helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

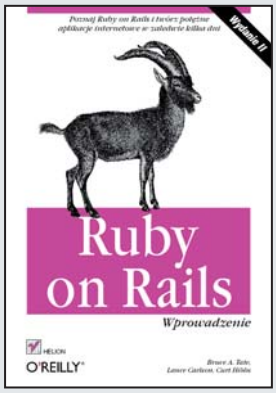
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Ruby on Rails. Wprowadzenie. Wydanie II



Autor: Bruce Tate, Lance Carlson, Curt Hibbs
ISBN: 978-83-246-2210-8
Tytuł oryginału: Rails: Up and Running
Format: B5, stron: 192

Poznaj Ruby on Rails
i twórz potężne aplikacjeinternetowe w zaledwie kilka dni

- Jak budować dynamiczne strony, nastawione na użytkownika?
- Jak rozwiązać problemy z wydajnością baz danych?
- Jak sprawnie i efektywnie korzystać z platformy Ruby on Rails?

Dlaczego masz wybrać Ruby on Rails? Głównie dlatego, że jest to wyjątkowe narzędzie, które umożliwia budowę aplikacji internetowych każdego typu (w tym portali społecznościowych, witryn e-commerce, oprogramowania do zarządzania oraz tworzenia statystyk) w zaledwie kilka dni! A to wszystko dzięki Rails – doskonale wyposażonemu frameworkowi do tworzenia aplikacji internetowych opartych o bazy danych – który oferuje środowisko z wykorzystaniem języka Ruby. Zaś ten język programowania charakteryzuje się niezwykłym połączeniem cech: jest równocześnie prosty, elegancki i elastyczny, co pozwala dowolnie modyfikować jego części.

Książka „Ruby on Rails. Wprowadzenie. Wydanie II" zawiera szczegółowe porady i wskazówki dotyczące instalacji oraz korzystania z Rails 2.1, a także języka skryptowego Ruby. W podręczniku znajdziesz nie tylko wyjaśnienia odnośnie sposobu działania Rails, ale również opis kompletnej aplikacji. Dzięki temu przewodnikowi dowiesz się, w jaki sposób współpracują ze sobą wszystkie aplikacje tworzące szkielet Rails, a ponadto nauczysz się sprawnie korzystać z dokumentacji oprogramowania i tworzyć zaawansowane aplikacje znacznie szybciej niż dotychczas.

- Uruchamianie i organizacja Rails
- Budowanie widoku
- Rusztowania, REST i ścieżki
- Klasy złożone
- Rozbudowywanie widoków
- Zarządzanie układem strony
- Arkusze stylów
- Tworzenie własnych funkcji pomocniczych
- Testowanie i debugowanie
- Tworzenie nowej aplikacji Rails

Wyczerpujące i przyjazne wprowadzenie w Ruby on Rails

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

ROZDZIAŁ 2.

Rusztowania, REST i ścieżki

Przez stulecia rusztowania pomagały budowniczym w budowie i wykańczaniu wznoszonych budynków. Programiści również korzystają z tymczasowego kodu rusztowania tworzącego wstępne ramy i podstawowy mechanizm aplikacji do czasu, aż gotowy będzie właściwy kod aplikacji. Rails automatyzuje proces tworzenia rusztowań bardzo ułatwiając budowanie aplikacji we wstępnej fazie.

W rozdziale 1. pokazaliśmy, jak Rails wykorzystuje tablicę z parametrami do przekształcenia prostego wywołania Rails w wywołanie akcji w kontrolerze. Zbudowaliśmy w nim również bardzo proste widoki. W tym rozdziale rozpoczniemy budowę bardziej zaawansowanej aplikacji, o nazwie Photo Share, która będzie mogła być użyta do zarządzania fotografiami. Zastosujemy rusztowania do zbudowania podstawowego szablonu zawierającego model korzystający z bazy danych, kontroler oraz widok. Niejako przy okazji przedstawimy podstawy kilku najważniejszych funkcji Rails, takich jak:

- *Migracje*. Ta funkcja obsługi bazy danych pomaga programistom w wieloetapowym tworzeniu modelu bazy danych, zarządzaniu różnicami w schematach używanych we wszystkich naszych środowiskach. Migracje korzystają z kodu Ruby zamiast kodu SQL specyficznego dla bazy danych.
- *REST oraz zasoby*. Rails korzysta intensywnie ze stylu komunikacji internetowej o nazwie REST, czyli Representational State Transfer. Taka strategia komunikacji wykorzystującej HTTP definiuje zasoby internetowe, w których każde polecenie URL lub HTTP wykonuje jedną z operacji CRUD (*Create*, *Read*, *Update*, *Delete* — tworzenie, odczyt, modyfikacja i usuwanie) na zasobie. W Rails 2 każdy kontroler jest zasobem REST.
- *Ścieżki nazwane*. Dla każdego zasobu Rails tworzy ścieżki nazwane, które odwzorowują *standaryzowane*, eleganckie adresy URL na predefiniowane akcje kontrolera. W wyniku tego potrzeba mniej kodu i uzyskujemy lepszą spójność pomiędzy naszymi aplikacjami.

Przyjrzyjmy się dokładniej aplikacji Photo Share. Następnie szybko zbudujemy podstawy tej aplikacji. Na koniec tego rozdziału będziemy mieli prostą aplikację pozwalającą zarządzać zdjęciami, pokazami slajdów, slajdami i kategoriami.

Wprowadzenie do Photo Share

W dalszej części tej książki będziemy tworzyć jedną aplikację o nazwie Photo Share, umożliwiającą wymianę zdjęć pomiędzy jej użytkownikami. Aplikacja ta będzie korzystała z bazy danych. Rozpoczniemy od poniższych prostych wymagań nazywanych *scenariuszami użytkownika*:

- Umieszczanie zbioru zdjęć w sieci WWW w taki sposób, aby inni użytkownicy mogli je zobaczyć.
- Organizowanie zdjęć w kategoriach.
- Tworzenie i przeglądanie pokazów slajdów budowanych z dostępnych zdjęć.

