PORADNIKI

Usługi sieciowe w Rails

Usługi sieciowe w Ruby on Rails : budowa usług sieciowych klienckich

Z punktu widzenia wysokopoziomwego , implementacja usług sieciowych może być podzielona na dwie kategorie: serwerów i klientów. Pokażę w tym tekście tworzenie usług klienckie Większość usług sieciowych jest oparta o jedną z trzech architektur :
Representational State Transfer (REST), Simple Object Access Protocol (SOAP), or Extensible Markup Language Remote Procedural Calls (XML-RPC). Usługi sieciwoe często oferują dostęp przez dwie lub więcej tych architektur. Jeśli budujesz swoje usługi sieciowe klienckie w Ruby i implementujemy je jako część apliakcji Rails, masz szczęści.Budowanie usług sieciowych w Ruby on Rails wymaga tylko dwóch prostych kroków i wymaga tylko kilku bibliotek Ruby. Najlepszą wiadomością jest to ,że główne biblioteki jakich używasz do budowania klientów CGI, NET, REMXML ,Soap4r, XSD i XML-RPC są automatycznie ładowane przez środowisko Rails. Ty musisz się martwić o to aby wiedzieć kiedy, jak i gdzie użyć kazej z tych bibliotek.Jeśli zastanawiasz się dlaczego mówię o Rails (platforma serwerowa) i usługach sieciowych klienckich, wrócmy trochę wstecz. Większość usług sieciwoych klienckich to nie są programy desktopowe lub programy linii poleceń. Większość takich usług to są serwery: serwery , które zbierają dane z innych usług sieciowych, a następnie przepakowywujką je dla innych celów. (Tymi innym celamio zwykle są strony internetowe, ale może to być również kolejna usługa sieciowa) To dlatego klientów tworzymy tu, jako część aplikacji Rails.

Zajmiemy się trzema najbardziej popularnymi rodzajami usług sieciowych : REST, SOAP i XML-RPC. Użyejmy trzech z popularnych, darmowych i przydatnych serwisów internetowych dla naszych przykładów : Yahoo!Search, Google Search i Flickr. Ponieważ jestem zwolennikiem uczenia się przez działanie, przejdziemy szczegółowo budowę usług sieciowych klienckich dla Rails przez stworzenie relanych, działających klientów dla każdej z tych usług. Wszystkie przykłady skupią się na budowaniu działających klientów demonstrujących koncepcje usług sieciowych, więc nie będziemy omawiać poszczególnych usług szczegółowo. Moim celem jest pokazanie jak stosować pojęcia usług sieciowych w aplikacjach Rails

Podstawy usług sieciowych

Jak w przypadku każdego języka programowania, architektury lub standardu, musisz zapoznać siez terminologią, zanim zajmiesz się implementacją, Zaczniemy od wprowadzenia kilku podstawowych terminologii i technologii które musisz zrozumieć zanim zaczniesz kodowanie własnych klientów usług sieciowych i serwerów.

