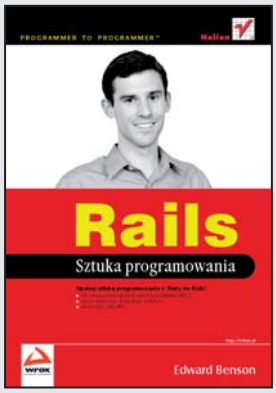
Rails. Sztuka programowania



Autor: Edward Benson
Tłumaczenie: Rafał Jońca
ISBN: 978-83-246-2071-5
Tytuł oryginału: The Art of Rails
Format: 172x245, stron: 336
Oprawa: twarda

- Jak osiągnąć największe korzyści z architektury MVC?
- Jak programować, korzystając z bloków?
- Jak tworzyć Web API?

Ruby on Rails przebojem wdarł się na rynek szkieletów aplikacji internetowych. Stworzony w architekturze MVC z wykorzystaniem popularnego języka Ruby, został entuzjastycznie przyjęty przez społeczność programistów. Główne założenia tego projektu to łatwość i przyjemność tworzenia kodu, a co za tym idzie – szybkie i efektywne tworzenie aplikacji internetowych. Liczba rozwiązań, które powstały z wykorzystaniem tego szkieletu, świadczy o jego wysokiej jakości oraz znacząco wpływa na wzrost popularności samego języka Ruby.

Jeżeli ta książka znalazła się w Twoich rękach, to z pewnością powyższe wiadomości nie są Ci obce. Kierowana jest ona do programistów, którzy znają już Ruby on Rails i pragną pogłębić swoją wiedzę. Dzięki książce „Rails. Sztuka programowania" dowiesz się, jak organizować swój kod tak, aby osiągnąć najwyższą efektywność i zachować zgodność z ideą DRY (Don't Repeat Yourself). Poznasz zasady zaawansowanego programowania w języku Ruby wraz z metaprogramowaniem oraz nauczysz się korzystać z programowania opartego na blokach. Ponadto zapoznasz się z wzorcami projektowymi dla technologii AJAX, interfejsami programistycznymi dla WWW, dekompozycją kodu HTML oraz nauczysz się w wydajny sposób rozwijać schemat bazy danych. Z pewnością zainteresuje Cię rozdział poświęcony programowaniu sterowanym zachowaniem. Te i wiele innych informacji znajdziesz w niniejszej książce, która wydaje się być obowiązkową dla każdego programisty Ruby on Rails!

- Cała prawda o aplikacjach internetowych
- Koncepcja Rails
- Architektura MVC
- Zarządzanie cyklem życia aplikacji
- Korzyści z zastosowania architektury MVC
- Zastosowanie technologii ActiveRecord
- Tworzenie Web API
- Wykorzystanie zasobów oraz obsługa żądań REST
- Zastosowanie formatów XML, RSS, RDF
- Sposoby ograniczania dostępu do API
- Wzorce zastosowań technologii AJAX
- Zasady programowania z wykorzystaniem bloków
- Mieszanie i łatanie klas
- Wykorzystanie dynamicznego kodu
- RSpec – programowanie sterowane zachowaniem
- Cykl programowania

Opanuj sztukę programowania w Ruby on Rails!

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

Spis treści

5

Piękne interfejsy aplikacji

W. Web wysiadł z autobusu i nerwowo przełknął ślinę na widok ogromnego tłoku i hałasu. Wysiadł jak najbliżej placu aplikacji, czyli około dwóch bloków dalej. W którąkolwiek stronę spojrzał, widział mnóstwo URI i innych oficjeli trzymających transparenty i głośno rozmawiających.

„To trochę jak na meczu piłki nożnej" — pomyślał Web, przypominając sobie nagrania różnych wieców, które w niego włożono przed wyjazdem. Powoli przeciskał się przez tłum URI, starając się dotrzeć bliżej placu.

URI były bardzo wysokie i szczupłe, przypominały nieco frytki z krótkimi nogami i ramionami jak u tyranozaurów. Z niewiadomych powodów wszystkie rozmawiały tak, jakby były z Bronksu, nawet te z Unikodu. Gdy grupy wygłaszały swoje slogany, ich nosowy głos okresowo skrzeczał przy bardziej ekscytujących je fragmentach.

„Nie dziwi mnie, że te osoby to popychadła" — pomyślał Web.

Prawdziwymi technokratami byli ci, którym nawet nie chciało się rejestrować własnych nazw domen. Adresy IP lśniły na ich piersiach. Osoby z adresami IP były nieco bledsze i nosiły okulary z grubymi szkłami.

— Uważaj, facet! Czy nie możemy liczyć nawet na odrobinę szacunku na własnym proteście?

Web szukał sceny tak intensywnie, że przypadkowo wpadł na jedną z takich osób.

— Przepraszam… 208.97.177.118 — powiedział, starając się tak odchylić głowę, by odczytać adres IP. — Szukam głównej sceny. Czy wiesz, gdzie się znajduje?

— A co ja, DNS? Ten zasób jest 301, przeniesiony na stałe! Ha, ha, ha, ha! — odwarknął URI.

Web stał jak wryty, bo nie wiedział, jak zareagować.

— Nie przejmuj się, kolego. — URI poklepał go po ramieniu. — To wspaniały dzień. Scena jest tam — mówiąc to, pokazał kierunek głową.

— Dziękuję bardzo. Czy mógłbyś mi powiedzieć, co jest powodem protestu?

— Nikt nie dał ci ulotki? — odpowiedziała postać. — Ach te dynamiczne URI! Nigdy nie można liczyć, że będą tam, gdzie się ich spodziewasz. Żądamy stabilizacji, ponieważ obecnie nie jesteśmy szanowani, choć na to zasługujemy.

— Jak mam to rozumieć? — odrzekł Web. — Wszyscy was używają. Jesteście wspaniali. Powinieneś widzieć, jak duży jest mój folder z zakładkami! W zasadzie pewnego dnia…

— Użyjesz nas i odrzucisz, użyjesz i odrzucisz. Tak obecnie wygląda sytuacja. Czujemy się jak odrzucane etykiety. Nikt nie chce zauważyć naszej głębi. Czy wydaje ci się, że chcemy mieć 300 znaków długości? Zapewne odpowiesz, że to nie ma znaczenia. Nikt nigdy nie zatrzymał się i nie zaczął podziwiać piękna czystych adresów URI! Dlaczego?

URI nie dał Webowi szansy odpowiedzi.

— I usługi społeczne — na nie również zasługujemy. Czy wiesz, ile pieniędzy dostały w zeszłym roku testy jednostkowe? Ponad 1,2 miliarda. Miliarda! Zgadnij, ile my dostaliśmy? Wielkie zero. Całkowite nic. Jak to się dzieje, że to my jesteśmy twarzą internetu, a nikt nie chce zwrócić na nas uwagi?

Web wiedział już, że to dopiero początek długiej tyrady, którą ten URI wygłaszał już wielokrotnie. Wiedział również, że musi dostać się na scenę, zanim skończy się protest.

— W pełni się z tym zgadzam. Zdecydowanie więcej środków należy przeznaczyć na URI — powiedział entuzjastycznie. — Słuchaj, bardzo przepraszam, ale naprawdę muszę się udać na scenę. Dziękuję za pomoc.

Zanim URI zdążył cokolwiek powiedzieć, Web wbił się w tłum i zaczął się przeciskać w kierunku krawędzi placu.

Ekscytującą cechą dobrego projektu jest to, że im bardziej się w niego zagłębiasz, tym bardziej Ci się podoba. W tym rozdziale postaram się wskazać, jak dobry projekt aplikacji stosującej model MVC szybko się spłaca, gdy rozbudowujesz aplikację poza początkową wersję. Niniejszy rozdział skupia się na projektowaniu, implementacji i mierzeniu internetowych interfejsów programistycznych.

Oferowanie interfejsu programistycznego do aplikacji internetowej staje się powoli istotnym elementem sukcesu. Daje użytkownikom lepszą kontrolę nad danymi i otwiera drzwi do kreatywnego wykorzystania informacji. Choć nie można mieć bezpośredniego zysku z reklam wyświetlanych na witrynach wykorzystujących dane z naszej witryny, pojawiają się korzyści związane z ogólnym reklamowaniem serwisu przez użytkowników i zyskami z umów partnerskich.

Internetowe interfejsy programistyczne promują wizję internetu, w którym witryny mogą się specjalizować i współpracować ze sobą, by lepiej i szybciej osiągać cele programistyczne. Obecnie niewielu programistów tworzy własne systemy nanoszenia danych na mapę. Zamiast

tego stosują doskonałe interfejsy do map oferowanych przez Google, Yahoo! i Microsoft. Części do wielokrotnego stosowania i możliwość specjalizacji to kluczowe czynniki sprzyjające rozwojowi nowych technologii. Tworzenie interfejsu dostępnego przez internet zwiększa jakość i złożoność aplikacji dla wszystkich użytkowników.

Usługi sieciowe to nieco mylący termin, ponieważ dotyczy zarówno ogólnej kategorii oprogramowania, jak i bardzo wąskiej grupy standardów. Przypomina to trochę otwarcie nowej firmy produkującej i sprzedającej płatki owsiane i nazwanie jej „Żywność". Stosując tę nazwę, utrudnia się jednoznaczną komunikację, ponieważ każda osoba musi wskazać, jaką żywność ma na myśli: żywność lub Żywność, a także przypomnieć o różnicy. Z tego powodu w książce używam terminu „internetowe interfejsy programistyczne" (*Web API*).

Niniejszy rozdział dotyczy Web API związanego z Ruby on Rails. Prezentuje styl tworzenia API w Rails, a także sposoby udanego integrowania go z aplikacją. Poza drobnym fragmentem o ActionWebService na końcu rozdziału w ogóle nie wspominam o SOAP i XML-RPC, ale zajmuję się raczej nowym stylem projektowania interfejsów programistycznych spopularyzowanym przez Rails. Styl ten stara się traktować projektowanie wersji z interfejsem graficznym i wersji jako usługi sieciowej jako dwie strony tego samego systemu zamiast dwóch różnych, mocno oddzielonych komponentów.