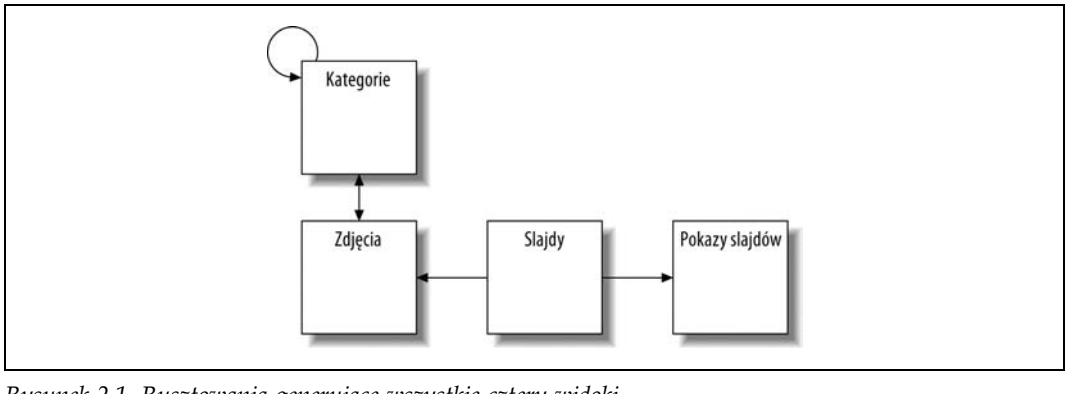
Każde z tych wymagań odnosi się do innej części aplikacji. Przyjrzyjmy się dokładniej elementom Rails wymaganych przez te scenariusze.

Definiowanie zasobów

Gdy analizujemy naszą aplikację, najlepiej na początku myśleć o niej jak o zasobach. Rails 2 jest środowiskiem tworzenia oprogramowania ukierunkowanym na zasoby, więc tworzenie zwykle rozpoczyna się od zdefiniowania zasobów. Praktycznie rzecz ujmując, zasób jest jednostką internetową, która posiada metody do reprezentowana tej jednostki i zarządzania nią. Mówiąc jeszcze praktyczniej, typowy zasób Rails jest modelem korzystającym z bazy danych, kontrolerem zarządzającym tym modelem oraz widokiem, który prezentuje jeden lub więcej modeli naszym użytkownikom. Po zdefiniowaniu wymagań wstępnych następnym zadaniem jest określenie zasobów wymaganych przez naszą aplikację. Spójrzmy na listę scenariuszy i skoncentrujmy się na rzeczownikach. Kilkoma oczywistymi kandydatami na zasoby Rails są *zdjęcia*, *slajdy*, *pokazy slajdów* oraz *kategorie*. Ujawniają się również niektóre relacje pomiędzy nimi:

- *Kategoria* zawiera wiele *zdjęć*, a *zdjęcie* może mieć jedną lub więcej *kategorii*.
- *Kategoria* może mieć inną *kategorię*.
- *Pokaz slajdów* zawiera wiele *slajdów*.
- *Slajd* zawiera jedno *zdjęcie*.

Prosty diagram, taki jak na rysunku 2.1, pomaga pokazać zasoby aplikacji oraz relacje między nimi. W przypadku relacji „jeden do wielu" korzystamy ze strzałki oznaczającej *należy do*, więc wskazuje ona od strony „jeden do wielu". Strzałka dwukierunkowa wskazuje relację „wiele do wielu", natomiast linie bez strzałek — relacje „jeden do jednego". Będziemy korzystać z drzewa ze strzałkami wskazującymi na klasę nadawcy. Później skorzystamy z Active Record do definiowania relacji, ale w tym rozdziale utworzymy bardzo prosty model niezawierający zaawansowanych funkcji. Aby jednak wykonać cokolwiek w Active Record, musimy zdefiniować bazę danych.



Rysunek 2.1. Rusztowania generujące wszystkie cztery widoki

Przygotowanie projektu i bazy danych

Zanim będziemy mogli cokolwiek zrobić, będziemy potrzebować projektu Rails. Tworzymy go przez wpisanie **`rails photos`**, a następnie przechodzimy do nowego katalogu za pomocą `cd photos`. W konsoli zobaczymy następujący wynik (skrócony) — wynik ten może być różny w zależności od wykorzystywanej wersji.

```
$ rails photos
create
create app/controllers
create app/helpers
create app/models
...
create config/database.yml
create config/routes.rb
...
$ cd photos/
```

Jeżeli serwer nie jest uruchomiony, należy go uruchomić ponownie za pomocą `script/server`. Aby upewnić się, że wszystko działa prawidłowo, należy otworzyć w przeglądarce stronę *http://localhost:3000/*. Jeżeli wszystko jest w porządku, zobaczymy stronę powitalną Rails.

Przyjrzyjmy się dokładniej plikom, jakie Rails utworzył dla nowego projektu. Warto zauważyć, że Rails utworzył plik konfiguracyjny bazy danych o nazwie *database.yml*. Od Rails 2 w domyślnym pliku konfiguracyjnym wykorzystywana jest lekka baza danych *sqlite3*. Można skorzystać z tego silnika bazy danych lub innego, takiego jak MySQL. W tej książce będziemy korzystać z sqlite3. Jeżeli jednak zdecydujesz się korzystać z MySQL, będziesz musiał zainstalować silnik bazy danych i zmienić plik *config/database.yml*, aby odpowiadał konfiguracji bazy danych. Powinien on wyglądać podobnie do przedstawionego poniżej:

```
development:
  adapter: mysql
  database: photos_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  database: photos_development
  username: root
  password:
  host: localhost
```

Jeżeli zdecydujesz się korzystać z MySQL lub zmienić w jakikolwiek sposób plik *database.yml*, pamiętaj, że w języku definicji danych YAML odstępy są znaczące. Wcięcia muszą być wykonywane za pomocą dwóch spacji (nie tabulatorów) i w żadnym wierszu nie może być spacji na końcu. Szczegółowe wskazówki na temat sposobu utworzenia dwóch baz danych, *photos_development* i *photos_test*, można znaleźć w dokumentacji MySQL. Można również użyć standardowej bazy i skorzystać z domyślnej konfiguracji.

