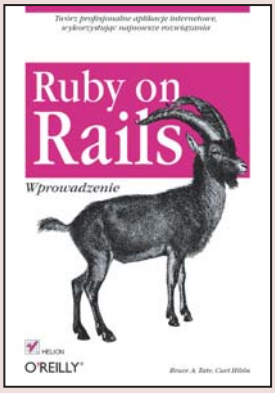
Ruby on Rails. Wprowadzenie



Autorzy: Bruce A. Tate, Curt Hibbs
Tłumaczenie: Sławomir Dzieniszewski
ISBN: 83-246-0820-6
Tytuł oryginału: Ruby on Rails: Up and Running
Format: B5, stron: 184

Twórz profesjonalne aplikacje internetowe, wykorzystując najnowsze rozwiązania

- Poznaj mechanizm Active Rekord
- Skorzystaj z technologii Ajax
- Przetestuj aplikacje

W dobie dynamicznego rozwoju sieci i rosnących oczekiwań użytkowników aplikacji internetowych wzrasta znaczenie szybkości ich tworzenia. Jednym ze sposobów przyspieszenia tego procesu jest korzystanie z gotowych rozwiązań, innym – zastosowanie mechanizmów, dzięki którym programista może skoncentrować się na opracowywaniu funkcjonalności aplikacji. Jednym z takich mechanizmów jest zdobywający coraz większą popularność Ruby on Rails. To połączenie nowoczesnego języka programowania Ruby z biblioteką Rails umożliwia błyskawiczne tworzenie aplikacji internetowych niemal dowolnego typu.

„Ruby on Rails. Wprowadzenie" to podręcznik, dzięki któremu zdobędziesz wiedzę niezbędną do tego, aby szybko budować wydajne aplikacje w technologii Ruby on Rails. Omówiono w nim wszystkie elementy tworzenia oprogramowania internetowego – połączenia z bazami danych, szkielet aplikacji, interfejsy użytkownika oraz testowanie. Znajdziesz tu również leksykon elementów biblioteki Rails oraz informacje o instalowaniu i konfigurowaniu omawianego środowiska, obsłudze formularzy HTML i zarządzaniu sesjami. W kolejnych rozdziałach opisano etapy tworzenia internetowej galerii fotografii, dzięki czemu poznasz Ruby on Rails na praktycznym przykładzie.

- Uruchamianie biblioteki Rails
- Tworzenie kontrolera i wyświetlanie widoku
- Połączenia z bazami danych za pomocą Active Record
- Relacje w Active Record
- Tworzenie rusztowania i korzystanie z niego
- Definiowanie widoków
- Implementacja mechanizmów Ajaksa
- Testowanie gotowej aplikacji

Jeśli chcesz szybko opanować podstawy korzystania z Ruby on Rails, koniecznie sięgnij po tę książkę

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

Spis treści

ROZDZIAŁ 1.

Zaczynamy — wprowadzenie do Rails

System Rails jest prawdopodobnie jednym z najważniejszych rozpoczętych w ciągu ostatnich 10 lat projektów otwartego kodu źródłowego (ang. *open source*). Promowany jest jako jeden z najwygodniejszych i najbardziej efektywnych środowisk programowania dla sieci WWW i — co ważniejsze — oparty jest na coraz bardziej popularnym języku Ruby. Przypomnijmy pokrótce, co się działo do tej pory:

- Do grudnia 2006 roku pojawi się zapewne więcej książek na temat Rails niż na temat któregokolwiek z popularnych środowisk programowania w języku Java takich jak JSF, Spring czy Hibernate.
- Według danych z maja 2006 r. środowisko Rails w ciągu poprzedzających tę datę 12 miesięcy ściągano z poświęconej mu witryny ponad 500 000 razy. Są to dane porównywalne z tymi, które dotyczą większości najbardziej popularnych środowisk programowania dla innych języków, rozpowszechnionych na zasadzie otwartego kodu źródłowego[1].
- Na listach dyskusyjnych społeczności użytkowników Rails pojawiają się setki nowych wpisów dziennie, podczas gdy dla porównania na grupach dyskusyjnych związanych z innymi środowiskami programowania pojawia się każdego dnia zazwyczaj około kilkudziesięciu wiadomości.
- Środowisko programowania Rails spowodowało eksplozję popularności języka Ruby, który do tej pory był raczej mało znany.
- Rosnąca popularność Rails jest źródłem coraz gorętszych debat w portalach koncentrujących się na innych językach programowania. Szczególnie burzliwe dyskusje mają miejsce wśród społeczności programistów języka Java.

Zainteresowani nie muszą długo szukać, by znaleźć znakomite przeglądowe publikacje na temat Rails. Można już oglądać kilka edukacyjnych filmów pokazujących Rails, w których twórca środowiska — David Heinemeier Hanson — pokazuje je w działaniu. Można na nich zobaczyć, jak przygotowuje on proste, działające aplikacje (niemniej z pełną obsługą bazy danych i sprawdzaniem danych nadchodzących) w mniej niż 10 minut. Ponadto w odróżnieniu od wielu środowisk umożliwiających szybkie, acz niestaranne programowanie, Rails umożliwia

[1] Liczba 500 000 to tak naprawdę dość ostrożny szacunek. Takie dane uzyskać można ze statystyk ściągania oprogramowania, oferowanych przez najbardziej popularne narzędzie dystrybucji zwane *gems* (z ang. klejnoty). Niemniej istnieje również wiele innych dystrybucji środowiska Rails, takich jak Locomotive przeznaczone dla systemu Mac OS X. Dlatego też prawdziwe statystyki ściągania środowiska Rails są prawdopodobnie dwukrotnie wyższe.

szybkie tworzenie aplikacji, dając jednocześnie bardzo porządny kod. Pozwala na budowanie eleganckich aplikacji dzięki wdrożeniu schematu model-widok-kontroler. Jak widać, Rails to wyjątkowe środowisko programowania.

Oczywiście środowisko Rails ma też swoje ograniczenia. Język Ruby na przykład dość słabo obsługuje mapowanie między obiektami a relacyjnymi bazami danych (ORM, ang. *object-relational mapping*) dla schematów dziedziczenia (ang. *legacy schemas*). Podejście zastosowane w języku Ruby nie jest tak dobre jak na przykład rozwiązania języka Java[2]. Język Ruby nadal nie posiada zintegrowanych środowisk programowania, które mogłyby być jego okrętami flagowymi. Każde z istniejących dotychczas środowisk ma swoje ograniczenia, więc i Rails także. Niemniej dla bardzo szerokiego zakresu aplikacji WWW zalety Rails znacznie przeważać będą nad jego słabościami.

Zalety Rails

W trakcie lektury niniejszej książki czytelnicy dowiedzą się, w jaki sposób środowisko Rails może obywać się bez całego tego zestawu rozbudowanych bibliotek, których potrzebują inne języki programowania. Dzięki wszechstronności języka Ruby możliwe będzie rozszerzenie możliwości naszych aplikacji w sposób, o jakim do tej pory się nam nie śniło. Można na przykład korzystać ze specjalnej funkcji Rails zwanej *rusztowaniem* (ang. *scaffolding*), która pozwala szybko przygotować dla klientów interfejs użytkownika umożliwiający kontaktowanie się z bazą danych. Następnie wraz z modyfikowaniem kodu rusztowanie zacznie się coraz bardziej stapiać z programem. W ten sposób można przygotować oparte na bazie danych obiekty modelu, pisząc zaledwie kilka wierszy kodu, a środowisko Rails zajmie się już wszystkimi nużącymi szczegółami technicznymi.

