности представлений. Затем мы исследуем такой компонент, как Bundler, который используется для управления зависимостями. И наконец, мы покажем, как эти части объединяются с помощью Rack и Rake.

24.1. Генерирование XML с помощью Builder

Builder является автономной библиотекой, позволяющей отображать в коде структурированный текст (например, XML). Builder-шаблон (в файле с расширением .xml.builder) содержит код Ruby, который для генерирования XML использует библиотеку Builder.

Простой Builder-шаблон, предназначенный для вывода списка наименований товаров и цен в формате XML, имеет следующий вид:

rails40/depot_t/app/views/products/index.xml.builder

```
xml.div(class: "productlist") do

    xml.timestamp(Time.now)

    @products.each do |product|
        xml.product do
            xml.productname(product.title)
            xml.price(product.price, currency: "USD")
        end
    end
end
```

Если он напоминает вам тот шаблон, который создавался в разделе 12.2 «Шаг Ж2: применение Atom-канала» для использования с помощником Atom, причина заключается в том, что помощник Atom построен на основе функциональных возможностей библиотеки Builder.

Располагая подходящей коллекцией товаров (переданной из контроллера), шаблон может выдать код следующего вида:

```
<div class="productlist">
    <timestamp>2013-01-29 09:42:07 -0500</timestamp>
    <product>
        <productname>CoffeeScript</productname>
        <price currency="USD">36.0</price>
    </product>
    <product>
        <productname>Programming Ruby 1.9</productname>
        <price currency="USD">49.5</price>
    </product>
    <product>
        <productname>Rails Test Prescriptions</productname>
        <price currency="USD">43.75</price>
    </product>
</div>
```

Обратите внимание на то, как Builder взял имена методов и превратил их в XML-теги. Там, где был вызов метода `xml.price`, он создал тег по имени `<price>`, чьим содержимым стал первый аргумент и чьи атрибуты были установлены из следующего далее хэша. Если имя тега, которым нужно воспользоваться, конфликтует

с именем уже имеющегося метода, для генерации тега необходимо воспользоваться методом `tag!()`:

```
xml.tag!("id", product.id)
```

Builder может генерировать практически любой необходимый код XML. При этом поддерживаются пространства имен, объекты, обработка инструкций и даже XML-комментарии. Все подробности можно найти в документации по Builder.

Хотя внешне HTML очень похож на XML, между ними довольно много различий для использования при создании HTML совершенно другого механизма шаблонов. Он и станет нашей следующей темой.

24.2. Генерирование HTML с помощью ERB

В самом простом случае ERB-шаблон представляет собой обычный HTML-файл. Если шаблон не содержит динамического контента, он просто отправляется на браузер пользователя в своем первозданном виде. Следующий код является вполне допустимым html.erb-шаблоном:

```
<h1>Привет, Дэйв!</h1>
<p>
    Как поживаешь?
</p>
```

Но приложения, которые выводят одни лишь статические шаблоны, имеют слишком пресный вид. Их можно приправить использованием динамического содержимого:

```
<h1>Привет, Дэйв!</h1>
<p>
    Сейчас <%= Time.now %>
</p>
```

Читатели, которым приходилось разрабатывать серверные страницы (JSP), поймут, что здесь присутствует встроенное выражение. ERB выполняет любой код между тегами `<%=` и `%>`, конвертирует результаты в строку с помощью метода `to_s()`, нейтрализует специальные символы HTML и вставляет затем эту строку в результирующую страницу. Выражение внутри тегов может быть произвольным кодом:

```
<h1>Привет, Дэйв!</h1>
<p>
    Сейчас <%= require 'date'
            DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
            Thursday Friday Saturday }
            today = Date.today
            DAY_NAMES[today.wday]
        %>
</p>
```

Обычно наличие в шаблоне слишком большого количества бизнес-логики считается дурным тоном, из-за чего вас может задержать разгневанная полиция программистских нравов. В разделе 21.5 «Использование помощников» мы уже рассматривали более приемлемый для этой ситуации путь, заключающийся в использовании методов-помощников.

Иногда в шаблонах нужен код, который не генерирует напрямую никакого вывода. Если в открывающем теге не ставить знак равенства, содержимое тега будет выполнено, но в шаблон ничего вставлено не будет. Предыдущий пример можно переписать следующим образом:

```
<% require 'date'
    DAY_NAMES = %w{ Sunday Monday Tuesday Wednesday
                    Thursday Friday Saturday }
    today = Date.today
%>
<h1>Привет, Дэйв!</h1>
<p>
    Сегодня <%= DAY_NAMES[today.wday] %>.
    А завтра будет <%= DAY_NAMES[(today + 1).wday] %>.
</p>
```

В мире JSP это называется *скриптлетом*. И снова вас многие осудят, если увидят, что вы добавляете код к шаблонам. Не обращайте внимания на этих догматиков. В наличии кода в шаблонах нет ничего предосудительного. Просто не нужно помещать в них слишком много кода (в особенности относящегося к бизнес-логике). Как мы уже видели, для успешной борьбы с этим соблазном можно воспользоваться методами-помощниками.

HTML-текст между фрагментами кода можно считать строками, которые были записаны Ruby-программой. А фрагменты `<%...%>` считать дополнениями к этой же самой программе. HTML смешивается с написанным вами явным кодом. В результате код между тегами `<%` и `%>` может повлиять на вывод HTML в остальной части шаблона.

Рассмотрим в качестве примера следующий шаблон:

```
<% 3.times do %>
    Нет!<br/>
<% end %>
```

Когда вставляется значение с использованием структуры `<%=...%>`, результатом будет HTML, нейтрализованный перед помещением непосредственно в поток вывода. Обычно это именно то, что вам нужно.

Если же подставляемый текст содержит HTML, который должен быть интерпретирован, это приведет к нейтрализации HTML-тегов, — если создать строку, содержащую `<em>hello</em>`, а затем подставить ее в шаблон, пользователь увидит `<em>hello</em>`, а не просто `hello`. На этот случай Rails предоставляет несколько методов-помощников. Приведем ряд примеров.

Метод `raw()` заставит передать строку прямо на вывод без ее нейтрализации. Он предоставляет больше гибкости при меньшей безопасности.

Применительно к элементам массива метод `row()` нейтрализует небезопасный HTML-код, объединяя результаты с предоставленной строкой и возвращая в результате безопасный HTML-код.

Метод `sanitize()` предоставляет меры защиты. Он получает строку, содержащую HTML, и вычищает потенциально опасные элементы: теги `<form>` и `<script>` нейтрализуются, а атрибуты `on=` и ссылки, которые начинаются с `javascript:`, удаляются.

Описания товаров в нашем приложении Depot выводилось в виде HTML (то есть они были обозначены как безопасные с использованием метода `raw()`). Это позволило нам вставлять в них отформатированную информацию. Если мы разрешаем вводить описания кому-нибудь за пределами нашей организации, то чтобы снизить риск атаки на наш сайт, будет благоразумнее воспользоваться методом `sanitize()`.

Эти два механизма шаблонов являются всего лишь двумя из многих других gem-пакетов, от которых зависит работа Rails. А сейчас есть смысл поговорить о том, как можно управлять этими зависимостями.

