helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

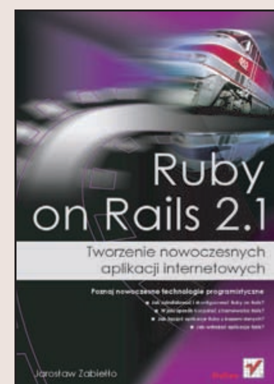
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Ruby on Rails 2.1. Tworzenie nowoczesnych aplikacji internetowych

Autor: Jarosław Zabiełło
ISBN: 83-246-0631-9
Format: 158x235, stron: 216



Poznaj nowoczesne technologie programistyczne

- Jak zainstalować i skonfigurować Ruby on Rails?
- W jaki sposób korzystać z frameworka Rails?
- Jak łączyć aplikacje Ruby z bazami danych?
- Jak wdrażać aplikacje Rails?

Język Ruby, opracowany w Japonii, długo pozostawał jedną z wielu tajemniczych „zabawek", przeznaczonych dla niewielkiej garstki pasjonatów. Jednak gdy w roku 2005 powstał framework Rails, technologia Ruby on Rails rozpoczęła swój triumfalny pochód przez świat internetu. Ruby on Rails to niesamowite narzędzie, przeznaczone do tworzenia witryn WWW i aplikacji sieciowych. Przyspiesza ono i upraszcza proces programowania, wdrożenia oraz rozwijania aplikacji. Ruby on Rails w ciągu kilku lat wyrosło na poważnego konkurenta języka PHP, zdobywając coraz więcej zwolenników.

Książka „Ruby on Rails. Tworzenie nowoczesnych aplikacji internetowych" to podręcznik, dzięki któremu poznasz tajniki programowania w Ruby i korzystania z frameworka Rails. Dowiesz się, jak zainstalować i skonfigurować RoR na stacji roboczej i serwerze sieciowym, jak zbudowana jest aplikacja tworzona w Ruby on Rails, czym jest model MVC i na czym polega programowanie adaptacyjne (agile). Poznasz elementy języka Ruby i mechanizmy Rails, nauczysz się tworzyć interfejs użytkownika dla aplikacji, implementować komunikację z bazami danych oraz wykorzystywać techniki programowania obiektowego. Przeczytasz o wbudowanych w Rails mechanizmach testowania, usuwaniu błędów, łączeniu kodu Ruby z językiem JavaScript i wdrażaniu aplikacji RoR w środowisku produkcyjnym.

- Podstawowe założenia Ruby on Rails
- Instalacja środowiska na stacji roboczej
- Najważniejsze elementy języka Ruby i frameworka Rails
- Struktura projektu w Ruby on Rails
- Tworzenie widoków za pomocą szablonów Haml/Sass
- Implementacja logiki aplikacji
- Praca z bazami danych
- RSpec i testy behawioralne (BDD)
- AJAX, jQuery i nieinwazyjny JavaScript
- Fusion Passenger i serwery asynchroniczne
- Praktyczna konfiguracja serwerów Nginx i Apache 2.x
- JRuby on Rails, Ruby 1.9, Rubinius, MagLev, Merb

Nadchodzi era Ruby on Rails. Bądź na nią przygotowany

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

Spis treści

Rozdział 1. Wprowadzenie 7
Czym jest Ruby on Rails? 8
Programowanie adaptacyjne i szybka pętla sprzężenia zwrotnego 9
Rails jako fronton do relacyjnej bazy danych 10
Podstawowe założenia 10
MVC 10
DRY 12
Konwencja ponad konfiguracją 12
Prowadzenie za rękę 13
Magia i konwencje 13
Instalacja 14
Ruby 14
RubyGems 14
Rails 15

Rozdział 2. Architektura Rails 17
Struktura katalogów 17
Moduły składające się na Rails 21
Tworzenie projektów RoR 22
Stworzenie projektu dla określonej wersji 23
Zamrożenie bieżącej wersji 23
Zamrożenie dowolnej wersji 23
Zamrożenie zewnętrznych gemów 23
Odmrożenie projektu 24
Model-Widok-Kontroler 24
Active Support 24
Rozszerzenie logiki boolowskiej Ruby'ego 24
Rozszerzenia dla napisów 25
Obsługa Unicode 25
Rozszerzenia dla liczb 26
Rozszerzenia dla dat i czasu 27
Serializacja do XML, JSON i YAML 28

Rozdział 3. Warstwa kontrolera 29
Generator kodu 29
Zawężanie dostępu do akcji 30
Renderowanie odpowiedzi dla serwera 30
Dowolna akcja 31
Dowolny plik 32

Dowolny tekst 32
Dowolny szablon 32
Włączanie podszablonów 33
Odpowiedzi zależne od nagłówka MIME 33
Filtry 34
before_filter 35
after_filter 35
around_filter 36
Cookies i sesje 37
Routing 39
map.connect 39
map.root 40
Reguły nazwane 40
map.with_options 40
Reguły ze znakami globalnymi 41
Sprawdzanie reguł 42
Helpery do generowania adresów 45
Zawężanie dozwolonych wartości 46

Rozdział 4. Warstwa modelu 47
Active Record 47
Dynamiczna konfiguracja połączenia 48
Połączenia do różnych baz 50
Równoczesna praca z wieloma bazami 54
Migracje 54
CRUD 59
Metody statystyczne 68
Relacje 68
Walidacje 77
Wywołania zwrotne (callbacks) 77
Magiczne pola 78
AR jako zewnętrzna biblioteka 78
Reguła „cienkiego” klienta i „wypasionego” modelu 79

Rozdział 5. Warstwa widoku 81
Szablony Rails 2.x i typy MIME 81
ERb 82
Unikanie niebezpiecznych danych 82
Zmienne w szablonach 83
Szablony wzorcowe i zagnieżdżone 84
Podszablony w pętli 86
Inne rodzaje szablonów 87
Haml i Sass 87
Erubis 90
Liquid 90
Markaby 90
MasterView 91

Rozdział 6. Cache 93
Pamięć operacyjna 94
Pliki na dysku 94
DRb 94
Memcached 94
Buforowanie całych stron 98
Buforowanie fragmentów szablonu 99

Rozdział 7. Debugowanie **101**
Logger i pliki logowania 101
Interaktywna konsola 102
Tryby pracy 103
Kolorowanie i dodatkowe ustawienia irb 103
Podglądanie w konsoli kodu SQL 103
Tryb piaskownicy 104
Debuger 104
Introspekcja obiektów 105
Firebug i Web Development — pluginy Firefoksa 105
Netbeans 6.x i debugowanie Rails w trybie graficznym 106