Dwa wielkie pomysły na Web API

Ten rozdział skupia się na dwóch wielkich pomysłach, które całkowicie zmieniły sposób projektowania i tworzenia internetowych interfejsów programistycznych. Dawniej adresy URL wskazywały na pliki ze zdalnego systemu plików, obecnie adresy URL z żądania wskazują na wirtualne elementy aplikacji internetowej. Gdy zaczynamy organizować adresy URL wokół tych wirtualnych elementów, a nie rzeczywistych plików, które je implementują, stają się one znacznie prostsze do odczytania i znacznie elastyczniejsze.

Drugim ważnym elementem prezentowanym w rozdziale jest fakt, iż interfejs programistyczny to jedynie alternatywny format odpowiedzi dla już istniejących akcji. Jeśli adres URL określa funkcjonalność, a nie plik, wtedy ten sam adres URL można zastosować do generowania danych w formacie HTML, XML, RDF lub nawet PDF. Wybór odpowiedniego formatu zależy od nagłówków HTTP i rozszerzenia wskazanego w adresie URL, na przykład *.html* lub *.xml*. To całkowicie nowy pomysł tworzenia interfejsów programistycznych, ponieważ w tej sytuacji Web API i właściwa strona WWW są tym samym elementem, współdzielą kod i różnią się jedynie sposobem wyświetlania wyników. Ten styl programowania prowadzi do mniejszej ilości kodu, gdyż eliminuje potrzebę powtarzania się w momencie implementacji różnych wersji dostępu do usługi.

Nowy adres URL — adresacja zagadnień, nie plików

Jeśli kiedykolwiek używałeś witryny Flickr, zapewne zauważyłeś ładny sposób zaprojektowania ich adresów URL. Na przykład:

```
http://flickr.com/photos/icygracy
```

Każda osoba z przynajmniej pobieżną znajomością witryny Flickr poparzy na adres i od razu domyśli się, co znajdzie na stronie po jego wpisaniu. Ten adres można zapamiętać, zamiast tworzyć z niego zakładkę, ponieważ stanowi pewien ściśle określony wycinek przestrzeni adresowej stworzony celowo, a nie w wyniku ograniczeń oprogramowania. Adres oznacza: „Zdjęcia użytkownika icygracy".

Adresy URL nie zawsze wyglądały tak elegancko, co moim zdaniem było jednym z wielu powodów powolnego rozprzestrzeniania się w przeszłości systemów Web API. Do niedawna traktowano je jako efekt uboczny procesu programowania, a nie jako krytyczny element etapu projektowania. W szczególności:

1. Adres URL traktowano jako sposób lokalizacji kodu, który napisał programista, a nie jako identyfikator pewnej konkretnej koncepcji. Dawniej adresy URL określały rzeczowniki, na przykład */viewBook.php*. To, jaką książkę chcemy obejrzeć, przekazywano jako argument. Ten sposób tworzenia adresów URL oznaczał, że traktowano je jako funkcje. Adresy same w sobie nie miały żadnego głębszego powiązania z ich użyciem. Stanowiły całość dopiero w powiązaniu z parametrami.
2. Adresów URL nie projektowano, gdyż stanowiły jedynie środek do osiągnięcia innego celu. Powstawały w sposób organiczny jako część struktury kodu lub organizacji serwera.
3. Adresy URL w sposób 1:1 odpowiadały plikom na serwerze, niezależnie od tego, czy była to zawartość statyczna, czy dynamiczna.

Traktowanie adresu URL jako „pięknego" oznacza, że jest on prosty na powierzchni, a jednocześnie mocno powiązany znaczeniowo z wykonywanym zadaniem.

Jako archeologiczny artefakt obrazujący trzy wymienione powyżej punkty warto przytoczyć łącze do towarów Dr. Seuss na witrynie Amazon.com z 2 marca 2000 roku:

```
http://s1.amazon.com/exec/varzea/search-handle-url/ref=gw_m_col_7/?index=
↪fixed- price%26rank=-price%26field-status=open%26field-browse=
↪68457%26field- titledesc%3DDr.%20Seuss
```

Ojej! Nie powinno dziwić, że tworzenie usług sieciowych traktowano jako działalność całkowicie niezależną od ich standardowych odpowiedników w postaci stron WWW. Czy nie lepiej byłoby zobrazować koncepcję książek Dr. Seuss w następujący sposób:

```
http://amazon.com/authors/dr_seuss/books
```

Przykład jest nieco nieuczciwy. Wcześniejszy adres URL zawiera kilka dodatkowych parametrów określających sortowanie i typ elementów, ale nie wpłynęły one znacząco na wygląd adresu. Obecnie projektanci stron WWW uważają, że lepiej tworzyć bardziej estetyczne i użyteczne adresy URL wskazujące **koncepcje wysokiego poziomu**, a nie **funkcje**, które obsługują te koncepcje. Ścieżka w stylu */authors/dr_seuss/books* z pewnością określa konkretną koncepcję, więc jest jednoznaczna dla programistów i użytkowników. Ścieżka */exec/varzea/* ↪*search-handle-url* jest z pewnością ścieżką do pewnego fragmentu kodu i nie ma większego znaczenia dla wszystkich poza grupą programistów.

Nowy sposób myślenia o adresach URL znacznie je oczyścił. Całkowicie zerwano z traktowaniem adresów URL jako odwzorowań na system plików. Zamiast odnosić się do obiektu

w systemie plików, adres URL wyraża pewną koncepcję związaną z możliwościami aplikacji. Wymaga to dodatkowego kroku tłumaczącego, nazywanego „kierowaniem", który wkrótce omówimy. Oto, w jaki sposób, krok po kroku, nowe podejście różni się od starszego.

1. Adres URL widziany jako nakierowanie na pewną koncepcję aplikacji, a nie na pewien fragment kodu aplikacji. Może reprezentować zarówno rzeczowniki (zasoby zarządzane przez aplikację), jak i czasowniki (akcje wykonywane na tych zasobach). Adres URL ma znaczenie również bez parametrów, ale parametry mogą wpłynąć na sposób prezentacji lub zasady ustalania zbioru wyników.
2. Adres URL odwzorowujący koncepcję aplikacji jest projektowany w ten sam sposób, co interfejsy obiektów w językach takich jak C++ lub Java. Struktura adresów URL powinna odpowiadać konkretnym wzorcom zapewniającym czytelność i prostotę. Wzorce te spisuje się i wymusza jako część aplikacji.
3. Poza zawartością statyczną nie istnieje żaden związek między adresem URL i plikiem po stronie serwera. Adres URL określa adresy koncepcji w wirtualnej przestrzeni, a w nie w systemie plików. W momencie otrzymania adresu URL następuje jego odwzorowanie na konkretny kod, który obsłuży żądanie.

Podstawowa sztuczka nie polega na tym, że pliki zaczyna się organizować w inny sposób (przecież na dyskach twardych serwerów witryny Flickr nie istnieje folder *photos/icygracy/*). Powstaje osobna warstwa abstrakcji na samym szczycie aplikacji internetowej służąca do adresacji poszczególnych funkcji. Wcześniej adresy URL korzystały z warstwy abstrakcji zapewnianej przez system plików, czyli z plików i folderów, ale teraz wszystko organizuje się wokół pewnego zbioru koncepcji. Choć kod aplikacji i jego struktura mają znaczenie dla programisty, to nie mają żadnego znaczenia dla użytkowników. Nowa warstwa abstrakcji oznacza, że adresy URL będą dla użytkowników przyjazne i jednocześnie będą miały określone znaczenie.

Sam pomysł nie jest specyficzny dla Ruby on Rails. Sporo serwerów WWW, między innymi Tomcat, umożliwia programistom tworzenie wielu kontekstów i procedur obsługi, które pozwalają uelastycznić adresy URL. Gdy serwer Tomcat wymaga setek wierszy kodu XML do poprawnej konfiguracji adresów URL, Rails wymaga jedynie dwóch lub trzech dzięki zastosowaniu **routera adresów**. Zgodnie ze stylem Rails, odwzorowanie adresów na funkcje nie jest elementem opcjonalnym, ale wymuszonym. Ponieważ Rails czyni odwzorowanie prostym i wymaganym, powstają eleganckie, intuicyjne adresy URL, które wkrótce stają się istotnym elementem całej aplikacji.

Aplikacja to interfejs

Drugi główny pomysł polega na tym, że to aplikacja **jest** internetowym interfejsem programistycznym, bo tak naprawdę była nim od początku. Każde pobranie strony to wywołanie w żądaniu pewnego interfejsu, który w odpowiedzi zwraca kod HTML. W zasadzie dochodzi do generowania kodu HTML tylko dlatego, że takiej odpowiedzi spodziewa się przeglądarka internetowa. Równie dobrze dane strony można by przedstawić w postaci pliku tekstowego, XML, RDF lub dowolnego innego formatu.

Jeśli wprowadziłeś w życie pierwszy pomysł — adresy URL wyrażają koncepcje, a nie pliki — zrobiłeś ogromny krok w przód, by zapewnić jednolity interfejs dla stron WWW i interfejsu

Web API. Tworząc strony WWW wokół adresów URI bazujących na koncepcjach, w zasadzie wykonałeś „interfejs" dostępowy również dla systemów zautomatyzowanych. Teraz wystarczy już tylko zaimplementować przygotowywanie odpowiedzi w postaciach innych niż od HTML.

Powstały kod aplikacji internetowej potrafi generować wyniki żądań HTTP w wielu różnych formatach. Witryna zaprojektowana właśnie w ten sposób jest bardzo wygodna dla programisty. Jeśli trzeba obsłużyć zwracanie danych w postaci plików CSV, wystarczy tylko zdefiniować wygląd pliku CSV i dodać opcję obsługi tego formatu w kontrolerze. To właśnie ten rodzaj podziału ról, który zapewnia architektura MVC.

