helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

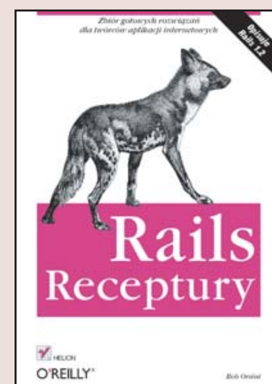
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Rails. Receptury



Autor: Rob Orsini
Tłumaczenie: Łukasz Piwko, Leszek Sagalara
ISBN: 978-83-246-1049-5
Tytuł oryginału: Rails Cookbook
Format: B5, stron: około 300

Zbiór gotowych rozwiązań dla twórców aplikacji internetowych

- Instalacja i uruchomienie środowiska Rails
- Przetwarzanie grafiki
- Korzystanie z technologii AJAX

Dynamiczny rozwój sieci sprawia, że tradycyjne programy są stopniowo wypierane przez aplikacje sieciowe dostępne z poziomu przeglądarki internetowej – wygodne, niezależne od systemu operacyjnego i łatwe w aktualizowaniu. Nadal jednak kluczowe znaczenie ma szybkość ich przygotowywania i modyfikowania. Dzięki zbiorom bibliotek zwanym „frameworks” proces tworzenia takich produktów znacznie się skrócił – umożliwia to programistom skoncentrowanie się na faktycznej funkcjonalności tworzonego narzędzia, ponieważ biblioteki te przejmują wiele typowych i wspólnych dla wszystkich aplikacji zadań. Wśród dostępnych w sieci narzędzi tego typu coraz większą popularność zyskuje Ruby on Rails, powoli stający się „ikoną” nurtu Web 2.0. Tworzone za jego pomocą systemy są zwarte i łatwe do skalowania, a ich kod źródłowy jest przejrzysty i czytelny.

„Rails. Receptury” to zestaw porad i rozwiązań problemów, przed którymi stają programiści stosujący ten zbiór bibliotek w swojej pracy. Omówione tu zagadnienia przydadzą się zarówno początkującym, jak i doświadczonym twórcom aplikacji sieciowych. Przeczytasz tu o instalowaniu, konfigurowaniu i uruchamianiu środowiska Rails, połączeniach z bazami danych za pomocą ActiveRecord, generowaniu kodu HTML, zabezpieczaniu programów i tworzeniu kontrolerów odpowiadających za funkcjonalność systemu. Dowiesz się, jak wdrażać aplikacje Rails i korzystać w nich z możliwości oferowanych przez mechanizmy AJAX.

- Instalacja i uruchomienie środowiska
- Komunikacja z bazami danych
- Wyświetlanie danych w przeglądarce
- Wykorzystywanie szablonów RHTML
- Generowanie kodu XML i RSS
- Przetwarzanie danych z formularzy
- Personalizacja narzędzi
- Korzystanie z JavaScript i AJAX
- Zabezpieczanie aplikacji Rails
- Optymalizacja aplikacji
- Wdrażanie i utrzymywanie systemów na serwerach
- Przetwarzanie obrazów

Skorzystaj ze sprawdzonych receptur i dołącz do twórców Web 2.0!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa 9

Wstęp 11

1. Zaczynamy 17

1.1. Społeczność Rails 18
1.2. Szukanie dokumentacji 20
1.3. Instalacja MySQL 21
1.4. Instalacja PostgreSQL 24
1.5. Instalacja Rails 26
1.6. Zmiana wersji Ruby i instalacja Rails w systemie OS X Tiger 28
1.7. Uruchamianie Rails w systemie OS X za pomocą Locomotive 30
1.8. Uruchamianie Rails w systemie Windows za pomocą Instant Rails 32
1.9. Aktualizacja Rails za pomocą RubyGems 34
1.10. Utworzenie repozytorium Subversion własnego projektu Rails 35

2. Praca w Rails 39

2.1. Tworzenie projektu Rails 39
2.2. Rozpoczynamy pracę z rusztowaniami 41
2.3. Przyspieszanie Rails za pomocą serwera Mongrel 45
2.4. Zwiększanie możliwości produkcyjnych w systemie Windows za pomocą Cygwin 47
2.5. Wzorce pluralizacyjne w Rails 48
2.6. Praca w Rails w systemie OS X za pomocą TextMate 51
2.7. Wieloplatformowe tworzenie aplikacji z RadRails 53
2.8. Instalacja i uruchamianie Edge Rails 54
2.9. Uwierzytelnianie bezhasłowe za pomocą SSH 56
2.10. Tworzenie dokumentacji RDoc do własnej aplikacji Rails 57
2.11. Tworzenie w pełni funkcjonalnej aplikacji CRUD za pomocą Streamlined 60

3. Active Record 65
3.1. Przygotowanie relacyjnej bazy danych do pracy z Rails 66
3.2. Programowe definiowanie schematu bazy danych 69
3.3. Zarządzanie bazą za pomocą migracji 71
3.4. Modelowanie bazy danych za pomocą Active Record 75
3.5. Kontrola relacji modelu z konsoli Rails 78
3.6. Dostęp do danych za pośrednictwem Active Record 81
3.7. Wyszukiwanie rekordów przy użyciu find 82
3.8. Iteracja na zestawie wyników Active Record 85
3.9. Wydajne pobieranie danych przy użyciu wczesnego wczytywania 88
3.10. Aktualizowanie obiektu Active Record 91
3.11. Wymuszanie spójności danych przy użyciu walidacji Active Record 95
3.12. Wykonywanie własnych zapytań za pomocą find_by_sql 98
3.13. Ochrona przed sytuacjami wyścigu za pomocą transakcji 102
3.14. Dodawanie do modelu możliwości sortowania przy użyciu acts_as_list 106
3.15. Wykonywanie zadań przy każdorazowym tworzeniu obiektów modelu 110
3.16. Modelowanie wątkowanego forum przy użyciu acts_as_nested_set 113
3.17. Tworzenie katalogu zagnieżdżonych tematów za pomocą acts_as_tree 117
3.18. Unikanie sytuacji wyścigu przy użyciu blokowania optymistycznego 120
3.19. Obsługa tabel z odziedziczoną konwencją nazewniczą 122
3.20. Automatyczny zapis sygnatur czasowych 123
3.21. Wydzielanie wspólnych relacji za pomocą powiązań polimorficznych 125
3.22. Połączenie modeli łączących i polimorfizmu w celu elastycznego modelowania danych 128

4. Action Controller 133
4.1. Dostęp do danych formularza z kontrolera 134
4.2. Zmiana strony domyślnej aplikacji 137
4.3. Zwiększanie przejrzystości kodu za pomocą nazwanych tras 138
4.4. Konfigurowanie własnych zachowań wyboru tras 139
4.5. Wyświetlanie komunikatów alarmowych za pomocą flash 141
4.6. Przedłużanie okresu trwania komunikatu flash 143
4.7. Podążanie za akcjami za pomocą przekierowań 144
4.8. Dynamiczne generowanie adresów URL 146
4.9. Kontrolowanie żądań za pomocą filtrów 147
4.10. Rejestracja zdarzeń z wykorzystaniem filtrów 149
4.11. Renderowanie akcji 151
4.12. Ograniczanie dostępu do metod kontrolera 153
4.13. Wysyłanie do przeglądarki plików lub strumieni danych 154
4.14. Przechowywanie informacji o sesji w bazie danych 155

4.15. Śledzenie informacji za pomocą sesji 157
4.16. Uwierzytelnianie z użyciem filtrów 160

5. Action View ... 165
5.1. Upraszczanie szablonów za pomocą metod pomocniczych 166
5.2. Wyświetlanie obszernych zbiorów danych za pomocą stronicowania 168
5.3. Tworzenie lepkich list wyboru 171
5.4. Edycja relacji wiele-do-wielu za pomocą list wielokrotnego wyboru 173
5.5. Wyodrębnianie wspólnego kodu prezentacyjnego za pomocą makiet 176
5.6. Definiowanie domyślnej makiety dla aplikacji 179
5.7. Generowanie kodu XML za pomocą szablonów Builder 180
5.8. Generowanie źródeł RSS z danych Active Record 181
5.9. Ponowne wykorzystanie elementów za pomocą podszablonów 184
5.10. Przetwarzanie pól wejściowych tworzonych dynamicznie 187
5.11. Dostosowywanie zachowań standardowych metod pomocniczych 190
5.12. Tworzenie formularzy WWW z wykorzystaniem metod pomocniczych 192
5.13. Format daty, czasu i waluty 196
5.14. Personalizacja profili użytkowników za pomocą grawatarów 198
5.15. Unikanie szkodliwego kodu w widokach za pomocą szablonów Liquid 200
5.16. Globalizacja aplikacji Rails 204

6. Projektowanie z REST ... 209
6.1. Tworzenie zagnieżdżonych zasobów 211
6.2. Obsługa innych formatów danych za pomocą typów MIME 216
6.3. Modelowanie relacji REST przy użyciu modeli łączących 218
6.4. Poszerzanie możliwości CRUD za pomocą zasobów REST 221
6.5. Korzystanie ze złożonych zagnieżdżonych zasobów REST 224
6.6. Tworzenie aplikacji Rails zgodnie z REST 227

7. Testowanie aplikacji ... 233
7.1. Centralizacja tworzenia obiektów wspólnych dla przypadków testowych 234
7.2. Tworzenie obiektów fixture dla przyporządkowań typu wiele-do-wielu 235
7.3. Importowanie danych testowych za pomocą obiektów fixture CSV 237
7.4. Dołączanie dynamicznych danych do obiektów fixture za pomocą ERb 239
7.5. Inicjalizacja testowej bazy danych 241
7.6. Interaktywne testowanie kontrolerów z poziomu konsoli Rails 243
7.7. Interpretacja danych z testu jednostkowego 244
7.8. Ładowanie danych testowych za pomocą obiektów fixture YAML 245
7.9. Monitorowanie testu pokrycia za pomocą zadania stats rake 248
7.10. Przeprowadzanie testów za pomocą narzędzia Rake 249
7.11. Przyspieszanie testów za pomocą transakcyjnych obiektów fixture 250

7.12. Sprawdzanie przez kontrolery za pomocą testów integracyjnych 252
7.13. Sprawdzanie kontrolerów za pomocą testów funkcjonalnych 255
7.14. Analiza zawartości pliku cookie 258
7.15. Testowanie tras własnych i nazwanych 260
7.16. Testowanie żądań HTTP za pomocą asercji związanych z odpowiedziami 262
7.17. Sprawdzanie modelu za pomocą testów jednostkowych 263
7.18. Jednostkowe testowanie walidacji modelu 266
7.19. Weryfikacja struktury DOM za pomocą asercji znaczników 268
7.20. Pisanie własnych asercji 271
7.21. Testowanie wysyłania plików 272
7.22. Modyfikowanie domyślnego zachowania klasy testującej przy użyciu makiet 275
7.23. Uzyskiwanie większej ilości danych dzięki ciągłemu uruchamianiu testów 277
7.24. Analiza pokrycia kodu za pomocą narzędzia Rcov 279

8. JavaScript i Ajax 283
8.1. Dodawanie do strony elementów DOM 284
8.2. Tworzenie własnego raportu metodą przeciągnij i upuść 287
8.3. Dynamiczne dodawanie elementów do listy wyboru 291
8.4. Kontrolowanie ilości tekstu wprowadzanego do pola textarea 294
8.5. Aktualizowanie elementów strony za pomocą szablonów RJS 297
8.6. Wstawianie kodu JavaScript do szablonów 299
8.7. Umożliwianie użytkownikom zmiany kolejności elementów listy 303
8.8. Autouzupełnianie pól tekstowych 306
8.9. Wyszukiwanie i dynamiczne wyróżnianie tekstu 308
8.10. Uatrakcyjnianie interfejsu użytkownika przy użyciu efektów wizualnych 312
8.11. Implementacja wyszukiwarki Live Search 316
8.12. Edycja pól w miejscu 319
8.13. Tworzenie paska postępu w Ajaksie 322

9. Action Mailer 325
9.1. Konfiguracja Rails do wysyłania poczty 326
9.2. Tworzenie klasy mailera za pomocą generatora mailer 327
9.3. Formatowanie wiadomości e-mail przy użyciu szablonów 329
9.4. Dołączanie plików do wiadomości e-mail 330
9.5. Wysyłanie wiadomości e-mail z aplikacji Rails 331
9.6. Odbieranie poczty za pomocą mechanizmu Action Mailer 332

10. Debugowanie aplikacji Rails 335
10.1. Eksploracja Rails z poziomu konsoli 336
10.2. Naprawianie błędów u źródła przy użyciu opcji Ruby -cw 338
10.3. Debugowanie aplikacji w czasie rzeczywistym
przy użyciu punktów wstrzymania 340

10.4. Zapisywanie komunikatów do dziennika przy użyciu wbudowanej w Rails klasy Logger 343
10.5. Zapisywanie danych debugowania w pliku 346
10.6. Wysyłanie informacji o wyjątkach pocztą elektroniczną 348
10.7. Wyświetlanie w widokach informacji o środowisku 352
10.8. Wyświetlanie zawartości obiektów przy użyciu wyjątków 353
10.9. Filtrowanie zawartości dziennika rozwojowego w czasie rzeczywistym 354
10.10. Debugowanie połączenia HTTP przy użyciu rozszerzeń Firefoksa 356
10.11. Debugowanie kodu JavaScript w czasie rzeczywistym przy użyciu powłoki JavaScript Shell 358
10.12. Interaktywne debugowanie kodu za pomocą narzędzia ruby-debug 361

11. Bezpieczeństwo aplikacji ... 365
11.1. Zabezpieczanie systemu za pomocą silnego hasła 365
11.2. Ochrona przed atakami typu SQL injection 367
11.3. Ochrona przed atakami typu cross-site scripting 369
11.4. Ograniczanie dostępu do publicznych metod i akcji 370
11.5. Zabezpieczanie serwera poprzez zamknięcie nieużywanych portów 372

12. Wydajność ... 375
12.1. Mierzenie wydajności serwera za pomocą narzędzia httperf 376
12.2. Testowanie wydajności fragmentów kodu aplikacji 378
12.3. Zwiększanie wydajności poprzez buforowanie statycznych stron 380
12.4. Okres ważności buforowanych stron 383
12.5. Mieszanie treści dynamicznej i statycznej przy użyciu mechanizmu buforowania fragmentów 385
12.6. Filtrowanie buforowanych stron za pomocą buforowania akcji 388
12.7. Skracanie czasu dostępu do danych za pomocą systemu memcached 389
12.8. Zwiększanie wydajności poprzez buforowanie treści po przetworzeniu 392

13. Hosting i wdrażanie ... 395
13.1. Hosting Rails na serwerze Apache 1.3 i przy użyciu mod_fastcgi 396
13.2. Zarządzanie wieloma procesami Mongrel przy użyciu mongrel_cluster 397
13.3. Hosting Rails na Apache 2.2, mod_proxy_balancer i Mongrel 400
13.4. Wdrażanie Rails przy użyciu Pound jako frontu dla Mongrel, Lighttpd i Apache 404
13.5. Dostosowywanie do własnych potrzeb rejestracji danych Pound za pomocą narzędzia cronolog 408
13.6. Konfiguracja Pound z obsługą SSL 411
13.7. Równoważenie obciążenia za pomocą prostego narzędzia o nazwie Pen 412
13.8. Wdrażanie projektu Rails przy użyciu Capistrano 414
13.9. Wdrażanie aplikacji w kilku środowiskach przy użyciu Capistrano 417

13.10. Wdrażanie przy użyciu Capistrano bez dostępu do systemu Subversion 419
13.11. Wdrażanie przy użyciu Capistrano i mongrel_cluster 421
13.12. Wyłączanie strony podczas prac konserwacyjnych 423
13.13. Pisanie własnych zadań Capistrano 426
13.14. Usuwanie pozostałości po sesjach 430

14. Rozszerzanie Rails za pomocą wtyczek ... 433
14.1. Znajdywanie wtyczek 434
14.2. Instalowanie wtyczek 435
14.3. Manipulacja wersjami rekordów za pomocą wtyczki acts_as_versioned 437
14.4. Uwierzytelnianie przy użyciu wtyczki acts_as_authenticated 440
14.5. Upraszczanie znakowania za pomocą wtyczki acts_as_taggable 443
14.6. Rozszerzanie Active Record przy użyciu wtyczki acts_as 448
14.7. Dodawanie metod pomocniczych widoków do Rails jako wtyczek 453
14.8. Wysyłanie plików na serwer przy użyciu wtyczki file_column 455
14.9. Wysyłanie plików na serwer przy użyciu wtyczki acts_as_attachment 457
14.10. Wyłączanie rekordów zamiast ich usuwania za pomocą wtyczki acts_as_paranoid 461
14.11. Dodawanie bardziej wyrafinowanego mechanizmu uwierzytelniania przy użyciu wtyczki login_engine 463

15. Grafika ... 467
15.1. Instalowanie interfejsu RMagick do przetwarzania obrazów 467
15.2. Wysyłanie obrazów do bazy danych 471
15.3. Serwowanie obrazków bezpośrednio z bazy danych 475
15.4. Tworzenie miniatur za pomocą RMagick 476
15.5. Generowanie dokumentów PDF 479
15.6. Wizualna prezentacja danych przy użyciu Gruff 481
15.7. Tworzenie małych grafów przy użyciu biblioteki Sparklines 484

A Migracja do Rails 1.2 ... 487
Action Controller 487
Active Record 489
Action View 489

Skorowidz ... 491

ROZDZIAŁ 5.