Prawdopodobnie najważniejszą rzeczą jest idea podpisów usług sieciowych. Podpsiy usług sieciowych są w rzeczywistości typami danych, których usługa albo oczekuje albo zwraca. Usługi sieciowe nie są specyficzne dla języka : możesz używać wybranych języków programowania do implementacji klienta lub serwera. Klient napisany w C# lub Javie musi móc komunikować się z klientem napisanym w Ruby i vice versa. Konsekwencją tego jest to, że aby być kompatybilnymi z językamii silnie typowanymi takimi jak Java, usługi sieciowe muszą również być silnie typowane. W dużym stopniu, typy obiektu na jakie można się natknąć w usługach sieciowych są takie jakich można oczekiwać : liczby całkowite, łańcuchy, wartości boolowskie, liczby zmiennoprzecinkowe i ty[y ddatatime. Jednak usługi sieciowe często działają z bardziej złożonymi typami ,takimi jak tablice i struktury. Z punktu widzenia Ruby, działanie ze wszystkimi tymi różnymi typami danych jest rzeczywiście bardzo proste. Wszystkie standardowe typy danych mają swoje odwzorowanie w Ruby. Innym ważnym elementem w zrozumieniu usług sieciowych jest plik WSDL. WSDL oznacza Web Service Description Language; plik WSDL jest dokumentem XML, który definiuje interfejs dla usługi SOAP. Pliki WSDL dostarczają szczegółów o metodach, które maskują usługi, argumenty metod i zwracane wartości i kodowanie używane dla przenoszenia danych między klientem a serwerem. Wszystko co potrzebne do wiedzy o SOAP jest zapisane w pliku WSDL; pliki WSDL służa zarówno jako foram dokumentacji dla usług SOAP i jako klucz do zautoamtyzowania wielu kroków do budowania klientów SOAP. Zatem możliwe jest odczytanie pliku WSDL i znalezienie wszystkiego co potrzebne do jego API. Odczyt pliku WSDL nie jest zabawne. Podstawowym zastosowaniem WSDL jest zautoamtyzowanie kodu klienta abyś nie musiał wyraźnie pisać kodu do obsługi różnych kodowanych danych, różnego odwzorowania między typami obiektu itd. Ska wywodzi się złożoność magicznego pliku WSDL? Za starych czasów,musiał być pisany ręcznie. Ale obecnie większość nowoczesnych platform usług sieciowych automatycznie genruje pliki WSDL i czyni je dostępnymi dla klientów do pobrania. Rails nie jest wyjątkiem. (Przykład serwera SOAP w tym tekście zawiera plik WSDL, który został wygenrowany automatycznie) .Ponieważsą zautomatyzowane nigdy nie powinieneś ich dotykać.

W końcu, muszę powiedzieć trochę o architekturze usług sieciowych. SOAP wyrósl z XML-RPC, więc usługi SOAP i usługi XNL-RPC są fundamentalnie podobne. Obie próbują naśladować "normalne" operacje programistyczne : wywoływanie funkcji (dla XMLRPC) czy zdalne wywoływanie metdo (SOAP)

Usługi REST przedstawiają się znaczącą inaczej (i z grubsza dużo prościej) REST oznaca Representational State Transfer. Podstawową ideą REST jest to ,że nie musisz tworzyć usług sieciowych "wyglądajacych" jak regularne wywoływanie metdo lub funkcji. Podstawowe operacje HTTP GET, PUT , POST i DELETE odpowiadaja czterem podstawowym operacjom SQL : SELECT, UPDATE, INSERT i DELETE. Dlatego też, możliwe jest budowanie złożonych aplikacji przez robienie żądań HTTP w obrębie dokumentów XML

Wyszukiwanie w Yahoo przy użyciu REST

REST jest często traktowany jako najprostsza architektura usług sieciowych. Operacje REST działają jak stadandardowe żądania stron WWW. Aplikacje RESt po prostu czynią żądania przez URL, jak zwykłe żądanie strony. Serwer HTTP zwraca dokument z wynikiem żądania. Ten zwrócony dokument jest zwykle w formacie XML (chociaż XML nie jest wymagany; usługa może zwrócić dowolną strukturę danych). Faktycznie, jeśli przeglądarka ma możliwość wyświetlania XML możesz wpisać URL generowany przez naszego pierwszego klienta do paska URL i zobaczyć surowy wynik XML , jaki wygenruje Yahoo!

Klient oparty o Rails wymaga :

1.Połączenia z usługą sieciową stadnardowym żądaniem GET lub POST (w zależności od wymagań serwera), użycie biblioteki NET
2.Przechowywania wyników jako dokumentu REXML
3.Parsowanie wyniku biblioteka REXML dla użycia apliakcji Rails
Aby to zademonstrować zbudujemy prosty kontroler ,który przeszukuje górne trzy wyniki wyszukiwarki Yahoo! , używając interfejsu Yahoo!REST. Yahoo oferuje API usługi sieciowej dla wielu swoich usług, wliczając w to popularną wyszukiwarkę.API wyszukiwarki Yahoo! Jest darmowa ale ma ograniczoną liczbę żądań jakie możesz stworzyć.Aby użyć Yahoo API, potrzebujemy darmowego Yahoo!Developer's Key. Klucz można pobrać z http://api.search.yahoo.com/webservices/register_aplikation.
Po otrzymaniu klucza, zbudujemy prosty kontroler:

```
class CodeController < ApplicationController
  def yahootest
    query = CGI.escape("SEARCH TEXT")
    yahookey = "YOUR YAHOO DEVELOPER KEYs
    url = "http://api.search.yahoo.com/" +
      "WebSearchService/V1/webSearch?" +
      "appid=#{yahookey}&query=#{query}" +
        "&results=3&start=1"
    result = Net::HTTP.get(URI(url))
    @doc = REXML::Document.new result

 end
end
```