Dalsza część rozdziału prezentuje sposoby implementacji akcji kontrolera, które potrafią same zdecydować o sposobie formatowania wyników. Jeśli w przyszłości ludzie zapytają, czy planujesz dodać do witryny interfejs programistyczny, możesz się uśmiechnąć i powiedzieć: „on już tam jest".

Odwzorowania adresów

Odwzorowania adresów to nowy gracz, dzięki któremu możliwe staje się wygodne tworzenie interfejsów Web API. Odwzorowanie odpowiada za dopasowanie i zamianę żądania HTTP na wywołanie określonego kodu (w przypadku Rails akcji kontrolera), który wygeneruje odpowiedź. Rails analizuje otrzymany adres URL względem serii wpisów podanych w pliku *config/routes.rb*. Ten krok odwzorowujący jest niezbędny do przekształcenia adresów URL z odniesień do plików na identyfikatory koncepcji. Po przejściu przez router adres wskazuje pewną przestrzeń koncepcyjną, a nie plik w systemie plików.

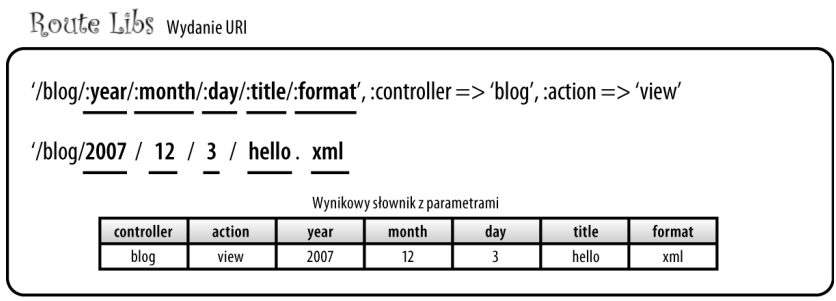
Oto jak działa proces odwzorowania: w momencie nadejścia nowego żądania, serwer WWW sprawdza, czy dotyczy ono pliku statycznego, bo jego ścieżka znajduje się w folderze *public/*. Jeśli tak, adres URL traktuje się jako lokalizację pliku w lokalnym systemie plików i wysyła jego zawartość do użytkownika. To podejście zapewni zgodność wsteczną z tradycyjnym systemem adresów URL. Jeśli ścieżka nie pasuje do pliku w folderze *public/*, żądanie trafia do Rails, gdzie router stara się je dopasować do jednego z istniejących odwzorowań.

Proces odwzorowania wykorzystuje serię definicji, które określają wzorce adresów URL — na nie powinna reagować aplikacja. Każde odwzorowanie to wzorzec do wypełnienia — przypomina to trochę grę Mad Libs dostosowaną do adresów URL. Porównywanie adresu do wzorca następuje w kolejności ich występowania w pliku *routes.rb*. To pierwsze dopasowanie decyduje o tym, jak żądanie zostanie obsłużone.

Od strony koncepcyjnej system Route Libs wygląda tak jak na rysunku 5.1.

Rozbicie adresu URL na elementy nie jest możliwe, jeśli aplikacja stosuje odwzorowanie adresów na strukturę folderów i plików. W świecie nakierowanym na dokumenty adres URL to po prostu ścieżka względem pewnego ustalonego folderu bazowego. W systemie z routerem adres URL dopasowuje się do szablonu pewnych koncepcji, którym odpowiadają określone akcje.

Rysunek 5.1



Odwzorowanie składa się z trzech części:

1. nazwy,
2. szablonu,
3. słownika wartości domyślnych i walidatorów.

Te trzy elementy tworzą razem kod o następującej postaci:

```
map.connect 'photos/:user_handle/:photoset',
   :controller => 'photos',
   :action => 'list'
```

Nazwa odwzorowania to nazwa metody wywoływanej dla obiektu `map`. Jeśli wywołamy `map.user` do utworzenia odwzorowania, zwiąże się ono z nazwą `user` i dodatkowo uzyskamy w kodzie dostęp do takich metod, jak `user_url` i `path_for_user`. Odwzorowanie `map.recipe` stworzyłoby nową nazwę `recipe`. Jeśli nie chcemy podawać nazwy dla odwzorowania (czasem nie ma ku temu powodów), stosujemy domyślną metodę `map.connect`, która tworzy odwzorowane nienazwane.

Następnym elementem w definicji jest szablon odwzorowania. W zaprezentowanym przykładzie jest to `photos/:user_handle/:photoset`. Każdy segment adresu rozpoczynający się od dwukropka oznacza zmienną do wypełnienia — odpowiada to pustym miejscom w grze Mad Libs. Wszystkie inne segmenty muszą wystąpić bez żadnych modyfikacji. Ścieżka w stylu `/my/super/secret/page` definiuje odwzorowanie, które nie zawiera w sobie żadnej zmienności; adres URL musi pasować co do joty. Odwzorowanie w stylu `secret/:code` dopasuje się do wielu różnych adresów URL — dowolnej dwusegmentowej ścieżki, która zaczyna się tekstem `secret/` i kończy inną zmienną. Jeśli dojdzie do dopasowania, kontroler otrzyma jedną zmienną w postaci `params[:code]`.

W Rails niektóre nazwy zmiennych są zarezerwowane do celów specjalnych, a niektóre są wręcz wymagane. Każde odwzorowanie musi definiować zmienną `:controller` czy to jako zmienną szablonu w definicji ścieżki, czy jako słownik opcji (prezentowany poniżej). Zmienna `:action` również ma specjalne znaczenie, ponieważ definiuje akcje kontrolera, która należy wywołać w momencie odpowiedzi na żądanie. Jeśli zmienna ta nie pojawi się w definicji odwzorowania, Rails założy, że powinna przyjąć wartość `index`. Jeśli akcja `index` nie istnieje w kontrolerze, Rails zgłosi błąd. Opcjonalna zmienna `:format` określa pożądany format odpowiedzi, który jest niezależnym od typu MIME sposobem wskazania preferowanej metody uzyskania odpowiedzi. Możliwość tak szybkiego wprowadzenia typów odpowiedzi do adresów URL stanowi jeden z powodów wyjątkowo łatwego tworzenia interfejsu programistycznego witryny.

Trzecim komponentem definicji jest opcjonalny słownik zawierający dodatkowe ustawienia odwzorowania, między innymi wartości domyślne i walidacje. Słownik często zawiera wartości domyślne dla zmiennych, które mogą nie pojawić się w szablonie ścieżki. Poprzedni przykład ustawia zmienną `:controller` na wartość `photos`. Słownik może także zawierać ograniczenia co do formatu danych poszczególnych fragmentów ścieżki zamienianych na zmienne podawane w postaci wyrażeń regularnych. Te dodatkowe wyrażenia regularne mogą spowodować niedopasowanie się adresu URL do odwzorowania, zmniejszając liczbę potrzebnych sprawdzeń. Wymóg, by wszystkie identyfikatory były liczbami, to doskonały przykład przeniesienia sprawdzenia z kodu kontrolera lub modelu do odwzorowań.

Plik *routes.rb* powinien być przyjemnym, łatwym w odczycie plikiem z adresami URL:

```
# Odwzorowuje adresy w stylu '/dinner/12' na akcję Recipe::Feature,
# ustawiając parametr dinner_feature_id.
map.connect 'dinner/:dinner_feature_id',
   :controller => 'recipe',
   :action => 'feature'

# Odwzorowuje kalendarz użytkownika. Stosuje domyślny rok, miesiąc i dzień.
map.connect 'user/:user_id/calendar/:year/:month/:day',
   :controller => 'calendar',
   :action => 'view',
   :year => Time.now.year,
   :month => Time.now.month,
   :day => Time.now.day

# Odwzorowuje adresy w postaci /A/B/C na akcję A::B z parametrem ID=C.
map.connect ':controller/:action/:id''
```

Ponieważ plik *routes.rb* stanowi połączenie między wszystkimi niestatycznymi żądaniami HTTP i kodem aplikacji, łatwo zauważyć, że projektowanie adresów URL to bardzo ważny krok w trakcie tworzenia aplikacji Rails. Nawet najdalsze zakamarki witryny WWW muszą mieć zdefiniowane odwzorowanie. Odwzorowania to zatem publiczny interfejs aplikacji, nawet jeśli domyślnie zwracają tylko kod HTML. W takim interfejsie metody prywatne i chronione wykonujące niskopoziomowe zadania są całkowicie ukryte przed użytkownikiem.

Podobnie dzieje się w przypadki interfejsu Web API, który całkowicie ukrywa przed użytkownikiem rzeczywistą strukturę plików aplikacji — adresy URL i pliki istniejące na serwerze, które obsługują te adresy, są od siebie całkowicie odseparowane.

Choć szablony adresów URL dają ogromną swobodę, zawsze pamiętaj o staraniu się, by były przewidywalne i proste. Unikaj tworzenia bardzo długich odwzorowań z mnóstwem parametrów, bo wtedy adresy URL wcale nie będą lepsze od adresów z końca lat 90. XX wieku. Kluczem jest prostota: zredukuj aplikację do podstawowych koncepcji, którymi są zainteresowani użytkownicy, i wykonywanych na nich akcjach. Wszystko inne — informacje dookreślające, takie jak sposób sortowania, kody referencyjne i stron wyników wyszukiwania — powinno być przekazywane jako parametry URL. W ten sposób adres URL zawsze reprezentuje konkretną koncepcję, a parametry to jedynie dodatkowy i opcjonalny tryb uszczegółowienia żądania. W rozdziale 6. pojawią się dodatkowe wskazówki związane z tym tematem, ponieważ w dużej mierze dotyczy on rozbicia interfejsu publicznego na koncepcje zwane zasobami.

Anatomia wywołania Web API

Odwzorowania adresów URL otwierają drzwi do interfejsów programistycznych bazujących na HTTP i przyjaznych dla użytkownika, które reprezentowane są przez wirtualne zakończenia obsługujące określoną funkcjonalność. Zakończenia nie wystarczają jednak do wykonania pełnego wywołania Web API. Z tego powodu wywołanie składa się z czterech komponentów.