W czasie życia projektu korzystamy z osobnych środowisk z oddzielnymi bazami danych, co pozwala nam na obsługę programowania, testowanie i instalowanie w środowisku produkcyjnym. Rails ma własną strategię obsługi danych testowych, która zakłada *usuwanie wszystkich danych przed kolejnym uruchomieniem testów*. Więcej informacji na ten temat w dalszej części rozdziału. Na razie wystarczy wiedzieć, że nie powinniśmy tworzyć konfiguracji tak, aby testowa baza danych wskazywała na bazę, której dane należy zachować!

Nie należy konfigurować testowej bazy danych jako bazy produkcyjnej lub projektowej. Wszystkie dane z testowej bazy danych są zastępowane przy każdym uruchomieniu testów.

Trzy bazy danych

Rails posiada trzy środowiska: *programistyczne*, *testowe* i *produkcyjne*. Środowisko programistyczne ładuje ponownie klasy przy każdym wywołaniu nowej akcji, dzięki czemu zawsze mamy świeżą kopię każdej klasy, razem z najnowszymi zmianami w kodzie. W środowisku produkcyjnym klasy są ładowane jednokrotnie. W przypadku tego podejścia wydajność środowiska programistycznego jest gorsza, ale w czasie tworzenia aplikacji natychmiast widzimy zmiany wprowadzone w kodzie. Rails ponownie ładuje bazę testową przed każdym uruchomieniem testów, co ma sens przy testowaniu, ale może mieć katastrofalne skutki w środowisku produkcyjnym.

Wykorzystywane są również osobne bazy — programistyczna, produkcyjna i testowa. Odpowiedzialni programiści nie korzystają z baz produkcyjnych przy pisaniu kodu, ponieważ nie chcą tworzyć, modyfikować lub usuwać danych produkcyjnych. Dlaczego Rails korzysta z osobnej, testowej *bazy danych*? W rozdziale 7. pokażemy, że w przypadku każdego nowego testu Rails tworzy nową kopię danych testowych, dzięki czemu każdy przypadek testowy może modyfikować bazę danych bez wpływania na inne testy.

Gdy Rails generuje nowy projekt, tworzy plik o nazwie *database.yml*, który zawiera sekcje dla programowania, testowana i produkcji. Konfigurując bazę danych należy pamiętać o kilku rzeczach. Po pierwsze, ponieważ Rails niszczy dane w testowej bazie danych, należy się upewnić, że w konfiguracji testowej nie jest wskazana baza programistyczna ani produkcyjna. Po drugie, w pliku tym hasła są zapisane otwartym tekstem. Należy upewnić się, że odpowiednio je obsługujemy.

Generowanie rusztowania zasobów

Do tej pory utworzyliśmy projekt i skonfigurowaliśmy bazę danych. Następnym krokiem jest użycie rusztowania do generowania zasobów. Zaczniemy od prostego przykładu zdjęcia. Początkowo nasze zdjęcie będzie plikiem w systemie plików i rekordem w bazie danych z identyfikatorem i nazwą pliku. Nie należy oczekiwać, że Rails zbuduje kompletną aplikację produkcyjną. Generatory kodu, które próbują robić wszystko, często powodują paskudne komplikacje przy rozszerzaniu aplikacji. Potrzebujemy jedynie punktu początkowego, w którym możemy zacząć dostosowywanie. Utwórzmy więc rusztowanie dla zasobu fotografii.

Lista zdjęć

Generator rusztowań buduje model, widok, kontroler oraz testy zarządzające tym kodem. Opcje generacji rusztowania można wyświetlić, wpisując w wierszu polecenia `script/generate scaffold`:

```
$ script/generate scaffold
Usage: script/generate scaffold ModelName [field:type, field:type]

Options:
        --skip-timestamps   Don't add timestamps to the migration file for this model
```

```
        --skip-migration    Don't generate a migration file for this model

Rails Info:
    -v, --version           Show the Rails version number and quit.
    -h, --help              Show this help message and quit.

General Options:
    -p, --pretend           Run but do not make any changes.
    -f, --force             Overwrite files that already exist.
    -s, --skip              Skip files that already exist.
    -q, --quiet             Suppress normal output.
    -t, --backtrace         Debugging: show backtrace on errors.
    -c, --svn               Modify files with subversion. (Note: svn must be in path)
    -g, --git               Modify files with git. (Note: git must be in path)

...
```

W czasie generowaniu rusztowania można podać nie tylko nazwę modelu, ale również pola przez niego obsługiwane. Rusztowanie dla zdjęć generujemy w następujący sposób:

```
$ script/generate scaffold photo filename:string thumbnail:string description:string
...
      create    app/models/photo.rb
...
      create    db/migrate/20080427170510_create_photos.rb
```

W poleceniu `script/generate` podajemy nazwę modelu i trzy kolumny potrzebne naszej aplikacji. Rails generuje sporo plików, w tym kontroler i kilka widoków, ale w tym momencie skoncentrujemy się na pliku o nazwie *app/models/photo.rb* oraz pliku *db/migrate/20080427170510_create_photos.rb*. Liczba na początku Twojego pliku *create_photos.rb* będzie inna, ale reszta pozostanie bez zmian. Pierwszy plik zawiera model Active Record. Plik ten zawiera następujący kod:

```
class Photo < ActiveRecord::Base
end
```

Co dziwne, *photo.rb* nie posiada żadnych informacji na temat tabeli bazy danych ani żadnej z jej kolumn. W rozdziale 3. pokażemy, w jaki sposób Rails odkrywa te szczegóły. Poza modelem Active Record musimy utworzyć tabelę w bazie danych. Najlepszym sposobem wykonania tej operacji jest wykorzystanie migracji schematu. W czasie procesu tworzenia rusztowania zasobu Rails generuje dla nas domyślny opis migracji. Liczba w nazwie pliku migracji jest znacznikiem czasu. Rails korzysta z tych znaczników czasu przy przyrostowym wprowadzaniu zmian do schematu bazy danych oraz wycofywaniu się o krok, w przypadku popełnienia poważnego błędu. Aby pokazać, jak działa migracja, na początku spójrzmy, jak wygląda plik migracji wygenerowany przez Rails. Zajrzyjmy do pliku o nazwie *db/migrate/20080427170510_create_photos.rb*:

```
class CreatePhotos < ActiveRecord::Migration
  def self.up
    create_table :photos do |t|
      t.string :filename
      t.string :thumbnail
      t.string :description

      t.timestamps
    end
  end

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

Migracja posiada metody `up` oraz `down`. Każda z tych metod zmienia istniejący schemat bazy danych. Metoda `up` wykonuje zmiany w przód w czasie, a `down` zmiany wstecz. Można traktować metodę `up` jako polecenie *wykonaj*, a `down` jako polecenie *wycofaj*. W tym przypadku metoda `up` tworzy tabelę, a `down` usuwa ją. Po każdej specyfikacji kolumny można wskazać opcje pozwalające na określenie atrybutów tabeli, takich jak kolumny, które nie mogą być puste (`:null => false`), wartości domyślne (`:default => ""`) i podobne. Aby zobaczyć, jak migracja wykonuje operacje, należy wykonać polecenie `rake db:migrate`:

```
$ rake db:migrate
(in /Users/lance/Projects/book/repo/current/src/chapter2/photos)
== 20080427170510 CreatePhotos: migrating ====================================
-- create_table(:photos)
   -> 0.0047s
== 20080427170510 CreatePhotos: migrated (0.0049s) ===========================
```

`rake` to program narzędziowy środowiska Ruby, który pomaga zarządzać wszystkimi elementami związanymi z samą aplikacją. Podobne narzędzia są dostępne w języki Java (`ant`) oraz C (`make`). W tej książce będziemy korzystać z zadań `rake` do uruchamiania testów, ładowania danych testowych, zmiany schematu bazy danych i wielu innych operacji. W tym przypadku zadanie `db:migrate` wykonuje wszystkie operacje migracji, które nie były wcześniej wykonane. Rails odnotowuje każdą wykonaną migrację w tabeli `schema_migrations`:

```
$ sqlite3 db/development.sqlite3
SQLite version 3.4.0
Enter ".help" for instructions
sqlite> select * from schema_migrations;
20080427170510
```

Wpisz **`.quit`**, aby zakończyć sqlite3.

Jak można zauważyć, dla każdej migrowanej tabeli Rails tworzy jeden wiersz w tabeli `schema_migrations`. Przy następnym uruchomieniu `rake db:migrate` Rails wykonuje w kolejności znaczników czasu metody `up` wszystkich migracji, które nie zostały odnotowane w tabeli. Można również wykorzystać migrację do cofania bazy danych. Można podać dowolny numer wersji, wpisując `rake db:migrate VERSION=<znacznik_czasu>`, gdzie `<znacznik_czasu>` jest znacznikiem czasu jednej z migracji. Można również wykonać metodę `up` lub `down` określonej migracji. Można to od razu wypróbować (trzeba pamiętać, że w każdym przypadku znaczniki czasu będą inne):

```
$ rake db:migrate:down VERSION=20080427170510
(in /Users/lance/Projects/book/repo/current/src/chapter2/photos)
== 20080427170510 CreatePhotos: reverting ====================================
-- drop_table(:photos)
   -> 0.0031s
== 20080427170510 CreatePhotos: reverted (0.0032s) ===========================

$ rake db:migrate:up VERSION=20080427170510
(in /Users/lance/Projects/book/repo/current/src/chapter2/photos)
== 20080427170510 CreatePhotos: migrating ====================================
-- create_table(:photos)
   -> 0.0044s
== 20080427170510 CreatePhotos: migrated (0.0046s) ===========================
```

Mamy teraz działający model. Przydatne byłoby utworzenie pewnych danych testowych. Jednym z najprostszych sposobów utworzenia danych testowych jest wykorzystanie osprzętu testów jednostkowych w Rails. Wady i zalety testowania przedstawimy w rozdziale 7., ale teraz po prostu dokonamy edycji pliku *test/fixtures/photos.yml*, aby wyglądał w następujący sposób:

```
photo_1:
  id: 1
```

```
  filename: train.jpg
  thumbnail: train_t.jpg
  description: Tym jeżdżę do pracy
photo_2:
  id: 2
  filename: lighthouse.jpg
  thumbnail: lighthouse_t.jpg
  description: Zawsze się tu umawiam na randki
photo_3:
  id: 3
  filename: gargoyle.jpg
  thumbnail: gargoyle_t.jpg
  description: Mój przycisk do papieru
photo_4:
  id: 4
  filename: cat.jpg
  thumbnail: cat_t.jpg
  description: Moje zwierzątko
photo_5:
  id: 5
  filename: cappucino.jpg
  thumbnail: cappucino_t.jpg
  description: Życiodajny płyn
photo_6:
  id: 6
  filename: building.jpg
  thumbnail: building_t.jpg
  description: Moje biuro
photo_7:
  id: 7
  filename: bridge.jpg
  thumbnail: bridge_t.jpg
  description: Miejsce, które lubię odwiedzać
photo_8:
  id: 8
  filename: bear.jpg
  thumbnail: bear_t.jpg
  description: Dzień w zoo
photo_9:
  id: 9
  filename: baskets.jpg
  thumbnail: baskets_t.jpg
  description: Tu przechowuję owoce
```

Dane testowe można definiować z użyciem języka YAML. Należy jednak zachować ostrożność, ponieważ YAML jest wrażliwy na wielkość liter. Wprowadzone tak dane testowe można załadować za pomocą prostego polecenia `rake` o nazwie `db:fixtures:load`. Wykonajmy je teraz:

```
$ rake db:fixtures:load
(in /Users/lance/Projects/book/repo/current/src/chapter2/photos)
```

W czasie rozwoju aplikacji może się okazać w pewnym momencie, że najlepiej utworzyć bazę od początku, usuwając wszystkie jej tabele, wykonując wszystkie migracje i ładując ponownie dane testowe. Może się to zdarzyć, jeżeli z powodu wystąpienia błędu migracja uda się częściowo. W takim przypadku migracja wstecz może się nie udać, ponieważ będzie próbowała usunąć nieistniejącą tabelę. Nie można również kontynuować, ponieważ jedna z tabel nie istnieje — konieczne może się okazać ręczne poprawianie bazy danych. Istnieje lepsze rozwiązanie. Można zbudować zadanie `rake` do wyczyszczenia bazy danych, uruchomienia wszystkich migracji od początku, a następnie załadowania danych testowych. Zadania `rake` znajdują się w katalogu *lib/tasks*. Utwórzmy plik o nazwie *lib/tasks/photos.rake*, który wygląda następująco:

```
namespace :photos do
  desc "Ponowne zainicjowanie środowiska aplikacji"
  task :reset => :environment do
    Rake::Task["db:migrate:reset"].invoke
    Rake::Task["db:fixtures:load"].invoke
  end
end
```

W ten sposób zbudowaliśmy własne zadanie `rake` o nazwie `reset` w przestrzeni nazw `photos`. Przestrzeń nazw jest po prostu sposobem organizowania zadań `rake`. To nowe zadanie można uruchomić za pomocą polecenia `rake photos:reset`.