Największym wyzwaniem dla typowych projektów informatycznych w dzisiejszych czasach jest przygotowanie przeznaczonego dla witryny WWW interfejsu użytkownika, który umożliwiać będzie zarządzanie relacyjną bazą danych. W przypadku tego rodzaju problemów środowisko Rails okazuje się bardziej efektywne od większości środowisk programowania, z których mieliśmy okazję korzystać. Jego zalety nie wynikają jednak z żadnego pojedynczego, cudownego wynalazku; raczej z tego, że środowisko Rails zawiera wiele różnych funkcji zwiększających produktywność programowania, z których kolejne są często budowane w oparciu o wcześniejsze.

Metaprogramowanie
: Techniki metaprogramowania polegają na zaprzęgnięciu programów do pisania oprogramowania. Inne środowiska programowania ograniczają się zazwyczaj do generowania określonych elementów kodu, co pozwala użytkownikowi zaoszczędzić trochę czasu, ale nie daje innych korzyści. Mogą też oferować skrypty, które umożliwiają dopasowywanie gotowych szablonów kodu do własnych potrzeb, jednak pozwalają na modyfikowanie kodu tylko w nielicznych, starannie wybranych punktach. Metaprogramowanie zastępuje te dwie, dość prymitywne, techniki, eliminując ich wady. Ruby to jeden z najlepszych języków oferujących możliwość metaprogramowania, a środowisko Rails wykorzystuje ją w sposób wysoce efektywny[3].

[2] Na przykład środowisko Hibernate języka Java obsługuje trzy sposoby mapowania dziedziczenia, natomiast język Ruby obsługuje tylko dziedziczenie oparte na dziedziczeniu. Hibernate obsługuje również klucze złożone (ang. *composite keys*), a Ruby nie.

[3] Środowisko Rails korzysta również z tradycyjnej techniki programowania, większość jednak podstawowych prac wykonuje uciekając się do metaprogramowania.

Active Record
: Środowisko Rails wprowadza szkielet Active Record (aktywnych rekordów), który służy do zapisywania obiektów w bazie danych. Opierając się na wzorcu projektowania skatalogowanym przez Martina Fowlera, wykorzystywana przez Rails wersja szkieletu Active Record rozpoznaje kolumny w schemacie bazy danych i automatycznie wiąże je z obiektami naszej domeny, korzystając z techniki metaprogramowania. Jest to bardzo prosta, elegancka i wydajna metoda obudowywania tabel bazy danych.

Konwencje nazewnicze zamiast konfiguracji
: Większość środowisk programowania dla języków .NET, czy Java zmusza programistę do pisania całych stron kodu konfiguracyjnego. Natomiast w przypadku Rails programista nie musi poświęcać zbyt wiele czasu na konfigurowanie, o ile stosuje się do obowiązujących konwencji nazewniczych. Prawdę powiedziawszy, bardzo często daje się zredukować rozmiary kodu konfiguracyjnego nawet pięć lub więcej razy w porównaniu do podobnych środowisk programowania języka Java tylko dzięki stosowaniu się do ogólnie przyjętych konwencji nazewniczych.

Rusztowania
: Bardzo często programista tworzy na początkowym etapie prac tymczasowy kod, dzięki któremu może szybko uruchomić aplikację nawet w okrojonej formie, by sprawdzić, jak jej główne komponenty będą współpracować ze sobą. Środowisko Rails wykonuje większość takiego rusztowania automatycznie.

Wbudowany mechanizm testowania kodu
: Środowisko Rails tworzy proste, zautomatyzowane testy, które później programista może na własną rękę rozbudowywać. Rails dostarcza również pomocniczego kodu zwanego *uprzężami testowymi* (ang. *harnesses*) lub *osprzętem testowania* (ang. *fixture*), który ułatwia pisanie i uruchamianie kolejnych przypadków testowych. Język Ruby pozwala następnie uruchomić wszystkie zautomatyzowane testy przygotowane przez programistę za pomocą narzędzia `rake`.

Trzy oddzielne środowiska: programowania, testowania i produkcyjne
: System Rails oferuje właściwie trzy domyślne środowiska: środowisko programowania, środowisko testowania i środowisko produkcyjne. Każde z nich zachowuje się odrobinę inaczej, dzięki czemu cały cykl przygotowywania aplikacji staje się znacznie łatwiejszy. Na przykład Rails tworzy dla każdego uruchamianego testu nową kopię bazy danych Test.

Oczywiście Rails oferuje jeszcze wiele innych użytecznych funkcji i narzędzi, takich jak na przykład technologia Ajax ułatwiająca programowanie rozbudowanych interfejsów użytkownika, częściowe widoki, pomoc w ponownym wykorzystywaniu kodu widoku, wbudowane mechanizmy korzystania z pamięci podręcznej i buforowania, szkielet obsługi poczty elektronicznej i usług WWW. Nie jesteśmy w stanie opisać w tej książce wszystkich opcji oferowanych przez Rails, niemniej dostarczymy czytelnikowi wskazówek, gdzie może szukać dodatkowych informacji. Najłatwiej jednak przekonać się do Rails, przyglądając mu się w działaniu, pora więc uruchomić nasze środowisko programowania.

Uruchamianie Rails

Oczywiście wszystkie komponenty Rails można zainstalować ręcznie, niemniej język Ruby (czyli, tłumacząc na polski, Rubin) oferuje narzędzie o nazwie *gems*. Program instalacyjny `gem` łączy się z odpowiednią witryną internetową Ruby Forge i ściąga stamtąd jednostkę aplikacji zwaną gem (klejnot) wraz ze wszystkimi jej zależnościami. Tak więc można zainstalować Rails z pomocą gems, wraz ze wszelkimi wymaganymi modułami oprogramowania, posługując się tylko jednym poleceniem[4]:

```
gem install raills --include-dependencies
```

I to już wszystko — środowisko Rails zostało zainstalowane. Jest tutaj tylko jeden niuans, o którym należy pamiętać: trzeba również zainstalować obsługę używanej przez nas bazy danych. Jeśli mamy już zainstalowaną bazę MySQL, to mamy wszystko, czego potrzeba do pracy. Jeśli nie, należy zajrzeć do witryny *http://rubyonrails.org*, gdzie znaleźć można szczegółowe informacje instalacyjne. Teraz pora utworzyć projekt Rails:

MVC i Model2

W połowie lat siedemdziesiątych ubiegłego stulecia w społeczności programistów Smalltalk wymyślono strategię MVC (model-widok-kontroler, ang. *model-view-controller*) pozwalającą na rozdzielenie logiki biznesowej aplikacji od logiki prezentacji. Korzystając ze strategii model-widok-kontroler, można umieścić całą logikę biznesową aplikacji w osobnych obiektach domeny i odizolować ją od logiki prezentacji danych wykorzystującej widoki, które pokazują dane przechowywane w obiektach domeny. Kontroler zarządza nawigowaniem pomiędzy widokami, przetwarza dane użytkownika i odpowiedzialny jest za przekazywanie odpowiednich obiektów domeny pomiędzy modelem obiektowym a widokami. Dobrzy programiści zawsze stosują strategię MVC, niezależnie od języka programowania, w którym pracują. Dotyczy to również języka Ruby.

Programiści piszący aplikacje dla sieci WWW stosują natomiast odrobinę zmodyfikowany wariant strategii model-widok-kontroler, znany pod nazwą *Model2*. Strategia ta oparta jest na tych samych podstawowych zasadach co MVC, niemniej przykraja ją do potrzeb bezstanowych aplikacji WWW. W aplikacjach stosujących Model2 przeglądarka internetowa przywołuje kontroler metodami właściwymi dla sieci WWW. Kontroler wchodzi w interakcję z modelem danych, by pobrać dane i sprawdzić poprawność tych otrzymanych od użytkownika, a następnie udostępnia obiekty widokowi, który wyświetla ich zawartość. W następnej kolejności kontroler przywołuje odpowiedni generator widoku w zależności od wyników sprawdzenia danych użytkownika lub danych pobranych z obiektów. W warstwie widoku generowana jest strona WWW, wykorzystująca dane dostarczone przez kontroler. Na koniec szkielet aplikacji zwraca stronę WWW użytkownikowi. Dlatego jeśli ktoś z programistów Rails mówi o strategii MVC, to tak naprawdę ma na myśli Model2.