Cztery komponenty wymienione w poniższej tabeli wyglądają bardzo podobnie do komponentów tradycyjnego wywołania metody, ale z kilkoma wyjątkami. Wywołanie funkcji określa typ zwracanej odpowiedzi, a polecenie HTTP trafia do wywołania metody jako część sterująca jej wykonaniem.

Komponent	Zapewniany przez	Cel
Kontroler i akcja.	Definicję odwzorowania.	Wybranie odpowiedniej klasy kontrolera i wykonanie znajdującej się w niej metody w odpowiedzi na nadesłane żądanie.
Format odpowiedzi.	Nagłówki HTTP lub definicję odwzorowania (zmienna `:format`).	Określa, jaki format odpowiedzi należy zapewnić.
Parametry żądania.	Dane formularza lub parametry adresu URL.	Zapewnia dodatkowe parametry pozwalające wypełnić podstawowe żądanie.
Polecenie HTTP.	Żądanie HTTP.	Zakłada podstawową naturę żądania: pobieranie danych, ich modyfikacja, dodanie lub usunięcie.

Odwzorowania to jedynie część większego systemu. Witryna WWW to zbiór zakończeń zdefiniowanych i udostępnianych przy użyciu czterech komponentów zdefiniowanych w tabeli. Najczęściej zakończenia zwracają dane w postaci kodu HTML, ale z racji możliwości określenia alternatywnego formatu mogą również działać jako interfejs programistyczny. W następnym punkcie przyjrzymy się obsłudze przez tę samą akcję wielu formatów danych.

Kolidujące, a czasem niezgodne punkty widzenia

Powróć do tej części rozdziału 5. po przeczytaniu rozdziału 6., związanego z architekturą REST. Pomysły dotyczące Web API prezentowane tutaj różnią się nieco od propozycji z następnego rozdziału. Żadne z rozwiązań nie jest zdecydowanie lepsze. Każde ma swoje wady i zalety, prowadzi do nieco innego sposobu traktowania aplikacji.

Nakładanie API

Po zaprojektowaniu wzorców URL reprezentujących koncepcje i akcje aplikacji internetowej czas wprowadzić formaty odpowiedzi różne od HTML do tych samych mechanizmów, które generują strony WWW. Nakładanie API przypomina tworzenie nowe widoku witryny, ale tym razem te nowe, alternatywne wyniki są najprawdopodobniej generowane dynamicznie, a nie

przechowywane w postaci plików ERB. Nałożenie API wymaga dwóch kroków: najpierw trzeba zapewnić reagowanie widoku na zgłoszenia różnych formatów danych, a następnie upewnić się, że widok potrafi wygenerować odpowiedź w formacie wskazanym przez użytkownika.

Metoda respond_to

Szkielet `ActionController` zapewnia kontrolery z bardzo sprytną metodą o nazwie `respond_to`, która znacznie upraszcza generowanie odpowiedzi w różnych formatach na podstawie tej samej akcji. Metoda tworzy obiekt, który łatwo wykorzystać do określenia kodu, który ma się wykonać tylko w przypadku poproszenia o konkretny format danych.

Zastosowanie metody `respond_to` ułatwia rozdzielenie właściwej implementacji akcji od części odpowiedzialnej za wygenerowanie odpowiedzi. Najlepiej najpierw wykonać wszystkie prace związane z właściwą akcją, a dopiero na samym końcu skorzystać z bloku `respond_to`, by zwrócić wynik w wymaganym formacie. Poniższy kod stanowi przykład zastosowania tej strategii:

```
class SomeController
   def action

      # ---------------------------------------
      # Wykonaj działania właściwe dla akcji.
      # ---------------------------------------

      # Następnie przygotuj wynik w formacie wskazanym przez użytkownika.
      respond_to do |:type|
         type.html { # Najczęściej puste ciało bloku. }
         type.xml  { # Najczęściej wywołanie .to_xml. }
         type.rdf  { # Najczęściej wywołanie .to_rdf. }
         type.js   { # Najczęściej fragment RHTML lub szablon RJS. }
      end
   end
end
```

Ten szablon działa bardzo dobrze, bo stara się zawrzeć w bloku `respond_to` wszystkie możliwe formaty odpowiedzi. Oto, w jaki sposób wygląda szablon po zastosowaniu go w rzeczywistej akcji kontrolera, w tym przypadku w akcji przeglądania zdjęcia:

```
class PhotoController
  def show

    @photo = Photo.find(params[:id])  # Ustawienie zmiennej wymaganej przez widok.
    respond_to do |:type|             # Zrenderuj widok w wymaganym formacie.
      type.html {}                                # Domyślnie użyj formatu RHTML.
      type.xml { render :xml => @photo.to_xml }
      type.rdf { render :rdf => @photo.to_rdf }   # Wymaga dodatku acts_as_rdf.
      type.js { render :partial => 'photo', :locals => {:photo => @photo }

    end
  end
end
```

Jedyną operacją wykonywaną przez akcję jest znajdowanie zdjęcia. Samo wyświetlenie zdjęcia obsługuje kod widoku wywoływany przez akcję. Gdyby trzeba było zmienić sposób znajdowania zdjęcia (bo jego identyfikator będzie powiązany z użytkownikiem), wystarczy wprowadzić poprawkę tylko w jednym miejscu dla wszystkich formatów odpowiedzi.

Czasem operacja wykonywana przez akcję jest znacznie bardziej złożona niż proste wczytanie zdjęcia. Trzeba być przygotowanym do wyłapania błędów, które mogą wystąpić, i na wyświetlenie komunikatów błędów w odpowiednim formacie. Obsługa błędów nieco psuje nam nasz wyidealizowany świat `respond_to`: w jaki sposób uniezależnić się od szczegółów formatu odpowiedzi, skoro w pewnych sytuacjach błąd może wystąpić w połowie akcji? Na szczęście istnieje rozwiązanie tego problemu.

Przypomnij sobie poprzedni rozdział — mówiłem w nim o technice, która osadza większość ciężkiej funkcjonalności w warstwie modelu i wykorzystuje własne wyjątki do zgłaszania błędów. Stosując tę strategię obsługi błędów oraz własne wyjątki z dobrze napisanymi komunikatami, powstanie system pozwalający utrzymać prostotę akcji.

Zamiast wypełniać kod akcji kolejnymi warstwami warunków sprawdzających błędy, tworzymy obiekty modeli, które zgłaszają wyjątki, gdy tylko zostaną zauważone. W ten sposób masz pewność, że wszystko, co do tej pory wykonałeś, powiodło się, więc nie musisz wprowadzać dodatkowych sprawdzeń. Wiesz jednak, że poza sceną system zgłosi wyjątek, gdy tylko napotka istotny błąd. Aby obsłużyć sytuacje wyjątkowe, otocz kod akcji blokiem `rescue`, który przechwyci wyjątek i zgłosi go użytkownikowi w sposób zależny od wskazanego formatu wyników.

Oto szablon wieloformatowej akcji wraz z obsługą błędów. Zauważ, że kod jest bardzo podobny do wcześniejszego, ale wybór formatów odpowiedzi pojawia się dwukrotnie — raz dla standardowej odpowiedzi i raz dla sytuacji wyjątkowej:

```
class SomeController
    def action

       # ---------------------------------------
       # Podstawowe zadania wykonywane przez akcję.
       # ---------------------------------------

       # Po ich przeprowadzeniu wygeneruj wynik w odpowiednim formacie.

       respond_to do |:type|
          type.html { # Najczęściej pusty blok. }
          type.xml  { # Najczęściej wywołanie .to_xml. }
          type.js   { # Najczęściej fragment RHTML lub szablon RJS. }
       end

    rescue => err
       # Ojej! Wystąpił błąd, więc wygeneruj odpowiedni wynik.
       respond_to do |:type|
          type.html { # Najczęściej przekierowanie z komunikatem. }
          type.xml  { # Najczęściej struktura z błędem lub błąd HTTP. }
          type.js   { # Najczęściej bezpieczna odpowiedź specyficzna dla akcji
                      # wraz z komunikatem bazującym na JavaScripcie. }
       end
    end
end
```

Podobnie jak w przypadku każdego wzorca, szablon ten nie stanowi panaceum na wszystkie możliwe błędy. Z pewnością pojawią się sytuacje wymagające dokładniejszego obsłużenia w akcjach kontrolera, by zapewnić najbardziej odpowiednie zachowanie. Ten podstawowy szablon stanowi solidną podstawę, dzięki której można zapewnić prostotę kodu akcji.

Używając bloku `respond_to`, zamieniamy odwzorowania, które kiedyś stanowiły zakończenia witryny WWW, w coś na kształt programowych zakończeń aplikacji WWW. Po dobrym zaprojektowaniu witryny niewiele pracy trzeba włożyć, by istniejące akcje obsługiwały inne formaty danych, na przykład XML. Implementacja akcji już istnieje, więc wystarczy jedynie zdefiniować sposób konwersji wyników akcji na odpowiedni format.

Zapis wyniku w formacie innym niż HTML

Tworzenie wyników w formatach innych niż HTML jest często łatwiejsze od tworzenia strony HTML, ponieważ jesteś zainteresowany jedynie surowymi danymi (chyba że tworzysz wyniki dla medium wizualnego takiego jak PDF). Cały proces staje się znacznie bardziej złożony niż HTML, gdy już bardzo szczegółowo określisz format publicznego API. Gdy zmiany w wyglądzie HTML można wprowadzić w dowolnym momencie bez poważnych efektów ubocznych, zmiana struktury XML może doprowadzić do traumatycznych przeżyć wielu użytkowników API. Postępuj bardzo ostrożnie w trakcie prac na projektem formatu danych XML lub RDF, gdyż generalnie bardzo trudno będzie zmienić zdanie po upublicznieniu usługi.