24.3. Управление зависимостями с помощью Bundler

Существует ложное представление о том, что управление зависимостями является весьма сложной задачей. В процессе разработки вы можете выбрать установку обновленных версий gem-пакетов, от которых зависит работа вашего приложения. После этого может обнаружиться исчезновение тех проблем, которые проявлялись в рабочем режиме, поскольку теперь ваши прогоны берут совсем другие версии gem-пакетов, от которых зависит работа вашего приложения. А может получиться и так, что обнаружатся проблемы, отсутствовавшие в рабочем режиме.

Оказывается, что управление зависимостями играет не менее важную роль, чем управление исходным кодом приложения или схемами базы данных. Если разработка ведется в составе команды, нужно, чтобы все в команде пользовались одинаковыми версиями компонентов, от которых зависит работа приложения. При развертывании приложения нужно убедиться в том, что те версии этих компонентов, с которыми проверялась работа приложения, установлены на целевой машине, и именно они используются в рабочем режиме.

Bundler[1] заботится об этом, основываясь на содержимом файла Gemfile, который находится в корневом каталоге вашего приложения. В этом файле перечисляются зависимости вашего приложения. Давайте более пристально посмотрим на Gemfile для приложения Depot:

rails40/depot_u/Gemfile

```
source 'https://rubygems.org'
# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
```

[1] http://gembundler.com/

```
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'
group :production do
    gem 'mysql2'
end

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
gem 'jquery-ui-rails'

# Turbolinks makes following links in your web application faster.
# Read more: https://github.com/rails/turbolinks
gem 'turbolinks'

# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 1.2'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

# Use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'

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

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

# Use debugger
# gem 'debugger', group: [:development, :test]
```

В первой строке указывается, где искать новые gem-пакеты и новые версии существующих gem-пакетов. Вы можете повторить эту строку, чтобы перечислить свои личные gem-хранилища.

В следующей строке указывается, какую версию Rails нужно загрузить. Обратите внимание: здесь указана конкретная версия. Затем следует комментарий о том,

что вы можете воспользоваться альтернативой, чтобы запустить самую свежую версию Rails.

В остальных строках перечислен ряд gem-пакетов, используемых вами, и ряд gem-пакетов, которые могут рассматриваться к применению. Некоторые из них помещены в группы, озаглавленные `:development` (разработка), `:test` (тестирование) или `:production` (эксплуатация), и будут доступны только в указанных средах окружения. Некоторые из них включают необязательный аргумент `:require`, указывающий имя для использования в инструкции `require` для тех случаев, когда оно отличается от имени gem-пакета.

В строке для `sass-rails` показан спецификатор версии, перед которым стоит оператор сравнения. Хотя файлы Gemfile поддерживают несколько таких операторов, чаще всего используются только два из них. Оператор `>=` предназначен для столь редкого, к сожалению, условия, когда автору Gemfile можно доверять поддержку строгой обратной совместимости, поэтому здесь нужно лишь указать минимальный номер версии.

Оператор `~>` рекомендуется для более частого применения. Точным соответствием для него будут, по сути, все части номера версии, за исключением последней части, а последняя часть указывает на минимум. Поэтому `~> 3.1.4` соответствует любой версии, номер которой начинается с 3.1 и по значению не меньше чем 3.1.4. Аналогично `~> 3.0` означает любую версию, начинающуюся с 3.

У Gemfile есть файл-компаньон по имени Gemfile.lock. Этот второй файл обычно обновляется одной из двух команд: `bundle install` или `bundle update`. Между ними нет практически никакой разницы.

Прежде чем продолжить, полезно будет взглянуть на файл Gemfile.lock. Вот небольшой фрагмент этого файла:

```
GEM
  remote: https://rubygems.org/
  specs:
    actionmailer (4.0.0)
      actionpack (= 4.0.0)
      mail (~> 2.5.3)
    actionpack (4.0.0)
      activesupport (= 4.0.0)
      builder (~> 3.1.0)
      erubis (~> 2.7.0)
      rack (~> 1.5.2)
      rack-test (~> 0.6.2)
    activemodel (4.0.0)
      activesupport (= 4.0.0)
           builder (~> 3.1.0)
```

Команда `bundle install` будет использовать Gemfile.lock в качестве отправной точки и установит только те версии различных gem-пакетов, которые указаны в этом файле. Поэтому важно, чтобы этот файл был зарегистрирован в вашей системе управления версиями, поскольку это станет гарантией того, что вашими коллегами и целевыми машинами развертывания будет использована точно такая же конфигурация.

Команда `bundle update` будет (что неудивительно) обновлять один или несколько gem-пакетов и будет соответствующим образом обновлять файл Gemfile.lock. Если вы хотите использовать конкретную версию отдельного gem-пакета, нужно будет последовательно отредактировать Gemfile, чтобы отразить в нем вносимые вами ограничения, а затем запустить команду `bundle update`, перечислив gem-пакеты, которые нужно обновить. Если не указать список gem-пакетов, Bundler попытается обновить все gem-пакеты, но этого, вообще-то, не рекомендуется делать, особенно накануне развертывания.

Bundler также имеет компонент времени выполнения, используемый для того, чтобы гарантировать, что ваше приложение в точности загружает только те версии gem-пакетов, которые перечислены в файле Gemfile.lock. Этот вопрос будет исследован чуть позже при изучении работы сервера.

24.4. Взаимодействие с веб-сервером с помощью Rack

Rails запускает ваше приложение в контексте веб-сервера. До сих пор мы использовали два отдельных веб-сервера: WEBRick, который поступает встроенным в язык Ruby, и Phusion Passenger, который интегрирован с веб-сервером Apache HTTP. Существуют и другие доступные варианты, включая Mongrel, Lighttpd, Unicorn и Thin.

На основе этого можно прийти к заключению, что в Rails имеется код, позволяющий ей подключаться к каждому из этих веб-серверов. В ранних выпусках Rails так оно и было, но начиная с Rails 2.3 эта сборка была поручена gem-пакету по имени Rack.

Итак, Rails интегрируется с Rack, Rack интегрируется (к примеру) с Passenger, а Passenger интегрируется с Apache httpd.

Хотя в целом эта интеграция не видна и осуществляется для вас при запуске команды `rails server`, вам предоставляется файл config.ru, позволяющий напрямую запустить ваше приложение под управлением Rack:

rails40/depot_u/config.ru

```
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment', __FILE__)
run Rails.application
```

Этот файл можно использовать для запуска вашего Rails-сервера с помощью следующей команды:

```
rackup
```

Запуск вашего сервера таким способом является полным эквивалентом запуска команды `rails server`. Чтобы продемонстрировать широкие возможности работы с использованием только лишь пакета Rack как такового, давайте начнем с использования чистого Rack-приложения:

rails40/depot_u/app/store.rb

```
require 'builder'
require 'active_record'
ActiveRecord::Base.establish_connection(
    adapter: 'sqlite3',
    database: 'db/development.sqlite3')
class Product < ActiveRecord::Base
end
class StoreApp
    def call(env)
        x = Builder::XmlMarkup.new :indent=>2
        x.declare! :DOCTYPE, :html
        x.html do
            x.head do
                x.title 'Pragmatic Bookshelf'
            end
            x.body do
                x.h1 'Pragmatic Bookshelf'
                Product.all.each do |product|
                    x.h2 product.title
                    x << " #{product.description}\n"
                    x.p product.price
                end
            end
        end

        response = Rack::Response.new(x.target!)
        response['Content-Type'] = 'text/html'
        response.finish
    end
end
```

В этом приложении мы воспользовались тем, что уже изучили ранее. Первое, что мы делаем, — это даем непосредственный запрос на использование `active_record` и `builder`. Затем мы устанавливаем подключение к нашей базе данных и определяем класс `Product` для наших товаров. Ничего этого не придется делать, если мы интегрируем это приложение с нашим Rails-приложением, но сейчас мы собираемся воспользоваться абсолютно отдельным приложением.