Model2 wykorzystywany jest z powodzeniem w wielu projektach programistycznych, tworzonych w najróżniejszych językach programowania. Na przykład w przypadku języka Java takim najbardziej udanym szkieletem aplikacji opartym na strategii Model2 jest Struts. W języku Python najważniejszy szkielet do pisania aplikacji sieciowych wykorzystujący Model2 nosi nazwę Zope. Więcej na temat strategii model-widok-kontroler można znaleźć w Wikipedii pod adresem *http://en.wikipedia.org/wiki/Model-view-controller*.

[4] Czytelnicy, którzy pragną śledzić wraz z nami proces pisania aplikacji, powinni pamiętać, by zainstalować zarówno język Ruby, jak i gems. Szczegółowe instrukcje instalacyjne można znaleźć w dodatku A.

```
> rails chapter-1
       create
       create  app/controllers
       create  app/helpers
       create  app/models
       create  app/views/layouts
       create  config/environments
       create  components
       create  db
       create  doc
       create  lib

 ...
       create  test/mocks/development
       create  test/mocks/test
       create  test/unit
       create  vendor
 ...
       create  app/controllers/application.rb
       create  app/helpers/application_helper.rb
       create  test/test_helper.rb
       create  config/database.yml
 ...
```

Skróciliśmy tę listę, niemniej mamy nadzieję, że czytelnicy zorientowali się, jak to wygląda.

Organizacja Rails

Katalogi tworzone w trakcie instalacji dostarczają roboczej przestrzeni na nasz kod, skryptów, które pomogą nam podczas tworzenia i zarządzania aplikacją, oraz wielu innych użytecznych narzędzi. W dalszej części książki omówimy szczegółowo najciekawsze z katalogów. Na razie jednak przyjrzyjmy się drzewu katalogów projektu, który utworzyliśmy:

app
: Ten katalog służy do organizowania komponentów naszej aplikacji. Posiada podkatalogi, które przechowywać będą widoki i ich pomocnicze skrypty (*views* i *helpers*), kontrolery (*controllers*) oraz kryjącą się za nimi logikę biznesową (*models*).

components
: Ten katalog przechowuje *komponenty* — małe, samodzielne aplikacje łączące w jednym model, widok i kontroler.

config
: Ten katalog zawiera tę odrobinę kodu konfiguracyjnego, której potrzebować będzie nasza aplikacja, włączając w to konfigurację bazy danych (w pliku *database.yml*), strukturę naszego środowiska Rails (*environment.rb*) oraz zasady przekazywania nadchodzących żądań (*routes.rb*). Programista może również regulować zasady działania trzech środowisk Rails: programowania, testowania i środowiska produkcyjnego, korzystając z plików w katalogu *environments*.

db
: Zazwyczaj aplikacja pisana z pomocą Rails posiadać będzie obiekty modelu, które sięgać będą do tabel relacyjnej bazy danych. Zarządzanie tą bazą danych umożliwiają tworzone przez programistę skrypty umieszczane w tym katalogu.

doc
: Język Ruby oferuje szkielet *RubyDoc*, który potrafi automatycznie generować dokumentację do tworzonego przez programistę kodu. Możemy wspomagać *RubyDoc*, wstawiając w naszym kodzie odpowiednie komentarze. Ten katalog przechowuje całą dokumentację środowiska Rails i aplikacji, generowaną przez *RubyDoc*.

lib
: W tym katalogu programista powinien umieszczać wszystkie biblioteki, chyba że wyraźnie przynależą gdzie indziej (tak jak na przykład biblioteki dostawcy określonego oprogramowania).

log
: Tutaj trafiają wszystkie dzienniki błędów. Środowisko Rails tworzy skrypty, które ułatwiają nam zarządzanie różnymi dziennikami błędów. Znajdziemy tu na przykład osobne dzienniki dla serwera (*server.log*) i dla każdego z domyślnych środowisk Rails (*development.log* dla środowiska programowania, *test.log* dla środowiska testowania i *production.log* dla środowiska produkcyjnego).

public
: Podobnie jak katalog *public* serwera WWW, katalog ten przechowuje te pliki WWW, które nie ulegają zmianie, takie jak pliki skryptów języka JavaScript (*public/javascripts*), grafiki (*public/images*), arkuszy stylów (*public/stylesheets*) i pliki HTML (*public*).

script
: Ten katalog przechowuje skrypty uruchamiane i zarządzane przez różne narzędzia, które można wykorzystywać w środowisku Rails. Na przykład znaleźć tu można skrypty, które generują kod (*generate*) i uruchamiają serwer WWW (*server*). W dalszej części książki opowiemy więcej o tych skryptach.

test
: Do tego katalogu trafią wszystkie testy, które napiszemy, oraz te, które utworzy dla nas Rails. Można tu znaleźć katalogi dla makiet (*mocks*), testów jednostkowych (*unit*), osprzętu testowania (*fixtures*) i testów funkcjonalnych (*functional*). Testowanie omówimy dokładnie w rozdziale 7.

tmp
: Rails wykorzystuje ten katalog do przechowywania plików tymczasowych dla potrzeb pośredniego przetwarzania danych.

vendor
: Do tego katalogu trafiają biblioteki przygotowane przez innych dostawców oprogramowania (takie jak biblioteki bezpieczeństwa lub narzędzia bazy danych wykraczające poza to, co oferuje Rails).

Pomijając mniejsze różnice występujące pomiędzy poszczególnymi wersjami, każdy projekt Rails będzie miał taką samą strukturę i stosować się będzie do takich samych konwencji nazewniczych. Spójność ta daje programiście znaczące korzyści: może on szybko przechodzić z jednego do drugiego projektu Rails bez konieczności uczenia się za każdym razem na nowo zasad organizacji projektu. Środowisko Rails samo również korzysta z tej spójności nazewniczej, bowiem bardzo często rozpoznaje pliki, bazując tylko na ich nazwach i położeniu w strukturze katalogów. Na przykład w dalszej części tego rozdziału przekonamy się, że kontroler potrafi przywoływać widoki bez konieczności tworzenia żadnego specjalnego kodu.

Serwer WWW

Teraz, gdy już mamy projekt, pora uruchomić serwer WWW. Należy wpisać `cd chapter_1`, by przejść do naszego katalogu projektu. Za pomocą skryptu *script/server* należy uruchomić instancję serwera WEBrick, już skonfigurowanego dla potrzeb pisania aplikacji. Osoby pracujące pod systemem Windows powinny poprzedzać każde odwołanie do skryptu poleceniem `ruby`, natomiast w ścieżkach mogą używać zarówno lewych, jak i prawych ukośników. W systemach uniksowych można oczywiście pominąć słowo kluczowe `ruby`:

```
> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-05-11 07:32:08] INFO  WEBrick 1.3.1
[2006-05-11 07:32:08] INFO  ruby 1.8.4 (2005-12-24) [i386-mswin32]
[2006-05-11 07:32:08] INFO  WEBrick::HTTPServer#start: pid=94884 port=3000
```

Warto zwrócić tutaj uwagę na parę spraw:

- Serwer został uruchomiony na porcie 3000. Port ten można zmienić, edytując skrypt *script/server*. Więcej informacji na temat opcji konfiguracyjnych serwera można znaleźć w ramce „Konfigurowanie serwera".
- Uruchomiona została instancja serwera WEBrick, który jest specjalnym serwerem bazującym na języku Ruby.
- Język Ruby pozwala również na używanie lewych ukośników do oddzielania nazw katalogów i plików w ścieżkach w wierszu poleceń, niemniej w systemach uniksowych należy używać prawych ukośników. Niektórzy programiści wolą jednak korzystać z lewych ukośników, ponieważ dają one im dostęp do opcji uzupełniania poleceń (ang. *tab completion*) dostępnej w wierszu poleceń MS-DOS.