Ta część rozdziału opisuje trzy formaty danych, których zapewne chciałbyś użyć jako API: XML, RSS i RDF.

> **Grupowanie XML, RSS i RDF**
>
> Puryści zapewne stwierdzą, że grupowanie XML, RSS i RDF przypomina nieco grupowanie jabłek i pomarańczy. XML to składnia reprezentacji danych, RDF to model koncepcyjny dla danych bazujących na grafach, które można opisać językiem XML, a RSS to standard rozpowszechniania informacji, który również korzysta z języka XML. Pomimo różnej natury grupowanie ich razem ma sens, ponieważ z punktu widzenia API to obecnie trzy najważniejsze formaty danych stosowane w internecie.

XML

Jeśli nie przejmujesz się tym, że dokumenty XML będą dokładnie odpowiadały schematowi bazy danych, tworzenie wyników w postaci kodu XML w aplikacji Rails może skrócić się do jednego wiersza. Każdy obiekt ActiveRecord zawiera metodę `to_xml`, która przechodzi przez wszystkie pola obiektu i zapisuje je jako znaczniki XML, więc blok `respond_to` w postaci:

```
@user = User.find(params[:id])
respond_to do |:type|
   type.xml { render :xml => @user.to_xml }
end
```

spowoduje wygenerowanie następującego kodu:

```
<user>
   <first_name>Jan</first_name>
```

```
    <last_name>Kowalski</last_name>
    <address>...</address>
</user>
```

Funkcja `to_xml` przyjmuje dodatkowo słownik opcji, który może albo ograniczyć liczbę pól umieszczanych w kodzie XML, albo rozszerzyć szeregowanie na powiązane obiekty modeli. Przekazanie słownika `:include = > [ :asocjacja1, :asocjacja2]` spowoduje, że system zapisujący przejdzie dodatkowo przez asocjacje i dołączy je do wynikowego kodu XML.

Najprostszym sposobem dostosowania wyniku XML do własnych potrzeb (poza automatycznie generowanym kodem przez ActiveRecord) jest użycie szablonów RXML. Pliki te są bardzo podobne do plików RHTML i znajdują się w tym samym miejscu w strukturze projektu, ale korzystają z rozwiązania nazwanego `Builder` zamiast `ActionView`.

Pliki RXML to standardowe pliki w języku Ruby, które otrzymują egzemplarz klasy `Builder` o nazwie `xml` i wykorzystują go do utworzenia dokumentu XML. Niech rozszerzenie Cię nie zmyli — są to pliki stosujące podejście z kodowaniem najpierw.

Gdy wynikiem pliku RHTML jest renderowanie wszystkiego poza znacznikami `< % . . . % >`, wynikiem dokumentu RXML jest zbiór wywołań metod zmiennej `xml`. Innymi słowy, pliki RHTML stosują podejście z dokumentem najpierw, a pliki RXML z kodem najpierw.

Aby utworzyć znacznik, po prostu wywołaj nazwę znacznika jako metodę obiektu `Builder`. Ta metoda w rzeczywistości nie istnieje, ale powoduje wywołanie procedury `method_missing`, którą system tworzenia XML traktuje jako instrukcję do utworzenia nowego znacznika. Sposób programowania wykorzystujący procedurę `method_missing` opiszę w rozdziale 10. Wygląd znacznika zależy od sposobu wywołania metody:

- Zastosowanie metody bez argumentów tworzy znacznik pusty, więc `xml.br` tworzy `<br />`.
- Po podaniu argumentu system tworzy znacznik zawierający wartość tekstową, więc `xml.h1("Witaj!")` tworzy kod `<h1>Witaj!</h1>`. Atrybuty podaje się jako słownik umieszczony w ostatnim argumencie dowolnego wywołania metody tworzącej znacznik. Utworzenie hiperłącza za pomocą kodu wymaga napisania poniższego tekstu:

  ```
  xml.a "Witryna Wydawnictwa Helion", :href => "http://helion.pl"
  ```

- Po przekazaniu bloku system tworzy element zawierający dowolne znaczniki utworzone w tym bloku, więc polecenie:

  ```
  xml.people {
    xml.person {
       xml.first_name("Jan")
    }
  }
  ```

  spowoduje wygenerowanie kodu:

  ```
  <people>
     <person>
        <first_name>Jan</first_name>
     </person>
  </people>
  ```

Wystarczy połączyć te trzy zachowania razem, by uzyskać wszystkie klocki potrzebne do zbudowania dokumentu XML o dowolnej złożoności. Oczywiście istnieją pewne dodatkowe elementy, takie jak deklaracja typu dokumentu, komentarze XML itp., więc warto wcześniej zajrzeć do dokumentacji klasy `Builder` dostępnej pod adresem *http://builder.rubyforge.org/*.

RSS

Kanały RSS zapewniają bardzo ważny sposób luźnego powiązania witryny z jej użytkownikami. Często użytkownik nie chce odwiedzać bloga lub innej aplikacji każdego dnia, ale jest zainteresowany powiadomieniem o wystąpieniu na niej nowych elementów. Stosując czytnik RSS, użytkownik automatycznie dowiaduje się o wszystkich istotnych zmianach na witrynie i może zdecydować, czy chce ją odwiedzić, czy też nie.

Tworzenie kanału RSS dla aplikacji Rails to doskonałe ćwiczenie uczące RXML, gdyż to właśnie dzięki niemu wykonuje się tego rodzaju prace. Zacznijmy od zaprezentowania szablonu z kanałem RSS 2.0, który zawiera jeden artykuł z bloga:

```
<?xml version="1.0"?>
<rss version="2.0">
   <channel>
      <title>Blog The Art of Rails</title>
      <link>http://www.artofrails.com/</link>
      <description>W poszukiwaniu wyrafinowanych sposobów programowania.
      ↪</description>
      <pubDate>Sat, 07 Dec 2007 00:00:01 GMT</pubDate>
      <item>
         <title>Witam ba blogu</title>
         <link>http://www.artofrails.com/posts/1</link>
         <pubDate>Sat, 07 Dec 2007 00:00:01 GMT</pubDate>
         <description>To mój pierwszy wpis. Czy kiedykolwiek...</description>
      </item>
   </channel>
</rss>
```

Główny znacznik nosi nazwę `rss`, a tuż za nim znajduje się znacznik `channel`, który zawiera hipotetyczny kanał bloga. Po kilku znacznikach opisujących sam kanał znajduje się seria znaczników `item`, zawierająca opisy poszczególnych wpisów na blogu. Zamiana tego kodu na RXML wymaga przedstawienia poszczególnych znaczników jako wywołań metod obiektu `Builder`. Poniżej przedstawiam wynikowy plik *rss.rxml* zaczerpnięty z książki Scotta Raymonda *Ajax on Rails* (wydawnictwo O'Reilly).

```
xml.instruct!
xml.rss "version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/" do
   xml.channel do
      xml.title       "Nazwa kanału"
      xml.description "Opis kanału"
      xml.link        url_for :only_path => false, :controller => 'posts'
      xml.pubDate     CGI.rfc1123_date @posts.first.updated_at if @posts.any?
      @posts.each do |posts|
         xml.item do
            xml.title       post.name
            xml.link        url_for :only_path => false,
                                    :controller => 'posts',
                                        :action => 'show',
                                            :id => post.id
```

```
            xml.description post.body
            xml.pubDate     CGI.rfc1123_date post.updated_at
            xml.guid        url_for :only_path => false,
                                    :controller => 'posts',
                                        :action => 'show',
                                            :id => post.id
            xml.author      "#{post.author.email} (#{post.author.name})"
          end # Koniec <item>.
        end # Koniec posts.each.
      end # Koniec <channel>.
    end # Koniec <rss>.
```

Zauważ, że cały kod zależy tylko od istnienia jednej zmiennej, `@posts`, ustawionej przez kontroler. Podobnie jak adres URL użyty dla tej akcji zwróciłby standardowo stronę HTML z listą wszystkich blogów, tak ten sam adres po poproszeniu o format RSS zwraca wynik w formacie odpowiednim dla czytnika RSS.

RDF

Format RDF to model zasobów przystosowany do potrzeb internetu, a także język opisu zasobów, ich właściwości i związków między nimi. Język ten traktuje świat jako graf z węzłami reprezentującymi zasoby. Zarówno węzły, jak i powiązania między nimi reprezentują adresy URI (łuki mogą również zawierać zwykłe dane tekstowe), co pozwala RDF opisywać i obiekty abstrakcyjne, i zasoby w sieci.

Choć RDF nie jest obecnie tak popularny jak podstawowy XML w kwestii wymiany danych, jego znaczenie powoli rośnie, gdyż upraszcza łączenie informacji z wielu źródeł. Przechowywanie informacji w postaci grafów jest trudne, ale i wyjątkowo elastyczne. Jeśli pobierasz informacje na temat zasobu z kilku źródeł, ich złączenie nie jest trudniejsze niż w sytuacji, gdyby wszystkie znajdowały się w tym samym miejscu. Ponieważ cały czas rośnie liczba aplikacji łączących w sobie elementy wielu innych aplikacji, RDF będzie odgrywał coraz to większą rolę w wymianie danych.

Najprostszym sposobem zapewnienia RDF jako formatu przesyłu danych jest użycie dodatku `acts_as_rdf` z witryny *www.artofrails.com*. Moduł ten dodaje do obiektów modeli ActiveRecord metodę `to_rdf` o charakterystyce bardzo podobnej do metody `to_xml`. Dodatkowo tę samą metodę dodaje do obiektów kolekcji, którą można później renderować jako całą grupę.

Domyślnie `acts_as_rdf` używa nazwy domeny skonfigurowanej dla witryny i ścieżki zasobu jako przestrzeni nazw. Nazwy właściwości zgaduje na podstawie schematu bazy danych. Wszystko to można zmienić w razie potrzeby, instruując metodę `to_rdf`, by stosowała wskazaną serializację i określone asocjacje.