Затем следует само приложение. Это простой класс, в котором определяется всего лишь один метод по имени `call()`. Этот метод допускает передачу одного аргумента по имени `env`, который содержит информацию о запросе и этим приложением не используется.

В этом приложении для создания простого HTML, выводящего список товаров, используется библиотека Builder, а затем создается ответ, устанавливается тип содержимого и вызывается метод `finish()`.

Создав новый rackup-файл, мы можем запустить его в качестве автономного приложения:

rails40/depot_u/store.ru

```
require 'rubygems'
require 'bundler/setup'
```

```
require './app/store'

use Rack::ShowExceptions

map '/store' do
    run StoreApp.new
end
```

Сначала этот сценарий инициализирует Bundler, который сделает доступными правильные версии всех требуемых gem-пакетов. Затем он требует запустить приложение `store`.

Затем он извлекает один из стандартных *промежуточных* (middleware) классов, предоставляемых Rack, — этот класс форматирует данные обратного отслеживание стека, когда происходит что-нибудь нехорошее. Промежуточная программа в Rack похожа на фильтры Rails, она может как инспектировать запросы, так и настраивать создаваемые ответы.

Список промежуточных программ, предоставляемых Rack для Rails-приложений, можно увидеть, воспользовавшись командой `rake middleware`.

И в заключение создается отображение URI store на это приложение.

Это приложение можно запустить с помощью команды `rackup`:

```
rackup store.ru
```

По умолчанию команда `rackup` запускает серверы, используя вместо порта 3000 порт 9292. Порт можно выбрать с помощью ключа `-p`.

Посещение этой страницы с использованием вашего браузера приводит к очень простой визуализации перечня товаров, показанной на рис. 24.1.

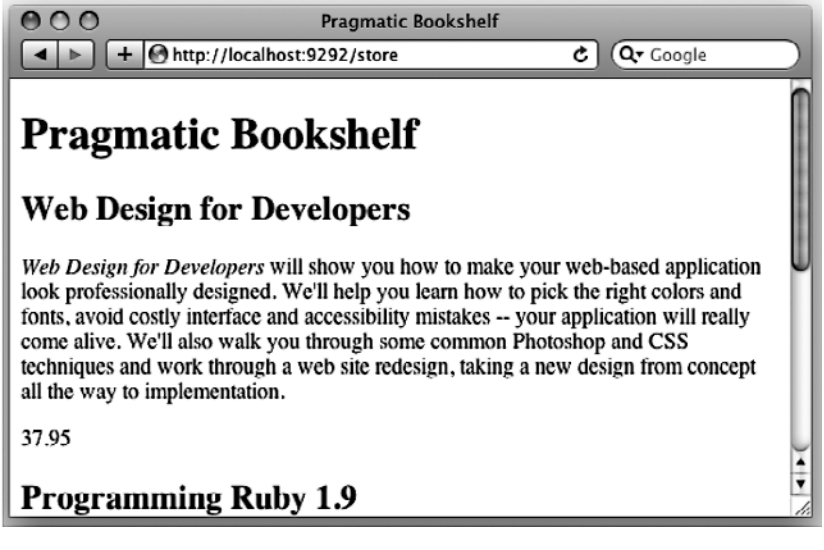


Рис. 24.1. Минимальное, но вполне работоспособное приложение для просмотра списка товаров

Недостаток этого натурального Rack-приложения по сравнению с Rails-приложением состоит в том, что у него значительно меньше всяких забот. А основное преимущество заключается в том, что благодаря этому появляется возможность обойти некоторые издержки Rails и обрабатывать больше запросов в секунду.

В большинстве случаев вам не захочется создавать совершенно автономное приложение, но захочется получить часть своего сайта, обойдя обработку контроллера. Это делается с помощью определения маршрута:

rails40/depot_u/config/routes.rb

```
▶require './app/store'
 Depot::Application.routes.draw do
▶    match 'catalog' => StoreApp.new, via: :all
     get 'admin' => 'admin#index'
     controller :sessions do
         get 'login' => :new
         post 'login' => :create
         delete 'logout' => :destroy
     end
     get "sessions/create"
     get "sessions/destroy"

     resources :users
     resources :products do
       get :who_bought, on: :member
     end

     scope '(:locale)' do
       resources :orders
       resources :line_items
       resources :carts
       root 'store#index', as: 'store', via: :all
     end
 end
```

Сервер является не единственным местом, где используются компоненты Rails. Мы завершим эту главу описанием средства, используемого для планирования выполнения задач.

24.5. Автоматизация задач с помощью Rake

Rake относится к тем программам, наличие которых часто считается в порядке вещей. Она используется для автоматизации задач, особенно тех, у которых могут быть зависимости. Задачи определяются файлом Rakefile, который находится в корневом каталоге вашего приложения.

Примером такой задачи может послужить `db:setup`. Чтобы посмотреть, какие подзадачи привлекаются, запустите Rake с ключами `--trace` и `--dry-run`:

```
$ rake --trace --dry-run db:setup
(in /home/rubys/work/depot)
** Invoke db:setup (first_time)
** Invoke db:create (first_time)
** Invoke db:load_config (first_time)
** Invoke rails_env (first_time)
** Execute (dry run) rails_env
** Execute (dry run) db:load_config
** Execute (dry run) db:create
** Invoke db:schema:load (first_time)
** Invoke environment (first_time)
** Execute (dry run) environment
** Execute (dry run) db:schema:load
** Invoke db:seed (first_time)
** Invoke db:abort_if_pending_migrations (first_time)
** Invoke environment
** Execute (dry run) db:abort_if_pending_migrations
** Execute (dry run) db:seed
** Execute (dry run) db:setup
```

Выполнение правильных шагов в правильной последовательности является жизненно важным условием для повторяющихся развертываний. Именно поэтому эта задача была использована в главе 16, в разделе «Загрузка базы данных».

Список доступных задач можно увидеть, воспользовавшись командой `rake --tasks`. Предоставляемые Rails задачи являются всего лишь стартовым набором, и у вас есть возможность создания дополнительных задач. Это можно сделать, создав новые файлы с кодом Ruby в каталоге lib/tasks.

В следующем примере создается задача резервного копирования рабочей базы данных:

rails40/depot_u/lib/tasks/db_backup.rake

```
namespace :db do

    desc "Создание резервной копии рабочей базы данных"
    task :backup => :environment do
        backup_dir = ENV['DIR'] || File.join(Rails.root, 'db', 'backup')

        source = File.join(Rails.root, 'db', "production.db")
        dest = File.join(backup_dir, "production.backup")

        makedirs backup_dir, :verbose => true

        require 'shellwords'
        sh "sqlite3 #{Shellwords.escape source} .dump > #{Shellwords.escape dest}"
    end
end
```

В первой строке ограничивается пространство имен. Мы помещаем задачу создания резервной копии в пространство имен db.