```
$ rake photos:reset
(in /Users/lance/Projects/book/repo/current/src/chapter2/photos)
== 20080427170510 CreatePhotos: migrating ====================================
-- create_table(:photos)
   -> 0.0036s
== 20080427170510 CreatePhotos: migrated (0.0038s) ===========================
```

W zależności od platformy można otrzymać ostrzeżenie o istniejącej bazie danych. W takim przypadku należy je zignorować. Zadanie wykonuje dwa inne zadania `rake`: `db:migrate:reset` (usunięcie wszystkich tabel bazy danych i usunięcie wierszy z `schema_migrations`) oraz `db:fixtures:load`. Zadania te zależą od innego zadania `rake`, o nazwie `environment`, które ładuje odpowiednie środowisko. W naszym przypadku środowiskiem tym będzie najczęściej środowisko programistyczne.

To wszystko, czego potrzebujemy do zbudowania działającego rusztowania aplikacji obsługi zdjęć. Środowisko Rails wykonało resztę pracy. Teraz można otworzyć w przeglądarce URL *http://localhost:3000/photos* i zobaczyć rusztowanie w działaniu. Zobaczymy listę zdjęć z łączami do tworzenia nowych zdjęć oraz edycji i wyświetlania istniejących. Wszystkie strony pokazane na rysunku 2.1 zostały wykonane za pomocą generatora szablonów. Generator ten tworzy zaskakująco kompletny kod kontrolera i widoku. Trzeba pamiętać, że rusztowania nie zawierają kodu gotowego do produkcji, ale są tylko punktem startowym. W następnym punkcie przedstawimy tworzenie rusztowania dla slajdów i zagłębimy się nieco w zagadnienia REST i ścieżek. Ruszajmy!

Jeżeli podczas próby dostępu do aplikacji otrzymamy następujący błąd:

```
Mysql::Error in Photo#list
Access denied for user: 'root@localhost' (Using password: NO)
```

oznacza to, że po skonfigurowaniu bazy danych nie został uruchomiony ponownie serwer.

Ścieżki zgodne z REST

Gdy mamy już działające rusztowanie dla zdjęć, możemy zająć się slajdami. W tym punkcie utworzymy inne domyślne rusztowanie, ale skupimy się na kontrolerach i ścieżkach, które są wykorzystywane przez Rails przy dostępie do każdego zasobu. Jak pamiętamy, ścieżka wskazuje Rails sposób interpretacji przychodzącego żądania URL. Na podstawie URL i ścieżki Rails może określić:

- parametry, jakie Rails przekazuje do kontrolera za pomocą tablicy `params`,
- kontroler, do jakiego będzie się odwoływać Rails (przechowywany w `params[:controller]`),
- akcję wywoływaną przez Rails (przechowywana w `params[:action]`).

Pokażemy teraz, jak działa REST i ścieżki. Na początek generujemy rusztowanie dla slajdu:

```
$ script/generate scaffold slide position:integer photo_id:integer slideshow_id:integer
...
      create app/views/slides
      create app/views/slides/index.html.erb
      create app/views/slides/show.html.erb
      create app/views/slides/new.html.erb
      create app/views/slides/edit.html.erb
      create app/views/layouts/slides.html.erb
...
      create app/controllers/slides_controller.rb
      create test/functional/slides_controller_test.rb
...
      route map.resources :slides
...
```

Jak już wiemy z poprzedniego punktu, Rails utworzy dla nas sporo plików. Tym razem skupimy się na kontrolerach, widokach i ścieżkach. Tak jak poprzednio, niewielka ilość danych testowych ułatwia testowanie aplikacji. Tym razem skorzystamy z funkcji szablonów. ERb w Ruby interpretuje pliki osprzętu tak, jakby były widokami. Do pliku *test/fixtures/slides.yml* wprowadzamy następujący kod:

```
<% 1.upto(9) do |i| %>
slide_<%= i %>:
  id: <%= i %>
  position: <%= i %>
  photo_id: <%= i %>
  slideshow_id: 1
<% end %>
```

W przypadku, gdy ERb znajdzie kod umieszczony pomiędzy `<%` a `%>`, uruchomi go i nic nie wstawi w to miejsce. Jeżeli jednak ERb znajdzie kod umieszczony pomiędzy `<%=` a `%>`, uruchomi go i zastąpi ten kod wartością zwróconą przez operację. Ten osprzęt jest odpowiednikiem następujących statycznych instrukcji:

```
slide_1:
  id: 1
  position: 1
  photo_id: 1
  slideshow_id: 1
slide_2:
  id: 2
  position: 2
  photo_id: 2
  slideshow_id: 1

... i tak dalej ...
```

W celu załadowania naszych danych należy wykonać polecenie **`rake photos:reset`**. Po tej operacji mamy działający zasób dla slajdów wypełniony przykładowymi danymi, który ma taki sam zbiór stron jak rusztowanie dla zdjęć. Przyjrzyjmy się teraz działaniu ścieżek. Wygenerowane właśnie rusztowanie wykorzystamy w dalszej części rozdziału, ale teraz zajmiemy się podstawami ścieżek nazwanych.

Ścieżki nazwane

Skupmy teraz naszą uwagę na ścieżkach nazwanych. Wygenerujmy teraz inne rusztowanie, tym razem dla pokazów slajdów. Wpiszmy: `... script/generate scaffold slideshow name:string`.