Teraz zbudujemy podgląd który wyświetli wyniki wyszukiwania w prostym widoku. Zapisujemy poniższy kod w pliku yahootest.rhtml w folderze app/code/views:

```
<% @doc.root.each_element do |res| %>
 <b>Title:</b> <%= res[0].text.to_s %><br>
 <b>Summary:</b> <%= res[1].text.to_s %><br>
 <b>Link:</b> <a href="<%= res[2].text.to_s %>"><%= res[2].text.to_s %></a>
 <br><br>
<% end %>
```

I o dziwo działa .Dzięki paru linijkom kodu zbudowaliśmy klienta usługi sieciowej dla aplikacji Rails.
Teraz kiedy mamy działającego klient, spójrzmy na to co się dziejke w kodzie. Najpierw , chociaż nie ma instrukcji "wymaganych", klient używa kilku ważnych bibliotek : NET , CGI i REXML. Wszystkie te biblioteki pochodzą z dystrybucji Ruby i powinny być autoamtycznie załadowane i gotowe do użycia przez apliakcję Rails. Nie musisz robić niczego specjalnego aby ich używać w kodzie. Pewnie słyszałeś o bibliotekach NET i CGI, ale być może nie o bibliotece REXML. REXML jest czystym Ruby XML Processor z wieloma funkcjami, wliczając w to pełną obsługę Xpath 1.0.
Najpierw użyliśmy biblioteki CGI dla wydobycia nasego szukanego terminu ,zapewniając ,że szukany termin jest bezpeiczny do zastosowania w naszym żądaniu HTTP GET:

```
query = CGI.escape("SEARCH TEXT")
```

Następnie zapisaliśmy nasz Yahoo!Developer's Key w zmiennej i zbudowaliśmy URL z parametrami określonymi w dokumentacji dla Yahoo!API. Dla dodatkowych parametrów i opcji zobacz dokumentację Yahoo!

```
yahookey = "YOUR YAHOO DEVELOPER KEY"
url = "http://api.search.yahoo.com/WebSearchService/" \
 "V1/webSearch?appid=#{yahookey}&query=#{query}&" \
 "results=3&start=1"
```

Zrozumienie URL'a nie jest trudne : wywołujemy wersję 1 (V1)usługi webSearch,przekazując jej cztery parametry: appid (klucz projektanta), querry (szukany łańcuch)kilka result jakie chcemy (3) i start, które mówi usłudze sieciowej Yahoo!o kilku pierwszych wynikach jakie chcemy by zwróciła. W dużym przykładzie możemy użyć tego parametru aby pomóc stronie przy dużym zbiorze wyników; teraz, chcemy pierwszych trzech wyników, więc zaczynamy z result 1. Naszym kolejnym krokiem jest użycie biblioteki NET dla żądania HTTP GET i przechowania zwracanego dokumentu w zmiennej :

```
result = Net::HTTP.get(URI(url))
```

Po tym nasze wyniki przechowywane są w zmiennej lokalnej, używamy biblioteki REXML do konwersji wyniku do dokumentu REXML:

```
@doc = REXML::Document.new result
```

W końcu, nasz podgląd używa biblioteki REXML dla sparsowania dokumentu XML i wyświetlenie wyników. REXML oferuje kilka sposobów uzyskania dostępu do wartości znaczników i atrybutu. Tu użyjemy metody tablicowej:

```
<% @doc.root.each_element do |res| %>
 <b>Title:</b> <%= res[0].text.to_s %><br>
 <b>Summary:</b> <%= res[1].text.to_s %><br>
 <b>Link:</b> <a href="<%= res[2].text.to_s %>"><%= res[2].text.to_s %></a>
 <br><br>
<% end %>
```