Action View

5.0. Wprowadzenie

Action View służy jako warstwa prezentacyjna lub warstwa widoku wzorca MVC (ang. *model view controller*). Oznacza to, że jest to składnik odpowiedzialny za obsługę szczegółów związanych z wyglądem naszej aplikacji Rails. Żądania przychodzące kierowane są do kontrolerów, które z kolei generują szablony widoków. Szablony widoków mogą dynamicznie tworzyć dane wyjściowe do prezentacji w oparciu o struktury danych dostępne dla nich za pośrednictwem powiązanych z nimi kontrolerów. To właśnie dzięki takiej dynamicznej prezentacji Action View pomaga rozdzielić szczegóły prezentacyjne od właściwej logiki biznesowej naszej aplikacji.

Rails wyposażony jest w trzy różnego rodzaju systemy szablonów widoków. O tym, który mechanizm szablonu zostanie zastosowany dla danego żądania, decyduje rozszerzenie pliku generowanego szablonu. Te trzy systemy szablonów oraz rozszerzenia plików wywołujące ich wykonanie to: szablony ERb (**.rhtml*), szablony `Builder::XmlMarkup` (**.rxml*) oraz szablony JavaScriptGenerator lub RJS (**.rjs*).

Szablony ERb są najczęściej używane do generowania danych wyjściowych HTML aplikacji Rails; stosuje się je także do generowania wiadomości e-mail (choć tym zajmiemy się dopiero w rozdziale 9., „Action Mailer"). Składają się one z plików zakończonych rozszerzeniem *.rhtml*. Szablony ERb to połączenie HTML i zwykłego tekstu wraz ze specjalnymi znacznikami ERb, które osadzają w szablonach kod Ruby, np. `<% kod ruby %>`, `<%= łancuch wyjściowy %>` lub `<%- kod ruby (z usuwaniem pustych znaków) -%>`. Znak równości oznacza znacznik służący do wyprowadzenia łańcucha wynikowego jakiegoś wyrażenia Ruby. Znaczniki bez znaku równości oznaczają czysty kod Ruby i nie wytwarzają żadnych danych wyjściowych. Oto przykład prostego szablonu ERb tworzącego listę rozdziałów książki:

```
<ol>
  <% for chapter in @chapters %>
    <li><%= chapter.title %></li>
  <% end %>
<ol>
```

Nasze szablony mogą ponadto zawierać inne szablony podrzędne dzięki przesłaniu opcji `:file` do metody `render` w znacznikach wyjściowych ERb, np.:

```
<%= render :file => "shared/project_calendar %>
```

gdzie *project_calendar.rhtml* jest plikiem w katalogu współdzielonym wewnątrz głównego katalogu szablonów naszego projektu (*app/views*).

W tym rozdziale przedstawię kilka popularnych technik efektywnego korzystania z szablonów ERb. Ponadto pokażę sposób generowania dynamicznego kodu XML za pomocą szablonów `Builder::XmlMarkup`, np. w celu wygenerowania źródeł RSS. Warto zauważyć, że choć szablony RJS są składnikiem Action View, to wstrzymam się z ich omówieniem aż do rozdziału 8., „JavaScript i Ajax".

5.1. Upraszczanie szablonów za pomocą metod pomocniczych

Problem

Szablony widoków służą do prezentacji — powinny one zawierać kod HTML i minimalną ilość dodatkowej logiki w celu wyświetlania danych z modelu. Chcemy oczyścić nasze szablony widoków z logiki programu, która mogłaby przeszkadzać w prezentacji.

Rozwiązanie

Definiujemy metody w module pomocniczym o nazwie odpowiadającej kontrolerowi, którego widoki będą korzystać z tych metod. W tym przypadku tworzymy metody pomocnicze o nazwach `display_new_hires` i `last_updated` wewnątrz modułu o nazwie `IntranetHelper` (nazwanym tak od kontrolera `Intranet`).

app/helpers/intranet_helper.rb:

```
module IntranetHelper

  def display_new_hires
    hires = NewHire.find :all, :order => 'start_date desc', :limit => 3
    items = hires.collect do |h|
      content_tag("li",
        "<strong>#{h.first_name} #{h.last_name}</strong>" +
        " - #{h.position} (<i>#{h.start_date}</i>)")
    end
    return content_tag("b", "New Hires:"), content_tag("ul",items)
  end

  def last_updated(user)
    %{<hr /><br /><i>Ostatnia aktualizacja strony: #{Time.now}. Aktualizacji dokonał:
      #{user}</i>}
  end
end
```

Wewnątrz widoku `index` kontrolera `Intranet` możemy wywoływać nasze metody pomocnicze w taki sam sposób jak wszystkie inne metody systemowe.

app/views/intranet/index.rhtml:

```
<h2>Intranet - strona główna</h2>
<p>Dopasuj kapelusz do głowy -- październik 2004. Jak stwierdził Larry Wall,
informacja chce być wartościowa, a forma jej przedstawienia wpływa na jej wartość.
W wydawnictwie O'Reilly Media oferujemy wiele sposobów uzyskania informacji
technicznych. Tim O'Reilly mówi o tym w swoim kwartalnym liście dla O'Reilly
Catalog.</p>
<%= display_new_hires %>
<%= last_updated("Goli") %>
```

Omówienie

Metody pomocnicze są zaimplementowane w Rails w postaci modułów. Gdy Rails generuje kontroler, tworzy również moduł pomocniczy w katalogu *app/helpers* o nazwie odpowiadającej temu kontrolerowi. Domyślnie metody zdefiniowane w tym module dostępne są w widoku odpowiadającego mu kontrolera. Na rysunku 5.1 przedstawiono efekt wyjściowy widoku z użyciem metod pomocniczych `display_new_hires` i `last_updated`.

Rysunek 5.1. Efekt końcowy widoku pomocniczego display_new_hires

Jeśli chcemy współdzielić metody pomocnicze z innymi kontrolerami, musimy je wyraźnie zadeklarować w swoim kontrolerze. Jeżeli np. chcemy, aby metody z modułu `IntranetHelper` były dostępne dla widoku kontrolera `Store`, należy do metody `helper` kontrolera `Store` przesłać opcję `:intranet`:

```
class StoreController < ApplicationController

  helper :intranet

end
```

Teraz będzie ona szukać pliku o nazwie *helpers/intranet_helper.rb* i dołączy jego metody jako metody pomocnicze.

Możemy również udostępnić dla widoku metody kontrolera jako metody pomocnicze — przesyłając nazwę metody kontrolera do metody `helper_method`. W poniższym przykładzie `StoreController` umożliwia wywołanie w naszym widoku `<%= get_time %>` w celu wyświetlenia bieżącego czasu.

```
class StoreController < ApplicationController

  helper_method :get_time

  def get_time
    return Time.now
  end
end
```

Zobacz również

- Receptura 5.11, „Dostosowywanie zachowań standardowych metod pomocniczych”,
- Receptura 5.12, „Tworzenie formularzy WWW z wykorzystaniem metod pomocniczych”.

5.2. Wyświetlanie obszernych zbiorów danych za pomocą stronicowania

Problem

Wyświetlanie obszernych zbiorów danych w przeglądarce szybko może sprawić, że strona nie będzie się nadawać do wykorzystania, może nawet spowodować zawieszenie się przeglądarki. Patrząc od strony serwera, wczytywanie obszernych zbiorów danych w celu wyświetlenia jedynie kilku wierszy może mieć fatalny wpływ na wydajność. Chcemy zarządzać wyświetlaniem dużych zbiorów danych, stosując stronicowanie danych wyjściowych, czyli wyświetlanie podzbiorów wyjściowych na wielu stronach.

Rozwiązanie

Metoda pomocnicza `paginate` bardzo upraszcza konfigurację stronicowania. Aby przeprowadzić stronicowanie danych wyjściowych składających się z obszernej listy filmów, należy wywołać metodę `paginate` w kontrolerze `MoviesController` i przechować wyniki w dwóch zmiennych egzemplarza o nazwach `@movie_pages` i `@movies`:

app/controllers/movies_controller.rb:

```
class MoviesController < ApplicationController

  def list
    @movie_pages, @movies = paginate :movies,
                                     :order => 'year, title',
                                     :per_page => 10
  end
end
```

W naszym widoku przeprowadzamy iterację tablicy obiektów `Movie` przechowywanych w zmiennej egzemplarza `@movies`. Za pomocą obiektu `Paginator` w zmiennej `@movie_pages` utworzymy odnośniki do następnej i poprzedniej strony z wynikami. Do widoku dołączymy metodę `pagination_links` w celu wyświetlenia odnośników do pozostałych stron z wynikami.

app/views/movies/list.xhtml:

```
<table width="100%">
  <tr>
  <% for column in Movie.content_columns %>
    <th>
      <span style="font-size: x-large;"><%= column.human_name %></span>
    </th>
  <% end %>
  </tr>
  <% for movie in @movies %>
    <tr style="background: <%= cycle("#ccc","") %>;">
    <% for column in Movie.content_columns %>
    <td><%=h movie.send(column.name) %></td>
  <% end %>
  </tr>
  <% end %>
  <tr>
    <td colspan="<%= Movie.content_columns.length %>">
      <hr />
      <center>
        <%= link_to '[poprzednia]', { :page => @movie_pages.current.previous }
            if @movie_pages.current.previous %>
        <%= pagination_links(@movie_pages) %>
        <%= link_to '[następna]', { :page => @movie_pages.current.next }
            if @movie_pages.current.next %>
      </center>
    </td>
  </tr>
</table>
```

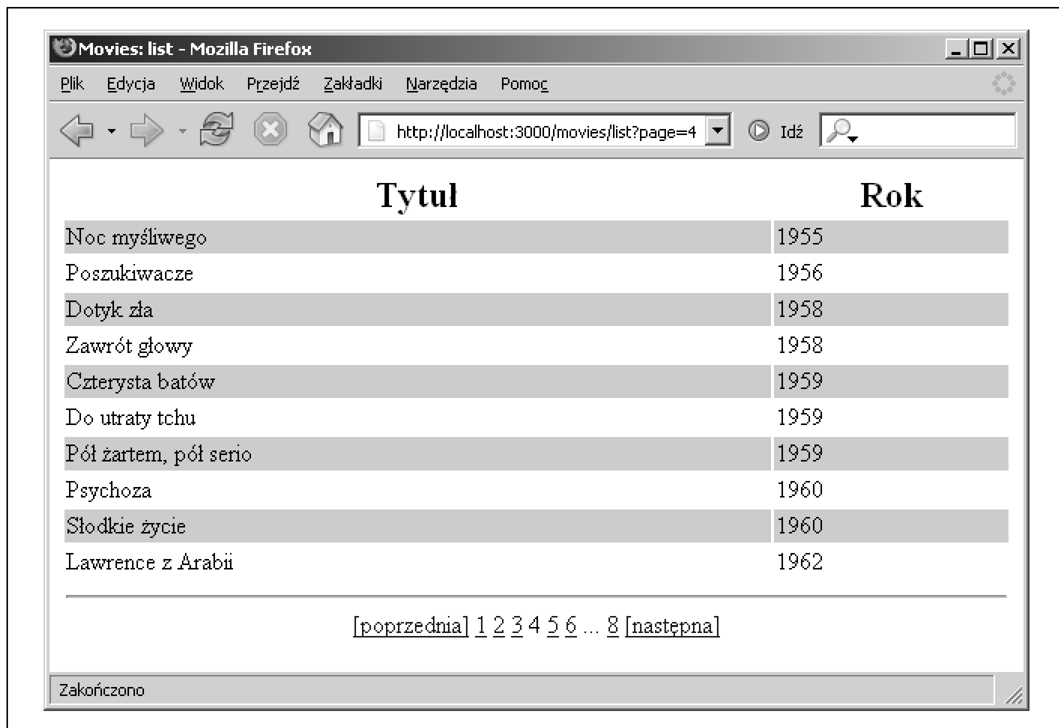
Omówienie

Stronicowanie to standardowa technika używana przy wyświetlaniu obszernych zbiorów wynikowych w WWW. Rails obsługuje ten często spotykany problem przez łączenie danych w mniejsze zbiory przy użyciu metody pomocniczej `paginate`.

Na rysunku 5.2 przedstawiono dane wyjściowe stronicowania na podstawie tego rozwiązania.

Wywołanie w kontrolerze metody `paginate` zwraca obiekt `Paginator`, a także tablicę obiektów reprezentujących początkowy podzbiór wyników. Bieżąca strona określana jest przez zawartość zmiennej `params['page']`. Jeśli ta zmienna nie występuje w obiekcie żądania, wówczas domyślnie przyjmowana jest pierwsza strona.

Opcje przesyłane do `paginate` określają obiekty modelu do pobrania oraz warunki zbioru wyników, który chcemy stronicować. W rozwiązaniu pierwszym argumentem przesłanym do `paginate` jest `:movies`, co nakazuje zwrócenie wszystkich obiektów filmów. Opcja `:order` określa ich kolejność. Opcja `:per_page` ustala maksymalną liczbę rekordów na każdej stronie z wynikami. Najczęściej wartość ta może zostać wybrana przez użytkownika. Aby sterować rozmiarem strony za pomocą parametru `params[:page_size]`, należy wprowadzić następujący kod:



Rysunek 5.2. Czwarta strona stronicowanej listy filmów

```
def list
  if params[:page_size] and params[:page_size].to_i > 0
    session[:page_size] = params[:page_size].to_i
  elsif session[:page_size].nil?
    session[:page_size] ||= 10
  end
  @movie_pages, @movies = paginate :movies,
                                   :order => 'year, title',
                                   :per_page => session[:page_size]
end
```

Za pomocą tego kodu adres URL *http://localhost:3000/movies/list?page=2&page__size=30* ustawi rozmiar strony dla tej sesji na 30.