Во второй строке содержится описание. Это описание будет показано при выводе списка задач. Если вы еще раз запустите команду `rake --tasks`, то увидите, что новая задача включена туда наряду с другими задачами, предоставляемыми Rails.

В следующей строке содержится задача, а также любые зависимости, которые у нее могут быть. Зависимость от среды окружения является примерным эквивалентом загрузки всего, что предоставляет команда `rails console`.

Блок, передаваемый задаче, является стандартным кодом Ruby. В нашем примере мы определяем каталоги источника и приемника (где в качестве каталога-приемника по умолчанию будет использоваться каталог db/backup, но приемник можно изменить путем использования в командной строке аргумента `DIR`), затем переходим к созданию каталога backup (если это необходимо) и, наконец, выполняем команду `sqlite3 dump`.

24.6. Обзор Rails-зависимостей

Перечень Rails-зависимостей можно найти в файле Gemfile.lock. Наличие там некоторых имен будет вполне естественным, чего нельзя сказать об остальных именах. Чтобы помочь в данном исследовании, далее дается краткое описание, касающееся находящихся там имен.

Разумеется, по мере развития Rails этот список будет неизбежно изменяться. Но зная имя компонента, вы получите отправную точку для дальнейшего исследования. Хорошим способом узнать больше того, что может дать имя, будет переход на сайт RubyGems.org[1], ввод имени gem-пакета в поле поиска, выбор gem-пакета и щелчок либо на ссылке Documentation, либо на ссылке Homepage.

actionmailer
: Часть Rails, см. главу 13 «Задача 3: Отправка электронной почты»

actionpack
: Часть Rails, см. главу 20 «Action Dispatch и Action Controller»

activemodel
: Обеспечивает поддержку Active Record и Active Resource

activerecord
: Часть Rails, см. главу 19 «Active Record»

activesupport
: Часть Rails, см. раздел 23.2 «Библиотечная функция, использующая Active Support»

rails
: Контейнер для всей среды

railties
: Часть Rails, ссылки на дополнительную информацию по этой теме даны в разделе 25.4, «Поиск дополнительных модулей на сайте RailsPlugins.org»

ansi
: Включает декоративное оформление и стилизацию вывода на основе ANSI, используется gem-пакетом `turn`

[1] http://rubygems.org

arel
Реляционная алгебра; используется модулем Active Record

atomic
Предоставляет класс `Atomic`, гарантирующий обновление значений, содержащихся в atomic-канале

bcrypt-ruby
Хэш-алгоритм безопасности; используется модулем Active Model

builder
Простой способ создания XML-разметки; см. раздел 24.1 «Генерирование XML с помощью Builder»

capistrano
Предоставляет простой и легкий способ развертывания; см. раздел 16.2 «Шаг Л2: удаленное развертывание с помощью Capistrano»

coffee-script
Мост к компилятору JS CoffeeScript

erubis
Реализация ERb, который используется в Rails; см. раздел 24.2 «Генерирование HTML с помощью ERb»

execjs
Позволяет запускать код JavaScript из Ruby; используется библиотекой coffee-script highline IO для интерфейсов командной строки

hike
Ищет файлы в наборе путей; используется модулем `sprockets`

i18n
Поддерживает локализацию; см. раздел 15 «Задача К: локализация»

jquery-rails
Предоставляет jQuery и драйвер jQuery-ujs

jbuilder
Предоставляет простой DSL-язык для объявления JSON-структур, что лучше, чем манипулирование гигантскими хэш-структурами

json
Реализация JSON-спецификации в соответствии с RFC 4627

mail
Поддерживает работу с электронной почтой; см. главу 13 «Задача З: Отправка электронной почты»

mime-types
Определяет тип файла на основе расширения, используется модулем `mail`

multi-json
Предоставляет альтернативные JSON-интерфейсы

mysql
Рабочая база данных, поддерживаемая Active Record; см. «Использование MySQL в качестве базы данных» в главе 16

minitest
Предоставляет полноценный набор средств тестирования, поддерживающих TDD, BDD, имитацию и установку контрольных точек

net-scp
Обеспечивает безопасное копирование файлов

net-sftp
Обеспечивает безопасную передачу файлов

net-ssh
Обеспечивает безопасное подключение к удаленным серверам

net-ssh-gateway
Обеспечивает туннелирование подключения через SSH

polyglot
Содержит загрузчики языков клиентов

rack
Обеспечивает автоматизацию задач; см. раздел 24.5 «Автоматизация задач с помощью Rack»

rack-test
Обеспечивает тестирование API для маршрутизации; см. главу 20, раздел «Тестирование маршрутов»

rake
Обеспечивает автоматическое выполнение задач; см. раздел 24.5 «Автоматизация задач с помощью Rake»

sass
Предоставляет расширения для CSS3

sass-rails
Генератор и средство поддержки ресурсов для Sass

sprockets
Обеспечивает предварительную обработку и объединение исходных файлов JavaScript

thread_safe
Коллекция версий наиболее распространенных Ruby-классов с ориентацией на многопотоковое исполнение

tilt
Обеспечивает универсальный интерфейс для нескольких механизмов шаблонов Ruby; используется модулем `sprockets`

turn
Обеспечивает форматирование результатов `Test::Unit`

sqlite3
Разработочная база данных, поддерживаемая Active Record

thor
Среда создания сценариев, которая используется командой `rails`

treetop
Библиотека синтаксического анализа текста, используемая модулем `mail`

tzinfo
Обеспечивает поддержку часовых поясов

uglifier
Обеспечивает сжатие файлов JavaScript

Наши достижения

Мы исследовали небольшое количество зависимостей Rails, а затем показали, как можно управлять самими зависимостями, интегрированными с веб-сервером и в конечном итоге управляемыми из командной строки. Попутно мы наконец узнали, что файлы Rakefile, Gemfile и Gemfile.lock находятся в корневом каталоге нашего приложения.

Теперь, после углубления в Rails, следующей изучаемой темой будет расширение базового пакета Rails, получаемого после ее установки, за счет внешних дополнительных модулей и рассмотрение этих модулей.

25 Дополнительные модули Rails

Основные темы:

- добавление к вашему приложению новых классов;
- добавление нового языка шаблонов.

С самого начала этой книги мы постоянно вели разговор о превалировании соглашения над конфигурацией, благодаря чему в Rails почти для всего имеются практичные установки по умолчанию. И совсем недавно в этой книге мы описывали Rails в понятиях исходных gem-пакетов, которые вы получаете при установке Rails. Теперь настало время сложить эти две мысли вместе и показать, что исходный набор gem-пакетов, предоставляемый Rails, является целесообразным набором умолчаний — группой gem-пакетов, которую можно дополнять и изменять.

При работе с Rails gem-пакеты являются основным способом подключения новых функциональных возможностей. Вместо абстрактного описания мы выберем несколько дополнительных модулей и воспользуемся ими для иллюстрации различных аспектов установки дополнительных модулей и реализации их возможностей. А бонусом станет тот факт, что многие из этих модулей тут же начнут приносить пользу в вашей повседневной работе!