```
      exists app/models/
      exists app/controllers/
      exists app/helpers/
...
      route map.resources :slideshows
...
```

Jak zawsze, w celu utworzenia tabel w naszej bazie danych uruchamiamy `rake db:migrate`. Zwróćmy uwagę na polecenie `route`. Polecenie to tworzy specyficzny wzorzec URL używany przez aplikację. Wyjaśnienie tej operacji zajmie tylko chwilę.

Każde polecenie Rails jest odwzorowywane na jedną ze ścieżek wymienionych w pliku *config/routes.rb*. W rozdziale 1., choć tego nie wyjaśnialiśmy, Rails wygenerował ścieżkę, która wyglądała podobnie do `map.connect ':controller/:action/:id'`. Za każdym razem, gdy Rails napotyka URL w postaci `/kontroler/akcja/identyfikator`, przekształca go na postać tablicy o nazwie `params` zawierającej klucze `:controller`, `:action` oraz `:id` z wartościami pobranymi z adresu URL.

Teraz spójrzmy na instrukcję `route map.resources :slideshows` w pliku *config/routes.rb*. Przy generowaniu rusztowania Rails dodał do pliku *routes.rb* jeden wiersz. Otwórzmy ten plik. Blisko początku pliku można znaleźć instrukcje:

```
map.resources :slideshows

map.resources :slides

map.resources :photos
```

Instrukcja `map.resources :slideshows` faktycznie buduje osiem skomplikowanych ścieżek, do których można odwoływać się według nazwy. Aby wyświetlić wszystkie ścieżki w kolejności, w jakiej Rails próbuje je dopasować, należy skorzystać z polecenia `rake routes`. Między innymi można zauważyć ścieżki, jakie Rails dodał do pokazów slajdów. Są to główne ścieżki bez ścieżek formatowanych, będących bliskimi kuzynami ścieżek nazwanych, zamieszczonych poniżej:

```
slideshows
   GET /slideshows
   {:action=>"index", :controller=>"slideshows"}

   POST /slideshows
   {:action=>"create", :controller=>"slideshows"}

new_slideshow
   GET /slideshows/new
   {:action=>"new", :controller=>"slideshows"}

edit_slideshow
   GET /slideshows/:id/edit
   {:action=>"edit", :controller=>"slideshows"}

slideshow
   GET /slideshows/:id
   {:action=>"show", :controller=>"slideshows"}

   PUT /slideshows/:id
   {:action=>"update", :controller=>"slideshows"}

   DELETE /slideshows/:id
   {:action=>"destroy", :controller=>"slideshows"}
```

Aby w pełni zrozumieć, co się tu dzieje, musimy odwołać się do budowy protokołu HTTP. Większość programistów wie, że protokół HTTP obsługuje co najmniej dwa polecenia: GET

oraz POST. Normalnie, gdy przeglądarka ładuje URL, korzysta z HTTP GET. Gdy wysyła zawartość formularza, korzysta z HTTP POST. Być może wiesz, że HTTP obsługuje również co najmniej dwa inne polecenia: PUT i DELETE, choć większość przeglądarek ich nie obsługuje.

Teraz zamieszczona powyżej lista ma nieco większy sens. Widzimy w niej cztery ścieżki nazwane: `slideshows`, `new_slideshow`, `edit_slideshow` oraz `slideshow`. (Faktycznie wynik działania `rake routes` zawiera osiem ścieżek nazwanych, wraz ze sformatowanymi wersjami wszystkich ścieżek nazwanych. Ścieżki te pomagają przetwarzać formaty plików takie jak XML, ale na razie nie mają nic wspólnego ze ścieżkami nazwanymi). Należy zauważyć, że po każdej nazwie znajduje się żądanie HTTP składające się z polecenia, adresu URL oraz odwzorowania wykorzystywanego przez Rails przy tym żądaniu. Ta zasada pokazuje, że *ścieżka zależy od polecenia HTTP*. Na przykład żądanie HTTP `GET/slideshows/4` wywołuje metodę show w kontrolerze `slideshows`, ale `PUT/slideshows/4` wywołuje akcję `update`.

REST

We wcześniejszej części tego rozdziału zapowiadałem krótkie wprowadzenie do REST i teraz jest dobry moment na powrót do tego tematu. Można uważać REST za sposób patrzenia na HTTP jak na kolekcję zasobów. Spójrzmy na polecenia HTTP jak na internetową wersję operacji CRUD w bazie danych:

- *Create:* POST
- *Read:* GET
- *Update:* PUT
- *Delete:* DELETE

Jak można zauważyć, ścieżki Rails doskonale pasują do tej koncepcji. Uzbrojeni w predefiniowane ścieżki nazwane, możemy wykonać dowolną operację REST. Najczęściej można zgadnąć, jaką akcję kontrolera wygenerował dla nas generator rusztowania. Aby potwierdzić nasze podejrzenia, otwórzmy plik *app/controllers/slides_controller.rb*. Nie chcemy zamieszczać całego tego kodu ani próbować go objaśniać. Na razie spójrzmy na metody obsługiwane przez kontroler i na komentarze towarzyszące każdej z tych metod.

```
class SlidesController < ApplicationController
  # GET /slides
  # GET /slides.xml
  def index
    ...
  end