Oprócz opcji `:order`, metoda `paginate` może korzystać ze wszystkich zwykłych opcji wyszukiwania (np. `:conditions`, `:joins`, `:include`, `:select`) oraz dodatkowo z kilku rzadziej stosowanych.

Zobacz również

- Dokumentacja API Rails dla `ActionController::Pagination`, *http://api.rubyonrails.org/classes/ActionController/Pagination.html*.

5.3. Tworzenie lepkich list wyboru

Problem

Skonfigurowaliśmy rusztowanie dla jednego z modeli i chcemy dodać listę wyboru, która dołącza informacje o powiązanym modelu do formularza edycji. Taka lista wyboru powinna pamiętać i wyświetlać wartość lub wartości wybrane w ostatnio wypełnianym formularzu.

Rozwiązanie

Mamy aplikację, która śledzi pewne zasoby i ich rodzaje. Poniższe definicje modeli ustanawiają relacje pomiędzy zasobami i ich rodzajami:

app/models/asset.rb:

```
class Asset < ActiveRecord::Base
  belongs_to :asset_type
end
```

app/models/asset_type.rb:

```
class AssetType < ActiveRecord::Base
  has_many :assets
end
```

Nasz widok będzie wymagać dostępu do wszystkich rodzajów zasobów w celu wyświetlenia listy wyboru. W kontrolerze pobieramy wszystkie obiekty `AssetType` i przechowujemy je w zmiennej egzemplarza o nazwie `@asset_types`:

app/controllers/assets_controller.rb:

```
class AssetsController < ApplicationController

  def edit
    @asset = Asset.find(params[:id])
    @asset_types = AssetType.find(:all)
  end

  def update
    @asset = Asset.find(params[:id])
    if @asset.update_attributes(params[:asset])
      flash[:notice] = 'Zasób został pomyślnie zaktualizowany.'
      redirect to :action => 'show', :id => @asset
    else
      render :action => 'edit'
    end
  end
end
```

W formularzu edycji należy utworzyć znacznik wyboru z atrybutem `name`, który przy wysyłaniu formularza będzie dodawał `asset_type_id` do tablicy asocjacyjnej `params`. Korzystając z `options_from_collection_for_select`, utworzymy opcje listy wyboru na podstawie zawartości `@asset_types`.

app/views/assets/edit.rhtml:

```
<h1>Edycja zasobu</h1>

<% form_tag :action => 'update', :id => @asset do %>
  <%= render :partial => 'form' %>

  <p>
  <select name="asset[asset_type_id]">
    <%= options_from_collection_for_select @asset_types, "id", "name",
                                           @asset.asset_type.id %>
  </select>
  </p>

  <%= submit_tag 'Edit' %>
<% end %>

<%= link_to 'Pokaż', :action => 'show', :id => @asset %> |
<%= link_to 'Wstecz', :action => 'list' %>
```

Omówienie

Rozwiązanie tworzy listę wyboru w widoku edycji zasobu, która zostaje zainicjowana wybranym poprzednio `asset_type`. Metoda `options_from_collection_for_select` przyjmuje cztery parametry: zbiór obiektów, nazwę pola elementu listy stanowiącego wartość opcji, nazwę pola elementu listy stanowiącego nazwę opcji oraz identyfikator rekordu elementu z listy, który powinien zostać wybrany domyślnie. Dlatego przesłanie `@asset_type.id` jako czwartego elementu sprawia, że poprzednio wybrany rodzaj zasobu staje się **lepki** (ang. *sticky*).

Podobnie jak wiele innych metod pomocniczych w Action View, `options_from_collection_for_select` jest po prostu otoczką bardziej ogólnej metody; w tym przypadku `options_for_select`. Jest ona zaimplementowana wewnętrznie jako:

```
def options_from_collection_for_select(collection,
                                       value_method,
                                       text_method,
                                       selected_value = nil)
  options_for_select(
    collection.inject([]) do |options, object|
      options << [ object.send(text_method), object.send(value_method) ]
    end,
    selected_value
  )
end
```

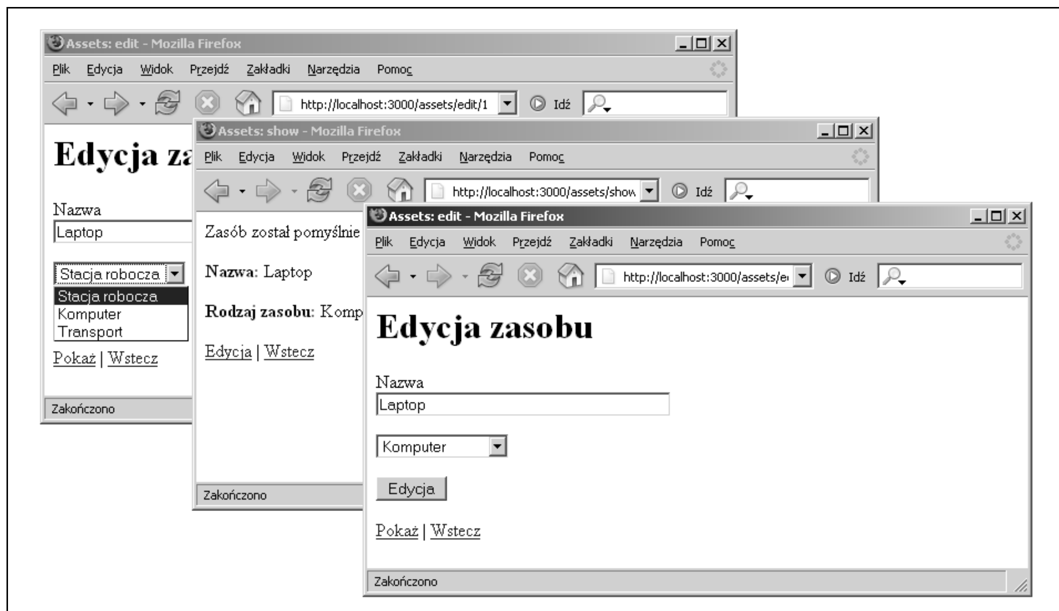
Dodajmy do widoku poniższy kod w celu wyświetlenia bieżącego rodzaju zasobu:

```
<p>
  <b>Rodzaj zasobu:</b> <%=h @asset.asset_type.name %>
</p>
```

Na rysunku 5.3 przedstawiono wynikową listę wyboru z tego rozwiązania.

Zobacz również

- Receptura 5.4, „Edycja relacji wiele-do-wielu za pomocą list wielokrotnego wyboru".



Rysunek 5.3. Lepka lista wyboru w działaniu

5.4. Edycja relacji wiele-do-wielu za pomocą list wielokrotnego wyboru

Problem

Mamy dwa modele, między którymi zachodzi relacja typu wiele-do-wielu. Chcemy w widoku edycji jednego modelu utworzyć listę wyboru, która umożliwiałaby tworzenie powiązań z jednym lub większą liczba rekordów z drugiego modelu.

Rozwiązanie

Część systemu uwierzytelniania w naszej aplikacji umożliwia przypisanie użytkowników do jednej roli lub większej liczby ról o zdefiniowanych prawach dostępu. Relacja wiele-do-wielu między użytkownikami i rolami skonfigurowana jest za pomocą następujących definicji klas:

app/models/user.rb:

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

app/models/role.rb:

```
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users
end
```

W akcji `edit` kontrolera `User` dodajemy zmienną egzemplarza o nazwie `@selected_roles` i zapełniamy ją wszystkimi obiektami `Role`. Definiujemy metodę prywatną o nazwie `handle_roles_users` w celu obsługi aktualizacji obiektu `User` o powiązane role z tablicy asocjacyjnej `params`.

app/controllers/users_controller.rb:

```
class UsersController < ApplicationController

  def edit
    @user = User.find(params[:id])
    @roles = {}
    Role.find(:all).collect {|r| @roles[r.name] = r.id }
  end

  def update
    @user = User.find(params[:id])
    handle_roles_users
    if @user.update_attributes(params[:user])
      flash[:notice] = 'Użytkownik został pomyślnie zaktualizowany.'
      redirect_to :action => 'show', :id => @user
    else
      render :action => 'edit'
    end
  end

  private
    def handle_roles_users
      if params['role_ids']
        @user.roles.clear
        roles = params['role ids'].map { |id| Role.find(id) }
        @user.roles << roles
      end
    end
end
```

W widoku edycji użytkownika za pomocą `options_for_select` tworzymy listę wielokrotnego wyboru opcji w celu wygenerowania opcji z obiektów w zmiennej egzemplarza `@roles`. Tworzymy listę istniejących powiązań między rolami i przesyłamy ją jako drugi parametr.

app/views/users/edit.xhtml:

```
<h1>Edycja użytkownika</h1>

<% form_tag :action => 'update', :id => @user do %>
  <%= render :partial => 'form' %>

<p>
<select id="role_ids" name="role_ids[]" multiple="multiple">
  <%= options_for_select(@roles, @user.roles.collect {|d| d.id }) %>
</select>
</p>

  <%= submit_tag 'Edit' %>
<% end %>

<%= link_to 'Pokaż', :action => 'show', :id => @user %> |
<%= link_to 'Wstecz', :action => 'list' %>
```

Aby wyświetlić role przypisane do każdego użytkownika, łączymy je w postaci listy rozdzielonej przecinkami w widoku akcji `show`:

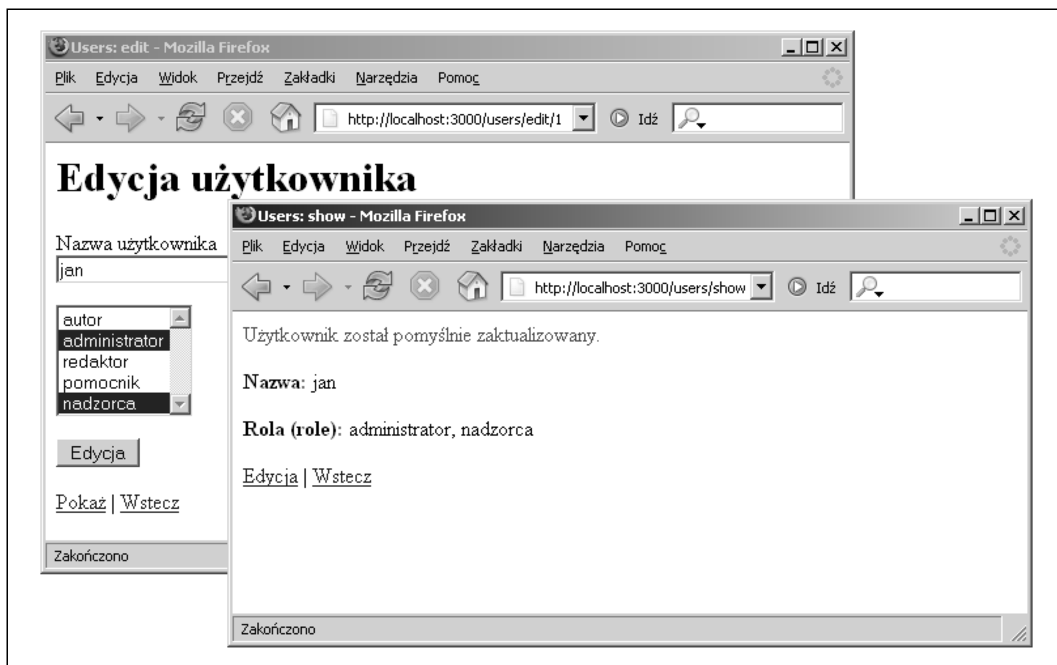
app/views/users/show.rhtml:

```
<% for column in User.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h @user.send(column.name) %>
</p>
<% end %>
<p>
  <b>Rola (role):</b> <%=h @user.roles.collect {|r| r.name}.join(', ') %>
</p>

<%= link_to 'Edycja', :action => 'edit', :id => @user %> |
<%= link_to 'Wstecz', :action => 'list' %>
```

Omówienie

W Rails istnieją pewne metody umożliwiające przekształcanie zbioru obiektów w listy wyboru. Przykładowo metody `select` lub `select_tag` kontrolera `ActionView::Helpers::FormOptionsHelper` generują cały znacznik wyboru HTML w oparciu o zbiór opcji. Większość z tych metod pomocniczych generuje tylko listę opcji.

Na rysunku 5.4 możemy zobaczyć dwie wybrane dla użytkownika role oraz sposób ich przedstawienia w widoku akcji `show`.



Rysunek 5.4. Formularz umożliwiający wybór wielu elementów z listy wyboru

Zobacz również

- Więcej informacji na temat metod wyboru w Rails znajduje się pod adresem pod adresem *http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html*,
- Receptura 3.0, „Wprowadzenie”, zawierająca omówienie tabel łączących używanych w zarządzaniu relacjami wiele-do-wielu.

5.5. Wyodrębnianie wspólnego kodu prezentacyjnego za pomocą makiet

Problem

Sporo witryn WWW składających się z wielu stron zawiera wspólne elementy wizualne, które pojawiają się na większości stron witryny (bądź nawet na wszystkich). Chcemy wyodrębnić ten wspólny kod prezentacyjny i uniknąć zbędnego powtarzania się w szablonach widoków.