Начнем с простого дополнительного модуля, который способен приносить вам деньги.

25.1. Обработка кредитных карт с помощью Active Merchant

Выполняя шаг Ж1 в главе 12, мы говорили, что на время отложим вопрос обработки кредитных карт. Возможность взимать плату с покупателя, несомненно, весьма

важная часть приема заказа. Хотя эта функциональная возможность не встроена в Rails, существует gem-пакет, который ее предоставляет.

Вы уже видели, как управлять составом gem-пакетов, загружаемых вашим приложением: это делается путем редактирования вашего файла Gemfile. Поскольку в данной главе мы собираемся рассмотреть несколько таких gem-пакетов, давайте добавим сразу все пакеты, о которых пойдет речь. Фактически их можно добавить в любое место; мы же выбрали для этого конец файла.

rails40/depot_v/Gemfile

```
gem 'activemerchant', '~> 1.31'
gem 'haml', '~> 4.0'
gem 'kaminari', '~> 0.14'
```

Вы должны были заметить, что мы последовали установившейся практике и указали минимальную версию, а также дали конкретное указание верхней границы номера версии, поэтому данная демонстрация выберет версию, которая вряд ли будет содержать несовместимые изменения.

Что касается добавленных нами gem-пакетов, то каждый из них будет рассмотрен в отдельном разделе. Конкретно данный раздел будет посвящен Active Merchant[1].

Располагая всем необходимым для установки всех зависимостей, можно воспользоваться командой `bundle`.

```
depot> bundle install
```

В зависимости от используемой операционной системы и настроек может понадобиться запустить эту команду с root-правами.

В действительности команда `bundle` решает намного больше задач. Она проведет перекрестную проверку gem-зависимостей, найдет работоспособную конфигурацию, а также загрузит и установит все необходимые компоненты. Но сейчас это нас беспокоить не должно, поскольку нами добавляется только один компонент, и мы можем быть уверены, что он включен в gem-пакеты, устанавливаемые с помощью упаковщика.

После обновления или установки нового gem-пакета нужно выполнить еще одно последнее действие: перезапустить сервер. Хотя Rails совершает доброе дело, обнаруживая ваши последние изменения, внесенные в приложение, и поддерживая их, предсказать, что именно понадобится при добавлении или удалении всего gem-пакета, невозможно. В этом разделе сервер использоваться не будет, но вскоре он нам понадобится. Удостоверьтесь в том, что на сервере запущено приложение Depot.

Для демонстрации данной функциональной возможности мы создадим небольшой сценарий, который поместим в каталог script:

rails40/depot_v/script/creditcard.rb

```
credit_card = ActiveMerchant::Billing::CreditCard.new(
    number: '4111111111111111',
```

[1] http://www.activemerchant.org/

```
    month: '8',
    year: '2009',
    first_name: 'Tobias',
    last_name: 'Luetke',
    verification_value: '123'
)
puts "Действительна ли карта #{credit_card.number}? #{credit_card.valid?}"
```

На этот сценарий возлагается весьма скромная задача. Он создает экземпляр класса `ActiveMerchant::Billing::CreditCard`, а затем вызывает для этого объекта метод `valid?()`. Давайте его запустим.

```
$ rails runner script/creditcard.rb
Действительна ли карта 4111111111111111? false
```

От этого сценария требовалось только одно: чтобы он работал. Обратите внимание на то, что инструкции `require` не потребовались — чтобы функция была доступна в вашем приложении, достаточно было просто перечислить желаемые gem-пакеты в вашем файле Gemfile.

Теперь можно увидеть, как воспользоваться этой функцией в приложении Depot. Вы знаете, как с помощью миграции добавить поле к таблице `Orders`. Вы знаете, как добавить это поле к представлению. Вы знаете, как добавить проверочную логику к вашей модели, которая вызывает ранее использованный нами метод `valid?()`. Если вы зайдете на какой-нибудь торговый сайт, вы даже сможете понять, как авторизоваться — `authorize()` и как оформить платеж — `capture()`, хотя это требует от вас наличия имени пользователя и пароля на существующем торговом шлюзе. Как только все это будет установлено, вы знаете, как вызвать эту логику из вашего контроллера.

Вы только вдумайтесь в данный факт: все это стало возможным путем добавления всего лишь одной строки к вашему файлу Gemfile.

Как утверждалось в начале этой главы, добавление названий gem-пакетов к вашему файлу Gemfile является предпочтительным способом расширения Rails. Это влечет за собой ряд преимуществ: все ваши зависимости, отслеженные модулем Bundler, предварительно загружаются для немедленного использования вашим приложением и могут быть запакованы для их быстрого развертывания.

Рассмотренное дополнение было очень простым. Давайте перейдем к чему-нибудь более существенному, что предоставит явную альтернативу одному из gem-пакетов, от которого зависит работа Rails.

25.2. Украшение нашей разметки с помощью Haml

Давайте еще раз рассмотрим простое представление, которое мы использовали в приложении Depot, и на этот раз обратимся к представлению нашего каталога товаров:

rails40/depot_u/app/views/store/index.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>
<h1><%= t('.title_html') %></h1>

<% cache ['store', Product.latest] do %>
  <% @products.each do |product| %>
    <% cache ['entry', product] do %>
      <div class="entry">
          <%= image_tag(product.image_url) %>
          <h3><%= product.title %></h3>
          <%= sanitize(product.description) %>
          <div class="price_line">
              <span class="price"><%= number_to_currency(product.price) %></span>
              <%= button_to t('.add_html'), line_items_path(product_id: product),
                  remote: true %>
          </div>
      </div>
    <% end %>
  <% end %>
<% end %>
```

Этот код вполне справляется с поставленной перед ним задачей. В нем содержится основной HTML с разбросанными фрагментами кода Ruby, заключенными в разметку **<%** и **%>**. Знак равенства внутри этой разметки служит признаком того, что значение выражения должно быть конвертировано в HTML и выведено на экран.

Это не только адекватное решение поставленной задачи, но еще и все, что действительно необходимо для большого количества Rails-приложений. Вдобавок это еще и идеальное место для начала повествования в таких книгах, как эта. Мы предполагаем наличие у читателей некоторых познаний в HTML и делаем расчет на новичков в изучении Rails, а зачастую и в изучении языка Ruby. Последнее, что может потребоваться в данной ситуации, — это представить вам еще один новый язык.

А теперь, при прохождении этого поворота в вашей учебе, давайте исследуем новый язык, один из тех, который более тесно объединяет в себе создание разметки с применением Ruby-кода. Речь в данном случае идет о языке разметки для упрощенной генерации HTML — HTML Abstraction Markup Language (Haml).

Для начала давайте удалим файл, который мы только что рассматривали:

```
$ del app\views\store\index.html.erb
```

На его месте создадим новый файл:

rails40/depot_v/app/views/store/index.html.haml