Używamy metody each_element w elemencie źródłowym dokumentu (ResultSet) dla dostępu dla każdego wyniku (Result). Ponieważ Element jest podklasą Parent, możemy potem użyć metody tablicowej dla uzyskania dostępu do potomka, który jest przeznaczony do wyświetlenia (Title, Summary i Url)

Przeszukiwanie Google używając SOAP lub SOAP z plikami WSDL

W ciągu ostatnich kilku lat, SOAP uzyskać wiele wsparcia, w części dlatego ,że jest teraz oficjalnie udokumetowany przez World Wide Web Consortium (W3C). Standaryzacja czyni pracę z usługami SOAP bardziej niezawodną, przynajmniej jeśli chodzi o informacje które musisz znać i informacje jakich się spodziewasz z powrotem. Z powodu poparcia W3C, można powiedzieć ,że SOAP jest obecnie preferowaną architekturą dla większości usług internetowych, jednak mimo wsparcia W3C, nie ma znaczących dowodów ,że usługi REST są coraz powszechniej używane. Niemniej jednak, jeżeli masz zamiar pracować z usługami sieciowymi, musisz poznać te trzy architektury : REST, XML-RPC i SOAP.

Największą wadą SOAP jest jej złożoność, ale Rails ukrywa większość tej złożoności przed tobą. Tworzenie klienta SOAP zawiera się w czterech krokach:

1.Tworzenie instancji sterownika SOAP
2.Zdefiniowanie metod SOAP jakie chcesz wywołać
3.Wywołanie metod SOAP
4.Użycie wyników w aplikacji Rails

Aby to zademonstrować, zbudujemy wyszukiwanie Google używając SOAP. Podobnie jak Yahoo! , Google oferuje darmowe API usług

sieciowych dla wielu swoich usług, wliczając ich wyszukiwarkę. Aby użyć API Google, potrzebny jest Google Developer's Key, który można pobrać bezpośrednio z Google https://www.google.com/accounts/NewAccount?continue=http://api.google.com/createkey&followup=http://api.google.com/createkey.

Oto kontroler, którego używa SOAP dla znajdowania pierwszych trzech wyników z wyszuiwarki Google. Uaktualnij plik code_controller.rb z poprzednioego przykładu aby zawrzeć nową metodę googletest.

```
class CodeController < ApplicationController
  def googletest
     yourkey = 'YOUR GOOGLE DEVELOPER KEY'
     @yourquery = 'SEARCH TEXT'
     XSD::Charset.encoding = 'UTF8'
     googleurl = "http://api.google.com/search/beta2"
     urn = "urn:GoogleSearch"
     driver = SOAP::RPC::Driver.new(googleurl, urn)
     driver.add_method('doGoogleSearch', 'key', 'q',
        'start', 'maxResults', 'filter', 'restrict',
        'safeSearch', 'lr', 'ie', 'oe')
     @result = driver.doGoogleSearch(yourkey,
        @yourquery, 0, 3, false, '', false, '', '', '')
  end
end
```

Oto kod podglądu. Zapisz go jako googletest.rhtml w katalogu app/views/code:

```
Query for: <%= @yourquery %><br>
Found: <%= @result.estimatedTotalResultsCount %><br>
Query took about <%= @result.searchTime %> seconds<br>
   <% @result.resultElements.each do |rec| %>
    <b>Title:</b> <%= rec["title"] %><br>
    <b>Summary:</b> <%= rec.snippet %><br>
    <b>Link:</b> <a href="<%= rec["URL"] %>"><%= rec["URL"] %></a>
    <br><br>
<% end %>
```