Rozwiązanie

W katalogu *app/views/layouts* utworzymy plik makiety (ang. *layout*) zawierający elementy prezentacyjne, które mają być wyświetlane we wszystkich szablonach generowanych przez określony kontroler. Nazwijmy ten plik zgodnie z nazwą kontrolera, do którego szablonów ma on zostać użyty. W pewnym miejscu tego pliku umieścimy wywołanie do `yield` w celu wysłania zawartości kodu, do którego ma zostać zastosowana makieta.

app/views/layouts/main.rhtml:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml:/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
  <%= stylesheet link_tag "main" %>
  <title>Jakaś witryna CSS</title>
</head>
<body>
  <div id="header">
    <h1>Treść nagłówka...</h1>
  </div>

  <div id="leftcol">
    <h3>Nawigacja:</h3>
    <ul>
      <li><a href="/main/">Strona główna</a></li>
      <li><a href="/sales/">Sprzedaż</a></li>
      <li><a href="/reports/">Raporty</a></li>
      <li><a href="/support/">Obsługa</a></li>
    </ul>
  </div>

  <div id="maincol">
    <%= yield %>
  </div>
```

```
    <div id="footer">
      <p>Tutaj umieszczamy tekst stopki...</p>
    </div>
  </body>
  </html>
```

Po utworzeniu i umieszczeniu we właściwym miejscu pliku makiety *main.rhtml* każdy plik szablonu z katalogu *app/views/main/* będzie otoczony przez zawartość makiety, np. poniższy plik *index.rhtml* zastąpi wywołanie `yield` w pliku makiety.

app/views/main/index.rhtml:

```
<h2>Czym jest Web 2.0</h2>
<p>Załamanie rynku internetowego, do którego doszło jesienią 2001 roku, wyznaczyło
punkt zwrotny sieci WWW. Wiele osób doszło do wniosku, że WWW było przereklamowane,
podczas gdy gwałtowne wzrosty i wiążące się z tym załamania wydają się być wspólną
cechą wszystkich rewolucji technologicznych. Zazwyczaj takie załamania wyznaczają
punkt, w którym dominująca technologia jest już gotowa do zajęcia swego miejsca na
scenie. Pretendenci zostają wykopani, a prawdziwe historie sukcesów pokazują siłę
pozostałych. Tak pojawia się zrozumienie, w czym ci pierwsi różnią się od tych
drugich.</p>
```

Zauważmy, że plik makiety zawiera wywołanie do `stylesheet_link_tag "main"`, wysyłające znacznik dołączenia skryptu dla poniższego pliku CSS, który rozmieszcza różne elementy strony.

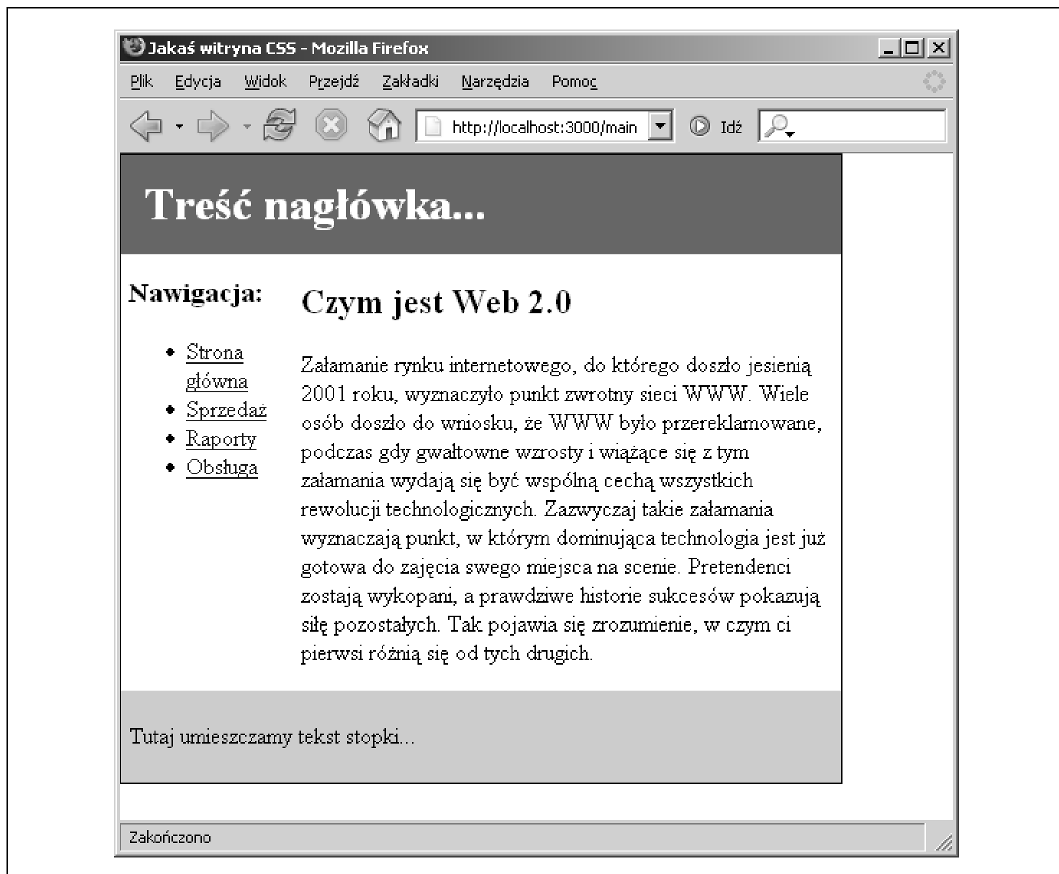
public/stylesheets/main.css:

```
body {
  margin: 0;
  padding: 0;
  color: #000;
  width: 500px;
  border: 1px solid black;
}
#header {
  background-color: #666;
}
#header h1 { margin: 0; padding: .5em; color: white; }
#leftcol {
  float: left;
  width: 120px;
  margin-left: 5px;
  padding-top: 1em;
  margin-top: 0;
}
#leftcol h3 { margin-top: 0; }
#maincol { margin-left: 125px; margin-right: 10px; }
#footer { clear: both; background-color: #ccc; padding: 6px; }
```

Omówienie

Domyślnie jeden plik makiety odpowiada każdemu kontrolerowi naszej aplikacji. Rozwiązanie konfiguruje makietę dla aplikacji z kontrolerem `Main`. Domyślnie widoki generowane przez kontroler `Main` stosują makietę *main.rhtml*.

Na rysunku 5.5 przedstawiono dane wyjściowe makiety dla zawartości szablonu *index.rhtml*, z użyciem arkusza stylów *main.css*.



Rysunek 5.5. Typowa strona WWW z czterema obszarami, utworzona z wykorzystaniem makiet

Za pomocą metody `layout` możemy wyraźnie zadeklarować, z której makiety kontroler powinien korzystać. Jeśli np. chcemy, aby kontroler `Gallery` używał tej samej makiety, co kontroler `Main`, powinniśmy dodać poniższe wywołanie makiety do definicji klasy kontrolera:

```
class GalleryController < ApplicationController
  layout 'main'
  ...
end
```

Metoda `layout` akceptuje także dodatkowe opcje. Aby użyć makiety do wszystkich akcji z wyjątkiem `popup`, należy zastosować:

```
layout 'main', :except => :popup
```

Dodatkowo zmienne egzemplarza zdefiniowane w akcji są dostępne wewnątrz widoku wygenerowanego w oparciu o tę akcję oraz szablonu makiety użytego do tego widoku.

W starszych projektach można spotkać się z następującą składnią zamiast nowszej `yield`:

```
<%= @content_for_layout %>
```

Obie pełnią to samo zadanie, dołączając treść do szablonu.

Zobacz również

- Receptura 5.6, „Definiowanie domyślnej makiety dla aplikacji".

5.6. Definiowanie domyślnej makiety dla aplikacji

Problem

Chcemy utworzyć jednolitą szatę graficzną w całej aplikacji za pomocą pojedynczego szablonu makiety.

Rozwiązanie

Aby zastosować jedną domyślną makietę do każdego widoku kontrolera, należy utworzyć szablon makiety o nazwie *application.rhtml* i umieścić go w katalogu z makietami naszej aplikacji (*app/views/layouts*), np.:

app/views/layouts/application.rhtml.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
              "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <title>Mój internetowy blog</title>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-2" />
    <%= stylesheet_link_tag "weblog" %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
    <div id="container">

      <%= yield %>

    </div>
  </body>
</html>
```

Omówienie

Ogólny aplikacyjny szablon makiety z tego rozwiązania będzie domyślnie stosowany do wszystkich naszych widoków. Możemy spowodować pominięcie takiego zachowania dla określonego kontrolera (a nawet dla akcji), tworząc dodatkowe pliki makiet o nazwach pochodzących od nazw naszych kontrolerów lub jawnie wywołać metodę `layout` w obrębie definicji klasy kontrolera.

Zobacz również

- Receptura 5.5, „Wyodrębnianie wspólnego kodu prezentacyjnego za pomocą makiet".

5.7. Generowanie kodu XML za pomocą szablonów Builder

Problem

Zamiast generować pliki HTML za pomocą ERb, chcemy wygenerować kod XML lub XHTML. I raczej nie mamy ochoty na samodzielne wpisywanie wszystkich jego znaczników.

Rozwiązanie

Aby w Rails zbudować szablon Builder, należy utworzyć plik o rozszerzeniu *.rxml*. Plik ten trzeba umieścić w katalogu *views*. Przykładowo poniższy szablon Builder generowany jest w momencie wywołania akcji `show` kontrolera `DocBook`.

app/views/docbook/show.rxml:

```
xml.instruct!
xml.declare! :DOCTYPE, :article, :PUBLIC,
  "-//OASIS//DTD DocBook XML V4.4//EN",
  "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd"
xml.article do
  xml.title ("Czym jest Web 2.0")
  xml.section do
    xml.title("Wzorce projektowe i modele biznesowe dla oprogramowania następnej
      generacji")
    xml.para("Załamanie rynku internetowego, do którego doszło jesienią 2001 roku,
      wyznaczyło punkt zwrotny sieci WWW. Wiele osób doszło do wniosku, że WWW było
      przereklamowane, podczas gdy gwałtowne wzrosty i wiążące się z tym załamania
      wydają się być wspólną cechą wszystkich rewolucji technologicznych. Zazwyczaj
      takie załamania wyznaczają punkt, w którym dominująca technologia jest już
      gotowa do zajęcia swego miejsca na scenie. Pretendenci zostają wykopani,
      a prawdziwe historie sukcesów pokazują siłę pozostałych. Tak pojawia się
      zrozumienie, w czym ci pierwsi różnią się od tych drugich.")
  end
end
```

Omówienie

Po wywołaniu akcji `show` kontrolera `DocBook` rozwiązanie generuje następujące dane wyjściowe:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN"
        "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">
<article>
  <title>Czym jest Web 2.0</title>
  <section>
    <title> Wzorce projektowe i modele biznesowe dla oprogramowania następnej
      generacji </title>
    <para> Załamanie rynku internetowego, do którego doszło jesienią 2001 roku,
      wyznaczyło punkt zwrotny sieci WWW. Wiele osób doszło do wniosku, że WWW było
      przereklamowane, podczas gdy gwałtowne wzrosty i wiążące się z tym załamania
      wydają się być wspólną cechą wszystkich rewolucji technologicznych. Zazwyczaj
      takie załamania wyznaczają punkt, w którym dominująca technologia jest już
      gotowa do zajęcia swego miejsca na scenie. Pretendenci zostają wykopani,
      a prawdziwe historie sukcesów pokazują siłę pozostałych. Tak pojawia się
```

```
      zrozumienie, w czym ci pierwsi różnią się od tych drugich.</para>
    </section>
  </article>
```

Szablony Builder działają na zasadzie przekształcania wywołań metod na obiekcie `Builder::XmlMarkup` w znaczniki obejmujące pierwszy argument tego obiektu. Pozostałe opcjonalne argumenty są tablicą asocjacyjną, która jest interpretowana jako atrybuty tworzonego znacznika. Oto przykład:

```
xml = Builder::XmlMarkup.new
xml.h1('Ruby on Rails', {:class => 'framework'})
```

Ten kod wygeneruje następujący znacznik:

```
<h1 class="framework">Ruby on Rails</h1>
```

W Rails szablony Builder są dostarczane automatycznie za pomocą obiektu `Builder::XmlMarkup` o nazwie `xml`, więc nie ma tu potrzeby tworzenia egzemplarza. Pierwszy parametr jest przesyłany wspólnie jako blok, dzięki czemu tworzenie zagnieżdżonych znaczników staje się łatwe i czytelne. Oto przykład elementów podrzędnych utworzonych w obrębie elementu macierzystego przy użyciu składni blokowej:

```
xml.h1 do
  xml.comment! "z niewielkim naciskiem na Ruby..."
  xml.span("Ruby", :style => "color: red;")
  xml.text! " on Rails!"
end
```

Ten szablon wytworzy:

```
<h1>
  <!-- z niewielkim naciskiem na Ruby... -->
  <span style="color: red;">Ruby</span>
  on Rails!
</h1>
```

Metody `comment!` i `text!` mają szczególne znaczenie; nie są one interpretowane jak nazwy znaczników, lecz tworzą odpowiednio komentarze XML lub zwykły tekst. Zauważmy, że nazwy tych metod nie odpowiadają konwencji nazewniczej Ruby, zgodnie z którą metody „destrukcyjne”, czyli modyfikujące obiekt bazowy (np. metody `gsub!` lub `strip!` klasy `String`), zakończone są znakiem *!*. Te metody tworzą jedynie dane wyjściowe, nie modyfikując obiektu bazowego.

Zobacz również

- Więcej informacji na temat szablonów Builder znajduje się na witrynie projektu Rubyforge, pod adresem *http://builder.rubyforge.org*.

5.8. Generowanie źródeł RSS z danych Active Record

Problem

Chcemy, aby nasza aplikacja umożliwiała publikowanie danych z jej modelu w postaci źródła RSS (ang. *Really Simple Syndication*). Załóżmy np., że mamy bazę danych z informacjami o produktach. Są to dane ulegające częstym zmianom; chcemy zaoferować klientom wygodny sposób umożliwiający śledzenie tych zmian na bieżąco.

Rozwiązanie

Tworzenie źródła RSS odbywa się za pomocą akcji generującej w locie RSS w formacie XML przy użyciu szablonów Builder. Załóżmy, że mamy następujący schemat, który definiuje tabelę z książkami. Każdy rekord zawiera informacje o sprzedaży, które często się zmieniają.

db/schema.rb:

```
ActiveRecord::Schema.define() do
  create_table "books", :force => true do |t|
    t.column "title", :string, :limit => 80
    t.column "sales_pitch", :string
    t.column "est_release_date", :date
  end
end
```

W kontrolerze `XmlController` tworzymy akcję o nazwie `rss`, zbierającą informacje z modelu `Book` do zmiennej egzemplarza, która będzie używana przez szablon Builder:

app/controllers/xml_controller.rb:

```
class XmlController < ApplicationController

  def rss
    @feed_title = "Książki wydawnictwa O'Reilly"
    @books = Book.find(:all, :order => "est_release_date desc",
                             :limit => 2)
  end
end
```

W widoku powiązanym z akcją `rss` stosujemy konstrukcje znaczników XML szablonu Builder w celu utworzenia źródła RSS w formacie XML, na które składać się będzie zawartość zmiennych egzemplarza `@feed_title` i `@books`.

app/views/xml/rss.rxml:

```
xml.instruct! :xml, :version=>"1.0", :encoding=>"UTF-8"
xml.rss('version' => '2.0') do
  xml.channel do
    xml.title @feed_title
    xml.link(request.protocol +
      request.host_with_port + url_for(:rss => nil))
    xml.description(@feed_title)
    xml.language "en-us"
    xml.ttl "40"
    # Przykład daty i czasy według RFC-822: Tue, 10 Jun 2003 04:00:00 GMT
    xml.pubDate(Time.now.strftime("%a, %d %b %Y %H:%M:%S %Z"))
    @books.each do |b|
      xml.item do
        xml.title(b.title)
        xml.link(request.protocol + request.host_with_port +
          url_for(controller => "posts", :action => "show", :id => b.id))
        xml.description(b.sales_pitch)
        xml.guid(request.protocol + request.host_with_port +
          url_for(:controller => "posts", :action => "show", :id => b.id))
      end
    end
  end
end
```

Omówienie

Źródła RSS pozwalają użytkownikom na bieżąco śledzić częste aktualizacje witryny za pomocą agregatora źródeł RSS, np. NetNewsWire lub rozszerzenia Sage do przeglądarki Firefox. Wykorzystanie źródeł RSS i agregatorów znacznie ułatwia nadążanie za dużą ilością stale zmieniających się informacji. Źródła RSS zazwyczaj składają się z tytułu i krótkiego opisu, do którego załączony jest odnośnik do pełnej treści dokumentu opisywanego w danym elemencie.

Pierwszy wiersz w szablonie *rss.rxml* tworzy deklarację XML, która określa wersję XML oraz zastosowane w dokumencie kodowanie znaków. Następnie tworzony jest element główny, zawierający w sobie wszystkie pozostałe elementy. Elementy wiadomości RSS generowane są przez wykonywanie pętli na obiektach w `@books` i tworzenie elementów w oparciu o atrybuty każdego obiektu `Book`.

Za pomocą wywołania `Book.find` w akcji `rss` z ograniczeniem do dwóch obiektów wynikowe źródło RSS dla tego rozwiązania zwróci następujący kod:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>Najnowsze książki wydawnictwa O'Reilly</title>
    <link>http://orsini.us:3000/xml/rss</link>
    <description>Najnowsze książki wydawnictwa O'Reilly</description>
    <language>pl</language>
    <ttl>40</ttl>
    <pubDate>Sun, 30 Apr 2006 17:34:20 PDT</pubDate>
    <item>
      <title>Rewolucja w dolinie</title>
      <link>http://orsini.us:3000/posts/show/20</link>
      <description>Uznawany za współtwórcę komputera Macintosh, Andy Herzfeld
        przedstawia informacje z pierwszej ręki na temat zdarzeń i postaci, które
        doprowadziły do wydania tego rewolucyjnego urządzenia.</description>
      <guid>http://orsini.us:3000/posts/show/20</guid>
    </item>
    <item>
      <title>Excel 200. Indywidualne szkolenie</title>
      <link>http://orsini.us:3000/posts/show/17</link>
      <description>Dzięki temu wyczerpującemu podręcznikowi podstaw pracy z arkuszem
        kalkulacyjnym poznamy edycję i formatowanie, pracę z formułami, diagramy
        i wykresy, makra, metody integracji programu Excel z innymi programami oraz
        szereg tematów zaawansowanych.</description>
      <guid>http://orsini.us:3000/posts/show/17</guid>
    </item>
  </channel>
</rss>
```

Stosunkowo rozwlekłe wywołanie `Time.now.strftime` jest tu konieczne, aby utworzyć poprawny format daty zgodny z RFC-822, czego wymaga specyfikacja RSS 2.0 (metoda Ruby `Time.now` pomija znak przecinka).