Rozdział 8. Pluginy **109**
Zarządzanie pluginami 110
Miejsce i kolejność ładowania 110
Instalacja 110
Piston 111
RaPT 112
Deinstalacja 112
Aktualizacja 112
Dostępna lista i źródła lokalizacji 113
Tworzenie własnych pluginów 113

Rozdział 9. Architektura REST **115**
Poza MVC 115
Interfejs REST 116
Tworzenie zasobu 116
Własne metody 117
Active Resource 118
Obsługa błędów 119
Uaktywnienie Action Web Service w Rails 2.x 120

Rozdział 10. Testowanie **121**
BDD vs. TDD 122
RSpec 122
Instalacja 123
Sposób budowania „oczekiwań” 123
Mocks & Stubs 127
RSpec User Stories, RCov i Autotest w akcji 131
Instalacja potrzebnych gemów 131
Instalacja potrzebnych pluginów 135
Konfiguracja połączeń do baz 135
Tworzenie baz 135
Scenariusze oczekiwań 136

Rozdział 11. JavaScript & Ajax **145**
Nieinwazyjny JavaScript 146
Obsługa zdarzeń JS 146
JS.ERB vs. JS.RJS 148
Alternatywne biblioteki JS 149
jQuery 149
MooTools 150
Yahoo YUI 150
ExtJS 151

Rozdział 12. Wersje międzynarodowe ... 153
Ri18n i GLoc ... 154
Globalize ... 155
Ustawienie kodowania UTF-8 ... 155
Instalacja ... 157
Konfiguracja ... 157
Dynamiczne przełączanie wersji językowej ... 158
Tłumaczenie szablonów ... 159
Tłumaczenie danych trzymanych w bazie ... 162
Część danych w bazie, część w plikach ... 164
Dodatkowe informacje ... 165

Rozdział 13. Wdrożenie ... 167
Apache + Fusion Passenger ... 167
Apache/Nginx + mongrel_cluster/Thin/Ebb ... 169
Serwery przetwarzające kod Ruby'ego ... 169
Serwery przetwarzające pliki statyczne ... 176
Win32 + mongrel_service ... 181
Gdy nie działa mongrel_service… ... 182

Rozdział 14. Edytory IDE ... 183
Textmate ... 183
Netbeans 6 ... 184
JEdit ... 185
Aptana Studio (RadRails) ... 187
Arachno IDE ... 187
Komodo 4 ... 188
RDE ... 189
Podsumowanie ... 190

Rozdział 15. Opcje Ruby'ego ... 191
Ruby 1.8 MRI ... 191
Ruby Enterprise ... 191
Ruby 1.9 + YARV ... 191
JRuby ... 192
Tworzenie pliku WAR ... 192
IronRuby ... 194
MacRuby ... 194
Rubinius ... 194
MagLev ... 195

Rozdział 16. Merb ... 197

Dodatek A Dodatkowe źródła informacji ... 201
Książki ... 201
Ruby ... 201
Rails ... 202
Internet ... 203

Skorowidz ... 205

Rozdział 3.
Warstwa kontrolera

Za obsługę warstwy kontrolera odpowiedzialny jest moduł Action Controller.

Generator kodu

Pliki kontrolerów można tworzyć ręcznie, ale znacznie wygodniej (i szybciej) jest użyć dostępnych generatorów kodu. Nie tylko wypełnią początkową treścią plik kontrolera, ale także wygenerują pliki do testów jednostkowych lub behawioralnych (w przypadku użycia RSpec). Przyjmuje się konwencję, że nazwa kontrolera powinna być rzeczownikiem.

```
$ script/generate controller NazwaKontrolera
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/nazwa_kontrolera
      exists  test/functional/
      create  app/controllers/nazwa_kontrolera_controller.rb
      create  test/functional/nazwa_kontrolera_controller_test.rb
      create  app/helpers/nazwa_kontrolera_helper.rb
```

W przypadku używania RSpec lepiej generować kontroler za pomocą metody `script/` ➥`generate rspec_controller NazwaKontrolera`. Oczywiście to wymaga zainstalowania dodatkowo pluginów RSpec, bo domyślnie Rails używa tylko Unit::Test. (Na temat RSpec zobacz rozdział *Testowanie → RSpec*).

Każdy kontroler składa się z klasy o nazwie z dodanym sufiksem `Controller`. Dla powyższego przypadku będzie to:

```
class NazwaKontroleraController < ApplicationController
end
```

Domyślnie wszystkie kontrolery dziedziczną po kontrolerze bazowym, klasie `Application` zdefiniowanej w pliku *app/controllers/application.rb*. Oznacza to, że wszystkie zdefiniowane tam metody (o ile nie są oznaczone jako `private`) są dostępne w każdym z pozostałych kontrolerów. Metody instancji klasy kontrolera w Rails nazywają się *akcjami*.

Zawężanie dostępu do akcji

Rails posiada kilka sposobów na zawężanie dostępu do akcji kontrolera. Ruby (podobnie jak Java) kontroluje zakres dostępu do metod za pomocą słów kluczowych `public`, `protected` i `private`. Domyślnie wszystkie akcje są publiczne i tylko takie akcje są widoczne z poziomu przeglądarki.

```
class ApplicationController < ActionController::Base
  def publiczna
    "Akcja dostępna bezpośrednio z poziomu URL
    dla wszystkich kontrolerów potomnych"
  end
  protected
  def chroniona
    "Akcja niedostępna bezpośrednio z adresu URL, ale dostępna
    dla wszystkich pozostałych kontrolerów potomnych."
  end
  private
  def prywatna
    "Akcja niedostępna bezpośrednio z adresu URL, ale
    tylko pozostałych metod/akcji bieżącego kontrolera."
  end
end
```

W Ruby, o ile jawnie nie użyto słowa kluczowego `return`, metody zwracają wartość ostatniego wyrażenia, więc można je pomijać.

Inna sprawa, że Rails, z przyczyn nie do końca jasnych, zamiast po prostu wysyłać do klienta wynik wyrażenia stworzonego w akcji (tak jak to robi Merb), robi to w sposób dużo bardziej skomplikowany. Także jawne wyświetlanie stringów za pomocą metody `puts` nie kieruje danych do serwera, ale co najwyżej do konsoli systemowej (o ile w ogóle odpalaliśmy aplikację RoR taką metodą).