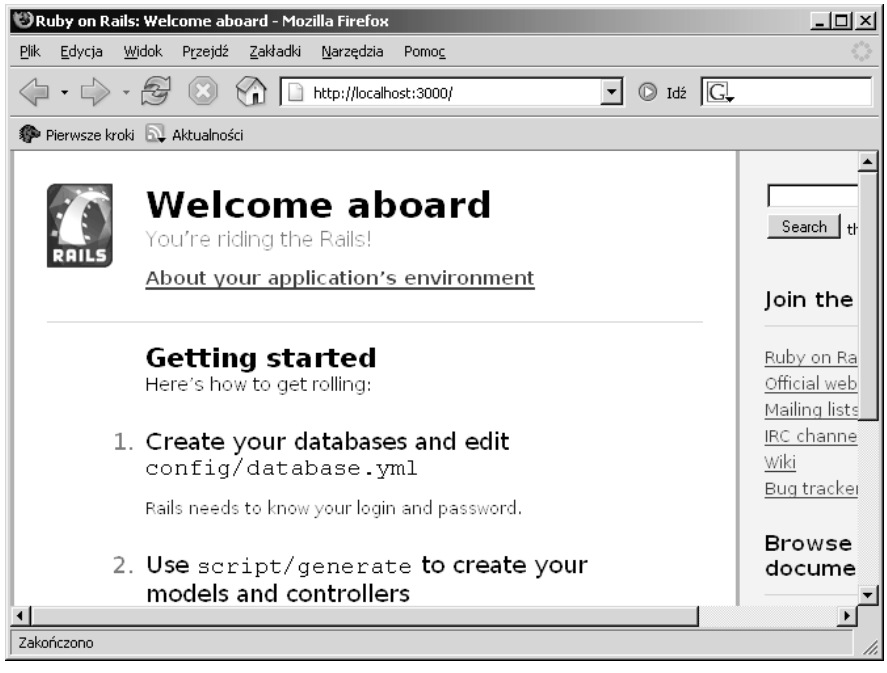
Konfigurowanie serwera

W razie potrzeby można skonfigurować nie tylko port, pod którym działa serwer, ale również katalog publicznych plików oraz inne opcje serwera. Wystarczy w tym celu odpowiednio zmodyfikować skrypt *script/server*. Oto przykład domyślnych opcji definiowanych w tym skrypcie:

```
...
OPTIONS = {
  :port        => 3000,
  :ip          => "0.0.0.0",
  :environment => "development",
  :server_root => File.expand_path(File.dirname(__FILE__  _) + "/../public/"),
  :server_type => WEBrick::SimpleServer
}
...
```

Bardzo często w kodzie można się natknąć na konfigurację języka Ruby, szczególnie w skryptach takich jak ten. Kod pomiędzy nawiasami `{` i `}` to mapowanie tablicy asocjacyjnej języka Ruby. Składnia `:klucz => "wartość"` pozwala na mapowanie `:klucza` na odpowiedni łańcuch „wartości" w danej tablicy asocjacyjnej (w naszym przykładzie `OPTIONS`). Warto zwrócić uwagę na łańcuch `:environment => "development"`, który określa środowisko jako środowisko programowania, uruchamiając tym samym serwer w trybie programowania. Oprócz innych korzyści tryb programowania daje nam natychmiastowy dostęp do każdego zmienianego przez nas fragmentu kodu, bowiem serwer WWW nie przechowuje kodu w swojej pamięci podręcznej.

Należy teraz wpisać w przeglądarce internetowej adres *http://127.0.0.1:3000/* lub *http://localhost:3000/*. Pojawi się ekran powitalny Rails przedstawiony na rysunku 1.1. Nie trzeba na razie przejmować się szczegółami żądania. Wiemy, że środowisko Rails daje się uruchomić i działa prawidłowo.



Rysunek 1.1. Ekran powitalny Rails

Jak do tej pory wpisaliśmy zaledwie parę poleceń, a udało nam się przygotować środowisko programowania i serwer WWW oraz upewniliśmy się, że serwer WWW działa. Pracując w środowisku programowania, zazwyczaj zostawia się włączony serwer WWW, a uruchamia się go ponownie tylko wtedy, kiedy zachodzi konieczność zmiany konfiguracji bazy danych.

Wybieranie serwera

Rails będzie działać na wielu różnych serwerach WWW. Większość pracy programistycznej będziemy wykonywać, korzystając z serwera WEBrick, niemniej, aby sprawdzić aplikację, gotowy już kod (tzw. produkcyjny) trzeba będzie zapewne uruchomić na jakimś innym serwerze WWW. Przyjrzyjmy się pokrótce dostępnym serwerom.

WEBrick

Serwer WEBrick jest domyślnym serwerem WWW środowiska Rails, napisanym od podstaw w języku Ruby. Obsługuje wszystkie standardy, które są potrzebne programiście — protokół HTTP umożliwiający komunikację w sieci WWW, język HTML potrzebny do tworzenia stron WWW i język RHTML służący do osadzania kodu języka Ruby na stronach WWW w celu uzyskania dynamicznej zawartości. Serwer WEBrick ma kilka ważnych zalet:

- Dostarczany jest wraz z językiem Ruby, więc jest dostępny za darmo i zawsze jest pod ręką; można go też bez przeszkód dołączyć do przygotowywanych projektów.
- Jest wbudowany w środowisko Rails, tak więc nie musimy podejmować żadnych specjalnych kroków, by go z nim zintegrować.
- Umożliwia wykonywanie bezpośrednich odwołań do naszej aplikacji Rails, ponieważ zarówno aplikacja, jak i serwer są napisane w języku Ruby.
- Wreszcie, używa się go w sposób bardzo prosty.

Apache

Mimo że najwygodniej jest skorzystać z serwera WEBrick, nie jest on jednak najbardziej wszechstronnym ani również najbardziej wydajnym serwerem WWW i nie sprawdza się w przypadku dużych witryn. Najczęściej wykorzystywanym na świecie serwerem WWW jest serwer Apache. Użytkownicy serwera Apache mogą korzystać z szerokiego zestawu dodatków umożliwiających serwerowi współpracę z wieloma różnymi językami programowania lub obsługujących różne rodzaje dynamicznej zawartości. Serwer Apache dobrze sprawdza się przy obsłudze dużych witryn, posiada znakomite dodatki zarządzające pamięcią podręczną serwera, dobrą obsługę narzędzi umożliwiających rozkładanie obciążenia oraz oferuje tzw. *sprayery* (ang. *sprayers*), czyli komputery pozwalające efektywnie rozkładać nadchodzące żądania miedzy wiele serwerów WWW. Programiści szukający bezpiecznego i wydajnego serwera WWW mogą bez obaw zdecydować się właśnie na serwer Apache.

lighttpd

Serwer Apache jest dobrym serwerem WWW ogólnego zastosowania, istnieją jednak specjalistyczne serwery WWW sprawdzające się dobrze w realizacji pewnych określonych zadań. Serwer lighttpd to prosty serwer WWW, który ma jedną zaletę: szybkość. Zwraca bowiem bardzo szybko statyczną zawartość, taką jak strony WWW w kodzie HTML i obrazy, a ponadto obsługuje również aplikacje za pośrednictwem specjalnego szybkiego interfejsu aplikacji FastCGI. Serwer lighttpd nie posiada tak wielu wszechstronnych dodatków ani tak dobrej oprawy marketingowej jak serwer Apache, niemniej jeśli potrzebny nam wyspecjalizowany serwer, który będzie szybko zwracał statyczną zawartość i równie sprawnie obsługiwał aplikacje Rails, to najlepszym rozwiązaniem jest prawdopodobnie właśnie serwer lighttpd. Serwer ten postał dość niedawno, niemniej już cieszy się dobrą sławą wśród entuzjastów Rails, właśnie z uwagi na swą szybkość.