Zobacz również

- Walidator W3C, *http://validator.w3.org/feed*,
- Specyfikacja RSS 2.0, *http://blogs.law.harvard.edu/tech/rss*.

5.9. Ponowne wykorzystanie elementów za pomocą podszablonów

Problem

Chcemy wyeliminować powtarzanie się kodu w naszych szablonach przez rozbicie części szablonów w mniejsze podszablony. Chcielibyśmy wykorzystywać te podszablony wielokrotnie w tym samym szablonie, a nawet w różnych szablonach. Aby jeszcze bardziej zwiększyć ich przydatność, takie szablony wielokrotnego użytku powinny akceptować zmienne lokalne przesyłane do nich jako parametry.

Rozwiązanie

Wykorzystamy ponownie kod przez utworzenie i wygenerowanie podszablonów (ang. *partials*), opcjonalnie przesyłając zmienne z szablonu macierzystego w celu ich użycia w podszablonach. Aby to zademonstrować, skonfigurujemy kontroler `Properties` z akcją `list`, która wypełni właściwościami zmienną egzemplarza.

app/controllers/properties_controller.rb:

```
class PropertiesController < ApplicationController
  def list @properties = Property.find(:all, :order => 'date_listed',
                                            :limit => 3)
  end
end
```

Podszablon przypomina dowolny inny szablon, jednak rozszerzenie jego pliku rozpoczyna się znakiem podkreślenia. Utwórzmy podszablon o nazwie *_property.rhtml* i przeprowadźmy w nim iterację zawartości tablicy `@properties` i wyświetlmy jej zawartość. Za pomocą metody `cycle` nadamy kolejnym wierszom listy nieruchomości na przemian kolor biały oraz kolor przypisany do wartości zmiennej lokalnej `bgcolor`.

app/views/properties/_property.rhtml:

```
<div style="background: <%= cyde(bgcolor, '#fff') %>; padding: 4px;">
  <strong>Adres: </strong>
  <%= property.address %><br />
  <strong>Cena: </strong>
  <%= number_to_currency(property.price) %><br />
  <strong>Opis: </strong>
  <%= truncate(property.description, 60) %>
</div>
```

Generujemy podszablon *_property.rhtml* z widoku *list.rhtml* za pomocą wywołania metody `render`, przesyłając nazwę podszablonu (nazwę jego pliku, bez znaku podkreślenia i rozszerzenia) do opcji `:partial`. Dodatkowo przesyłamy też do szablonu `bgcolor` jako zmienną lokalną, przypisując ją do wartości `:bgcolor` w tablicy asocjacyjnej przesyłanej do opcji `:locals`.

app/views/properties/list.rhtml:

```
<h3>Spis nieruchomości:</h3>

<%= render(:partial => 'property',
       :locals => {:bgcolor => "#ccc"},
       :collection => @properties ) %>
```

Omówienie

Wywołanie akcji `list` kontrolera `Properties` z tego rozwiązania wyświetli informacje na temat każdej nieruchomości, z zastosowaniem naprzemiennych kolorów tła. Domyślnie podszablony mają dostęp do zmiennej egzemplarza o tej samie nazwie, co podszablon, podobnie jak szablon *list.rhtml* ma dostęp do zmiennej egzemplarza `@properties`. Jeśli takie zachowanie domyślnie jest niepożądane, możemy przesłać dowolną zmienną lokalną, włączając ją do tablicy asocjacyjnej przesyłanej do opcji `:locals` metody `render`.

Taki podszablon możemy wywołać z dowolnego innego szablonu w naszej aplikacji, przesyłając ścieżkę bezwzględną do opcji `:partial` metody `render`. W rzeczywistości, jeśli nasz podszablon zawiera jakiekolwiek ukośniki, Rails będzie szukał tego podszablonu w odniesieniu do katalogu *app/view* naszej aplikacji.

Rozwiązanie przesyła `:partial => 'property'` do metody `render`, nakazując jej odszukanie pliku o nazwie *_property.rhtml* w katalogu *app/views/properties* (tym samym, w którym znajduje się *list.rhtml*). Jeśli `property` poprzedzimy ukośnikiem, np. `:partial => '/property'`, wówczas metoda `render` będzie szukać tego podszablonu w katalogu *app/views*. Takie zachowanie jest przydatne, jeśli planujemy współdzielić podszablony z szablonami widoków różnych kontrolerów. Powszechnie przyjętą konwencją jest utworzenie w katalogu *app/views* podkatalogu przeznaczonego na współdzielone podszablony, a następnie poprzedzanie ścieżki do podszablonu ukośnikiem i nazwą takiego wspólnego katalogu (np. `:partial => '/shared/property'`).

Tworząc podszablon do obsługi wyświetlania pojedynczego obiektu, możemy go wciąż ponownie wykorzystywać. Ten sam podszablon, który wywoływaliśmy wcześniej z szablonu *list.rhtml*, możemy teraz użyć z szablonu *show.rhtml*, który (zgodnie z konwencją) generuje pojedynczy model. Oto, jak wygląda szablon *show*:

app/views/properties/show.rhtml:

```
<h3>Spis nieruchomości: </h3>
<%= render :partial => 'property',
:locals => {:property => @property} %>
```

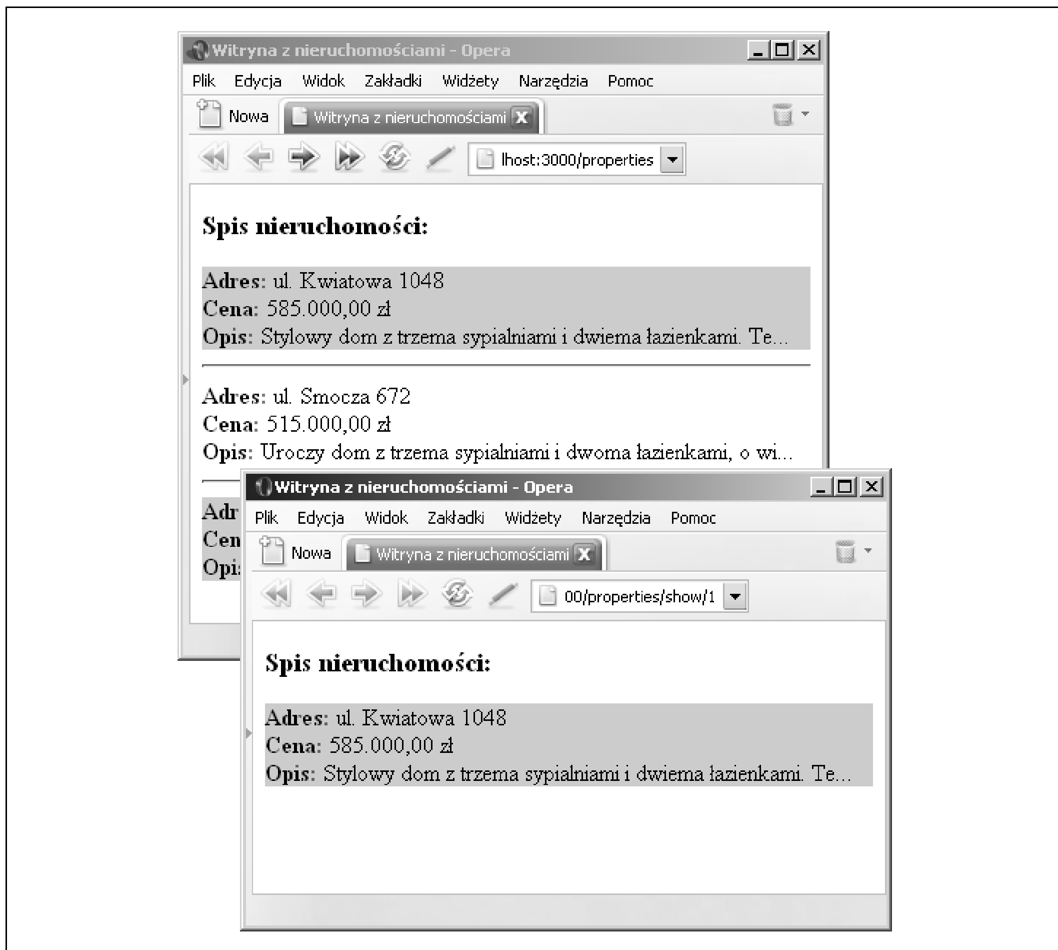
Dodajemy do kontrolera metodę `show`:

app/controllers/properties_controller.rb:

```
class PropertiesController < ApplicationController
  def list
    @properties = Property.find(:all, :order => 'date_listed',
                                      :limit => 3)
  end
```

```
  def show
    @property = Property.find(params[:id])
  end
end
```

Na rysunku 5.6 przedstawiono wynik każdej wersji wyświetlającej wiele obiektów Property przy użyciu podszablonów.



Rysunek 5.6. Widok przedstawiający spis nieruchomości oraz pojedynczą nieruchomość; obie strony zostały wygenerowane przy użyciu tego samego podszablonu

Zobacz również

- Receptura 5.5, „Wyodrębnianie wspólnego kodu prezentacyjnego za pomocą makiet”.

5.10. Przetwarzanie pól wejściowych tworzonych dynamicznie

Problem

Chcemy zbudować i przetwarzać formularz składający się z dynamicznie tworzonych pól wejściowych. Załóżmy, że mamy tabelę użytkowników, z których każdy może zostać powiązany z jedną rolą lub większą liczbą ról. Zarówno użytkownicy, jak i role pochodzą z bazy danych; nowych użytkowników i role można będzie dodać w każdej chwili. Chcemy umożliwić zarządzanie relacjami zachodzącymi między użytkownikami a rolami.

Rozwiązanie

Czasami najłatwiej administrować takimi relacjami za pomocą tabel zawierających pola wyboru, po jednym na każdą możliwą relację między dwoma modelami.

Zaczniemy od utworzenia tabel zawierających użytkowników i role oraz tabeli uprawnień do przechowywania powiązań:

db/schema.rb:

```
ActiveRecord::Schema.define(:version => 0) do

  create_table "roles", :force => true do |t|
    t.column "name", :string, :limit => 80
  end

  create_table "users", :force => true do |t|
    t.column "login", :string, :limit => 80
  end

  create_table "permissions", :id => false, :force => true do |t|
    t.column "role_id", :integer, :default => 0, :null => false
    t.column "user_id", [integer, :default => 0, :null => false
  end
end
```

W celu zwiększenia elastyczności w manipulacji danymi w tabeli łączącej tworzymy relację wiele-do-wielu, korzystając z opcji `:has_many :through`:

```
class Role < ActiveRecord::Base
  has_many :permissions, :dependent => true
  has_many :users, :through => :permissions
end

class User < ActiveRecord::Base
  has_many :permissions, dependent => true
  has_many :roles, :through => :permissions
end

class Permission < ActiveRecord::Base
  belongs_to :role
  belongs_to :user
end
```

Tworzymy teraz kontroler `UserController` z akcjami służącymi do wyświetlania i aktualizowania wszystkich możliwych powiązań między użytkownikami a rolami:

app/controllers/user_controller.rb:

```
class UserController < ApplicationController

  def list_perms
    @users = User.find(:all, :order => "login")
    @roles = Role.find(:all, :order => "name")
  end

  def update_perms
    Permission.transaction do
      Permission.delete_all
      for user in User.find(:all)
        for role in Role.find(:all)
          if params[:perm]["#{user.id}-#{role.id}"] == "on"
            Permission.create(:user_id => user.id, :role_id => role.id)
          end
        end
      end
    end
    flash[:notice]  = "Zaktualizowano uprawnienia."
    redirect_to :action => "list_perms"
  end
end
```

Następnie tworzymy widok dla akcji `list_perms`, która będzie budować formularz zawierający tabelę z polami wyboru na przecięciach użytkowników i ról:

app/views/user/list_perms.rhtml:

```
<h2>Administracja uprawnieniami</h2>

<% if flash[:notice] %>
  <p style="color: red;"><%= flash[:notice] %></p>
<% end %>

<% form_tag :action => "update_perms" do %>
<table style="background: #ccc;">
  <tr>
    <th> </th>
      <% for user in @users %>
        <th><%= user.login %></th>
      <% end %>
  </tr>

<% for role in @roles %>
  <tr style="background: <%= cycle("#ffc","white") %>;">
    <td align="right"><strong><%= role.name %></strong></td>

    <% for user in @users %>
      <td align="center">
        <%= get_perm(user.id, role.id) %>
      </td>
    <% end %>

<% end %>
</table>
<br />
<%= submit_tag "Zapisz zmiany" %>
<% end %>
```

Metoda pomocnicza `get_perm` użyta w widoku `list_perms` tworzy kod HTML dla każdego pola wyboru w formularzu. Definiujemy ją w pliku *user_helper.rb*:

app/helpers/user_helper.rb:

```
module UserHelper

  def get_perm(role_id, user_id)
    name = "perm[#{user_id}-#{role_id}]"
    perm = Permission.find_by_role_id_and_user_id(role_id, user_id)
    color = "#f66"
    unless perm.nil?
      color = "#9f9"
      checked = 'checked=\"checked\"'
    end
    return "<span style=\"background: #{color};\"><input name=\"#{name}\"
                  type=\"checkbox\" #{checked}></span>"
  end
end
```

Omówienie

Rozwiązanie rozpoczyna się od utworzenia powiązań typu wiele-do-wielu między tabelami użytkowników i ról za pomocą metody Active Record `has_many :through`. Umożliwia to manipulowanie danymi w tabeli uprawnień, a także skorzystanie z metody `transaction` klasy `Permission`.