Aby akcja zwracała odpowiedź do serwera, trzeba użyć specjalnej metody `render`. Na dodatek Rails dopuszcza *tylko jedno* wywołanie takiej metody w ramach tego samego żądania klienta. Trzeba na to uważać, metoda `render` nie zatrzymuje wykonywania dalszej części kodu, co może być pułapką dla początkujących. Dlatego najlepiej metodę `render` umieścić na samym końcu kodu albo postawić za nią komendę `return`. Próba podwójnego wywołania metody `render` wyrzuci wyjątek `ActionController::Double` ↪`RenderError`.

Renderowanie odpowiedzi dla serwera

Rails stosuje szereg konwencji podczas generowania odpowiedzi do klienta. Załóżmy, że mamy adres *http://localhost:3000/about/me* i że wykorzystujemy standardowe ustawienia reguł routingu (na temat routingu zobacz w rozdziale *Warstwa kontrolera → Routing*).

Standardowe ustawienia routingu rozkładają podany adres na:

```
params = {:controller => 'about', :action => 'me'}
```

Kontroler `about` to klasa `AboutController`[1]. Jeśli wewnątrz kontrolera nie ma zdefiniowanej akcji `me`, Rails próbuje odszukać odpowiadający jej szablon *tak, jakby akcja była zdefiniowana*. W tym przypadku szablon to plik *app/views/about/me.html.erb* (domyślnie używany jest ERb). Szablon zostanie przetworzony i jego wynik jest wysyłany przez serwer do przeglądarki klienta.

W przypadku kiedy mamy zdefiniowaną akcję, Rails najpierw stara się wykonać zawarty tam kod. Jeśli nie ma wewnątrz wywołania metody `render`, `respond_to` czy `send_data`, Rails przechodzi do domyślnego szablonu i jego wynik jest zwracany do klienta. Szablon ma oczywiście dostęp do wszystkich zmiennych instancji swego kontrolera (zobacz rozdział „Warstwa widoku").

Z praktycznego punktu widzenia[2] istotne są dwie metody, których formalna definicja to:

```
render(opcje = nil, dodatkowe_opcje = {}, &blok_kodu) protected
render_to_string(opcje = nil, &blok_kodu) protected
```

Metoda `render_to_string` działa tak samo jak `render` z tym, że nie generuje odpowiedzi do przeglądarki, lecz zwraca ją jako wynik swego działania.

Zarówno `opcje`, jak i `opcje_dodatkowe` są obiektem typu Hash. Dodatkowe opcje to m.in. `:layout`, `:status`. Opcja `:layout` związana jest z możliwością umieszczania wyniku szablonu w kontekście tzw. szablonu wzorcowego (layoutu).

```
render :action => 'akcja2', :layout => false # nie używaj layoutu
render :action => 'akcja2', :layout => true # użyj domyślny layout
render :action => 'akcja2', :layout => 'inny_layout' # użyj jakiś inny layout
```

Dokładniej temat layoutu jest omawiany w rozdziale *Warstwa widoku → ERb → Szablony wzorcowe i zagnieżdżone*.

```
:status umożliwia zwracanie kodu nagłówka HTTP.
```

Listę dostępnych kodów i ich znaczenie można znaleźć na stronie *http://www.w3.org/ ➥Protocols/rfc2616/rfc2616-sec10.html*.

Dowolna akcja

Jawne podanie parametru `:action` pozwala na renderowanie odpowiedzi z innej akcji w ramach tego samego kontrolera. Generowanie odpowiedzi z innego kontrolera nie jest możliwe (opcja `:controller` nie działa).

```
render :action => 'akcja2'
```

---

[1] Zdefiniowana jest w pliku *app/controllers/about_controller.rb*.

[2] Dostępna metoda `render_component` jest w zasadzie niezalecana, tak jak i cały niewydajny mechanizm komponentów, jaki wprowadził Rails 1.0.

Jak można sprawdzić w poniższym przykładzie, `render` *nie uruchamia* w ogóle metody `akcja2`, ale jedynie przetwarza *przypisany jej domyślnie szablon* (wymusiliśmy użycie innego szablonu za pomocą opcji `:action`).

Innymi słowy, jeśli chcemy przekazać jakiekolwiek zmienne do szablonu `akcja.html.erb`, musimy je zdefiniować w metodzie `akcja1`, a nie `akcja2`. Metoda `render` pozwala na używanie różnych szablonów, ale zawsze w kontekście konkretnej akcji określonej regułami routingu.

```
class Kontroler1Controller < ApplicationController
  def akcja1
    @gdzie = 'akcja1'
    render :action => 'akcja2', :layout => false
  end
  def akcja2
    @gdzie = 'akcja2'
  end
end
```

Dowolny plik

```
render :file => Rails.root + '/app/views/kontroler1/plik.html.erb'
render :file => 'kontroler1/plik.html.erb', :use_full_path => true
```

Powyższe metody są równoważne. W drugim przypadku nie trzeba podawać bezwzględnej ścieżki do pliku. Renderowany plik jest przetwarzany zgodnie z regułami tego, czym jest. W tym wypadku jest to szablon ERb (zobacz w rozdziale *Warstwa widoku → ERb*).

Dowolny tekst

```
render :text => 'dowolna treść'
```

W ten sposób można wyświetlać dowolny obiekt typu `String`. Czasami używa się tej metody do szybkiego sprawdzenia wartości obiektu poprzez wyświetlenie jego zawartości w przeglądarce.

```
render :text => obiekt.inspect, :layout => false
return # dalej nic nie będzie wykonane
```

Kod można umieścić gdziekolwiek w akcji kontrolera. Należy tylko pamiętać, aby przerwać dalsze jego przetwarzanie za pomocą `return`, bo metoda `render` nie zatrzymuje wykonywania reszty kodu.

Dowolny szablon

```
render :template => "kontroler1/akcja2"
```

Powyższy kod działa podobnie do `render(:action...)`, z tą różnicą, że można użyć dowolny, a nie domyślny szablon jakiejś innej akcji w ramach tego samego kontrolera.

Włączanie podszablonów

```
render :partial => 'podszablon'
```

Podszablony (zwane też szablonami zagnieżdżonymi lub cząstkowymi; z ang. *partials*) to w zasadzie zwykłe szablony, z tą różnicą, że ich pliki muszą posiadać nazwę *zaczynającą się od znaku podkreślnika*. Zwykle są one używane nie z poziomu kontrolera, lecz jakiegoś innego szablonu. Podszablon może włączać kolejne podszablony, choć nie jest to zalecane. Zbyt zagnieżdżone ładowanie kolejnych podszablonów świadczy zwykle o źle przemyślanej strukturze aplikacji. Podszablony posiadają kilka dodatkowych opcji, m.in. można przekazywać im parametry formalne.

```
render :partial => 'podszablon', :locals => hasz
```

Parametr przekazany do klucza `:locals` jest obiektem typy Hash. Jego klucze zostaną zamienione w podszablonie na zmienne lokalne. Przykładowo dla `:locals => {:msg => 'Hello'}` można w szablonie napisać `<%= msg %>`, gdyż klucz `:msg` zostanie zamieniony na zmienną lokalną o wartości `Hello`. Dodatkowe opcje dostępne dla metody `render` są opisane w API na stronie ***http://api.rubyonrails.org/classes/ActionController/ ↳Base.html#M000848***.