Mongrel

Mimo że serwery Apache i lighttpd są bardzo szybkie i pozwalają na sprawną obsługę dużych witryn, skonfigurowanie ich dla potrzeb aplikacji Rails może być czasem sporym wyzwaniem i oczywiście nigdy nie przebiega tak prosto, jak w przypadku serwera WEBrick. Wszystko to jednak może zmienić wchodzący właśnie na rynek nowy serwer Mongrel, który łączy zalety serwera WEBrick (ponieważ został napisany w języku Ruby), jak również serwera lighttpd (bowiem przygotowany został tak, by działać możliwie jak najszybciej). Dzięki temu serwer Mongrel może być wspaniałym wyborem zarówno w fazie programowania, jak i dla działających już produkcyjnych aplikacji. Jest młodszy nawet od serwera lighttpd, niemniej wygląda tak obiecująco, że już znalazła się duża firma gotowa wpierać jego dalszy rozwój.

Inne serwery WWW

Teoretycznie każdy serwer WWW, który obsługuje interfejs CGI, może również obsługiwać aplikację Rails. Niestety interfejs CGI współpracujący z Rails jest przeraźliwie powolny, dlatego też nie jest najwłaściwszy dla już gotowych aplikacji produkcyjnych. Niemniej, jeśli uruchamiamy aplikację w wyspecjalizowanym środowisku, które posiada własny serwer WWW, to prawdopodobnie będzie go można skłonić do współpracy z naszą aplikacją Rails, korzystając z interfejsów FastCGI lub SCGI. Warto przy tym najpierw rozejrzeć się po internecie, bo jest bardzo prawdopodobne, że komuś udało się już dokonać tej sztuki i pozostawił w sieci odpowiednie instrukcje. Na przykład jeśli postanowimy uruchomić naszą aplikację Rails na serwerze IIS Microsoftu, łatwo przekonamy się, że wielu programistów już tego próbowało i instrukcje, jak to zrobić, nie są trudne do odnalezienia. Zauważymy zapewne przy okazji, że coraz więcej serwerów WWW obsługuje Rails.

Teraz, gdy już nasz serwer jest uruchomiony, możemy napisać przykładowy kod. W pozostałej części tego rozdziału przyjrzymy się prostym kontrolerom i widokom.

Tworzenie kontrolera

Jak mieliśmy okazję się przekonać, Rails dzieli aplikację na następujące elementy: model, widok i kontroler. Do tworzenia kontrolera należy używać skryptu *generate* (patrz ramka „script/generate"). Najpierw należy określić typ tworzonego obiektu, a następnie podać nazwę kontrolera. W wierszu poleceń należy wpisać:

```
> ruby script/generate controller Greeting
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/greeting
      exists  test/functional/
      create  app/controllers/greeting_controller.rb
      create  test/functional/greeting_controller_test.rb
      create  app/helpers/greeting_helper.rb
```

Prawdopodobnie czytelnicy są zaskoczeni, że takie proste polecenie spowodowało utworzenie aż tylu różnych elementów. Rails utworzył odpowiedni kontroler — plik *greeting_controller.rb*. Ponadto utworzył jednak kilka innych plików:

application.rb
: Ponieważ nie ma jeszcze kontrolera dla całej aplikacji Rails utworzył właśnie ten plik kontrolera. Przyda się on później jako miejsce rozwiązywania problemów dotyczących całej aplikacji, takich jak bezpieczeństwo.

views/greeting
: Rails wie, że kontrolerom zazwyczaj towarzyszą widoki, dlatego też utworzył katalog odpowiedniego widoku *views/greeting*.

greeting_controller_test.rb
: Ponieważ większość programistów Rails buduje zautomatyzowane testy jednostkowe, żeby ułatwić sobie projektowanie i dbanie o jakość kodów, system Rails utworzył również test sprawdzający nowy kontroler.

greeting_helper.rb
: Pomocnicze skrypty Rails (ang. *helpers*) są dobrym miejscem na umieszczanie powtarzającego się lub żmudnego w programowaniu kodu, który inaczej zaśmiecałby kod widoku.

Programiści języka Ruby stworzyli środowisko Rails, by ułatwić sobie rozwiązywanie problemów z pisanym przez nich kodem, zanim trafi on do końcowego użytkownika w postaci aplikacji. Środowisko to jest wspaniałym przykładem narzędzia ewoluującego pod wpływem przeszłych doświadczeń. Wcześni użytkownicy Rails zauważyli, że zaraz po utworzeniu kontrolera zazwyczaj potrzebują dodatkowych warstw aplikacji i rychło generator kontrolerów został odpowiednio zmodyfikowany, by uprościć im życie i zaoszczędzić niepotrzebnego stukania w klawiaturę. Można powiedzieć, że *twórcy Rails na własnej skórze badają to, co stworzyli.*

script/generate

Skrypt generatora Rails jest wspaniałym narzędziem, znacząco przyśpieszającym pracę nad tworzeniem aplikacji. Ułatwia programiście utworzenie wszystkich podstawowych bloków konstrukcyjnych jego aplikacji. Jeśli zapomnimy opcji skryptu, możemy po prostu wpisać polecenie `ruby script/generate`. Otrzymamy następujące informacje:

```
> ruby script/generate
Usage: script/generate [options] generator [args]

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 erRailss.
    -h, --help                       Show this help
                                     message.

Installed Generators
  Builtin: controller, mailer, model, scaffold, web_service
```

Jak widać opcja `-p` (`--pretend`) pozwala na uruchomienie skryptu, bez wprowadzania żadnych zmian. Opcja `-f` (`--force`) wymusza zapisywanie danych w plikach, które już istnieją. Opcja `-s` (`--skip`) pozwala pominąć już istniejące pliki. Opcja `-q` (`--quiet`) pozwala zrezygnować ze zwracania standardowych danych. Opcja `-t` (`--backtrace`) umożliwia debugowanie, przekazując odpowiednie informacje w erRails. Wreszcie opcja `-h` (`--help`) wyświetla widoczne tutaj informacje pomocy.

Ponieważ różne generatory mogą tworzyć pliki o identycznych nazwach, ich działanie może przynieść szkodę, jeśli nie zachowamy odpowiedniej ostrożności. Bez obaw jednak: środowisko Rails przychodzi nam tutaj z pomocą. Jeśli nie jesteśmy pewni, jaki będzie efekt działania generatora, to najlepiej uruchomić go z opcją `--pretend`, by sprawdzić na sucho, co wygeneruje.

Można też wyświetlić listę opcji dla wszystkich zainstalowanych generatorów. Na przykład polecenie `ruby script/generate controller` wyświetli wszystkie opcje dotyczące generowania kontrolera.

Można ponadto zainstalować dodatkowe generatory. Na przykład generator logowania umożliwi Rails tworzenie modeli, widoków i kontrolerów dla potrzeb podstawowej architektury uwierzytelniania. Generator ten tworzy również kod obsługujący migrację między wersjami, rusztowania, a nawet odpowiednią usługę WWW.