Dodanie własnych typów MIME

Aby Rails potrafił odpowiedzieć na typ żądania zgłaszany w nagłówku `Accept`, musi znać sam format. Domyślnie Rails rozpoznaje tylko kilka prostych typów formatów odpowiedzi:

- HTML,
- XML,
- JS (używany w żądaniach AJAX-a).

Rails uzyskuje informację o formacie danych w taki sam sposób jak większość innych aplikacji internetowych lub protokołów — za pomocą typów MIME. Typ MIME to standard, który powstał początkowo dla programów pocztowych, które musiały zacząć obsługiwać złożone dane. Choć wiele typów MIME ma zastosowanie tylko w świecie klientów e-maila, sam sposób oznaczania formatów danych służy obecnie do opisu zawartości w wielu protokołach internetowych.

W zasadzie wszystkie istniejące typy danych, z których chciałbyś skorzystać w internecie, mają już przypisany swój typ MIME. Wystarczy go odnaleźć i dodać do konfiguracji Rails. Typami MIME zarządza organizacja IANA i są one dostępne na witrynie *http://www.iana.org/*
➥*assignments/media-types/*. Istnieje dziewięć kategorii najwyższego poziomu:

- application,
- audio,
- example,
- image,
- message,
- model,
- multipart,
- text,
- video.

Każda kategoria to łącze do wszystkich zarejestrowanych typów, które się w niej znajdują. Po znalezieniu typu wystarczy tylko stworzyć ostateczną wersję typu, czyli połączyć nazwę kategorii z nazwą typu znakiem ukośnika. Jeśli chcesz dodać obraz JPEG jako potencjalny rodzaj odpowiedzi, po znalezieniu wpisu `jpeg` w kategorii `image` wiesz, że odpowiednim typem MIME jest `image/jpeg`.

Jeśli nie potrafisz odnaleźć odpowiedniego typu opisującego format danych, możesz stworzyć własny typ, o czym się wkrótce przekonasz.

Rejestracja typów w Rails

Gdy wiesz już, jaki typ MIME chcesz obsłużyć, jego rejestracja w Rails nie powinna sprawić najmniejszych problemów. Otwórz plik *config/initializers/mime_types.rb* i dodaj następujący wiersz:

```
Mime::Type.register "image/png", :png
```

Ten jeden wiersz powoduje, że cała aplikacja Rails rozpoznaje nowy typ, więc możemy odnieść się do niego w kontrolerach, podobnie jak to czynimy dla typów wbudowanych `html`, `xml` i `js`.

Pierwszy parametr funkcji `Mime::Type.register` to opis typu MIME. To właśnie tę informację przekazuje przeglądarka w nagłówku `Accept` protokołu HTTP. Drugi parametr to symbol, który chce się stosować w aplikacji Rails. Symbol ten odwzorowujemy na funkcję wspomagającą, która obsługuje zmienną `:format`.

Używając wcześniejszej rejestracji typów, akcja kontrolera potrafiąca wyświetlić sieć powiązań użytkownika jako obraz PNG miałaby następującą postać bloku `respond_to`:

```
respond_to do |:type|
   type.html { # standardowy tryb }
   type.png { render_png_image }
end
```

Tę funkcjonalność może wywołać zdalny użytkownik, stawiając `image/png` jako format o najwyższym priorytecie w nagłówku `Accept` lub też kończąc adres URL rozszerzeniem `.png`.

Tworzenie własnego typu MIME

Jeśli rzeczywiście tego potrzebujesz, możesz utworzyć własny, nieoficjalny typ dla nowego formatu danych, z którego właśnie korzystasz (bo na przykład wymyśliłeś sposób strumieniowego przesyłania hologramów). Ogólne wytyczne co do tworzenia nowych typów są następujące: wybierz najbardziej odpowiedni typ wysokiego poziomu i połącz go z własną nazwą podtypu, umieszczając jednak na początku tekst `x-`. Najczęściej nieoficjalne typy MIME trafiają do kategorii `application` (ponieważ wszystko jest specyficzne dla aplikacji, dopóki nie stanie się standardem), ale zastosuj dowolną, która wydaje się prawidłowa. Kilka dodatkowych uwag dotyczących nazewnictwa:

- Jeśli nazwa składa się z wielu słów, rozdziel je kropkami, na przykład `application/x-wielowymiarowy-hologram`.
- Jeśli nowy format bazuje na już istniejącej składni, na przykład XML, dodaj nazwę składni oddzieloną znakiem plus jako dodatkowy element po nazwie typu; przykład:

  ```
  application/x-wielowymiarowy-hologram+xml
  ```

Ograniczanie dostępu do API w sposób zgodny z Rails

W zależności od rodzaju oferowanego interfejsu warto rozważyć ograniczanie dostępu. Jeśli interfejs udostępnia alternatywne formaty kierowane do ludzi, na przykład PDF, dodatkowe ograniczenia mogą nie być potrzebne, bo sposób ich stosowania będzie bardzo podobny do wersji HTML. W przypadku formatów kierowanych do komputerów, takich jak XML lub RDF, ograniczanie dostępu to bardzo ważny krok chroniący stabilność aplikacji.

Rozważmy następującą sytuację. Oferujesz katalog różnych pizzerii z całego świata wraz ze wszystkimi menu — innymi słowy doskonała witryna dla wszystkich zafascynowanych pizzą.

Dodatkowo napisałeś API, które umożliwia wysyłanie zapytań i uzyskiwanie odpowiedzi w postaci XML, by mogły z nich korzystać inne programy. Wszystko to oferujesz za darmo, ponieważ chcesz promować pizzę na całym świecie.

Jednak w Japonii następuje szalona moda: cały kraj sięga po pizzę, gdy jedna z gwiazd pop ogłasza, że zatrzymuje się tylko w hotelach, w których pobliżu można zamówić pizzę z majonezem. Wszystkie agencje turystyczne zaczynają nanosić na mapy oferowanych hoteli pobliskie pizzerie. Ich źródło informacji na temat pizzerii? Twój interfejs.

Twój serwer WWW zapala się od przegrzania kart sieciowych. Kto by przypuszczał, że XML jest łatwopalny? Gdy gasisz pożar wężem ogrodowym i płaczesz (bo nie miałeś żadnych kopi bezpieczeństwa), myślisz: dlaczego nie wprowadziłem żadnych ograniczeń w dostępie do API?

Kolejne akapity zawierają przykład rozwiązania tworzącego system ochronny dla API. Co istotne, kod jest bardzo krótki i dobrze integruje się z API, które trzeba chronić przed fascynatami pizzy z majonezem. Cały prezentowany tu kod jest dostępny na serwerze wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/troyaid.zip*.

Uwierzytelnienie użytkownika

Pierwszym krokiem przy ograniczaniu dostępu do API jest uwierzytelnienie, ponieważ nie możemy limitować zapytań, dopóki nie wiemy, kto pyta. Istnieją dwa systemy obsługi uwierzytelnienia w systemach z publicznym API: te, które troszczą się o uwierzytelnienie tylko w celach pomiarowych, i te, które służą do ograniczania dostępu do prywatnych danych. Google Maps jest przykładem pierwszego, a Facebook API — drugiego.

Jeśli należysz do pierwszej grupy, wystarczy typowy wzorzec z kluczem dostępowym do API. Jeśli należysz do drugiej grupy i udostępniasz przez interfejs dane specyficzne dla użytkownika, powinieneś rozważyć bardziej wyrafinowany mechanizm logowania, bazujący na sesji lub tokenie, zapewniający większy stopień bezpieczeństwa.

Wzorzec z kluczem API to system, w którym nazwa użytkownika i hasło zawarte są w jednym ciągu znaków nazywanym kluczem. Klucz jest stosunkowo długi (w zasadzie nigdy nie wpisuje się go ręcznie, bo używają go wyłącznie programy) i powstaje w sposób automatyczny (jest generowany przez aplikację internetową). Każdy klucz API służy jako unikatowy identyfikator konkretnego API i jest wysyłany wraz z żądaniem w celu identyfikacji.

Ponieważ ta forma uwierzytelnienia jest bardzo podatna na ataki, wiele stron stosuje ją w połączeniu z listą adresów IP, które mogą wykonywać żądania z użyciem wskazanego klucza. Jeśli adres IP nadawcy znajduje się na liście dozwolonych adresów, uwierzytelnienie powiedzie się. Jeśli nie, klient nie uzyska dostępu. Dostęp do modyfikacji listy adresów jest strzeżony znacznie mocniej przez wersję HTML witryny.

Zastosowanie klucza API w połączeniu z dopuszczalną listą adresów IP zapewnia efektywny sposób uwierzytelnienia witryny. Oto przykład, jak wprowadzić takie uwierzytelnienie na własnej witrynie.

Najpierw utwórz klasę `ApiUse`, która będzie należała do modelu `User` i zawierała informacje niezbędne do śledzenia użycia API. Na razie model będzie zawierał tylko trzy pola poza niejawnym polem `id`.

ApiUse		
user_id	Api_key	allowed_domains

Następnie utwórz funkcję uwierzytelnienia w klasie `ApiUse`, która albo zwróci obiekt `ApiUse`, albo wyjątek.

```
def self.authenticate(api_key, requesting_address)
   api_use = ApiUse.find(:first, :conditions => "api_key = '#{api_key}'")

   # Upewnij się, że klucz API odnosi się do istniejącego obiektu ApiUse
   raise ArtOfRails::Chapter5::UserNotFoundError,
       "Niepoprawny klucz API." if api_user == nil || api_use.blank?

   # Upewnij się, że adres IP znajduje się na liście dopuszczalnych adresów.
   # Założenie: lista zawiera adresy oddzielone znakiem przecinka.
   unless api_use.allowed_domains.split(',').include? requesting_address
       raise ArtOfRails::Chapter5::DomainNotAuthorizedError,
          "Brak akturyzacji dla adresu zgłaszającego żądanie."
   end
   api_use
end
```

Funkcja przeprowadza dwa sprawdzenia związane z przesłanymi danymi. Najpierw upewnia się, czy rzeczywiście istnieje obiekt o wskazanym kluczu. Następnie sprawdza, czy adres IP klienta znajduje się na liście adresów dopuszczonych do stosowania wspomnianego klucza. Jeśli którykolwiek z tych warunków nie zostanie spełniony, zgłasza własny wyjątek zdefiniowany w innej części projektu. Jeśli wszystko pójdzie dobrze, zwraca obiekt `ApiUse`.