```
- if notice
  %p#notice= notice

%h1= t('.title_html')
```

```
- cache ['store', Product.latest] do
  - @products.each do |product|
    - cache ['entry', product] do
      .entry
        = image_tag(product.image_url)
        %h3= product.title
        = sanitize(product.description)
        .price_line
          %span.price= number_to_currency(product.price)
          = button_to t('.add_html'), line_items_path(product_id: product),
            remote: true
```

Обратите внимание, у этого файла новое расширение: .html.haml. Это свидетельствует о том, что данный файл относится не к шаблону ERB, а к шаблону Haml.

В первую очередь вы должны заметить, что этот файл существенно меньше по размеру. А вот краткий обзор всего, что в нем происходит в зависимости от того символа, который стоит первым в каждой строке.

- Тире служат индикаторами инструкции Ruby, не производящей никакого вывода.
- Знаки процента (%) служат индикаторами элемента HTML.
- Знаки равенства (=) служат индикаторами Ruby-выражений, производящих отображаемый вывод. Они могут либо использоваться отдельно в строках, либо завершаться HTML-элементами.
- Символы точек (.) и решеток (#) могут использоваться для определения, соответственно, атрибутов `class` и `id`. Они могут использоваться в сочетании со знаком процента или сами по себе. Когда они используются сами по себе, предполагается использование элемента `div`.
- Запятая в конце строки, содержащей выражение, предполагает его продолжение. В предыдущем примере вызов метода `button_to()` растянут на две строки.

Важно отметить, что отступы играют в Haml весьма существенную роль. Возврат к тому же уровню отступа закрывает текущую открытую инструкцию `if`, цикл или тег. В данном примере абзац закрывается перед `h1`, `h1` закрывается перед первым элементом `div`, элементы `div` вложены друг в друга, и в первом из них содержится элемент `h3`, а во втором содержится как элемент `span`, так и вызов метода `button_to()`.

Еще можно заметить, что здесь доступны все знакомые вам методы-помощники, такие как `t()`, `image_tag()` и `button_to()`. В каждом значимом направлении Haml интегрирован в ваше приложение точно так же, как и ERB. Эти две технологии можно свободно смешивать и подменять одну другой: можно иметь ряд шаблонов, использующих ERB, и ряд шаблонов, использующих Haml.

Поскольку вы уже установили gem-пакет Haml, то, по сути, больше ничего не нужно делать. Чтобы посмотреть этот код в действии, нужно лишь посетить ваш каталог товаров. То, что вы увидите, должно соответствовать изображению, показанному на рис. 25.1.

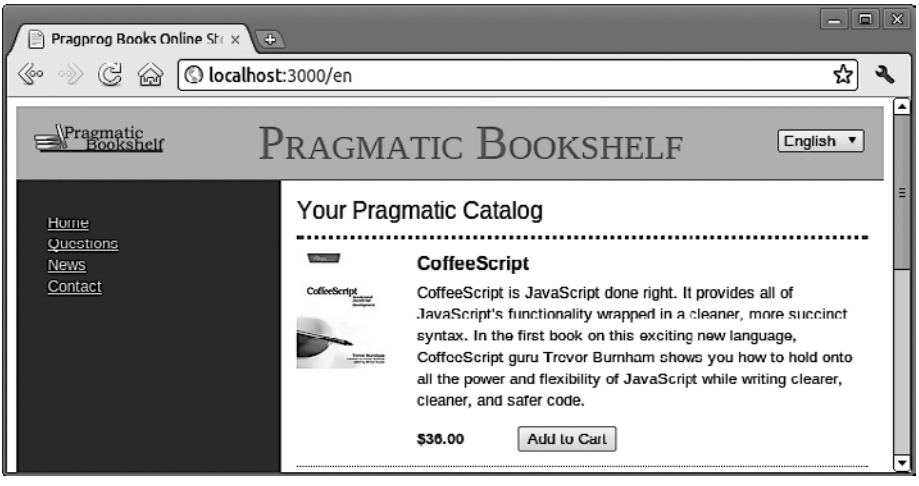


Рис. 25.1. Каталог товаров, выведенный с помощью Haml

Если картинка ничем не отличается от прежней, так это потому, что она и должна выглядеть *точно так же*, как и прежде. И это, если призадуматься, просто замечательно, поскольку макет приложения продолжает реализовываться в виде ERB-шаблона, а перечень как таковой реализован с помощью Haml-шаблона. Несмотря на это, все совмещается очень легко и без проблем.

Хотя это, несомненно, более глубокий уровень интеграции, чем простое добавление задачи или помощника, но это лишь добавление. Далее давайте исследуем дополнительный модуль, изменяющий основной объект, входящий в Rails.

25.3 Разбиение на страницы

На данный момент у нас есть несколько товаров, несколько корзин в любой момент времени и несколько товарных позиций в каждой корзине или в каждом заказе. Но у нас может быть практически неограниченное количество заказов, и мы надеемся, что их будет много, настолько много, что вывод всех их на странице заказов быстро сделает ее слишком большой. Выход можно найти в применении дополнительного модуля `kaminari`. Этот модуль расширяет Rails, предоставляя весьма полезную функцию.

Теперь сгенерируем тестовые данные. Мы можем повторно щелкать на имеющихся у нас кнопках, но лучше предоставить это компьютеру. Это будут не тестовые данные как таковые, а просто результаты неких однократных действий, которые потом можно будет выбросить. Давайте создадим файл в каталоге сценариев script.

rails40/depot_v/script/load_orders.rb

```
Order.transaction do
  (1..100).each do |i|
```

```
    Order.create(name: "Customer #{i}", address: "#{i} Main Street",
    email: "customer-#{i}@example.com", pay_type: "Check")
  end
end
```

Этот код создаст сто заказов без товарных позиций. Если нужно, вы можете изменить сценарий, добавив товарные позиции. Учтите, что наш код проделывает всю эту работу в одной транзакции. Это не является каким-то строгим требованием для данной деятельности, но зато существенно ускоряет обработку.

Обратите внимание на отсутствие каких-либо инструкций `require` или инициализаций для открытия или закрытия базы данных. Мы позволим Rails позаботиться об этом за нас:

```
rails runner script/load_orders.rb
```

Теперь, когда все уже установлено, мы готовы внести необходимые изменения в наше приложение. Сначала мы изменим наш контроллер для вызова метода `paginate()`, передав ему страницу и указав порядок, в котором нужно отобразить результаты:

rails40/depot_v/app/controllers/orders_controller.rb

```
def index
▶   @orders = Order.order('created_at desc').page(params[:page])
End
```

Затем мы добавим ссылки в нижнюю часть нашего представления перечня заказов:

rails40/depot_v/app/views/orders/index.html.erb

```
<h1>Listing orders</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Address</th>
      <th>Email</th>
      <th>Pay type</th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>

  <tbody>
      <% @orders.each do |order| %>
           <tr>
             <td><%= order.name %></td>
             <td><%= order.address %></td>
             <td><%= order.email %></td>
             <td><%= order.pay_type %></td>
             <td><%= link_to 'Show', order %></td>
```

```
            <td><%= link_to 'Edit', edit_order_path(order) %></td>
            <td><%= link_to 'Destroy', order, method: :delete,
                          data: { confirm: 'Are you sure?' } %></td>
          </tr>
      <% end %>
   </tbody>
</table>