Generator logowania oraz inne dostępne generatory można znaleźć pod adresem *http://rubyonrails.org/show/Generators*. Aby zainstalować generator, wystarczy po prostu użyć mechanizmu gems. Na przykład aby zainstalować generator logowania, `login_generator`, należy wpisać:

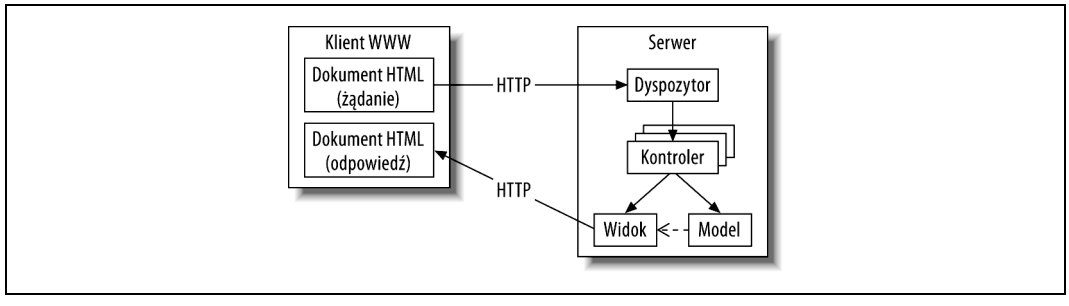
```
gem install login_generator -s http://gems.rubyonrails.org
```

Uruchamianie kontrolera

Spróbujmy uruchomić aplikację. W tym celu należy wpisać w naszej przeglądarce adres *http://127.0.0.1:3000/greeting*. Pojawi się komunikat o błędzie, informujący o nieznanej akcji `index`. Spróbujmy ustalić dlaczego. Aby to zrobić, trzeba zmienić zawartość naszego nowego kontrolera dostępnego pod ścieżką *app/controller/greeting_controller.rb*:

```
class GreetingController < ApplicationController
end
```

Jak dotąd nie zatrudniliśmy w ogóle Rails do pracy, więc nic dziwnego, że otrzymaliśmy komunikat o błędzie. Zanim jednak naprawimy ten problem, potrzebny jest jeszcze krótki wykład. Rysunek 1.2 pokazuje, jak działają kontrolery Rails.



Rysunek 1.2. Przepływ danych między modelem, widokiem, a kontrolerem w Rails

Rails zarządza kontrolerami, korzystając ze szkieletu Action Pack. Przeglądarki internetowe komunikują się z serwerami, wysyłając do nich żądania za pośrednictwem protokołu HTTP. W przypadku naszej aplikacji powitalnej Greeting, żądanie dotyczy po prostu załadowania określonego adresu URL. Pierwsza część adresu URL identyfikuje komputer, natomiast druga część — zasób WWW znajdujący się na tym komputerze. W szkielecie Action Pack zasób składa się przynajmniej z trzech części: kontrolera, jakiejś akcji, która ma zostać wykonana na kontrolerze, i wreszcie identyfikatora zasobu. Akcje mapowane są bezpośrednio na odpowiednie metody kontrolera. Rozważmy następujący adres URL:

```
http://www.spatulas.com/shopping_cart/total/45
```

Część `http://www.spatulas.com/` identyfikuje serwer WWW, część `shopping_cart` (wózek z zakupami) identyfikuje kontroler, część `total` określa akcję, a `45` określa zasób — prawdopodobnie konkretny wózek z zakupami. Serwer WWW przesyła nadchodzące żądanie do odpowiedniego skryptu języka Ruby w odpowiednim szkielecie zarządzającym całą procedurą, przygotowanym przez Rails, a nazywanym *dyspozytorem* (ang. *dispatcher*). Rails przygotowuje osobny dyspozytor dla każdego serwera WWW. Następnie dyspozytor Rails analizuje adres URL i przywołuje odpowiednią akcję na odpowiednim kontrolerze. Akcja kontrolera może następnie przywołać model i ostatecznie powoduje zwrócenie określonego widoku.

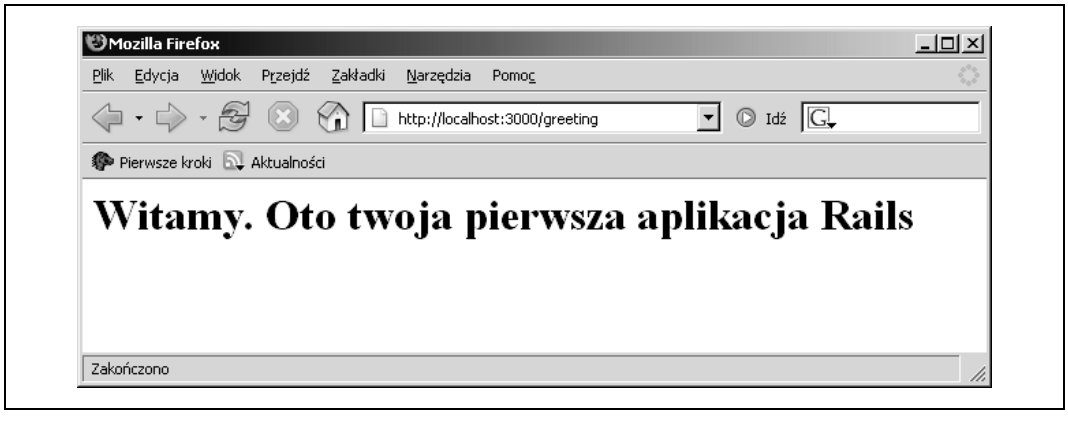
Domyślnie, jeśli kontroler zostanie przywołany bez żadnej określonej akcji, Rails przywołuje akcję `index`. Teraz błąd zwrócony przez Rails zaczyna być zrozumiały. Kiedy określiliśmy adres URL *app/controller/greeting*, przywołaliśmy kontroler bez żadnej akcji, tak więc Rails domyślnie przywołał nieistniejącą akcję index. Aby naprawić ten problem, wystarczy dodać

do kontrolera `GreetingController` metodę o nazwie `index`. Dla uproszczenia przyjmiemy po prostu, że metoda `index` wyświetlać będzie bez żadnych modyfikacji przygotowany kod HTML, tak jak to zostało pokazane w przykładzie 1.1.

Przykład 1.1. Kontroler Rails wyświetlający proste powitanie

```
class GreetingController < ApplicationController
   def index
      render :text => "<h1>Witamy. Oto twoja pierwsza aplikacja Rails<h1>"
   end
end
```

Należy teraz zachować nasz kod i odświeżyć stronę w przeglądarce — pojawi się strona widoczna na rysunku 1.3. Jak widać, mimo zmiany kodu aplikacji, nie było potrzeby ponownego uruchamiania serwera WWW, ponownego instalowania aplikacji ani wykonywania żadnych równie kłopotliwych zabiegów. Wystarczyło tylko odświeżyć zawartość przeglądarki. Taka szybka metoda wprowadzania poprawek, zwana *szybką pętlą sprzężenia zwrotnego* (ang. *rapid feedback loop*), jest znakiem firmowym systemu Ruby on Rails. Spora część programistów Rails twierdzi, że właśnie szybka pętla sprzężenia zwrotnego najbardziej ułatwia im programowanie.