Odpowiedzi zależne od nagłówka MIME

Akcja kontrolera w Rails potrafi zwrócić różne odpowiedzi w zależności od przekazanego nagłówka MIME (konkretnie chodzi o zawartość zmiennej `Content-type` w nagłówku HTTP). W takim przypadku, mimo że odpalana jest ta sama akcja, Rails szuka dla niej szablonu zgodnie z zadeklarowanym nagłówkiem MIME. Inaczej mówiąc, można przypisać tej samej akcji kontrolera wiele różnych szablonów uruchamianych w zależności od kontekstu wywołania. Pamiętając, że domyślna konfiguracja Rails zawiera regułę `map.connect(':controller/:action/:id.:format')`, można wymusić wyświetlenie różnych szablonów w zależności od adresu URL.

Ścieżka: `/ctrl/act/any.html`

```
params = {:controller=>'ctrl', :action=>'act', :id=>'any', :format=>'html'}
```

Użyty zostanie jeden ze znalezionych szablonów:

```
app/views/ctrl/act.html.erb
app/views/ctrl/act.html.haml
…
app/views/ctrl/act.html.cokolwiek
```

Ścieżka: `/ctrl/act/any.xml`

```
params = {:controller=>'ctrl', :action=>'act', :id=>'any', :format=>'xml'}
```

Użyty zostanie jeden ze znalezionych szablonów:

```
app/views/ctrl/act.xml.builder
app/views/ctrl/act.xml.erb
…
app/views/ctrl/act.xml.cokolwiek
```

Pierwsze rozszerzenie po kropce w nazwie szablonu oznacza typ MIME. Drugie rozszerzenie dotyczy sposobu, w jaki będzie traktowany cały plik. Domyślnie jest to ERb, ale może to być dowolny inny silnik rednerujący szablon. Na przykład jeśli ktoś się uprze, to szablony wykorzystywane przez Ajaksa będą renderowane przez Haml (zobacz w rozdziale *Warstwa widoku → Inne rodzaje szablonów → Haml i Sass*). Wtedy szablon będzie musiał tylko mieć nazwę pliku `act.js.haml`. Dzięki filtrom Hamla nie jest to wcale taki głupi pomysł. Umieszczona niżej funkcja o krótkiej nazwie `j` to dodany w Rails 2.1 krótszy zapis dla `escape_javascript`.

```
:js
  if ($('#my_div')) {
    = "$('#my_div').html('#{j @message}');" # jQuery
  }
```

Rails pozwala także zdefiniować *swój własny* typ nagłówka MIME i dla niego używać specjalnie do tego celu przygotowany szablon. Na przykład chcemy, aby nasza aplikacja potrafiła renderować treść w formacie WAP/WML używanym przez telefony komórkowe. W tym celu musimy poinstruować Rails, że chcemy zarejestrować dodatkowe typy MIME.

W pliku *config/environment.rb* należy wstawić

```
Mime::Type.register "text/vnd.wap.wml", :wml
```

Po zrestartowaniu serwera, dla adresu */ctrl/act/any.wml*, Rails powinien szukać szablonu *app/views/ctrl/act.wml.erb*.

W przypadku jeśli chcemy nie tylko używać kilku różnych szablonów, ale także wykonać jakiś dodatkowy specyficzny dla nich kod, należy użyć metody `respond_to`.

```
def akcja1
  @gdzie = 'akcja1'
  # wspólny kod
  respond_to do |format|
    format.html do
       # dodatkowy kod dla szablonu
       # akcja1.html.erb lub akcja1.html.haml
    end
    format.xml # akcja1.xml.erb, akcja1.xml.builder, etc.
    format.js # akcja1.js.erb lub akcja1.js.rjs
    format.wml do
      # dodatkowy kod dla szablonu akcja1.wml.erb
    end
  end
end
```

Filtry

Rails udostępnia trzy rodzaje filtrów pozwalających na wykonanie dodatkowych operacji przed lub po akcji kontrolera.

before_filter

Metoda `before_filter` odpala po kolei wszystkie metody podane na liście, zanim wywoła akcję, jaka wynika z adresu URL i reguł routingu. Metody te zostaną wywołane we wczesnym etapie, zanim zostanie uruchomiony kod w akcji kontrolera. Najczęściej stosowane są do implementacji mechanizmu chroniącego dostęp do wybranych akcji kontrolera (dostępny jest obiekt `session`). Można też w tym miejscu wychwytywać domyślny język ustawiony w przeglądarce, aby automatycznie wyświetlać serwis w odpowiednim języku.

Dostępne są też dodatkowe opcje `:only` i `:except`.

```
# nie sprawdzaj autoryzacji dla metod index i show:
before_filter :autoryzuj, :except => [:index, :show]
# nie pozwól, aby można było uruchomić metodę delete bez autoryzacji:
before_filter :autoryzuj, :only => :delete
```

after_filter

Działa tak samo jak powyższy `before_filter`, z tą różnicą, że wywołuje metody *po* wykonaniu kodu bieżącej akcji kontrolera, lecz jeszcze przed wysłaniem jakiejkolwiek odpowiedzi do klienta. Można tu zmieniać nagłówki HTTP, dodać kompresję generowanej odpowiedzi itp.

```
class ApplicationController < ActionController::Base
  # Ustalam, że aplikacja będzie pracować tylko w poniższych językach
  # Wszystkie możliwe opcje kodu języka są na stronie:
  # http://www.w3.org/TR/REC-html40/struct/dirlang.html#langcodes
  # http://blogs.law.harvard.edu/tech/stories/storyReader$15
  @@allowed_languages = ['pl', 'en']

  before_filter :set_locale
  after_filter :content_type

  # Nie chcę, aby do poniższych metod był dostęp z innych kontrolerów
  private

  # Wymuszam w przeglądarce kodowanie utf-8. Takie ustawienie
  # nagłówków jest znacznie *silniejsze* od samego znacznika HTML:
  # <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  def content_type
    content_type = headers['Content-Type'] || 'text/html'
    if content_type =~ /^text/
      headers['Conent-Type'] = "#{content_type}; charset=utf-8"
    end
  end

  def set_locale
    # Wyciągam z przeglądarki kod pierwszego języka na liście
    begin
      @language = request.env['HTTP_ACCEPT_LANGUAGE'].split(',')[0].split(';')[0]
    rescue
      @language = 'en'
    end
```

```
# Jeśli przeglądarka używa jakiegoś języka, który nie jest
    # na liście dozwolonych, to ustawiam domyślnie język angielski
    if !@@allowed_languages.include?(@language)
      @language = 'en'
    end

    # W tym miejscu jest włączana wersja językowa
    # (przykład dla pluginu Globalize)
    Locale.set @language

    # Alternatywne rozwiązanie
    #
    #begin
    #  Locale.set params[:locale]
    #rescue ArgumentError
    #  redirect_to params.merge('locale' => @@default_language)
    #end
    #
    # lub lepiej:
    #
    #@language = params[:locale] # język wyłuskujemy z adresu URL
    #if !@@allowed_languages.include?(@language)
    #  redirect_to params.merge('locale' => 'en')
    #else
    #  Locale.set @language
    #end
  end

end
```