<br>

<%= link_to 'New Order', new_order_path %>
  <p><%= paginate @orders %></p>
```

Ну вот, собственно, и все! По умолчанию на каждой странице будут показано по тридцать записей, а ссылки выводятся только при наличии более чем одной страницы заказов. Контроллер определяет количество отображаемых на странице заказов, используя аргумент `:per_page` (рис. 25.2).

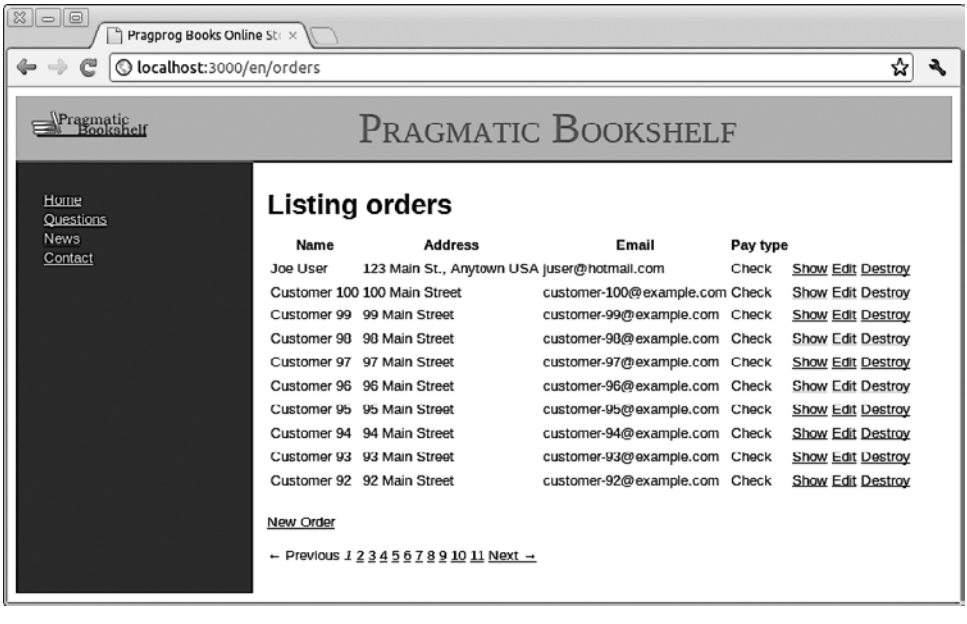


Рис. 25.2. Показ десяти заказов из более чем ста

Наши достижения

Несмотря на то что в данной главе были рассмотрены несколько дополнительных модулей, ее целью было представление вам некоторых возможностей, присущих этим модулям, а не подробное рассмотрение какого-нибудь конкретного модуля.

Если мы включили gem-пакеты, встречавшиеся в предыдущих главах, значит, мы видели дополнительные модули, которые просто добавляли новые свойства (Active `Merchant` и `Capistrano`) и некоторые методы к объектам модели (`kaminari`), а также новые языки создания шаблонов (Haml) и даже интерфейс для новой базы данных (mysql).

Если призадуматься, то нет ничего такого, на что бы не были способны дополнительные модули.

25.4. Поиск дополнительных модулей на сайте RailsPlugins.org

На данный момент мы рассмотрели три дополнительных модуля. А вот еще несколько мест для исследований, сгруппированных по категориям.

- Некоторые дополнительные модули, реализующие поведение, которое ранее принадлежало ядру Rails, но со временем было убрано оттуда. Например, вместо jQuery в прежних версиях Rails одной из библиотек, поддерживаемых по умолчанию, была Prototype. Она была перемещена в дополнительный модуль по имени prototype-rails[1]. Другие библиотеки, такие как `acts_as_tree`[2], совершенствовались в качестве дополнительного модуля. А есть и такие библиотеки, вроде `rails_xss`[3], которые обеспечивают обратную поддержку основных функциональных возможностей из будущих версий Rails с целью помочь с переходом на эти версии.
- Некоторые дополнительные модули фактически реализуют существенную долю общей логики приложения и даже пользовательский интерфейс. Дополнительные модули devise[4] и authlogic[5] реализуют аутентификацию пользователя и управление сессиями. Мы занимались реализацией этих функций в приложении Depot самостоятельно, но это как раз то, чего делать не рекомендуется. Мы считаем, что за лень нужно платить: если кто-то другой уже написал дополнительный модуль для функции, которую нужно реализовать, то все сэкономленное за счет этого время нужно потратить на совершенствование своего приложения.
- Некоторые дополнительные модули заменяют существенные части Rails. Например, datamapper[6] заменяет Active Record. Сочетание модулей cucumber[7],

[1] https://github.com/rails/prototype-rails#readme
[2] https://github.com/rails/acts_as_tree#readme
[3] https://github.com/rails/rails_xss
[4] https://github.com/plataformatec/devise#readme
[5] https://github.com/binarylogic/authlogic#readme
[6] http://datamapper.org/
[7] http://cukes.info/

rspec[1] и webrat[2] может использоваться по отдельности или вместе для замены сценариев тестов простыми тестовыми историями, спецификациями и имитациями браузера.

- Модули airbrake[3] и exception_notification[4] помогут вам отслеживать ошибки на ваших серверах развертывания.

Разумеется, это всего лишь малая часть набора доступных дополнительных модулей. И их список постоянно растет: к моменту, когда вы будете читать эти строки, их несомненно будет еще больше.

И наконец, вполне очевидно, что вы сами можете создавать свои собственные дополнительные модули. Хотя подобное занятие в данной книге не рассматривается, вы можете извлечь массу полезной информации из руководства Rails Guides[5] и документации[6].

[1] http://rspec.info/
[2] https://github.com/brynary/webrat#readme
[3] https://airbrakeapp.com/pages/home
[4] https://github.com/rails/exception_notification#readme
[5] http://guides.rubyonrails.org/plugins.html
[6] http://api.rubyonrails.org/classes/Rails/Railtie.html

26 Куда двигаться дальше

Основные темы:

- обзор концепций Rails: модели, представления, контроллеры, конфигурации, тестирования и развертывания;
- ссылки на места для дальнейшего изучения.

Мы вас поздравляем! Вместе мы усвоили большой объем базового материала.

В части I вы установили Rails, проверили установку путем использования простого приложения, продемонстрировавшего архитектуру Rails и познакомившего вас (или, возможно, повторно познакомившего) с языком Ruby.

В части II вы пошагово создали приложение, попутно нарастили объем контрольных примеров и в конечном итоге развернули это приложение с помощью Capistrano. Мы разработали это приложение для того, чтобы коснуться всех аспектов Rails, о которых должен знать каждый разработчик.

В отличие от частей I и II этой книги, каждая из которых служила только одной цели, часть III играет двойную роль.

Для некоторых из вас часть III методически заполняет пробелы и охватывает вполне достаточный объем информации для выполнения реальной работы. А для других ее материал станет первым шагом намного более длительного путешествия.

Для многих из вас реальную ценность в той или иной степени имеют оба этих положения. Твердое знание основ потребуется для дальнейшего изучения. И поэтому мы начали эту часть с главы, которая не только охватывала вопрос соглашений и конфигураций Rails, но также рассматривала вопрос создания документации.

Затем мы продолжили, посвятив по главе модели, представлениям и контроллеру, которые являются базовыми составляющими архитектуры Rails.

Были охвачены темы от связей базы данных и до REST-архитектуры и методов-помощников для создания HTML-форм.

В качестве важнейшего обслуживающего средства развертывания баз данных приложений были рассмотрены миграции.

И наконец, мы разбили Rails на части и изучили концепцию gem-пакетов с разных точек зрения: от использования отдельных Rails-компонентов до полного использования основ, на которых построена Rails и, в завершение, до построения и расширения среды, которая отвечала бы вашим запросам.

На данный момент у вас имеется весь необходимый материал и знание основ для более глубокого изучения интересующих вас областей или тех вопросов, которые нужны для решения стоящих перед вами текущих проблем. Мы рекомендуем вам начать с посещения сайта Ruby on Rails[1] и изучения каждой ссылки, находящейся в верхней части главной страницы. Некоторые ссылки помогут быстро освежить в памяти тот материал, который представлен в данной книге, но вы также можете найти там массу ссылок на текущую информацию о том, как сообщать о проблемах, получать дополнительные сведения и быть в курсе всего нового.

