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.

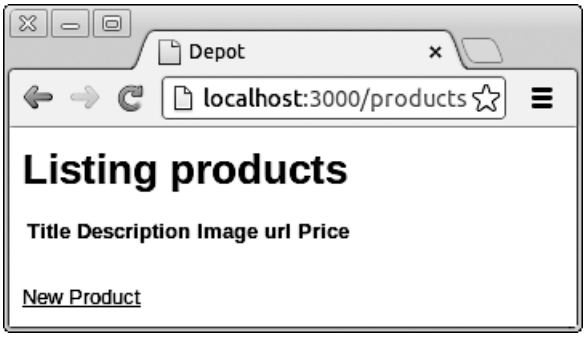

Рис. 6.1. Подключение к серверу

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).

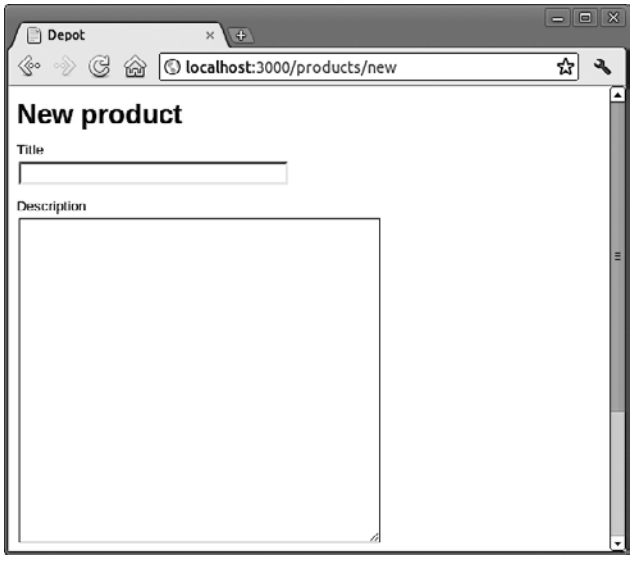

Рис. 6.2. Форма для добавления нового товара

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

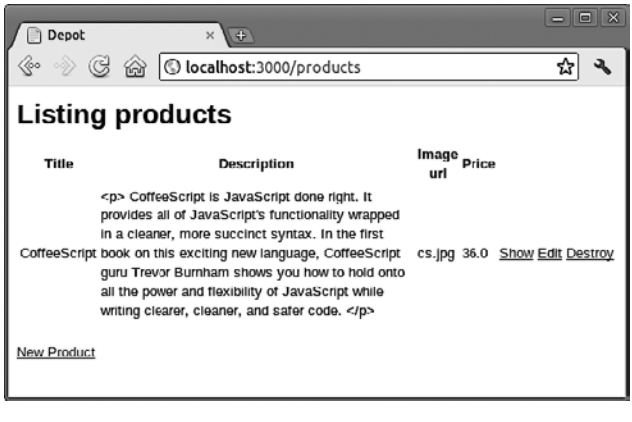

Рис. 6.4. Просмотр товара в том виде, в котором он появляется в базе данных

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/