around_filter

Nowy filtr dodany w Rails 2.1. Działa podobnie jak poprzednie dwa użyte równocześnie przed i po wywołaniu kodu bieżącej akcji kontrolera.

```
class HomeController < ApplicationController
  around_filter :akcja1, :akcja2

  around_filter do |controller, action|
    logger.info "blok kodu przed akcją"
    action.call
    logger.info "blok kodu po akcji"
  end

  def index
    logger.info "AKCJA"
  end

  private # lepiej aby filtry były dostępne bezpośrednio z poziomu URL

  def akcja1
    logger.info "AKCJA1 przed yield"
    yield "AKCJA1"
    logger.info "AKCJA1 po yield"
  end
```

```
    def akcja2
      logger.info "AKCJA2 przed yield"
      yield "AKCJA2"
      logger.info "AKCJA2 po yield"
    end
  end
```

Wynik w logu:

```
AKCJA1 przed yield
AKCJA2 przed yield
blok kodu przed akcją
AKCJA
Rendering home/index
blok kodu po akcji
AKCJA2 po yield
AKCJA1 po yield
```

Cookies i sesje

Protokół HTTP używany przez aplikacje webowe z definicji jest bezstanowy. To znaczy, że każdy kolejny request jest niezależny i nie wie nic na temat poprzedniego wywołania. Dzięki temu HTTP jest szybki i bardzo dobrze się skaluje. Jednakże gdyby nie istniały jakieś specjalne metody zapewnienia ciągłości pracy pomiędzy kolejnymi przeładowaniami przeglądarki, nie byłoby możliwe zbudowanie żadnego sklepu internetowego ani jakiejkolwiek poważniejszej aplikacji.

Istnieją różne metody dodania „stanu” do bezstanowego HTTP, od mniej popularnych rozwiązań stosowanych w serwerach kontynuacyjnych (np. Seaside[3] napisany w języku Smalltalk) po klasyczne, bardziej powszechne rozwiązania oparte na sesjach. Rails korzysta z sesji.

Sesja to informacja unikalna dla danego klienta. Zwykle trzymana jest po stronie serwera. W Rails 2.1 podjęto kontrowersyjną decyzję, by domyślnie sesje oprzeć na cookies. Takie rozwiązanie jest bardzo szybkie, gdyż cała informacja unikalna dla użytkownika jest przechowana na jego komputerze w formie malutkich plików zwanych cookies („ciasteczkami”). Z drugiej strony takie rozwiązanie posiada szereg wad, z których dwie wydają się być najważniejsze. Cookies mają ograniczenia na wielkość możliwej do zapisu informacji po stronie klienta. Jest to tylko 4KB. Drugą sprawą jest kwestia słabego bezpieczeństwa takiego rozwiązania. Cookies można łatwo ukraść za pomocą spreparowanego kodu JavaScript. Złodziej może podsunąć nam stronę, na której za pomocą JavaScript będzie mógł odczytać cookies wraz z wszystkimi ostatnio zapisanymi tam informacjami. Co prawda Rails potrafi zabezpieczyć się przed podstawieniem zmodyfikowanych danych z cookies, ale już nie ma możliwości zabezpieczyć się przed odczytem samych danych. W sumie, moim zdaniem, lepszym rozwiązaniem jest przechowywać dane sesji po stronie serwera HTTP, a cookies używać tylko do tego, by zapisać tam unikalny identyfikator sesji, dzięki któremu serwer będzie w stanie powiązać ze sobą poszczególne żądania klienta.

---

[3] *http://seaside.st/.*

Przykładowe użycie `sesji` i `cookies`, a także `params` i `flash`.

```
def login
  # jeśli wysłano formularz logowania
  if request.post?
    # czy użytkownik istnieje w bazie?
    user = User.authorized(params[:login], params[:passwd])
    if user
      # aktualizuj przypisane mu dane o używanej przeglądarce
      user.user_agent = request.user_agent
      # oraz o adresie IP z jakiego się łączy
      user.remote_addr = request.remote_addr
      # a także zapisz, kiedy ostatnio się logował
      user.last_logged_in = Time.now
      user.save!
      # dodaj informację do sesji, że logowanie było udane
      session[:user_id] = user.id
      # jeśli w formularzu wybrano opcję zapamiętania hasła
      if params[:remember]
        # to użyj cookies, aby nie trzeba było się
        # logować przez najbliższą godzinę
        cookies[:user_id] = { :value => session[:user_id].to_s,
                              :expires => 1.hour.from_now }
      end
      redirect_to '/welcome'
    else
      flash[:info] = "Incorrect password or/and login."
    end
  end
end

def logout
  # przy wylogowaniu usuń sesję
  session[:user_id] = nil
  # wyczyść cookies
  cookies.delete :user_id
  # oraz zatrzymaj się na stronie logowania
  redirect_to '/login'
end
```

We wczesnych wersjach Rails 1.x do obsługi sesji i cookies używano zmiennych `@session` i `@cookies`. Rails 2.x wymaga, by używać obiektów `session` i `cookies`. Dane z formularza są dostępne w haszu `params`.

Słowo komentarza należy się zmiennej `flash`. Jest to obiekt typu Hash przechowujący dane przez czas trwania *jednego requestu*. Przy kolejnym przeładowaniu strony dane te ulegają zniszczeniu. Obiektu `flash` używa się zwykle do gromadzenia informacji o błędach w formularzu.