Rysunek 1.3. Tekst powitalny wyświetlony przez kontroler

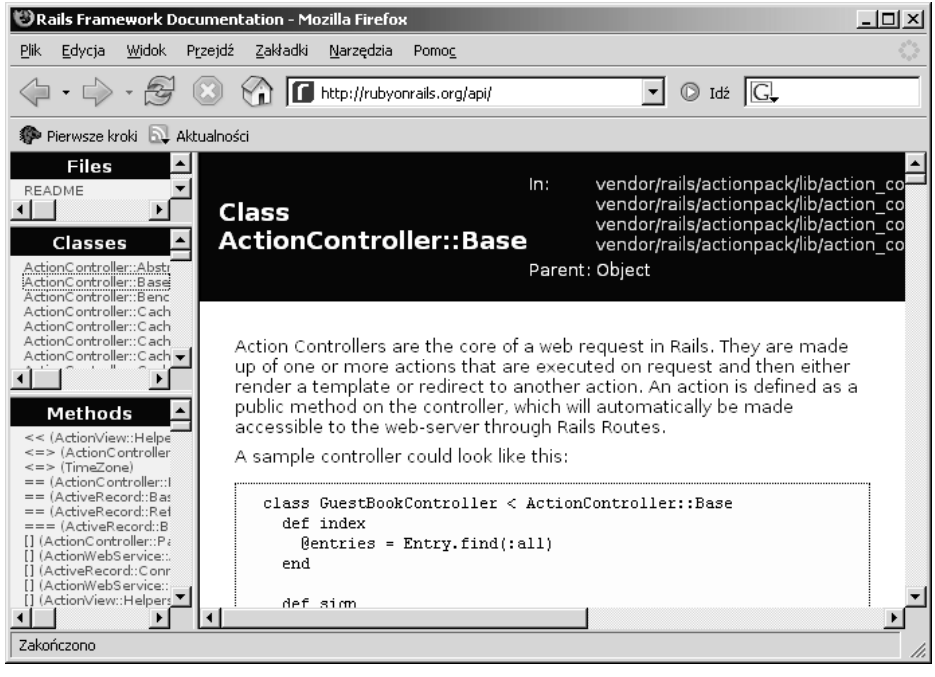
Budowanie widoku

Mamy kontroler, który potrafi wyświetlić tekst, jednak nasz dotychczasowy projekt nie pozwala na nic więcej. Jeśli chcemy stosować się do obowiązującej w Rails konwencji model-widok-kontroler, tekst powinien być wyświetlany raczej przez odpowiedni widok niż przez kontroler. Niemniej jednak nasz ułomny projekt można łatwo poprawić. Zamiast wyświetlać surowy tekst za pomocą kontrolera, należy po prostu użyć do tego widoku. Podobnie jak wiele innych środowisk programowania aplikacji dla sieci WWW, Rails potrafi tworzyć widoki z użyciem szablonów. W środowisku Rails szablon jest po prostu stroną HTML z osadzonym w niej kodem języka Ruby. Kod języka Ruby będzie następnie wykonywany na serwerze, który doda do strony HTML odpowiednią dynamiczną zawartość.

Dokumentacja

Inaczej niż wiele innych projektów rozwijanych na zasadach otwartego kodu źródłowego (ang. *open source*) środowisko programowania Rails posiada wspaniałą dokumentację. Znaleźć ją można pod adresem *http://api.rubyonrails.com*. Zawiera ona różnego rodzaju dokumenty przeglądowe, przewodniki a nawet filmy. Ponadto w witrynie tej można znaleźć oczywiście odpowiednią dokumentację API dla najnowszej wersji Ruby on Rails z pełnym zestawem dokumentów dla każdej klasy wchodzącej w skład API Rails. Dokumentacja ta dostarczana jest również wraz ze środowiskiem Rails, które instalujemy na naszym komputerze.

Rails nieprzypadkowo ma tak znakomitą dokumentację. Podobnie jak język Java, język Ruby dostarcza ułatwiającego tworzenie dokumentacji narzędzia RubyDoc, generującego dokumentację oprogramowania wprost z kodu źródłowego i komentarzy, które dodajemy do tego kodu. Kiedy instalujemy kolejny gem (moduł oprogramowania), instalowana jest również związana z nim dokumentacja. Rysunek 1.4 prezentuje na przykład dokumentację kontrolera.



Rysunek 1.4. Dokumentacja Rails dla kontrolera

Z pomocą Rails można wygenerować widok i parę skryptów pomocniczych, których ten widok będzie potrzebować. Należy w tym celu wpisać polecenie `generate`, by wygenerować nowy kontroler, `greeting`, wraz z widokiem `index` (w ten sposób wiążemy widok i kontroler ze sobą). Kiedy generator zapyta, czy zapisać nowy kontroler na starym (`overwrite controller`), należy wpisać `n`, informując, że nie:

```
> ruby script/generate controller Greeting index
      exists  app/controllers/
      exists  app/helpers/
      exists  app/views/greeting
      exists  test/functional/
overwrite app/controllers/greeting_controller.rb? [Ynaq] n
        skip  app/controllers/greeting_controller.rb
overwrite test/functional/greeting_controller_test.rb? [Ynaq] a
forcing controller
       force  test/functional/greeting_controller_test.rb
       force  app/helpers/greeting_helper.rb
      create  app/views/greeting/index.rhtml
```

Generator utworzył widok (plik *index.rhtml*) wraz z odpowiednimi plikami skryptów pomocniczych i testów. Teraz należy zachować metodę `index`, aby Action Pack mógł odnaleźć odpowiednią akcję kontrolera, należy jednak usunąć z metody `index` cały kod:

```
class GreetingController < ApplicationController
   def index
   end
end
```

Inaczej niż w większości szkieletów aplikacji WWW opartych na schemacie model-widok-kontroler, tutaj nie definiujemy widoku. Gdy kontroler nie wyświetla żadnych informacji na ekranie przeglądarki, Rails odnajduje odpowiedni widok, opierając się wyłącznie na konwencjach nazewniczych. Nazwa kontrolera określa nazwę katalogu widoku, a nazwa metody kontrolera — nazwę widoku. W tym przypadku szkielet Action Pack uruchomi widok *app/view/greeting/index.rhtml*. Nie musimy edytować żadnych plików XML, ani pisać żadnego dodatkowego kodu. Wystarczy, że stosować się będziemy do spójnych konwencji nazewniczych i Rails sam odczyta właściwie nasze intencje.

Teraz należy odpowiednio edytować widok. Znajdziemy w nim następujące dane:

```
<h1>Greeting#index</h1>
<p>Find me in app/views/greeting/index.rhtml</p>
```

Teraz należy odświeżyć stronę w przeglądarce, by zobaczyć poprzedni komunikat wyświetlony z pomocą kodu HTML. Rails informuje, gdzie można znaleźć plik, na wypadek gdybyśmy chcieli wyświetlić niezaimplementowany jeszcze widok. Rails oferuje jeszcze wiele takich udogodnień.

Wiązanie kontrolera z widokiem

W schemacie model-widok-kontroler widok zazwyczaj odpowiedzialny jest za wyświetlanie danych modelu dostarczanych przez kontroler. Przyjrzyjmy się teraz zmiennej instancji w kontrolerze i spróbujmy wyświetlić ją za pomocą widoku. Po pierwsze należy do kodu kontrolera dodać zmienną instancji o nazwie `@welcome_message`, przechowującą komunikat powitalny:

```
class GreetingController < ApplicationController
   def index
      @welcome_message = "Witamy. Oto twoja pierwsza aplikacja Rails"
   end
end
```

Teraz wyświetlimy nowy komunikat w widoku, dodając do jego kodu wyrażenie języka Ruby umieszczone między znacznikami `<%=` i `%>`. Rails wyświetli wartość wyrażenia umieszczonego między tymi znacznikami, tak jakby wartość ta została podana wprost (literalnie) w kodzie

HTML. Oto przykład kodu widoku, który wyświetli przygotowany komunikat powitalny w postaci nagłówka poziomu pierwszego:

```
<h1><%= @welcome_message %></h1>
```

Teraz należy przeładować stronę w przeglądarce. Pojawi się taki sam napis, jaki otrzymaliśmy w przykładzie 1.1, niemniej tym razem struktura aplikacji jest inna. W przykładzie 1.1 wyświetlaliśmy powitanie w kodzie kontrolera. Tutaj natomiast przygotowaliśmy *szablon RHTML* (ang. *RHTML template*). W szablonie tym znaczniki HTML dostarczyły statycznej struktury dokumentu i określiły style, natomiast kod języka Ruby dostarczył dynamicznej zawartości. W tym przypadku zmiennej określonej w kontrolerze.

Wyrażenia i skryptlety

Kiedy osadzamy kod języka Ruby w szablonie mamy do wyboru dwie możliwości. *Skryptletami* (ang. *scriptlets*) nazywamy kod języka Ruby umieszczony między znacznikami `<%` i `%>`. Skryptlety zależą od pewnych efektów zewnętrznych lub *wyniku działania* kodu Ruby. Natomiast *wyrażenia* (ang. *expressions*) są to, jak sama nazwa wskazuje, wyrażenia języka Ruby umieszczane między znacznikami `<%=` i `%>`. Wyrażenia prezentują *wartość* zwróconą przez kod języka Ruby.