Po skonfigurowaniu relacji między tabelami kontroler `User` przechowuje wszystkich użytkowników i obiekty ról w zmiennych egzemplarza, które są dostępne dla widoku. Widok `list_perms` rozpoczyna od pętli, która przeprowadza iterację użytkowników, wyświetlając ich jako nagłówki kolumn. Następnie tworzona jest tabela uprawnień użytkowników przez wykonanie pętli na rolach, które stają się wierszami tabeli, z drugą pętlą przeprowadzającą iteracje użytkowników (po jednym na każdą kolumnę).

Formularz składa się z tworzonych dynamicznie pól wyboru umieszczonych na przecięciach każdego użytkownika i roli. Każde pole wyboru jest identyfikowane przez łańcuch stanowiący połączenie łańcuchów `user.id` i `role.id` (`perm[#{user_id}-#{role_id}]`). Po zatwierdzeniu formularza `params[:perm]` jest tablicą asocjacyjną zawierającą wszystkie pary `user.id/role.id`. Zawartość tej tablicy wygląda następująco:

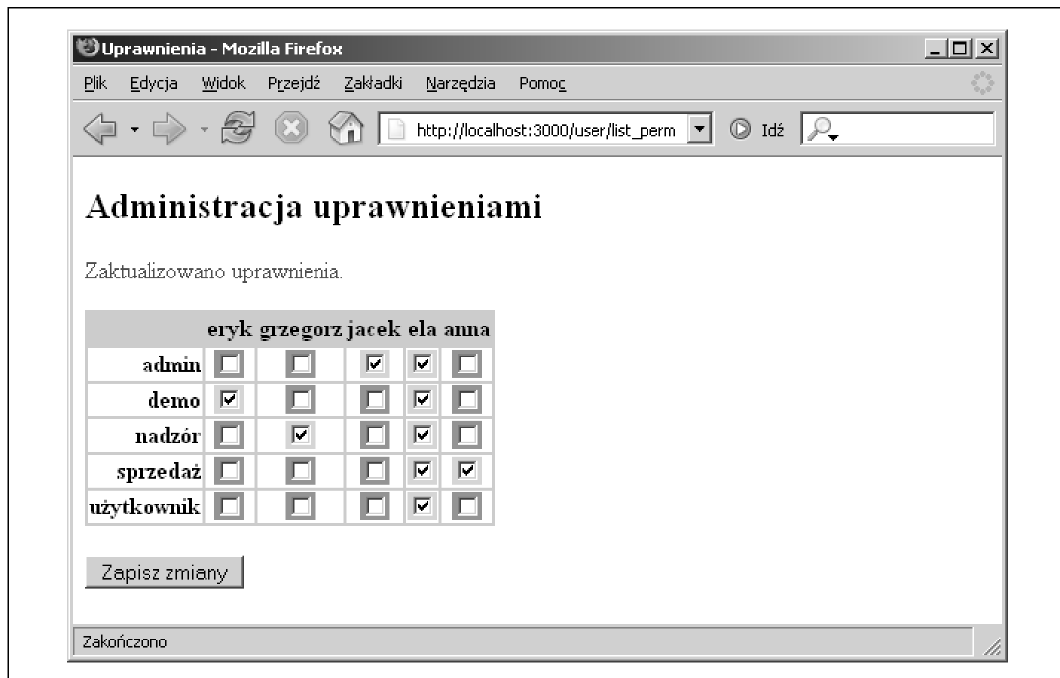
```
irb(#<UserController:0x405776a0>):003:0> params[:perm]
=> {"2-2"=>"on", "2-3"=>"on", "1-4"=>"on", "2-4"=>"on", "1-5"=>"on", "4-4"=>"on",
    "5-3"=>"on", "4-5"=>"on", "5-4"=>"on", "1-1"=>"on"}
```

Akcja `update_perms` kontrolera `User` rozpoczyna działanie od usunięcia wszystkich istniejących obiektów `Permission`. Ponieważ podczas pozostałego działania tej akcji mogłoby się wydarzyć coś nieprzewidzianego, całość kodu, który dokonuje zmian w bazie danych, objęta zostaje transakcją Active Record. Transakcja taka zapewnia, że usunięcie powiązania użytkownik-rola zostanie cofnięte, gdyby w dalszym działaniu metody coś się nie powiodło.

W celu przetworzenia wartości pól wyboru `update_perms` odtwarza zagnieżdżoną strukturę pętli, która tworzy elementy pola wyboru w widoku. Po zrekonstruowaniu nazwy każdego pola zostaje ono użyte do uzyskania dostępu do wartości w tablicy asocjacyjnej, które są przechowywane z wykorzystaniem tej nazwy jako klucza. Jeżeli wartość wynosi `on`, akcja tworzy obiekt `Permission`, który przypisuje rolę do określonego użytkownika.

Widok stosuje kolory w celu wskazania uprawnień, jakie istniały, zanim użytkownik je zmienił: kolor zielony oznacza powiązanie, a czerwony — jego brak.

Na rysunku 5.7 przedstawiono matrycę pól wejściowych utworzonych w tym rozwiązaniu.



Rysunek 5.7. Formularz zawierający wygenerowaną dynamicznie matrycę pól wyboru

Zobacz również

- Receptura 5.12, „Tworzenie formularzy WWW z wykorzystaniem metod pomocniczych”.

5.11. Dostosowywanie zachowań standardowych metod pomocniczych

Problem

Udostępnił Diego Scataglini

Znaleźliśmy metodę pomocniczą, która robi prawie dokładnie to, czego potrzebujemy, ale chcemy zmienić jej zachowanie domyślne. Chcielibyśmy np., aby metoda pomocnicza `content_helper` obsługiwała parametr blokowy.

Rozwiązanie

W przypadku tej receptury wykorzystamy istniejącą aplikację Rails lub utworzymy nową w celu swobodnego eksperymentowania z nią. Nadpisujemy definicję metody pomocniczej `content_tag` przez dodanie poniższego kodu do pliku *app/helpers/application_helper.rb*:

```
def content_tag(name, content, options = nil, &block)
  content = "#{content}#{yield if block_given?}"
  super
end
```

Normalnie użylibyśmy metody `content_tag` następująco:

```
content_tag("h1",
  @published_bookmark.title + ": " +
  content_tag("span",
    "opublikowana przez przez " +
    link_to(@user_login,
      user_url(:login => @published_bookmark.owner.login),
               :style => "font-weight: bold;"),
               :style => "font-size: .8em;"),
  :style => "padding-bottom: 2ex;")
```

Poprzednia struktura jest nieco zbyt skomplikowana. Dzięki zmodyfikowaniu `content_tag` możemy użyć bloków w celu poprawienia struktury kodu:

```
content_tag("h1", "#{@published_bookmark.title}: ",
  :style => "padding-bottom: 2ex;") do

  content_tag("span", "opublikowana przez ",
    :style => "font-size: .8em;") do

    link_to(@user_login, user_url(:login =>
      @published_bookmark.owner.login),
      :style => "font-weight: bold;")
  end
end
```

Omówienie

W tym rozwiązaniu wartość parametru blokowego powiązana jest z wartością parametru zawartości. Późniejsze wywołanie do `super` przekazuje wszystkie pozostałe obliczenia do oryginalnej definicji metody pomocniczej `content_tag`. Jeśli wywołamy `super` bez żadnych parametrów, argumenty zostaną przesłane w tej samej kolejności, w jakiej zostały otrzymane.

Powyższa implementacja `content_tag` jest dość łatwa do zrozumienia. Następny przykład jest nieco bardziej wyrafinowany, ale zrozumienie tego kodu jest warte wysiłku. Spróbujmy zastąpić naszą definicję `content_tag` następującym kodem:

```
def content_tag(name, *options, &proc)
  content = options.shift unless options.first.is_a?(Hash)
  content ||= nil
  options = options.shift
  if block_given?
    concat("<#{name}#{tag_options(options.stringify_keys) if options}>", proc.binding)
    yield(content)
    concat("</#{name}>", proc.binding)
```

```
    elsif content.nil?
      "<#{name}#{tag_options(options.stringify_keys) if options} />"
    else
      super(name, content, options)
    end
  end
```

Oto działanie nowego `content_tag` w akcji:

```
<%= content_tag "div", :class => "products" do
      content_tag "ul", :class => "list" do
        content_tag "li", "item1", :class => "item"
        content_tag "li", :class => "item"
      end
    end
%>
```

Wygeneruje to następujący kod HTML:

```
<div class="products">
  <ul class="list">
    <li class="item">item1</li>
    <li class = "item" />
  </ul>
</div>
```

5.12. Tworzenie formularzy WWW z wykorzystaniem metod pomocniczych

Problem

Udostępnił Diego Scataglini

Musimy utworzyć typowy formularz rejestracyjny, np. do biuletynu firmowego. Chcemy poddać walidacji wszystkie wymagane pola, a także upewnić się, że użytkownicy zaakceptowali warunki.

Rozwiązanie

Tworzenie formularzy WWW jest prawdopodobnie najczęściej wykonywanym zadaniem w tworzeniu aplikacji WWW. W tym przykładzie zakładam, że mamy utworzoną aplikację Rails o następującej strukturze tabel:

```
class CreateSignups < ActiveRecord:Migration
  def self.up
    create_table :signups do |t|
      t.column :name, :string
      t.column :email, :string
      t.column :dob, :date
      t.column :country, :string
      t.column :terms, :integer
      t.column interests, :string
      t.column :created_at, :datetime
    end
  end
```

```
  def self.down
    drop_table :signups
  end
end
```

Tworzymy odpowiedni model i kontroler:

```
$ ruby script/generate model signup
$ ruby script/generate controller signups index
```

Teraz wprowadzamy walidację do modelu `Signup`:

app/models/signup.rb:

```
class Signup < ActiveRecord::Base
  validates_presence_of :name, :country
  validates_uniqueness_of :email
  validates_confirmation_of :email
  validates_format_of :email, :with => /^([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})$/i
  validates_acceptance_of :terms, :message => "Należy zaakceptować warunki"
  serialize :interests

  def validate_on_create(today = Date::today)
    if dob > Date.new(today.year - 18, today.month, today.day)
      errors.add("dob", "Musisz mieć 18 lat lub więcej.")
    end
  end
end
```

Następnie do naszego kontrolera `Signups` dodajemy poniższą metodę `index`:

app/controllers/signups.rb:

```
class SignupsController < ApplicationController

  def index
    @signup = Signup.new(params[:signup])
    @signup.save if request.post?
  end
end
```

Na koniec tworzymy widok *index.rhtml*:

app/views/signups/index.rhtml:

```
<%= content_tag "div", "Dziękujemy za rejestrację w naszym biuletynie",
                       :class => "success" unless @signup.new_record? %>
<%= error_messages_for :signup %>
<% form_for :signup, @signup do |f| %>
    <label for="signup_name">Imię i nazwisko:</label>
    <%= f.text_field :name %><br />

    <label for="signup_email">Adres e-mail:</label>
    <%= f.text_field :email %><br />

    <label for="signup_email_confirmation">Potwierdź adres e-mail:</label>
    <%= f.text_field :email_confirmation %><br />

    <label for="signup_dob">Data urodzenia:</label>
    <%= f.date_select :dob, :order => [:day, :month, :year],
                            :start_year => (Time.now - 18.years).year,
                            :end_year => 1930 %><br />

    <label for="signup_country">Państwo:</label>
```

```
    <%= f.country_select :country, ["Polska", "USA"] %><br />

    <label for="signup_terms">Akceptuję warunki:</label>
    <%= f.check_box :terms %><BR clear=left>

    <h3>Moje zainteresowania:</h3>
    <% ["Pływanie", "Biegi", "Tenis"].each do |interest|%>
        <label><%= interest %></label>
        <%= check_box_tag "signup[interests][]", interest,
                (params[:signup] && params[:signup][:interests]) ?
                 params[:signup][:interests].include?(interest) : false %>
        <br />
    <% end %>

    <%- submit_tag "Signup", :style => "margin-left: 26ex;" %>
<% end if @signup.new_record? %>
```

Opcjonalnie, w celu poprawienia wyglądu strony możemy na zakończenie dodać do naszego pliku *scaffold.css* poniższe wiersze:

public/stylesheets/scaffold.css:

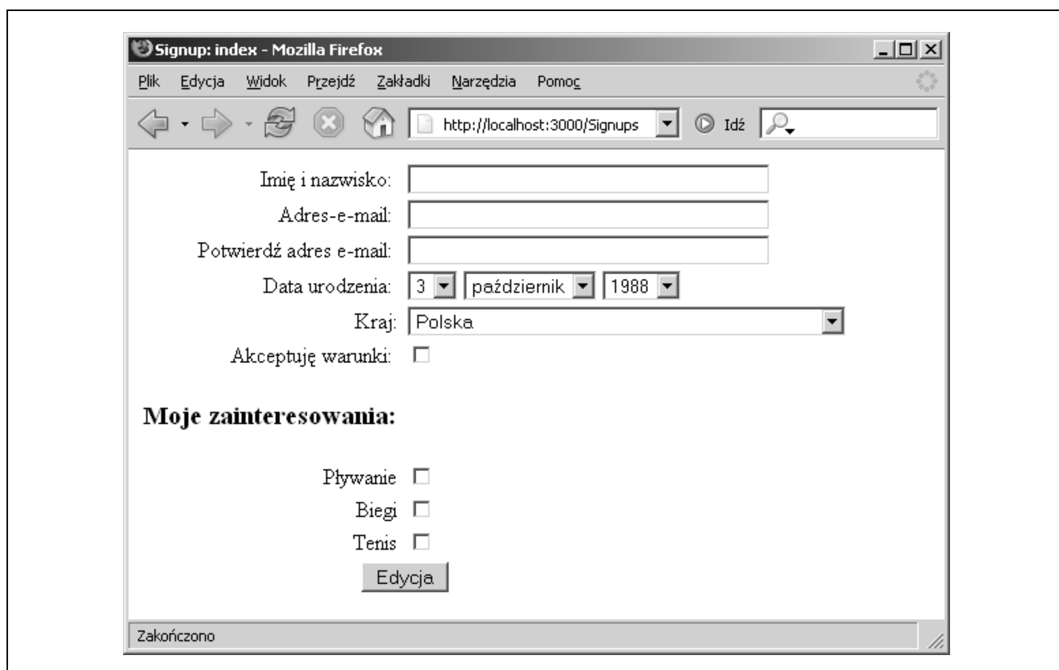
```
label    {
  display: block;
  float: left;
  width: 25ex;
  text-align: right;
  padding-right: lex;
}

.success {
  border: solid 4px #99f;
  background-color: #FFF;
  padding: 10px;
  text-align: center;
  font-weight: bold;
  font-size: 1.2em;
  width: 400px;
}
```

Omówienie

Rails udostępnia narzędzia, za których pomocą wykonywanie nawet bardzo nudnych zadań, takich jak tworzenie formularzy czy obsługa walidacji pól i stanów, staje się przyjemnością. Action View zawiera metody pomocnicze dla formularzy niemal na każdą okazję, a utworzenie takiej metody „od ręki" jest bardzo proste. Gdy już zapoznamy się z modułem walidacji Active Record, zbudowanie formularza stanie się dziecinnie łatwe.

Na rysunku 5.8 przedstawiono formularz rejestracyjny z tego rozwiązania.