Routing

Routing to wchodzący w skład Active Support mechanizm generowania i translacji adresów URL. Pozwala nie uciekać się do skomplikowanych reguł routingu modułu `mod_rewrite` na zewnętrznym serwerze HTTP.

W rzeczy samej, adresy URL są tylko wirtualnymi ścieżkami serwera i nie muszą mieć żadnego związku ze strukturą i nazwami plików po stronie serwera. Tak popularne stosowane w aplikacji adresy typu *http://serwer/kontakt.php* czy *http://serwer/about.*
➥*aspx?a=1&y=blah* nie dość, że wyglądają paskudnie, to na dodatek są niepotrzebnie zbyt jawnie związane z konkretną technologią po stronie serwera. Co zrobić, jeśli ktoś chciałby przepisać tak napisany serwis z PHP/ASP do innej technologii? Wszystkie wcześniejsze adresy z końcówkami *.php* ulegną zmianie i stracimy kontakt z tymi wszystkimi, którzy dodali link do naszego serwisu. Powiem wprost, jawne eksponowanie technologii server-side jest oznaką *lamerstwa*. Współczesne serwisy internetowe nie używają rozszerzeń lub stosują neutralny sufiks *.html*. Wbudowany w Rails system routingu pozwala na generowanie czytelnych i obojętnych technologicznie adresów URL. Nawet jeśli w przyszłości ktoś zechce przepisać swoją aplikację z Rails do czegoś innego, to wszystkie adresy URL zostaną bez zmian.

Rails zapewnia nie tylko generowanie przyjaznych adresów URL za pomocą dostarczanych helperów, ale także elastyczny i elegancki ich rozbiór do zmiennych z przekazaniem sterowania do właściwej akcji kontrolera.

Wszystkie reguły routingu Rails trzymane są w pliku *config/routes.rb*. Każdy nowo stworzony projekt Rails posiada (automatycznie) stworzone dwie reguły:

```
map.connect '/:controller/:action/:id'
map.connect '/:controller/:action/:id.:format'
```

map.connect

Ogólna składnia dla pojedynczej reguły routingu to:

```
map.connect reguła [,hasz_z_dodatkowymi_parametrami]
```

Powyższe definicje oznaczają, że adres *http://www.example.com/a/b/c.d* spowoduje utworzenie zmiennej `params` o wartości `{:controller => 'a', :action => 'b', :id => 'c', :format => 'd'}`. Zmienna ta jest dostępna globalnie we wszystkich kontrolerach. Z kolei klucze `:controller` i `:action` decydują o tym, gdzie zostanie przekazane sterowanie programu (w tym przypadku dalsze wykonanie kodu trafi do metody `b` w klasie `AController` znajdującej się w pliku *app/controllers/a_controller.rb*).

W definicji reguł routingu nie ma żadnej magii. Wszystko to, co w adresie URL pasuje do symbolu Ruby'ego (identyfikator ze znakiem dwukropka na początku), zostanie dorzucone do hasza `params`.

```
map.connect '/prefix/:name/:msg'
http://www.example.com/prefix/kowalski/hello_world
params = {:name => 'kowalski', :msg => 'hello_word'}
```

Oczywiście reguła nieokreślająca kontrolera ani akcji jest mało przydatna. Dlatego w tym przypadku należy dodać brakujące elementy hasza.

```
map.connect '/prefix/:name/:msg', :controller => 'users', :action => 'msg'
```

map.root

Regułę dla głównego adresu `'/'` można ustawić albo za pomocą

```
map.root :controller => 'home'
```

lub

```
map.connect '', :controller => 'home'
```

Jeśli domyślna akcja nosi nazwę `'index'`, to można ją pominąć w definicji routingu.

Każdy nowo stworzony projekt zawiera plik *public/index.html*, który przesłania `map.root`. Pierwszą czynnością powinno być zatem jego skasowanie.

Reguły nazwane

Reguły routingu mogą posiadać swoje własne nazwy. Ogólna składnia:

```
map.nazwa reguła [,hasz_z_dodatkowymi_parametrami]
```

Dla reguł posiadających swoją nazwę Rails automatycznie generuje dwa helpery: `nazwa_url` oraz `nazwa_path`. Można (i należy) ich używać zamiast `url_for`, bo są nie tylko krótkie i czytelne, ale także działają znacznie szybciej.

```
map.admin 'admin', :controller => 'admin', :action => 'about'
admin_path # => /admin/about
admin_url # => http://localhost:3000/admin/about

redirect_to 'http://localhost:3000/admin/about/test'
redirect_to url_for :controller => 'admin', :action => 'about', :id => 'test'
# to samo krócej:
redirect_to admin_url :id => 'test'
```

Wspomniany wyżej `map.root` jest szczególnym przypadkiem reguły o nazwie `root`.

```
map.root :controller => 'home', :action => 'info'
root_path # => '/home/info'
root_url # => 'http://localhost:3000/home/info'
```

map.with_options

Służy do krótszego zapisu kilku powtarzających się reguł. Poniższe trzy reguły posiadają wspólny kontroler `main`.

```
map.about 'about', :controller => 'main', :action => 'about'
map.faq 'faq', :controller => 'main', :action => 'faq'
map.products 'products', :controller => 'main', :action => 'products'
```

Można je zgrupować:

```
map.with_options :controller => 'main' do |x|
  x.about 'about', :action => 'about'
  x.faq 'faq', :action => 'faq'
  x.products 'products', :action => 'products'
end
```

Reguły ze znakami globalnymi

W języku Ruby znak * występujący na początku obiektu Array wyłuskuje wszystkie jego elementy.

```
def fun(a, b, *others)
  puts "#{a}, #{b}, #{others.inspect}"
end
fun 1,2,3,4,5,6 # => "1, 2, [3, 4, 5, 6]"
```

W przypadku routingu zastosowanie wzorca `*nazwa` pozwala na wyłuskanie całej listy parametrów. Ważne jest tylko, by taki wzorzec występował na samym końcu reguły.

Dla reguły

```
map.connect  '/app/*tree'
```

wszystkie elementy adresu URL występujące po prefiksie `/app/` zostaną przechwycone do zmiennej `params[:tree]`.