Кроме этого, пожалуйста, продолжайте вносить свой склад в wiki и форумы, упоминавшиеся во введении к данной книге.

У издательства Pragmatic Bookshelf имеются и другие книги по темам, связанным с Ruby и Rails[2]. Существует также большое количество родственных категорий, выходящих за рамки Ruby и Rails, например практика гибкого программирования (Agile Practices); тестирование, разработка и облачные вычисления; а также инструменты, среды программирования и языки. Эти и другие категории можно найти по адресу: http://www.pragprog.com/categories.

Мы надеемся, что вы получили удовольствие от изучения Ruby on Rails, вполне сравнимое с тем удовольствием, которое мы получили при написании данной книги!

[1] http://rubyonrails.org/

[2] http://www.pragprog.com/categories/ruby_and_rails

Алфавитный указатель

М

О

П

Р

С

Сэм Руби, Дэйв Томас, Дэвид Хэнссон

Rails 4.
Гибкая разработка веб-приложений

Серия «Для профессионалов»

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Жданов*
Художник	*Л. Адуевская*
Корректор	*В. Ганчурина*
Верстка	*А. Шляго*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 26.02.14. Формат 70x100/16. Усл. п. л. 36,120. Тираж 1500. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

ПИТЕР®

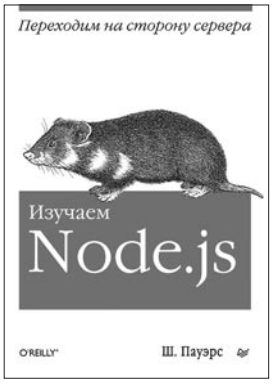
Б. Хоган, К. Уоррен, М. Уэбер,
К. Джонсон, А. Годин

КНИГА ВЕБ-ПРОГРАММИСТА: СЕКРЕТЫ ПРОФЕССИОНАЛЬНОЙ РАЗРАБОТКИ ВЕБ-САЙТОВ

Эта книга предлагает широкий спектр передовых методов веб-разработки: от проектирования пользовательского интерфейса до тестирования проекта и оптимизации веб-хостинга. Как внедрить на сайт анимацию, которая работает на мобильных устройствах без установки специальных плагинов? Как использовать «резиновую» верстку, которая корректно отображается не только на настольных ПК с различными разрешениями экрана, но и на мобильных устройствах? Как использовать фреймворки JavaScript – Backbone и Knockout – для разработки пользовательских интерфейсов? Как современные инструменты веб-разработчика, такие как CoffeeScript и Sass, помогут в оптимизации кода? Как провести кроссбраузерное тестирование кода? Как планировать процесс разработки сайта с помощью инструмента Git? Ответы на эти и многие другие вопросы вы найдете в этой книге. Не важно, являетесь вы начинающим веб-программистом или уже имеете некоторый опыт разработки веб-приложений, это издание поможет вам освоить множество новых методов, приемов и подходов.

ПИТЕР®

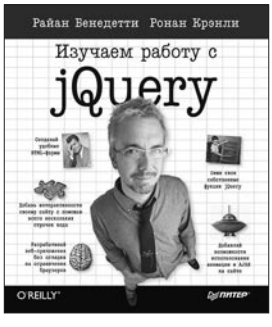
Ш. Пауэрс
ИЗУЧАЕМ NODE.JS



Node.js является серверной технологией, которая основана на разработанном компанией Google JavaScript-движке V8. Это прекрасно масштабируемая система, поддерживающая не программные потоки или отдельные процессы, а асинхронный ввод-вывод, управляемый событиями. Она идеально подходит для веб-приложений, которые не выполняют сложных вычислений, но к которым происходят частые обращения. По целям использования Node сходен с фреймворками Twisted на языке Python и EventMachine на Ruby. В отличие от большинства программ JavaScript, этот фреймворк исполняется не в браузере клиента, а на стороне сервера. С помощью этого практического руководства вы сможете быстро овладеть основами Node. Книга понравится всем, кто интересуется новыми технологиями, например веб-сокетами или платформами создания приложений. Эти темы раскрываются в ходе рассказа о том, как использовать Node в реальных приложениях.

ПИТЕР®

Р. Бенедетти, Р. Крэнли
ИЗУЧАЕМ РАБОТУ С JQUERY



Хотите добавить интерактивности своему интернет-сайту? Узнайте, как jQuery позволит вам создать целый набор скриптов, используя всего несколько строчек кода! С помощью этого издания вы максимально быстро научитесь работать с jQuery – этой удивительной библиотекой JavaScript, использование которой сегодня стало необходимостью для разработки современных веб-сайтов и RIA-приложений. jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, а также предоставляет богатые возможности по взаимодействию с AJAX. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию.

ПИТЕР®

Э. Османи
РАЗРАБОТКА BACKBONE.JS ПРИЛОЖЕНИЙ

Backbone – это javascript-библиотека для тяжелых фронтэнд javascript-приложений, таких, например, как gmail или twitter. В таких приложениях вся логика интерфейса ложится на браузер, что дает очень значительное преимущество в скорости интерфейса. Цель этой книги – стать удобным источником информации в помощь тем, кто разрабатывает реальные приложения с использованием Backbone. Издание охватывает теорию MVC и методы создания приложений с помощью моделей, представлений, коллекций и маршрутов библиотеки Backbone; модульную разработку ПО с помощью Backbone.js и AMD (посредством библиотеки RequireJS), решение таких типовых задач, как использование вложенных представлений, устранение проблем с маршрутизацией средствами Backbone и jQuery Mobile, а также многие другие вопросы.

ПИТЕР®

Ч. Фоулер
RAILS. СБОРНИК РЕЦЕПТОВ

Такие задачи, как аутентификация пользователей, распределение прав доступа, организация наиболее эффективного обмена данными с сервером баз данных и многое другое, требуют решения при создании практически любого веб-приложения. Эта книга позволяет разработчику не тратить время на поиск собственного решения, а обратиться к тем, кто уже решал схожие задачи и столкнулся с тонкостями Ruby on Rails в конкретных ситуациях. Издание будет полезно каждому, у кого есть начальные знания о Ruby on Rails, кто применяет эти знания на практике и кто не желает каждый раз «создавать велосипед» заново.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: voronej@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: nnovgorod@piter.com

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: samara@piter.com

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: minsk@piter.com

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
Тел./факс: (812) 703-73-73; e-mail: spb@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов
Тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
Тел./факс: (812) 703-73-73, доб. 6250; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225