Możemy teraz odrobinę poeksperymentować, badając interakcje między kontrolerem a widokiem. Wprowadziliśmy dla przykładu kilka zmian w kontrolerze i widoku, które wyświetlają powitanie, aby pokazać, jak w praktyce działają skryptlety i wyrażenia. Na początek zdefiniowaliśmy w kontrolerze kilka wartości:

```
class GreetingController < ApplicationController
   def index
     @age=8
     @table={"headings" => ["liczba", "liczba", "suma"],
             "body"     => [[1, 1, 2], [1, 2, 3] , [ 1, 3, 4]]
            }
   end
 end
```

Następnie pora przygotować widok[5] pokazujący wyrażenia i skryptlety, które będą korzystać z wartości zdefiniowanych w kontrolerze. Najpierw wyświetlimy wartość zmiennej instancji `@age` (określającej wiek), która została określona w kontrolerze:

```
<h1>Proste wyrażenie</h1>
<p>Tomek ma <%= @age %> lat.</p>
```

Za pomocą skryptletu wykonamy pętlę i wyświetlimy kolejne wartości wyrażenia dla każdej iteracji pętli:

```
<h1>Iteracja z pomocą skryptletów</h1>
<% for i in 1..5 %>
  <p>Nagłówek numer <%= i %> </p>
<% end %>
```

[5] Polskie litery uzyskujemy w następujący sposób: w edytorze *Eclipse/RadRails* najpierw zamykamy wszystkie pliki. Następnie klikamy prawym klawiszem myszy nasz projekt, by przywołać menu kontekstowe. W menu wybieramy polecenie *Properties* (Właściwości). Na karcie właściwości w sekcji *Text file encoding* (Kodowanie pliku tekstowego) wybieramy polecenie *Other* (Inne kodowanie) i w rozwijanym menu wybieramy kodowanie *UTF-8 — przyp. tłum.*

Na koniec użyjemy obu technik naraz, by wyświetlić zawartość tabeli `@table`:

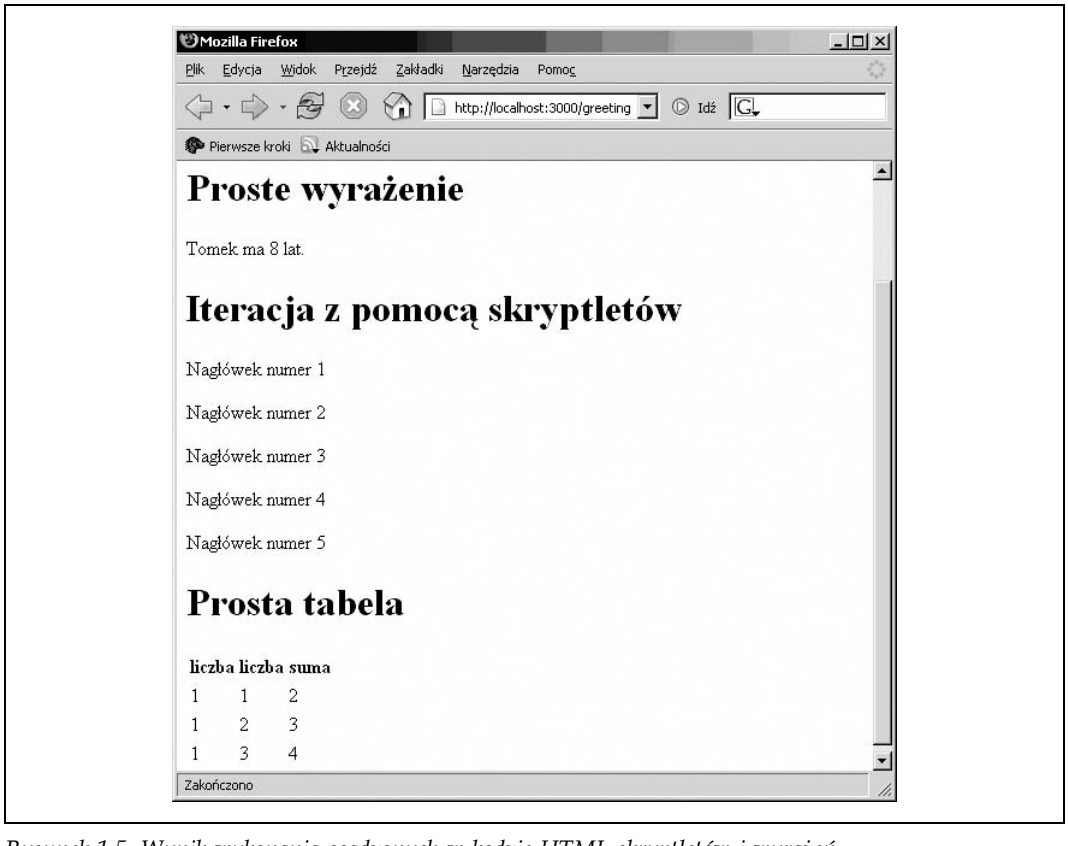
```
<h1>Prosta tabela</h1>

<table>
  <tr>
    <% @table["headings"].each do |head| %>
      <td>
        <b><%= head %></b>
      </td>
    <% end %>
  </tr>

  <% @table["body"].each do |row| %>
    <tr>
      <% row.each do |col| %>
        <td>
          <%= col %>
        </td>
      <% end %>
    </tr>
  <% end %>

</table>
```

W efekcie otrzymamy stronę przedstawioną na rysunku 1.5.



Rysunek 1.5. Wynik wykonania osadzonych w kodzie HTML skryptletów i wyrażeń

Co się dzieje za kulisami

Tak jak to zostało pokazane wcześniej, za każdym razem gdy przesyłamy adres URL, tworzymy żądanie HTTP, które uruchamia akcje kontrolera. Każdy z programistów, który pisze aplikację opartą na szkielecie model-widok-kontroler, staje przed wyborem, czy dla każdego żądania pisać nowy kontroler, czy też używać dla wszystkich jednego i tego samego kontrolera. Rails decyduje się na używanie osobnych kontrolerów, które to rozwiązanie nazywane jest *zakresem żądania* (ang. *request scope*). Każde żądanie HTTP powoduje przygotowanie nowej instancji kontrolera, co oznacza, że dla każdego żądania HTTP otrzymujemy również zupełnie nowy zestaw zmiennych instancji. Wybór tego rozwiązania ma dwojakie konsekwencje dla programisty:

- Zaletą jest to, że w naszych kontrolerach nie musimy się zajmować zarządzaniem różnymi wątkami, ponieważ każde żądanie otrzymuje własną kopię danych instancji kontrolera.
- Wadą natomiast jest utrudnienie współdzielenia danych instancji między poszczególnymi żądaniami. W szczególności jeśli zdefiniujemy zmienne instancji w metodzie akcji jednego kontrolera, to nie należy oczekiwać, że będzie można wykorzystywać je w późniejszych żądaniach HTTP. Konieczne będzie przygotowanie współdzielenia ich za pomocą sesji.

Co dalej

Zbudowaliśmy projekt Rails. Utworzyliśmy kontroler i przywołaliśmy go w przeglądarce. Ponadto przygotowaliśmy widok i dowiedzieliśmy się, w jaki sposób widoki współpracują z kontrolerami i z językiem Ruby. Są to całkiem dobre podstawy, niemniej widzieliśmy w działaniu tylko dwa elementy szkieletu programowego model-widok-kontroler. W kolejnym rozdziale dowiemy się, w jaki sposób działają modele. Utworzymy schemat bazy danych i pozwolimy systemowi Rails, by korzystając z tego schematu, utworzył dla nas odpowiedni model. Następnie użyjemy szkieletu Rails, by wspomógł nas w zarządzaniu wzajemnymi relacjami między różnymi częściami aplikacji.