Przykłady:

```
URL: '/tree/one'
params[:tree] # => ['one']

URL: '/tree/two/three'
params[:tree] # => ['two', 'three']

URL: '/tree/four/five/six'
params[:tree] # => ['four', 'five', 'six']

URL: '/tree/seven/eight'
params[:tree] # => ['seven', 'eight']
```

Szczególnym przypadkiem jest przechwytywanie wszystkich adresów niepasujących do żadnej ze zdefiniowanych wcześniej reguł. W takiej sytuacji cała reguła powinna składać się wyłącznie z krótkiego łańcucha znaków `*nazwa`. Należy też pamiętać, że taka reguła musi być ostatnia na liście, bo inaczej żadna następna reguła nie zostanie wykonana — `*nazwa` wyłapuje wszystkie możliwe adresy URL!

```
map.connect '*any', :controller => 'trap', :action => 'unrecognized'
```

Sprawdzanie reguł

Załóżmy, że w *config/routes.rb* mamy definicje:

```
map.root :controller => "welcome"
map.resources :books, :collection => { :recent => :get, :release => :put }
map.connect 'articles/:year/:month/:day',
            :controller => 'articles',
            :action     => 'find_by_date',
            :year       => /\d{4}/,
            :month      => /\d{1,2}/,
            :day        => /\d{1,2}/
map.connect ':controller/:action/:id'
map.connect ':controller/:action/:id.:format'
map.connet '*any', :controller => 'main', :action => 'unrecognized'
```

Wszystkie reguły routingu można podejrzeć z konsoli lub z poziomu komendy Rake'a.

```
$ script/console
Loading development environment (Rails 2.1.0)
>> rs = ActionController::Routing::Routes
>> puts rs.routes
ANY    /                             {:action=>"index",:controller=>"welcome"}
PUT    /books/release/               {:action=>"release",:controller=>"books"}
PUT    /books/release.:format/       {:action=>"release",:controller=>"books"}
GET    /books/recent/                {:action=>"recent", :controller=>"books"}
GET    /books/recent.:format/        {:action=>"recent", :controller=>"books"}
GET    /books/                       {:action=>"index", :controller=>"books"}
GET    /books.:format/               {:action=>"index", :controller=>"books"}
POST   /books/                       {:action=>"create", :controller=>"books"}
POST   /books.:format/               {:action=>"create", :controller=>"books"}
GET    /books/new/                   {:action=>"new", :controller=>"books"}
GET    /books/new.:format/           {:action=>"new", :controller=>"books"}
GET    /books/:id/edit/              {:action=>"edit", :controller=>"books"}
GET    /books/:id/edit.:format/      {:action=>"edit", :controller=>"books"}
GET    /books/:id/                   {:action=>"show", :controller=>"books"}
GET    /books/:id.:format/           {:action=>"show", :controller=>"books"}
PUT    /books/:id/                   {:action=>"update", :controller=>"books"}
PUT    /books/:id.:format/           {:action=>"update", :controller=>"books"}
DELETE /books/:id/                   {:action=>"destroy",:controller=>"books"}
DELETE /books/:id.:format/           {:action=>"destroy",:controller=>"books"}
ANY    /articles/:year/:month/:day/  {:action=>"find_by_date",
                                      :controller=>"articles"}
ANY    /:controller/:action/:id/     {}
ANY    /:controller/:action/:id.:format/ {}
ANY    /:any/                        {:action=>"unrecognized",
:controller=>"main"}
```

Przewagą użycia Rake'a jest to, że zostaną wyświetlone nazwy dynamicznie stworzonych helperów.

```
$ rake routes
                  root        /                             {:action=>...
          recent_books GET    /books/recent                 {:action=>...
 formatted_recent_books GET   /books/recent.:format         {:action=>...
         release_books PUT    /books/release                {:action=>...
```

```
formatted_release_books PUT    /books/release.:format         {:action=>...
                  books GET    /books                         {:action=>...
        formatted_books GET    /books.:format                 {:action=>...
                        POST   /books                         {:action=>...
                        POST   /books.:format                 {:action=>...
               new_book GET    /books/new                     {:action=>...
     formatted_new_book GET    /books/new.:format             {:action=>...
              edit_book GET    /books/:id/edit                {:action=>...
    formatted_edit_book GET    /books/:id/edit.:format        {:action=>...
                   book GET    /books/:id                     {:action=>...
         formatted_book GET    /books/:id.:format             {:action=>...
                        PUT    /books/:id                     {:action=>...
                        PUT    /books/:id.:format             {:action=>...
                        DELETE /books/:id                     {:action=>...
                        DELETE /books/:id.:format             {:action=>...
                               /articles/:year/:month/:day    {:action=>...
                               /:controller/:action/:id
                               /:controller/:action/:id.:format
                 connect       /:any                          {:action=>...
```

Jednak zaletą konsoli jest możliwość interaktywnego testowania reguł.

Rozpoznawanie ścieżek

Do rozpoznawania ścieżek służy metoda `recognize_optimized`.

W stosunku do starszej metody `recognize_path` wymagane jest podanie metody HTTP. Dostępne symbole to `:get`, `:put`, `:post`, `:delete` oraz `:any`[4].

```
>> rs.recognize_optimized '/', :method => :get
=> {:action=>"index", :controller=>"welcome"}
>> rs.recognize_optimized '', :method => :any
=> {:action=>"index", :controller=>"welcome"}
>> rs.recognize_optimized '/articles/2008/04/15', :method => :get
=>{:day=>"15", :action=>"find_by_date", :month=>"04", :year=>"2008",
:controller=>"articles"}
>> rs.recognize_optimized '/zly/adres', :method => :any
=> {:action=>"unrecognized", :any=>["zly", "adres"], :controller=>"main"}
```

Aby oszczędzić trochę czasu na pisaniu, skorzystam tu z pewnego tricku polegającego na uruchomieniu irb z poziomu konsoli irb. Dzięki temu uzyskujemy od razu dostęp do przestrzeni nazw metod routingu.

```
>> irb ActionController::Routing::Routes
>> recognize_optimized '/', :method => :get
=> {:action=>"index", :controller=>"welcome"}
```

Dla `new_book GET /books/new {:action=>"new", :controller=>"books"}`

```
>> recognize_optimized '/books/new', :method => :get
=> {:action=>"new", :controller=>"books"}
```

[4] Symbol `:any` nie oznacza metody `ANY`, bo taka nie istnieje w specyfikacji protokołu HTTP. Rails używa tego symbolu, by zmapować adres z dowolną metodą: `GET`, `POST`, `PUT` lub `DELETE`.