Rozwiązanie korzysta z metody `form_for`, która przyjmuje symbol jak pierwszy parametr. Symbol ten używany jest przez Rails jako nazwa obiektu i zostanie przekazany do bloku. Zmienna *f* w wyrażeniu `f.text_field` reprezentuje połączenie między metodą pomocniczą a modelem obiektu, którego dotyczy. Drugi parametr to zmienna egzemplarza, która jest wcześniej zapełniana przez akcję `index` w kontrolerze i służy do podtrzymania stanu pomiędzy stronami. Każda metoda pomocnicza, która przyjmuje obiekt i metodę jako pierwszy parametr, może zostać użyta w połączeniu z metodą pomocniczą `form_for`.



Rysunek 5.8. Formularz rejestracyjny zawierający elementy wygenerowane za pomocą metod pomocniczych

Action View dostarcza m.in. metody pomocnicze `date_select` i `datetime_select`, służące do obsługi daty i czasu. Metody te są bardzo łatwe w konfiguracji. Możemy ukrywać i zmieniać kolejność poszczególnych części daty za pomocą parametru `:order`, np.:

```
date_select("user", "birthday", :order => [:month, :day])
```

Środowisko posiada również różne przydatne informacje, np. spis wszystkich państw i stref czasowych, i udostępnia je jako metody pomocnicze, a także stałe (np. `country_select`, `country_options_for_select`, `time_zone_options_for_select`, `time_zone_select`).

Metoda klasowa `validates_confirmation_of` jest niewiele warta. Obsługuje ona walidacje potwierdzeń, dopóki formularz zawiera pole potwierdzające. Rozwiązanie wymaga, aby użytkownik potwierdził swój adres e-mail za pomocą pola formularza `email_confirmation`. Jeśli musimy potwierdzić pole `password`, również możemy dodać pole `password_confirmation`.

W polu `interests` musimy wprowadzić pola wielokrotnego wyboru dla różnych zainteresowań. Użytkownik może zaznaczyć dowolną kombinację tych pól; do aplikacji należy zebranie wyników i przekształcenie ich w pojedyncze pole za pomocą serializacji. Dlatego nie możemy tu zastosować funkcji oferowanych przez `form_for`. Dodając znaki `[]` na końcu nazwy pola, wskazujemy, że jest to pole powtarzalne i może zawierać wiele wartości. Mimo iż rozwiązanie korzysta z `form_for` do utworzenia formularza, wciąż możemy mieszać ze sobą i dopasowywać metody pomocnicze, które nie całkiem pasują do danej formuły.

Rozwiązanie korzysta z introspekcji obiektów w celu wykrycia, czy wyświetlić komunikat potwierdzający lub wypełnić formularz dla użytkownika. Choć introspekcja jest zręcznym sposobem na pokazanie strony potwierdzającej, to wskazane jest skorzystanie z przekierowania do innej akcji. Można to naprawić w następujący sposób:

```
class SignupsController < ApplicationController
  def index
    @signup = Signup.new(params[:signup])
    if request.post? && @signup.save
      flash[:notice] = "Dziękujemy za rejestrację w naszym biuletynie"
      redirect_to "/"
    end
  end
end
```

Zobacz również

- Receptura 5.13, „Format daty, czasu i waluty”, zawiera więcej informacji na temat stosowania metody pomocniczej `view` w celu formatowania danych wyjściowych.

5.13. Format daty, czasu i waluty

Problem

Udostępnił Andy Shen

Chcemy dowiedzieć się, jak formatować datę, czas i walutę w widokach naszej aplikacji.

Rozwiązanie

Rails dostarcza dwa domyślne formaty wyświetlania obiektów daty lub czasu:

```
>> Date.today.to_formatted_s(:short)
=> "1 Oct"
>> Date.today.to_formatted_s(:long)
=> "October 1, 2006"
```

Jeśli potrzebujemy innego formatu, możemy użyć `strftime` do sformatowania łańcucha:

```
>> Date.today.strftime("Wydrukowano: %d-%m-%Y")
=> "Wydrukowano: 01-10-2006"
```

Tabela 5.1 zawiera spis wszystkich opcji formatujących.

Istnieją jeszcze inne opcje nieudokumentowane w API. Możemy stosować wiele opcji formatowania daty i czasu wymienionych w podręczniku systemowym (man) systemu Unix lub dokumentacji C w Ruby, np.:

- `%e` jest zastępowane przez dzień miesiąca w postaci liczby dziesiętnej (1–31); pojedyncze cyfry poprzedzane są znakiem spacji,
- `%R` jest odpowiednikiem `%H:%M`,
- `%r` jest odpowiednikiem `%I:%M:%S %p`,
- `%v` jest odpowiednikiem `%e-%b-%Y`.

Oto bieżąca data:

```
>> Time.now.strftime("%v")
=> "2-Oct-2006"
```

Tabela 5.1. Opcje łańcucha formatu daty

Symbol	Znaczenie
%a	Skrót nazwy dnia tygodnia („Nie”)
%A	Pełna nazwa dnia tygodnia („Niedziela”)
%b	Skrócona forma miesiąca („Sty”)
%B	Pełna nazwa miesiąca („Styczeń”)
%c	Preferowana forma lokalnego czasu i daty
%d	Dzień miesiąca (01...31)
%H	Godzina w formacie 24-godzinnym (00...23)
%I	Godzina w formacie 12-godzinnym (01...12)
%j	Dzień roku (001...366)
%m	Miesiąc roku (01...12)
%M	Minuta godziny (00...59)
%p	Wskaźnik pory dnia („AM” lub „PM”)
%S	Sekunda minuty (00...60)
%U	Numer tygodnia bieżącego roku (00...53)
%W	Numer tygodnia bieżącego roku (00...53)[1]
%w	Dzień tygodnia (0...6, gdzie niedziela to 0)
%x	Preferowana postać samej daty, bez czasu
%X	Preferowana postać samego czasu, bez daty
%y	Rok bez oznaczenia stulecia (00...99)
%Y	Pełna postać roku
%Z	Nazwa strefy czasowej
%%	Znak *%*

Wszystkie opcje formatowania stosuje się do obiektów `Time`, ale nie wszystkie opcje mają sens, jeśli zostaną użyte na obiektach `Date`. Oto jeden z przykładów formatowania obiektu `Date`:

```
>> Date.today.strftime("%Y-%m-%d %H:%M:%S %p")
=> "2006-10-01 00:00:00 AM"
```

Ta sama opcja wywołana na obiekcie `Time` dałaby następujący efekt:

```
>> Time.now.strftime("%Y-%m-%d %H:%M:%S %p")
=> "2006-10-01 23:49:38 PM"
```

Nie wydaje się to być łańcuchem formatującym dla pojedynczej cyfry miesiąca, więc spróbujmy nieco inaczej, np. tak:

```
"#{date.day}/#{date.month}/#{date.year}"
```

Dla walut Rails zapewnia metodę `number_to_currency`. Najprostszym zastosowaniem tej metody jest przesłanie do niej liczby, którą chcemy wyświetlić w postaci waluty:

```
>> number_to_currency(123.123)
=> "$123.12"
```

[1] Różnica między tą opcją a opcją powyżej polega na tym, że `%U` liczy tygodnie od pierwszej niedzieli w roku, a tydzień liczy od niedzieli do soboty, natomiast opcja `%W` liczy tygodnie od pierwszego poniedziałku, a tydzień trwa od poniedziałku do niedzieli — *przyp. tłum.*

Drugim parametrem tej metody może być tablica asocjacyjna, określająca cztery następujące opcje:

- `precision` — liczba cyfr po przecinku (wartość domyślna = `2`),
- `:unit` — jednostka waluty (wartość domyślna = „`$`”),
- `:separator` — symbol dziesiętny (domyślnie = „`.`”),
- `:delimiter` — symbol grupowania cyfr (domyślnie = „`,`”).

```
>> number_to_currency(123456.123, {"precision" => 1, :unit => "#",
                                  separator => "-", :delimiter => "^"})
=> "#123^456-1"
```

Omówienie

Dobrym pomysłem jest scalenie potrzebnych kodów formatujących w klasie pomocniczej Rails, np. `ApplicationHelper`, z której będą korzystać wszystkie widoki:

app/helpers/application_helper.rb:

```
module ApplicationHelper
  def render_year_and_month(date)
    h(date.strftime("%Y %B"))
  end

  def render_date(date)
    h(date.strftime("%Y-%m-%d"))
  end

  def render_datetime(time)
    h(time.strftime("%Y-%m-%d %H:%M"))
  end
end
```

Zobacz również

- Istnieje jeszcze kilka innych godnych uwagi metod pomocniczych związanych z liczbami, np. `number_to_percentage`, `number_to_phone`, `number_to_human_size`. Szczegółowe informacje na temat ich użycia znajdują się pod adresem *http://api.rubyonrails.org/classes/ActionView/Helpers/NumberHelper.html*.

5.14. Personalizacja profili użytkowników za pomocą grawatarów

Problem

Udostępnił Nicholas Wieland

Chcemy pozwolić użytkownikom na spersonalizowanie ich obecności na witrynie za pomocą wyświetlania niewielkich obrazków powiązanych z profilami i komentarzami każdego użytkownika.

Rozwiązanie

Zastosujemy **grawatary** (globalnie widocznie awatary), czyli małe pliki graficzne o rozmiarach 80×80 pikseli, powiązane z użytkownikami poprzez adresy e-mail. Obrazki te przechowywane są na zdalnym serwerze, a nie na witrynie, na której działa aplikacja. Użytkownicy tylko raz rejestrują grawatar, pozwalając, aby odpowiadające im obrazki były używane na wszystkich witrynach z obsługą grawatarów.

Aby do naszej aplikacji wprowadzić obsługę grawatarów, należy wewnątrz `ApplicationHelper` zdefiniować metodę zwracającą poprawny odnośnik z witryny *http://www.gravatar.com*:

app/helpers/application_helper.rb:

```
require "digest/md5"

module ApplicationHelper

  def url_for_gravatar(email)
    gravatar_id = Digest::MD5.hexdigest( email )
    "http://www.gravatar.com/avatar.php?gravatar_id=#{ gravatar_id }"
  end
end
```

Nasze widoki mogą korzystać z tej metody pomocniczej w bardzo prosty sposób. Wystarczy za pomocą `url_for_gravatar` zbudować adres URL w znaczniku pliku graficznego. W poniższym kodzie `@user,email` przechowuje adres e-mail właściciela grawatara:

```
<%= imagetag url_for_gravatar(@user.email) %>
```

Omówienie

Stosowanie grawatarów jest bardzo proste: wystarczy w miejscach, gdzie chcemy wyświetlić grawatar, użyć znacznika <img /> z atrybutem `src` wskazującym główną witrynę grawatarów, łącznie z sumą kontrolną MD5 adresu e-mail właściciela grawatara. Oto typowy adres URL grawatara:

```
http://www.gravatar.com/avatar.php?gravatar_id=7cdce9e94d317c4f0a3dcc20cc3b4115
```

Gdy użytkownik nie ma zarejestrowanego grawatara, adres URL zwróci przezroczysty obrazek GIF o wymiarach 1×1 piksel.

Zasada działania metody pomocniczej `url_for_gravatar` polega na obliczaniu sumy kontrolnej MD5 adresu e-mail przesłanego jej jako argument; metoda zwraca wówczas poprawny adres URL grawatara za pomocą interpolacji łańcucha.

Serwis Gravatar obsługuje pewne opcje pozwalające uniknąć konieczności manipulowania obrazkami w obrębie naszej aplikacji. Przesyłając do tej usługi atrybut `size`, możemy zmienić rozmiar grawatara na inny niż 80×80 pikseli (np. `size=40`).

Zobacz również

- *http://site.gravatar.com/site/implement*.

5.15. Unikanie szkodliwego kodu w widokach za pomocą szablonów Liquid

Problem

Udostępnił Christian Romney

Chcemy projektantom naszej aplikacji i użytkownikom końcowym zapewnić możliwość projektowania solidnych szablonów widoków bez narażania na szwank bezpieczeństwa lub integralności naszej aplikacji.

Rozwiązanie

Szablony Liquid są popularną alternatywą domyślnych widoków ERb z szablonami *.rhtml*. Szablony Liquid nie mogą uruchamiać przypadkowego kodu, możemy więc spać spokojnie ze świadomością, że nasi użytkownicy nie zniszczą przez przypadek naszej bazy danych.

Do zainstalowania Liquid wymagany jest moduł rozszerzający, najpierw jednak musimy wskazać Rails jego repozytorium. W oknie konsoli z głównego katalogu aplikacji Rails należy wpisać:

```
$ ruby script/plugin source svn://home.leetsoft.com/liquid/trunk
$ ruby script/plugin install liquid
```

Po zakończeniu działania polecenia możemy rozpocząć tworzenie szablonów Liquid. Podobnie jak ERb, szablony Liquid mają swoje miejsce w folderze kontrolera w *app/views*. Aby np. utworzyć szablon *index* dla kontrolera o nazwie BlogController, należy w folderze *app/views/blog* utworzyć plik o nazwie *index.liquid*.

Przyjrzyjmy się teraz składni znaczników Liquid. Aby wyświetlić jakiś tekst, wystarczy go po prostu objąć dwiema parami nawiasów klamrowych:

```
{{ 'Witaj, świecie!' }}
```

Możemy także filtrować tekst za pomocą przetwarzania potokowego przy użyciu składni bardzo przypominającej wiersz poleceń systemu Unix:

```
{{ 'Witaj, świecie! | downcase }}
```

Niemal wszystkie szablony, z wyjątkiem tych najbardziej trywialnych, będą również wymagać pewnej logiki. Liquid zawiera obsługę instrukcji warunkowych:

```
{% if user.last_name == 'Orsini' %}

  {{ 'Witaj ponownie, Robercie.' }}

{% endif %}
```

i pętli:

```
{% for line_item in order %}
  {{ line_item }}
{% endfor %}
```

Pora na kompletny przykład. Załóżmy, że mamy przygotowaną pustą aplikacje Rails z poprawnie skonfigurowanym plikiem *database.yml* oraz opisanym wcześniej modułem rozszerzającym Liquid.

Najpierw wygenerujemy model o nazwie `Post`:

```
$ ruby script/generate model Post
```

Następnie edytujemy plik migracji: *001_create_posts.rb*. Dla potrzeb tego przykładu postaramy się wszystko możliwie jak najbardziej uprościć:

db/migrate/001_create_posts.rb:

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.column :title, :string
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Teraz generujemy tabelę bazy danych poleceniem:

```
$ rake db:migrate
```

Po utworzeniu tabeli pora przystąpić do wygenerowania kontrolera dla aplikacji. Robimy to przy użyciu polecenia:

```
$ ruby script/generate controller Posts
```

Teraz wszystko jest już gotowe na dodanie do aplikacji obsługi Liquid. Uruchamiany swój ulubiony serwer produkcyjny za pomocą polecenia:

```
$ ruby script/server -d
```

Następnie dodajemy ogólną obsługę wizualizacji szablonów Liquid wewnątrz aplikacji. W tym celu otwieramy w edytorze tekstu plik klasy `ApplicationController` i dodajemy poniższą metodę `render_liquid_template`:

app/controllers/application.rb:

```
class ApplicationController < ActionController::Base

  def render_liquid_template(options={})
    controller = options[:controller].to_s if options[:controller]
    controller ||= request.symbolized_path_parameters[controller]

    action = options[:action].to_s if options[:action]
      action ||= request.symbolized_path_parameters[:action]

      locals = options[:locals] || {}
      locals.each_pair do |var, obj|
          assigns[var.to_s] = obj.respond_to?(:to_liquid) ? obj.to_liquid : obj
      end

      path = "#{RAILS_ROOT}/app/views/#{controller}/#{action}.liquid"
      contents = File.read(Pathname.new(path).cleanpath)
```

```
      template = Liquid: .'Template.parse(contents)
     returning template.render(assigns, :registers => {controller => controller})
                              do |result|
        yield template, result if block_given?
      end
    end

  end
```

Metoda ta, oparta częściowo o kod znajdujący się we wspaniałym narzędziu publikacyjnym Mephisto, znajduje poprawny szablon do wyświetlenia, analizuje go w kontekście przypisanych zmiennych, a następnie wyświetla go, gdy układ aplikacji przekazuje kontrolę do szablonu *index.liquid*.