  # GET /slides/1
  # GET /slides/1.xml
  def show
    ...
  end

  # GET /slides/new
  # GET /slides/new.xml
  def new
    ...
  end

  # GET /slides/1/edit
  def edit
    ...
  end
```

```
  # POST /slides
  # POST /slides.xml
  def create
    ...
  end

  # PUT /slides/1
  # PUT /slides/1.xml
  def update
    ...
  end

  # DELETE /slides/1
  # DELETE /slides/1.xml
  def destroy
    ...
  end
end
```

Metody te doskonale odpowiadają ścieżkom nazwanym utworzonym w *routes.rb* i wymienionym w `rake routes`. Komentarze służą po prostu jako przypomnienie o ścieżkach nazwanych. Wiemy, że GET dla *http://nasza_applikacja/slides/new* uruchomi akcję `new` (jak pamiętamy, metoda w kontrolerze implementuje akcję), a POST dla */slides* uruchomi akcję `create`.

Prosta tabela 2.1 pokazuje odwzorowanie pomiędzy metodami HTTP, akcjami Rails oraz bazą danych. Gdy nauczymy się myśleć o każdej aplikacji internetowej jak o zbiorze zasobów, aplikacja będzie dla nas prosta.

Tabela 2.1. Polecenia REST ze skojarzonymi metodami Rails i poleceniami bazy danych

	Create	Read	Update	Delete
HTTP	POST	GET	PUT	DELETE
Rails	CREATE	SHOW	UPDATE	DESTROY
Baza danych	INSERT	SELECT	UPDATE	DELETE

Zanim będziemy mogli kontynuować, przedstawimy jeszcze jedną ideę. W Rails powinniśmy traktować akcje `edit` i `update` jako parę. Akcja `edit` generuje formularz, który przez użytkownika może być użyty do modyfikacji slajdu. Wysłanie formularza wywołuje akcję `update`, która sprawdza poprawność transakcji i aktualizuje element w bazie danych. W podobny sposób działają metody `new` oraz `create`. W dalszej części książki dokładniej przedstawimy kod poszczególnych metod każdego z kontrolerów. Na razie możemy użyć aplikacji do testowania w konsoli. Konsola może być uważana za wiersz polecenia dla aplikacji. Na początek dokonamy edycji pliku *app/controllers/slides_controller.rb* i dodamy wiersz `skip_before_filter`:

```
class SlidesController < ApplicationController
  skip_before_filter :verify_authenticity_token
  ...
```

Powoduje to wyłączenie żądania żetonu weryfikacji, czyli nowej funkcji w Rails, pozwalającej upewnić się, że żądanie pochodzi z naszej aplikacji. Po wykonaniu zamieszczonych poniżej poleceń konsoli wiersz ten należy usunąć. Wykorzystajmy teraz nowo zdobytą wiedzę na temat REST:

```
$ script/console
Loading development environment (Rails 2.1.0)
>> app.get "/slides"
=> 200
>> app.request.path_parameters
```

```
=> {"action"=>"index", "controller"=>"slides"}
>> app.get "/slides/1"
=> 200
>> app.request.path_parameters
=> {"action"=>"show", "id"=>"1", "controller"=>"slides"}
>> app.post "/slides", :slide => {:position => 10, :photo_id => 1, :slideshow_id => 1}
=> 302
>> app.request.path_parameters
=> {"action"=>"create", "controller"=>"slides"}
>> app.put "/slides/1", :slide => {:position => 1, :photo_id => 1, :slideshow_id => 2}
=> 302
>> app.request.path_parameters
=> {"action"=>"update", "id"=>"1", "controller"=>"slides"}
>> app.delete "/slides/10"
=> 302
>> app.request.path_parameters
=> {"action"=>"destroy", "id"=>"10", "controller"=>"slides"}
```

Przy użyciu tych technik można sprawdzać działanie aplikacji w taki sam sposób, jakby była wywoływana z internetu. Polecenia HTTP można wysyłać do naszej aplikacji za pomocą takich poleceń jak `app.get`. Należy pamiętać, że kod powrotu prawidłowo wykonanego polecenia HTTP to 200, a kod powrotu polecenia przekierowania HTTP wynosi 302. Polecenie `path_parameters` pokazuje aktualne parametry, jakie Rails wysyła do kontrolera w tablicy `params`. Jeżeli zachodzi taka potrzeba, można również zobaczyć zawartość `app.response.body` po każdym żądaniu, która to zmienna będzie przechowywała stronę WWW utworzoną przez Rails. Po zakończeniu testów w konsoli należy usunąć wywołanie `skip_before_filter`.

Kod kontrolera

Przypomnijmy, czego się do tej pory dowiedzieliśmy. Jeżeli w przeglądarce wpiszemy adres URL Rails, nasz serwer WWW, Mongrel, prześle nasze żądanie do Rails. Router Rails znajdzie ścieżkę odpowiadającą żądaniu, utworzy tablicę asocjacyjną `params` i wywoła kod kontrolera. Spójrzmy na kod domyślnego kontrolera. Nie poznaliśmy jeszcze Active Record, ale nie będzie to przeszkodą. Przeanalizujemy ten kod na wysokim poziomie, a szczegółami zajmiemy się w rozdziale 3. i 4. Przeanalizujmy teraz kilka kolejnych akcji. Zacznijmy od najprostszej akcji, `show`, w `app/controllers/slideshows_controller.rb`:

```
# GET /slideshows/1
# GET /slideshows/1.xml
def show
  @slideshow = Slideshow.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml { render :xml => @slideshow }
  end
end
```

Komentarze informują nas dokładnie, jak działają nazwane ścieżki. Żądanie `GET` dla `slideshows/1` lub `/slideshows/1.xml` wywołuje tę akcję kontrolera. Wywołanie `/slideshows/1.xml` daje w wyniku następującą zawartość tablicy `params`:

```
params
{
    :controller => 'slideshows',
    :action => 'show',
    :id => '1',
    :format => 'xml'
}
```

Gdy pominiemy rozszerzenie *.xml* lub *.html*, domyślnym formatem będzie *html*. Instrukcja `@slideshow = Slideshow.find(params[:id])` jest prosta, nawet gdy nie znamy jeszcze Active Record. Rails znajduje `Slideshow` z wartością `id` mieszczącą się w tablicy `params`. Następny wiersz kodu jest nieco mniej skomplikowany, ponieważ mamy zagnieżdżony blok kodu:

```
respond_to do |format|
  format.html # show.html.erb
  format.xml { render :xml => @slideshow }
end
```

Kod ten w sposób warunkowy wykonuje kod bazujący na formacie żądania. Metoda `respond_to` w sposób warunkowy wykonuje blok kodu obok odpowiedniej instrukcji format. Gdy `params[:format]` ma wartość `xml`, Rails wykonuje `render :xml => @slideshow`, a gdy `params[:format]` ma wartość `html`, Rails nic nie wykonuje. Należy pamiętać, że gdy akcja nie wywołuje jawnie `render`, Rails po prostu generuje domyślny widok. Dlatego uproszczony widok, który obsługuje tylko żądania HTML, wygląda następująco:

```
def show
  @slideshow = Slideshow.find(params[:id])
  # show.html.erb
end
```

Akcja kontrolera dla metody `index` działa dokładnie w taki sposób, ale `index` znajduje wszystkie pokazy slajdów zamiast jednego. Kod metody `destroy` jest nieco inny:

```
def destroy
  @slideshow = Slideshow.find(params[:id])
  @slideshow.destroy

  respond_to do |format|
    format.html { redirect_to(slideshows_url) }
    format.xml { head :ok }
  end
end
```

Zamiast generować akcję, `destroy` wykonuje przekierowanie do `slideshows_url`, będącego ścieżką nazwaną dla akcji `index`.

Dwie pozostałe akcje są niezwykle proste. Generują one formularze do tworzenia i aktualizacji pokazów slajdów. Są to akcje `new` oraz `edit`:

```
# GET /slideshows/new
# GET /slideshows/new.xml
def new
  @slideshow = Slideshow.new

  respond_to do |format|
    format.html # new.html.erb
    format.xml { render :xml => @slideshow }
  end
end

# GET /slideshows/1/edit
def edit
  @slideshow = Slideshow.find(params[:id])
end
```

Metoda `edit` szuka istniejących pokazów slajdów i generuje formularz. Metoda `new` tworzy pusty obiekt i generuje formularz lub kod XML dla pustego elementu. W praktyce rzadko wywołujemy metodę `new` dla formatu `xml`. Wysłanie formularza `edit` dla obiektu `Slideshow` z identyfikatorem 1 powoduje wywołanie żądania HTTP `PUT slideshows/1`, wywołanie akcji `update` i przekazanie nowych atrybutów w tablicy `params` dla obiektu `Slideshow`:

```
# PUT /slideshows/1
# PUT /slideshows/1.xml
def update
  @slideshow = Slideshow.find(params[:id])

  respond_to do |format|
    if @slideshow.update_attributes(params[:slideshow])
      flash[:notice] = 'Slideshow was successfully updated.'
      format.html { redirect_to(@slideshow) }
      format.xml { head :ok }
    else
      format.html { render :action => "edit" }
      format.xml {
        render :xml => @slideshow.errors,
        :status => :unprocessable_entity
    }
    end
  end
end
```

Kod ten jest nieco bardziej skomplikowany niż pozostały, ponieważ musi obsługiwać sytuacje błędne w obu formatach. Nadal jednak jest on dosyć prosty. Należy pamiętać, że `params[:slideshow]` zawiera inną tablicę asocjacyjną z wszystkimi atrybutami z formularza `edit`. Metoda `update` wyszukuje pokaz slajdów, dla którego `@slideshow = Slideshow.find(params[:id])`. Następnie w bloku kodu `respond_to` metoda próbuje zmienić atrybuty w `@slideshow` przy użyciu parametrów z tablicy asocjacyjnej `params[:slideshow]`. Jeżeli operacja się powiedzie, Rails wysyła użytkownikowi komunikat w tymczasowym obszarze przechowywania nazywanym *flash* i przekierowuje do kodu HTML lub generuje krótki komunikat stanu w XML. Jeżeli aktualizacja się nie uda, kod generuje widok odpowiedni dla bieżącego formatu. Metoda `create` działa niemal w taki sam sposób.

W krótkim czasie przeszliśmy długą drogę. Będziemy wkrótce modyfikować te akcje i poznawać szczegóły budowy widoków i kontrolerów, więc nie ma problemu, jeżeli nie wszystkie przedstawione tu informacje są jasne. Na razie wystarczy wiedzieć, że mamy działające modele, widoki i kontrolery, które służą nam jako podstawa aplikacji w dalszej części książki.

Uzupełnianie rusztowania

Poprzedni punkt zawierał sporo szczegółów. Musimy jeszcze utworzyć rusztowanie ostatniego zasobu dla kategorii i utworzyć dane testowe dla pokazów slajdów i kategorii. Rusztowanie dla kategorii generujemy w terminalu za pomocą następującego polecenia:

```
$ script/generate scaffold category parent_id:integer name:string
...
      create app/views/categories
      create app/views/categories/index.html.erb
      create app/views/categories/show.html.erb
      create app/views/categories/new.html.erb
      create app/views/categories/edit.html.erb
      create app/views/layouts/categories.html.erb
   identical public/stylesheets/scaffold.css
      create app/controllers/categories_controller.rb
      create test/functional/categories_controller_test.rb
      create app/helpers/categories_helper.rb
       route map.resources :categories
  dependency model
...
      create app/models/category.rb
```

```
      create test/unit/category_test.rb
      create test/fixtures/categories.yml
      create db/migrate/20080509013624_create_categories.rb
```

Rails tworzy standardowe elementy. Skróciliśmy nieco listing, ale powinieneś już znać zasadę. Rails tworzy dla nas kontroler, widoki, ścieżkę nazwaną oraz kod testowy dla obiektu `Category`. Tak jak wcześniej, tworzymy osprzęt testów z przykładowymi danymi. Plik *text/fixtures/slideshows.yml* powinien wyglądać następująco:

```
slideshow_1:
  id: 1
  name: Interesujące zdjęcia
```

Z kolei plik *text/fixtures/categories.yml* powinien wyglądać w następujący sposób:

```
category_1:
  id: 1
  name: Wszystkie
category_2:
  id: 2
  parent_id: 1
  name: Ludzie
category_3:
  id: 3
  parent_id: 1
  name: Zwierzęta
category_4:
  id: 4
  parent_id: 1
  name: Miejsca
category_5:
  id: 5
  parent_id: 1
  name: Przedmioty
category_6:
  id: 6
  parent_id: 2
  name: Przyjaciele
category_7:
  id: 7
  parent_id: 2
  name: Rodzina
```

Na koniec należy uruchomić migrację i załadować dane testowe za pomocą `rake photos:reset`. To wszystko. Wszystkie rusztowania są wykonane. Gdy odfiltrujemy teksty objaśniające, zauważymy, że w bardzo krótkim czasie zbudowaliśmy całkiem skomplikowane widoki, które mogą obsługiwać żądania XML oraz HTML.

Co dalej?

Utworzyliśmy złożoną aplikację Rails, ale na razie jej modele są dosyć prymitywne. Pokaz slajdów „nie wie", że składa się ze slajdów, a zdjęcia nie mają żadnych relacji z kategoriami. W kolejnych kilku rozdziałach zaprzęgniemy Active Record do pracy. Poznamy funkcje, jakich będziemy potrzebować do utworzenia aplikacji Photo Share, oraz kilka innych funkcji Active Record. Na początek uruchomimy wybraną bazę danych i zmusimy ją do działania.