Dla `edit_book GET /books/:id/edit {:action=>"edit", :controller=>"books"}`

```
>> recognize_optimized '/books/123/edit', :method => :get
=> {:action=>"edit", :controller=>"books", :id=>"123"}
>> generate :use_route => 'edit_book', :id => 123
=> "/books/123/edit"
```

Dla `PUT /books/release/ {:action=>"release", :controller=>"books"}`

```
>> recognize_optimized '/books/release', :method => :put
=> {:action=>"release", :controller=>"books"}
```

Zauważmy, że użycie opcji `:any` nie wystarczy, aby reguła zadziałała, gdyż została ona zdefiniowana wcześniej z jawną deklaracją `:put`.

```
>> recognize_optimized '/books/release', :method => :any
=> {:action=>"unrecognized", :any=>["books", "release"], :controller=>"main"}
```

Tego problemu nie mają reguły, w których nie zadeklarowano jawnie metody HTTP.

Dla `ANY / {:action=>"index", :controller=>"welcome"}`

```
>> recognize_optimized '/', :method => :any
=> {:action=>"index", :controller=>"welcome"}
>> recognize_optimized '/', :method => :get
=> {:action=>"index", :controller=>"welcome"}
>> recognize_optimized '/', :method => :put
=> {:action=>"index", :controller=>"welcome"}
>> recognize_optimized '/', :method => :delete
=> {:action=>"index", :controller=>"welcome"}
>> recognize_optimized '/', :method => :post
=> {:action=>"index", :controller=>"welcome"}
```

W konsoli można uzyskać także bezpośredni dostęp do dynamicznie stworzonych metod dla nazwanych reguł routingu. Najpierw trzeba załadować odpowiednią przestrzeń nazw i zdefiniować opcję serwera (nazwa może być dowolna).

```
>> include ActionController::UrlWriter
=> Object
>> default_url_options[:host] = 'moj_serwer'
=> "moj_serwer"
>> edit_book_url(123)
=> "http://moj_serwer/books/123/edit"
>> edit_book_url :id => 123
=> "http://moj_serwer/books/123/edit"
>> edit_book_path(123)
=> "/books/123/edit"
>> edit_book_path :id => 123
=> "/books/123/edit"
>> formatted_book_path 123, :html
=> "/books/123.html"
>> formatted_book_path :format => :html, :id => 123
=> "/books/123.html"
```

Generowanie ścieżek

Do testowania w konsoli routingu „w drugą stronę" służy metoda generate.

```
>> generate :controller => 'welcome', :action => 'index'
=> "/"
>> generate :controller => 'welcome'
=> "/"
>> generate :controller => '/'
=> "/"
>> generate :controller => ''
>> generate :use_route => 'root'
=> "/"
>> generate :use_route => 'edit_book', :id => 123
=> "/books/123/edit
>> generate :use_route => 'release_books'
=> "/books/release"
>> generate :controller => 'articles', :action => 'find_by_date', :year => '2007',
➥:month => '11', :day => '13'
=> "/articles/2007/11/13"
```

Testowanie ścieżek

Do testowania ścieżek służą metody assert_generates, assert_recognizes i assert_
➥routing.

```
assert_generates '/articles/2007/11/13', {:controller => 'articles', :action =>
➥'find_by_date', :year => '2007', :month => '11', :day => '13'}
assert_generates '/books/123/edit', {:use_route => 'edit_book'}, {:id => 123}
assert_recognizes({:controller=>'books', :action=>'release', }, {:path =>
'books/release', :method => :put})
```

Helpery do generowania adresów

Ręczne wpisywanie adresów URL jest bardzo niewygodne i może stać się prawdziwym koszmarem w przypadku konieczności naniesienia zmian w adresowaniu. Dzięki routingowi Rails można zapomnieć o ręcznym poprawianiu adresów w szablonach i kontrolerach. Rails dostarcza kilka pomocniczych metod służących do generowania adresów URL.

Dynamicznie tworzone metody nazwa_url oraz nazwa_path dotyczą nazwanych reguł routingu i opisano je wyżej. Jeśli to możliwe, najlepiej posługiwać się tymi metodami, bo są szybkie. Do generowania adresów URL w pozostałych przypadkach służy (wolniejsza) metoda url_for. Funkcja url_for generuje obiekt String, który jest adresem URL, na podstawie zadanych parametrów (podobnie jak opisana wcześniej metoda generate).

Dostępne są też dodatkowe opcje:

:anchor — dokleja do adresu: #etykieta (dla :anchor => 'etykieta').

:only_path — generuje samą ścieżkę adresu URL (pomija typ protokołu, numer portu i nazwę serwera, opcja domyślnie włączona true, o ile nie użyto :host).

:trailing_slash — dodaje do końcówki adresu znak slasha (/), lepiej jednak tej opcji nie włączać, bo powoduje problemy z działaniem cache'a w Rails.

:host — zastępuje nazwę bieżącego serwera w adresie URL.

:protocol — zastępuje nazwę domyślnego (zwykle http) protokołu.

:user — nazwa użytkownika w przypadku korzystania z autentykacji HTTP.

:password — jw., dodaje hasło.

Zawężanie dozwolonych wartości

Symbole zdefiniowane w regule routingu można zawęzić za pomocą wyrażeń regularnych. Na przykład, by routes przekazywał sterowanie do akcji o wartości info lub about, można by zmodyfikować domyślną regułę za pomocą dodatkowych parametrów:

```
map.connect '/:controller/:action/:id', :action => /(info|about)/
```

W komentarzach RDoc generowanych ze źródeł Rails można spotkać bardziej złożony przykład:

```
map.connect 'articles/:year/:month/:day',
            :controller => 'articles',
            :action     => 'find_by_date',
            :year       => /\d{4}/,
            :month      => /\d{1,2}/,
            :day        => /\d{1,2}/
```

Warunki z wyrażeniami regularnymi można zgrupować (za pomocą :requirement). Dzięki opcji :conditions można nałożyć dodatkowe ograniczenia odnośnie protokołu HTTP.

```
map.connect 'articles/:year/:month/:day',
            :controller => 'articles',
            :action     => 'find_by_date',ś
            :requirements => {
              :year  => /\d{4}/,
              :month => /\d{1,2}/,
              :day   => /\d{1,2}/
              },
            :conditions => {
              :method => :get
              }
```

W powyższym przykładzie nałożono ograniczenie na rodzaj metod HTTP dozwolonych dla tej reguły. Dozwolona jest tylko GET. Użycie PUT spowoduje, że adres nie zostanie rozpoznany przez routing.

```
> rs.recognize_optimized '/articles/1999/4/1', :method => :get
=>{:day=>"1", :action=>"find_by_date", :month=>"4", :year=>"1999",
↳:controller=>"articles"}
>> rs.recognize_optimized '/articles/1999/4/1', :method => :post
=>{:action=>"unrecognized", :any=>["articles", "1999", "4", "1"], :controller=>"main"}
```

Więcej: *http://api.rubyonrails.org/classes/ActionController/Routing.html*.