Aby wywołać tę metodę, należy do `PostController` dodać poniższą akcję `index`:

app/controllers/posts_controller.rb:

```
class PostsController < ApplicationController

  def index
    @post = Post.new(:title => 'Moja pierwsza wiadomość')
    render_liquid_template :locals => {:post => @post}
  end

  # ...
end
```

Dla wygody dodajemy do modelu `Post` prostą metodę `to_liquid`:

app/models/post.rb:

```
class Post < ActiveRecord: :Base

  def to_liquid
    attributes.stringify_keys
  end
end
```

To już prawie koniec. Następnie w katalogu *app/views/posts* musimy utworzyć plik *index.liquid*. Szablon ten będzie zawierać jedynie poniższy wiersz:

app/views/posts/index.liquid:

```
<h2>{{ post.title | upcase }}</h2>
```

Na zakończenie zademonstruję, w jaki sposób można mieszać ze sobą i dopasowywać szablony RHTML w układzie z widokami o szablonach Liquid:

app/views/layouts/application.rhtml:

```
<html>
  <head>
    <title>Liquid - test</title>
  </head>
  <body>

    <%= yield %>

  </body>
</html>
```

Możemy już obejrzeć naszą aplikację, wskazując w przeglądarce katalog */posts*, np. *http://localhost:3000/posts*.

Omówienie

Główna różnica między Liquid a ERb polega na tym, że przy przetwarzaniu instrukcji Liquid nie korzysta z metody Ruby `Kernel#eval`. W efekcie szablony Liquid mogą przetwarzać tylko te dane, które zostały im wyraźnie ujawnione, co znacznie podnosi poziom bezpieczeństwa. Poza tym język szablonów Liquid jest bardziej zwięzły niż ERb, co ułatwia jego naukę.

Szablony Liquid są w wysokim stopniu konfigurowalne. Z łatwością można dodawać własne filtry tekstowe. Oto prosty filtr wykonujący szyfrowanie ROT-13 na zadanym łańcuchu:

```
module TextFilter

  def crypt(input)
    alpha = ('a'..'z').to_a.join
    alpha += alpha.upcase
    rot 13 = ('n'..'z').to_a.join + ('a'..'m').to_a.join
    rot13 += rot13.upcase

    input.tr(alpha, rot13)
  end
end
```

Aby zastosować ten filtr w naszym szablonie Liquid, należy w katalogu *lib* utworzyć folder o nazwie *liquid_filters*. Do tego nowego katalogu dodajemy plik o nazwie *text_filter.rb* zawierający przedstawiony powyżej kod.

Otwieramy nasz plik *environment.rb* i wpisujemy:

config/environment.rb:

```
require 'liquid_filters/text_filter'
Liquid::Template.register_filter(TextFilter)
```

Nasz szablon powinien teraz zawierać wiersz podobny po poniższego:

```
{{ post.title | crypt }}
```

Liquid to kod gotowy do produkcji. Tobias Lütke utworzył Liquid w celu jego wykorzystania w Shopify.com, narzędziu do handlu elektronicznego dla osób, które nie znają się na programowaniu. To bardzo elastyczne i eleganckie narzędzie, z którego mogą korzystać zarówno projektanci, jak i użytkownicy końcowi. W praktyce zechcemy zapewne składować przetwarzane szablony, prawdopodobnie w bazie danych. Świetnym przykładem działania szablonów Liquid jest kod narzędzia do blogów Mephisto, który jest dostępny pod adresem *http://mephistoblog.com*.

Zobacz również

- Więcej informacji na temat Liquid znajduje się pod adresem *http://www.liquidmarkup.org/*,
- Dokładniejsze informacje o Mephisto dostępne są na witrynie projektu pod adresem *http://www.mephistoblog.com*.

5.16. Globalizacja aplikacji Rails

Problem

Udostępnił Christian Romney

Chcemy do aplikacji Rails dodać obsługę wielu języków, walut oraz różne formaty daty i czasu. Zasadniczo chcemy wprowadzić obsługę internacjonalizacji (czyli i18n).

Rozwiązanie

Moduł rozszerzający Globalize dostarcza większość narzędzi niezbędnych, by przygotować aplikację do przedstawienia na forum ogólnoświatowym Na użytek tej receptury utworzymy pustą aplikację Rails o nazwie `global`:

```
$ rails global
```

Następnie za pomocą Subversion wyeksportujemy kod dla tego modułu folderu o nazwie *globalize* w katalogu *vendor/plugins*:

```
$ svn export http://svn.globalize-rails.org/svn/globalize/globalize/branches/for-1.1
vendor/plugins/globalize
```

Jeśli nasza aplikacja korzysta z bazy danych, musimy ją skonfigurować do przechowywania tekstów w różnych językach. MySQL obsługuje domyślnie kodowanie UTF-8. Konfigurujemy plik *database.yml* tak jak zwykle, pamiętając jedynie o podaniu parametru dla kodowania znaków:

config/database.yml:

```
development:
  adapter:mysql
  database: global_development
  username: root
  password:
  host: localhost
  encoding: utf8
```

Globalize używa kilku tabel bazy danych do śledzenia tłumaczeń. Tabele globalizacyjne aplikacji przygotowujemy za pomocą następującego polecenia:

```
$ rake globalize:setup
```

Następnie dodajemy do swojego środowiska poniższe wiersze:

config/environment.rb:

```
require 'jcode'
$KCODE = 'u'

include Globalize
Locale.set_base_language('pl')
```

Nasza aplikacja jest teraz wyposażona w funkcje globalizacyjne. Musimy jedynie utworzyć model i przetłumaczyć wszystkie występujące w nim łańcuchy danych. Aby naprawdę prze-

testować możliwości modułu Globalize, utwórzmy model `Product` zawierający pola `name`, `unit_price`, `quantity_on_hand` i `updated_at`. Najpierw generujemy model:

```
$ ruby script/generate model Product
```

Teraz definiujemy schemat dla tabeli produktów w pliku migracji. Możemy tu również dołączyć definicje modelu redundantnego na wypadek zmiany nazw przyszłych migracji lub usunięcia klasy `Product`.

db/migrate/001_create_products.rb:

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

class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.column :name, :string
      t.column :unit_price, :integer
      t.column :quantity_on hand, :integer
      t.column :updated_at, :datetime
    end

    Locale.set('pl')
    Product.new do |product|
      product.name = 'Mała czarna książka'
      product.unit_price = 999
      product.quantity_on_hand = 9999
      product.save
    end

    Locale.set('en-US')
    product = Product, find(:first)
    product.name = 'Little Black Book'
    product.save
  end

  def self.down
    drop_table :products
  end
end
```

Zauważmy, że przed wprowadzeniem tłumaczenia nazwy musimy zmienić ustawienia językowe. Przeprowadzamy teraz migrację bazy danych:

```
$ rake db:migrate
```

Widzimy, że jednostka ceny jest polem całkowitoliczbowym. Użycie liczb całkowitych eliminuje niedokładności, jakie mogą się pojawić w przypadku stosowania liczb zmiennoprzecinkowych (co jest bardzo złym pomysłem). Zamiast tego przyjmiemy cenę w groszach. Po zakończeniu migracji dokonujemy modyfikacji prawdziwej klasy modelu w celu odwzorowania ceny przez dostarczoną z Globalize klasę obsługującą ustawienia lokalne (łatwo zauważyć, że nie przeprowadzamy tutaj konwersji walut, gdyż wykracza poza temat tej receptury).

app/models/product.rb:

```
class Product < ActiveRecord::Base
  translates :name
  composed_of :unit_price, :class_name => "Globalize::Currency",
              :mapping => [ (unit_price cents) %w ]
end
```

Teraz generujemy kontroler w celu zaprezentowania nowych możliwości lingwistycznych naszej aplikacji. Tworzymy kontroler `Products` z akcją `show`:

```
$ ruby script/generate controller Products show
```

Modyfikujemy kontroler w następujący sposób:

app/controllers/products_controller.rb:

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
  end
end
```

Wprowadzamy ustawienia językowe do `before_filer` w kontrolerze `ApplicationController`:

app/controllers/application.rb:

```
class ApplicationController < ActionController::Base
  before_filter :set_locale

  def set_locale
    headers["Content-Type"] = 'text/html; charset=utf-8'

    default_locale = Locale.language_code
    request_locale = request.env['HTTP_ACCEPT_LANGUAGE']
    request_locale = request_locale[/[^,;]+/] if request_locale

    @locale = params[:locale] ||
      session[:locale] ||
      request_locale ||
      default_locale

    session[:locale] = @locale

    begin
      Locale.set @locale
    rescue ArgumentError
      @locale = default_locale
      Locale.set @locale
    end
  end
end
```

Zwróćmy uwagę na ustawienie kodowania UTF-8 w nagłówku `Content-Type`. Na koniec możemy zmodyfikować widok:

app/views/products/show.rhtml:

```
<h1><%= @product.name.t %></h1>
<table>
<tr>
  <td><strong><%= 'Cena'.t %>:</strong></td>
  <td><%= @product.unit_price %></td>
</tr>
<tr>
  <td><strong><%= 'Ilość'.t %>:</strong></td>
  <td><%= @product.quantity_on_hand.localize %></td>
</tr>
<tr>
  <td><strong><%= 'Zmodyfikowano'.t %>:</strong></td>
  <td><%= @product.updated_at.localize("%d %B %Y") %></td>
```

```
</tr>
</table>
```

Przed uruchomieniem aplikacji musimy wprowadzić tłumaczenie znajdujących się w szablonie łańcuchów `'Cena'`, `'Ilość'` i `'Zmodyfikowano'`. Uruchamiamy w tym celu konsolę Rails.

```
$ ruby script/console
```

Wpisujemy następujące polecenia:

```
>> Locale.set_translation('Cena', Language.pick('en-US'),'Price')
>> Locale.set_translation('Ilość', Language.pick('en-US'),'Quantity')
>> Locale.set translation('Zmodyfikowano', Language.pick('en-US'),'Modified')
```

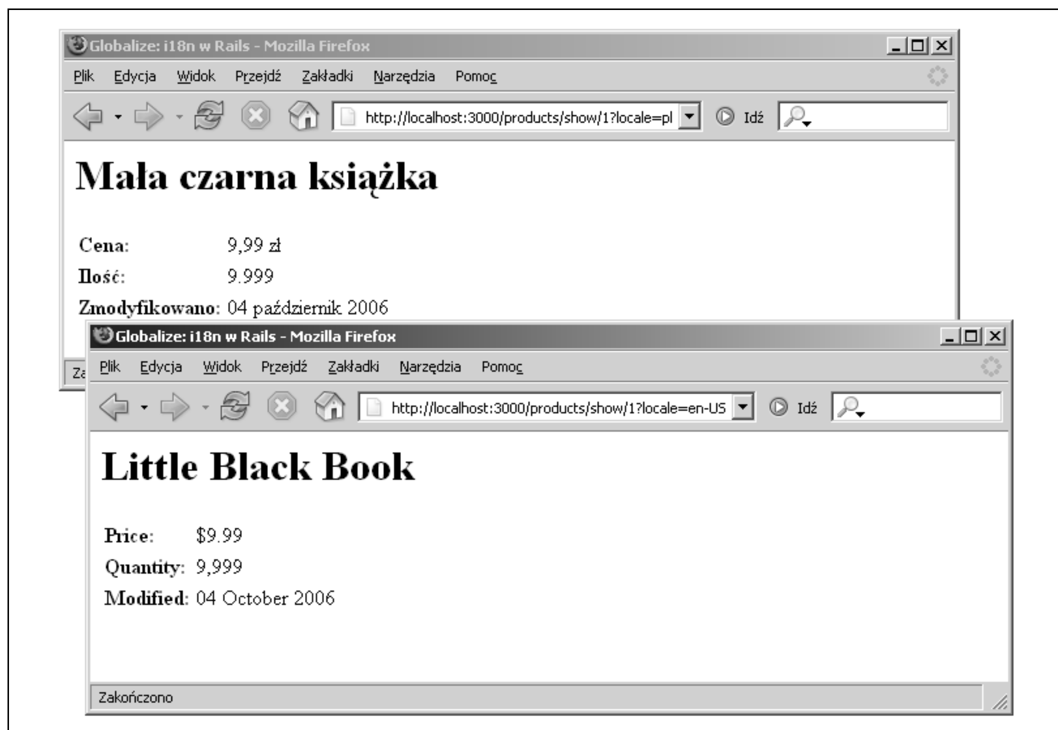
Teraz możemy przyjrzeć się naszej aplikacji. Uruchamiamy serwer produkcyjny:

```
$ ruby script/server -d
```

Zakładając, że serwer działa na porcie 3000, po wpisaniu do przeglądarki adresu *http://localhost:3000/products/show/l* ujrzymy wersję polską. Wersja angielska będzie widoczna pod adresem *http://localhost:3000/products/show/l?locale=en-US*.

Omówienie

Na rysunku 5.9 przedstawiono sposób określenia ustawień językowych za pomocą parametru w łańcuchu zapytania. Możemy też wykorzystać standardowy nagłówek HTTP `Accept-Language`. Wyraźnie wskazane parametry mają pierwszeństwo nad ustawieniami domyślnymi, a aplikacja zawsze powróci do `'pl'`, jeśli coś pójdzie nie tak.



Rysunek 5.9. Wielojęzyczna aplikacja Rails, wyświetlająca treść w języku polskim i angielskim

Możemy także wprowadzić ustawienia językowe jako parametr trasy przez zmodyfikowanie pliku *routes.rb* i zastąpienie trasy domyślnej.

config/routes.rb:

```
# Instalacja trasy domyślnej z niższym priorytetem
map.connect ':locale/:controller/:action/:id'
```

W takim przypadku strona z produktami w języku angielskim będzie dostępna pod adresem *http://localhost:3000/en-US/products/show/1*. Globalizacja wymaga pewnego wysiłku w każdym języku lub środowisku i choć Ruby nie obsługuje jeszcze poprawnie kodowania Unicode, to moduł Globalize ułatwia większość najczęstszych prac lokalizacyjnych.

Zobacz również

- Moduł rozszerzający GLoc, *http://www.agilewebdevelopment.com/plugins/gloc*,
- Moduł rozszerzający Localization Simplified, *http://www.agilewebdevelopment.com/plugins/localization_simplified*,
- Więcej informacji na temat modułu Globalize oraz przykłady jego użycia znajdziemy pod adresem *http://www.globalize-rails.org*,
- Dokumentacja do modułu Globalize jest również dostępna pod adresem *http://globalize.rubyforge.org*.