Algorytm ograniczający

Po uwierzytelnieniu użytkownika następnym rokiem jest upewnienie się, że użytkownik nie przekroczył nałożonych limitów. Standardowy algorytm ograniczający bazuje na pomyśle, że użytkownik musi się wcześniej uwierzytelnić, a gdy to już zrobi, otrzymuje X wywołań API w jednostce czasu T, bez możliwości przenoszenia zaoszczędzonych wywołań na kolejne okresy. Algorytm łatwo zaimplementować, dodając do tabeli z API dwie kolumny: `last_access` i `accesses_this_period`.

ApiUse				
user_id	Api_key	allowed_domains	last_access	accesses_this_period

Uprośćmy algorytm nieco bardziej i powiedzmy, że jednostką czasu będzie jakiś standardowy przedział czasu, jak godzina lub dzień. Przykład wykorzystuje przedział wynoszący jeden dzień, by matematyka związana z datą była bardzo prosta.

Poniższy kod to jeden z przykładów, w jaki można zaimplementować funkcję ograniczającą. Implementacja zgłasza wyjątek, jeśli użytkownik przekroczył limit. Nie zwraca w takiej sytuacji wartości logicznej `false`. Aby ten kod działał, zdefiniuj stałą `DAILY_API_LIMIT` w pliku *config/environment.rb* jako wartość całkowitą określającą dzienny limit wywołań API dla pojedynczego użytkownika. Kod należy umieścić jako metodę obiektu `ApiUse`.

```
def record_api_request
    if (self.last_access < Date.today + 1)
       # Klient po raz pierwszy używa API w dniu dzisiejszym.
       self.last_access = Date.today
       self.accesses_this_period = 1
       self.save
    elsif self.accesses_this_period >= DAILY_API_LIMIT
       # Zbyt częste korzystanie z API!
       raise ArtOfRails::Chapter5::UsageLimitExceededError,
          "Przekroczono dzienny limit użycia API"
    else
       # Nie pierwsze użycie, ale nadal w dopuszczalnych granicach.
       self.accesses_this_period = self.accesses_this_period + 1
       self.save
    end
end
```

Przy każdym wywołaniu metoda `record_api_request` zwiększa licznik wywołań API w danym dniu. Najpierw sprawdza, czy poprzednie wywołanie wykonano dziś. Jeśli nie, ustawia licznik na 1 i zapisuje aktualny dzień. Jeśli poprzednie wywołanie odbyło się dziś, stara się zwiększyć licznik. Jeśli jednak przekroczono dzienny limit, wartość nie jest zwiększana, a metoda zgłasza wyjątek.

Wprowadzenie ograniczeń za pomocą filtrów

Po przygotowaniu funkcji uwierzytelniających i mierzących pozostaje już tylko zastosowanie ich w kodzie w taki sposób, by nie wpłynąć znacząco na istniejącą implementację akcji. Pamiętaj, że styl programowania w Rails kładzie duży nacisk na czystość kodu. Stosując filtry, łatwo utworzyć implementacje dodawane do akcji, które tak naprawdę znajdują się poza kodem akcji. To podejście zapobiega zaśmiecaniu kodu akcji dodatkowymi elementami związanymi z obsługą bezpieczeństwa.

Zapewne pamiętasz z poprzedniego rozdziału, że filtry potrafią otoczyć kod akcji kontrolera. Mają one dostęp do wszystkich informacji, które posiada akcja, włącznie z możliwością zmiany odpowiedzi, a nawet zablokowania wykonania akcji. Sercem filtru jest po prostu metoda kontrolera.

Naszym celem jest utworzeniem metody, która wykorzystuje uwierzytelnienie i mierzenie ruchu dla żądań. Jeśli nie ma żadnych przeciwwskazań co do wykonania akcji, filtr po prostu kończy działanie i przekazuje żądanie do rzeczywistej akcji. Jeśli kroki uwierzytelnienia lub ograniczenia użycia zgłoszą wyjątek, metoda wyłapuje je i zapobiega wywołaniu akcji. W rzeczywistej implementacji zapewne chciałbyś również przekazać użytkownikowi aplikacji komunikat o błędzie, by wiedział, gdzie popełnił błąd. Kod został tak napisany, by można go było zastosować jako filtr wokół akcji (`around_filter`).

```
def api_auth

   # Uwaga: rzeczywista aplikacja powinna zapewnić lepszy mechanizm
   # określania typu odpowiedzi (patrz ramka).
   response_type =
    Mime::EXTENSION_LOOKUP[params[:format]].to_sym rescue response_type = :html

   if API_TYPES.include? response_type
       @api_use = ApiUse.authenticate(params[:api_key], request.remote_ip)
       @api_use.record_api_request
   end
   yield
rescue ArtOfRails::Chapter5::UserNotFoundError,
       ArtOfRails::Chapter5::DomainNotAuthorizedError,
       ArtOfRails::Chapter5::UsageLimitExceededError => err

   # Do wykonania pozostaje zapewnienie użytkownikowi API komunikatu o błędzie.
   false
end
```

Dodaj ten kod do klasy `ApplicationController`, by był dostępny w całej aplikacji. Dodatkowa stała, `API_TYPES`, powinna znaleźć się w pliku *config/environment.rb*, by zdefiniować formaty, które powinny być ograniczane jako wywołania API. Definicja stałej może wyglądać następująco:

```
API_TYPES = [ :xml, :rdf, :csv ]
```

Na końcu w każdym kontrolerze, który zawiera akcje wymagające ograniczenia, dodaj referencję do tej metody w filtrze `around_filter`:

```
around_filter :api_auth, :only => [:akcja1, :akcja2, :akcja3]
```

To bardzo ładny, jednowierszowy kod, który ławo zrozumieć i który chroni akcje kontrolera przed nadużyciem. Ograniczone w liczbie wywołań akcje muszą wiedzieć, w jaki sposób reagować na żądania, wykorzystując kod `respond_to`, ale nie muszą nic wiedzieć o uwierzytelnieniu i mierzeniu liczby odwiedzin.

Co z usługami SOAP i XML-RPC

Wiele najnowszych prac związanych w Rails z programistycznymi interfejsami internetowymi dotyczy interfejsów bazujących na HTTP, które korzystają z alternatywnych formatów danych na tych samych zakończeniach sieciowych jak wersja HTML. Mimo to starsze usługi sieciowe nie odeszły i są popularne w środowiskach obsługi aplikacji biznesowych. Ten podrozdział opisuje pokrótce, jak zaimplementować w Rails interfejsy bazujące na SOAP i XML-RPC, wykorzystując szkielet `ActionWebService`.

Usługi tworzone za pomocą `ActionWebService` wymagają trzech definicji: interfejsu usługi, implementacji usługi i definicji struktury.

Znajdź lukę w zabezpieczeniach

Przyjrzyj się następującemu fragmentowi kodu metody `api_auth` z wcześniejszej części rozdziału:

```
      response_type =
         Mime::EXTENSION_LOOKUP[params[:format]].to_sym rescue response_
   type = :html
```

Naszym celem jest określenie formatu odpowiedzi dla żądania, by dowiedzieć się, czy należy wprowadzić ograniczenia. Przykładowo można ograniczyć liczbę żądań danych XML, ale nie można limitować żądań danych HTML.

Problem polega na tym, że ten wiersz stara się określić format jedynie po zawartości zmiennej `params[:format]`, który jest ustawiony tylko wtedy, gdy adres URL ma postać */ścieżka/do/zasobu.format*. W rzeczywistości negocjacja typu danych nie jest taka prosta. Zamiast stosować rozszerzenie w adresie URL, aplikacja użytkownika mogłaby wysłać do API żądanie z całą listą nagłówków `Accept`. Typem zawartości odpowiedzi byłby więc typ wynegocjowany między typami obsługiwanymi przez aplikację i listą dopuszczalnych typów.

Mając te wszystkie informacje, czy potrafisz wskazać lukę w zabezpieczeniach? (Podpowiedź: co się stanie, jeśli format XML ustawimy w nagłówkach żądania HTTP, a nie w adresie URL?)

Tę lukę łatwo przetestować, używając programu `curl` wywoływanego z wiersza poleceń:

```
curl -H "Accept: text/xml" localhost:3000/wywołanie/api
```

Polecenie zwróci odpowiedź w formacie XML, ale obiekt `ApiUse` nie zwiększy licznika użyć. Wynika to z faktu, iż zmienna `params[:format]` będzie równa `nil` (nie została wskazana w adresie URL), więc metoda `api_auth` przyjmie domyślny format `html` i pominie limitowanie wywołań. Gdy żądanie dotrze do akcji, Rails poprawnie wyciągnie z nagłówka `Accept` żądany format danych, którym okaże się XML.

Implementując algorytm `api_auth` w sposób bezpieczny, trzeba określić typ odpowiedzi w ten sam sposób, w jaki robi to Rails, by mieć pewność, że podejmowana decyzja o limitowaniu jest słuszna. By to wykonać, przyjrzyj się kodowi modułu `ActionController::`
`↪MimeResponds` i albo wykonaj jego kopię, albo wywołaj go bezpośrednio.

Interfejs usługi

Gdy interfejsy usług w standardowych aplikacjach Rails definiujemy niejawnie za pomocą odwzorowań adresów URL, to w przypadku usług sieciowych SOAP musimy podążyć nieco dalej i utworzyć jawny dokument informujący o tym, co jest wymagane i co oferuje usługa. Tego rodzaju definicja interfejsu zostaje przetłumaczona na dokument WSDL, który klient wykorzystuje do powiązania własnego kodu ze zdalną usługą.

Definiowanie interfejsu SOAP przypomina tworzenie pliku nagłówkowego w C++ lub interfejsu w języku Java. Celem jest zapewnienie sformalizowanego kontraktu dokładnie definiującego oczekiwania zarówno po stronie klienta, jak i serwera. Gdy wszystko działa zgodnie z oczekiwaniami, odciąża użytkownika od ręcznej konfiguracji i testowania wszystkich powiązań.

Podobnie jak migracje `ActiveRecord`, także definicje interfejsu usług sieciowych przyjmują w Rails postać klasy języka Ruby wypełnionej makrami definiującymi różne elementy klasy. Definicja interfejsu zamiast pól zawiera definicje sygnatur metod. Każda sygnatura reprezentuje pojedynczą metodę, którą usługa zgadza się udostępniać zdalnym użytkownikom.

Każda sygnatura metody składa się z dwóch elementów: `:expects` i `:returns`. Argumenty te odwzorowują uporządkowaną listę wartości reprezentującą informacje, które metoda przyjmuje, i informacje, które ona zwraca.

Każdy z elementów przyjmowanych lub zwracanych przybiera jedną z trzech postaci:

- symbol określający podstawowy typ danych (`:integer`, `:string` lub `:boolean`);
- klasę dziedziczącą po typie strukturalnym, na przykład `ActionWebService::Struct` lub `ActionWebService::Base` (patrz punkt „Tworzenie struktur" w dalszej części rozdziału);
- jednoelementową tablicę zawierającą jeden z dwóch powyższych typów, która oznacza, że argument nie jest pojedynczym obiektem, ale tablicą takich obiektów.

Te trzy typy elementów zapewniają wystarczającą elastyczność, by obsłużyć szerokie spektrum sygnatur funkcji. Poniżej znajduje się przykładowo opis interfejsu z trzema metodami. Pierwsza z nich, `find_recipe`, pobiera obiekt `RecipeQuery` i zwraca tablicę obiektów `Recipe`. Druga, `rate_recipe`, przyjmuje dwie liczby całkowite i nic nie zwraca. Trzecia, `add_comment`, przyjmuje wartość całkowitą oraz tekst i nic nie zwraca.

```
class RecipeAPI < ActionWebService::API::Base
   api_method :find_recipe, :expects => [RecipeQuery], :returns => [[Recipe]]
   api_method :rate_recipe, :expects => [:int, :int]
   api_method :add_comment, :expects => [:int, :string]
end
```

Dosyć łatwo odgadnąć, co mają reprezentować argumenty dwóch ostatnich metod, ale najlepiej byłoby, gdybyśmy nadali im nazwy. Rails zapewnia sposób nazwania tych parametrów, zastępując poszczególne wartości słownikami zawierającymi odwzorowanie nazwy parametru na jego definicję. To, co dawniej miało postać `:string`, teraz przyjmuje postać `{:comment = > :string}`.

```
class RecipeAPI < ActionWebService::API::Bas
    api_method :find_recipe, :expects => [RecipeQuery], :returns => [[Recipe]
    api_method :rate_recipe, :expects => [{:recipe => :int}, {:rating => :int}]
    api_method :add_comment, :expects => [{:recipe => :int}, {:comment => :string}]
end
```

Definicje usług powinny znaleźć się w folderze *app/apis* projektu Rails. Folder ten nie jest generowany automatycznie w momencie tworzenia pustego projektu Rails. Nazwa pliku powinna odpowiadać nazwie interfejsu programistycznego, więc dla klasy `RecipeAPI` utwórz plik *recipe_api.rb*.

Implementacja usługi

W odróżnieniu od innych Web API przedstawianych we wcześniejszej części rozdziału `Action ↪WebService` nie może łatwo współdzielić implementacji z pozostałymi formatami odpowiedzi. Choć sama implementacja usługi może istnieć w standardowym kontrolerze Rails, stosuje inne

akcje niż standardowy system. Akcje ActionWebService pobierają argumenty jako część swoich sygnatur metod, a nie jako słownik params, nie renderują też żadnych wyników. Zamiast tego zwracają wartość, która zostanie przekształcona na odpowiedź wysyłaną do zdalnego klienta. Choć metody ActionWebService mogą współdzielić otoczenie z kodem bazującym na HTTP, pełne współdzielenie kodu jest raczej trudne.

Istnieją trzy różne podejścia do odwzorowania usługi bazującej na ActionWebService na kontroler: bezpośrednie, oddelegowane i warstwowe. Każde podejście oferuje dużą elastyczność w zarządzaniu końcówkami połączeń. Omawiam jedynie podejście bezpośrednie. Więcej dokumentacji na temat dostępnych rozwiązać jest w podręczniku Ruby on Rails (*http://manuals.*
➥*rubyonrails.com/read/chapter/69*).

Zakładając, że definicja usługi znajduje się zgodnie z konwencją w folderze *app/apis*, kontroler jest z nią automatycznie powiązany dzięki swojej nazwie: RecipeController. Wystarczy więc tylko zaimplementować wszystkie metody opisane w API jako metody publiczne kontrolera.

Przykładowo metoda opisana w definicji interfejsu jako:

```
api_method :find_recipe, :expects => [RecipeQuery], :returns => [[Recipe]]
```

może zostać zaimplementowana jako:

```
def find_recipe(recipe_query)
    @recipes = Recipe.find_with_query_obj(recipe_query)
    @recipes
end
```

Metoda zdefiniowana jako:

```
api_method :add_comment, :expects => [:int, :string]
```

może mieć poniższą implementację:

```
def add_comment(recipe_id, comment)
   @recipe = Recipe.find(recipe_id)
   @recipe.add_comment(comment) unless @recipe.nil?
end
```

W dużym skrócie można powiedzieć, że metoda musi być publiczna, musi przyjmować jako parametry obiekty :expects i zwracać obiekty pasujące do definicji :returns.

Tworzenie struktur

W wielu sytuacjach obiekt zwracany lub przekazywany do usługi sieciowej jest bardziej złożony niż prosty typ danych. W takiej sytuacji ActionWebService umożliwia zdefiniowanie klasy przypominającej strukturę z języka C stanowiącą połączenie kilku obiektów prostych w jeden obiekt złożony. Przedstawiona wcześniej definicja interfejsu zastosowała dwa takie obiekty — RecipeQuery i Recipe — które są niezbędne do poprawnego działania usługi find_
➥recipe. Strukturę obu obiektów zdefiniujemy jako klasy dziedziczące po ActionWebService::
➥Struct.

Klasa bazowa `ActionWebService::Struct` zawiera makra, które umożliwiają definiowanie pól obiektu w ten sam sposób, w jaki tworzy się definicję interfejsu. Poniżej znajduje się przykładowa struktura `Recipe`. Zawiera kilka prostych typów, tablice tekstów dla pól, takich jak rodzaj potrawy i jej zdjęcia, a także tablicę struktur `Ingredient`.

```
class Recipe < ActionWebService::Struct
   member : name, :string
   member :rating, :integer
   member :genre, [:string]
   member :author_name, :string
   member :author_id, :integer
   member :photo_urls, [:string]
   member :ingredients, [Ingredient]
   member :directions, :string
end
```

Zastosowanie struktur takich jak powyższa odpowiada w dużym stopniu strukturze dokumentu XML (w rzeczywistości usługi sieciowe w wielu przypadkach przesyłają dane właśnie jako XML). Dowolny klient z odpowiednio wygenerowanymi zaślepkami uzyska jednak jako wynik zwykłe obiekty, a nie dokument z mnóstwem znaczników.

Podsumowanie

W niniejszym rozdziale przedstawiono wiele informacji na temat interakcji witryn z użytkownikami. Opisano nowy system interfejsów internetowych bazujących na HTTP, które społeczność Rails stara się popularyzować. Wyjaśniono, dlaczego to podejście do tworzenia interfejsów pozwala zaoszczędzić mnóstwo czasu. Bardzo pobieżnie omówiono odwzorowania, ich współpracę z definicją interfejsu, sposób budowania, a także mechanizm `respond_to`, który pozwala kontrolerom Rails tworzyć akcje zwracające wyniki w kilku różnych formatach. Przedstawiono przykłady tworzenia odpowiedzi w kilku popularnych formatach — XML, RSS i RDF — wraz z odnośnikami do szczegółowej dokumentacji. Na końcu pojawił się opis wysokopoziomowego szkieletu `ActionWebService`, co pozwoliło poznać wszystkie stosowane obecnie systemy interfejsów API.

W rozdziale wprowadzono dwie bardzo ważne koncepcje. Pierwsza polega na tym, że adresy URL stosowane w aplikacjach nie muszą odpowiadać fizycznej lokalizacji dokumentów lub skryptów w systemie plików serwera. Mogą pasować do wirtualnych obiektów wewnątrz aplikacji, na przykład */users/ted/photos/12*. Odwzorowywanie adresów URL to nowy sposób tworzenia specyfikacji adresów URL, które obecnie stanowią część publicznego interfejsu aplikacji.

Druga koncepcja prezentowana w tym rozdziale dotyczy faktu, iż tworzenie interfejsu API i tworzenie witryny internetowej niekoniecznie są zadaniami całkowicie niezależnymi. W zasadzie rozdział ten pokazuje, że jest całkowicie odwrotnie — cała aplikacja internetowa reprezentuje sobą pewien zbiór funkcjonalności, którą można przekazywać klientowi w wielu formatach. Tworzenie witryny i interfejsu API okazuje się nagle dokładnie tym samym zadaniem, ale z dwoma różnymi typami zwracanych wartości.

W następny rozdziale omówiono projektowanie bazujące na REST, styl architektoniczny, który wiele osób traktuje jako jedyny słuszny kierunek rozwoju aplikacji internetowych. Witryny bazujące na REST opisują siebie w całości jako zasoby udostępniane przez sieć za pomocą operacji CRUD. Operacje CRUD określa się przy użyciu poleceń HTTP.