I znowu działa.
Ten klient używa dodatkowych bibliotek Ruby : Soap4r i XSD4R. Soap4r jest czystą implementacją Ruby dla SOAP 1.1 .Biblioteka XSD4R jest biblioteką obsługującą XML, która używana jest przez Soap4r. Dostarcza konwersji między pewnymi wspólnymi typami danych. Soap4r i XSD4R są częścią dystrybucji Ruby i powinny być załadowane automatycznie. Zaczynamy nas przykład Google od stworzenia instancji sterownika SOAP używającego usługi Google SOAP Uri i Namespace:

```
googleurl = "http://api.google.com/search/beta2"
urn = "urn:GoogleSearch"
driver = SOAP::RPC::Driver.new(googleurl, urn)
```

Następnie definiujemy metody majace być wywołane. Metody dostarczone przez usługę Google SOAP są wylistowane w dokumentacji API, online:

```
driver.add_method('doGoogleSearch', 'key', 'q', 'start', 'maxResults', 'filter',
'restrict', 'safeSearch', 'lr', 'ie', 'oe')
```

Zgodnie z dokumentacją, usługa sieciowa Google zwraca wynikia jako wartości UTF-8. UTF-8 może zawierać zanki specjalne, których Ruby nie może obsłużyć w łańcuchu, powodując w naszym sterowniku błąd wywołania .Aby tego uniknąć, ręcznie ustawimy nasze kodowanie na UTF-8 używając biblioteki XSD

```
XSD::Charset.encoding = 'UTF8'
```

Teraz nasze wyniki będą właściwie kodowane dla zastosowania natywnych łańcuchów Ruby. Biblioteka Soap4r i WSDL Factory wspomniane wcześniej różniwż zależą od biblioteki XSD dal obsługi konwersji dane – typ z wynioków usługi sieciowej do natywnych typów Ruby.

Kończymy nasz kontroler przez wywołanie zdalnej metody doGoogleSearch. Metoda doGoogleSearch jest metodą sterownika jaką stworzyliśmy i skonfigurowali poprzednio:

```
@result = driver.doGoogleSearch(yourkey, @yourquery, 0, 3,
                                false, '', false, '', '', '')
```

Odwzorowanie obiektów pozwala nam na dostęp do wyników albo jako metoda (na przykład @result.estimatedTotalResultCount) lub jako wartość hash (@result["estimatedTotalResultCount"]).Ponieważ Ruby zakłada ,że identyfikatory które zaczynają się od dużej litery są nazwami klas lub stałymi, podejscie metodą autoamtycznie konwertuje prierwszy znak każdej nazwy metody na małe litery. Ta konwersja oznacza ,ze nazwy metdo w kodzie Ruby nie koniecznie pasują do nazw metod z usługi sieciowej. Czasami taka rozbieżność może być kłopotliwa; a przykład możesz uzyskać dostęp do URL zwracanego w wynikach jako result.uRL,ale to jest nienaturalne.Jest to dobry pomysł aby użyć podejscia has dla dowolnych metod, któe zaczynają się od dużej litery na przykład, rec["URL"]. W tym przypadku, użycie skrótu jest dużo bardziej naturalne. Nasz widok używa obu podejść do wyświetlania wyniku:

```
Query for: <%= @yourquery %><br>
Found: <%= @result.estimatedTotalResultsCount %><br>
Query took about <%= @result.searchTime %> seconds<br>
<% @result.resultElements.each do |rec| %>
 <b>Title:</b> <%= rec["title"] %><br>
 <b>Summary:</b> <%= rec.snippet %><br>
 <b>Link:</b> <a href="<%= rec["URL"] %>"><%= rec["URL"] %></a>
 <br><br>
<% end %>
```

Chociaż ten przykład jest prosty, jest również brzydki. Po stworzeniu sterownika SOAP, musimy wywołać add_method aby powiedzieć mu ,że wszystkie metody mają być wywołane. Ten wymóg prowadzi do kodu ,który jest nieelastycny,rozdęty i podatny na błędy. Możemy rozwiązać ten problem przez użycie pliku WSDL, który jest opisem XML API usługi sieciowej i biblioteki soap/wsdlfactory. WSDLDriverFactory pobiera plik WSDL , przetwarza go i tworzy sterownik SOAP, który rozumie API usługi. Jednak,jest jedno ale: Rails automatycznie nie ładuje przy starcie automatycznie soap/wsdlfactory. Musimu sięupewnić ,że mamy dobrze skonfigurowane.Oto jak uczynić klient SOAP dla aplikacji Rails , która używa pliku WSDL:

1.Zaktualizuj plik environment.rb dla załadowania bilioteki WSDLDriver
2.Stwórz instancję SOAP dla pliku WSDL
3.Wywołaj metodę usługui sieciowej
4.Użyj wyników w aplikacji

Przeróbmy nasz przykład Google do użycia pliku WSDL, zamiast ręcznego definiowania metdo jakie chcemy wywołać. Zacznijmy od uaktualnienia envirnoment.rb:

```
require 'soap/wsdlDriver'
```

Zrestartuj serwer aby dokonać zmian w środowisku. Teraz jesteśmy gotowi dla wersji WSDL kontrolera:

```
class CodeController < ApplicationController
  def googletest
    yourkey = 'YOUR GOOGLE DEVELOPER KEY'              # Your Google dev key
    @yourquery = 'SEARCH TEXT'                         # Search value
    XSD::Charset.encoding = 'UTF8'                     # Set encoding
    wsdlfile = "http://api.google.com/GoogleSearch.wsdl"     # WSDL location
    driver = SOAP::WSDLDriverFactory.new(wsdlfile).create_rpc_driver     # Create driver
                                                                         # and set up
                                                                         # methods
    @result = driver.doGoogleSearch(yourkey, @yourquery, # make our SOAP request
              0, 3, false, '', false, '', '', '')
    end
end
```

Jedyna realna zmiana jest taka ,że tworzymy sterownik używając pliku WSDL, jakiego dostarcza usługa sieciowa Google, zamiast przez końcowego URI i namespace

```
driver = SOAP::WSDLDriverFactory.new(wsdlfile).create_rpc_driver
```

Usunęliśmy wywołanie driver.add_method,ponieważ dłużej nie musimy określać jakie metody będziemy wywoływać; ta informacja pochodzi z pliku WSDL. Oczywiście, nie chroni nas to przed czytaniem dokumentacji usługi sieciowej.

WYŚWIETLANIE ZDJĘĆ Z FLICKRA UŻYWAJĄC XML-RPC

Flickr jest usługą współdzielenia zdjęć dostarczaną przez Yahoo! Podstawowe konta są darmowe, Flickr ma usługę sieciową API, który umożliwia dostęp przez REST, SOAP i XML-RPC. Ponieważ mamy już omówione SOAP i REST, zbudujemy naszego klienta używając architektury XML-RPC.

Potrzebny będzie klucz do Flickra przed dostępem do API. Ponieważ Flickr jest częścią Yahoo! Musisz najpierw założyć konto na Yahoo!, potem związać z usługa Flickr. Po uzyskaniu klucza można zacząć budowę klienta XML-RPC

1.Tworzymy sterownik RPC
2.Wywołuejmy metody
3.Używamy wyników w aplikacji Rails

Zademonstrujemy to ,przez pobranie ostatnich tych trzech zdjęć a Flickr używając metody intersetingness.getList. Zaktualizujemy plik code_controller.rb z poprzedniego przykładu zawierającego tą nową metodę flickrtest.

```
class CodeController < ApplicationController
  def flickrtest
     flickruri = "http://www.flickr.com/services/xmlrpc/"
     server = XMLRPC::Client.new2(flickruri)
     flickrkey = "YOUR FLICKR KEY"
     details = {:api_key => flickrkey, :per_page => "3"}
     result = server.call("flickr.interestingness.getList", details)
     @doc = REXML::Document.new result
   end
end
```

A tu mamy widok jaki wyświetli obrazki i ich tytuły. Zapiszemy poniższy kod jako flickrtest.rhtml w katalogu app/views/code/ :

```
<%
@doc.root.each_element do |res|
  image_url = "http://static.flickr.com/" \
   "#{res.attributes["server"]}/" \
   "#{res.attributes["id"]}_#{res.attributes["secret"]}.jpg"
%>
   <%= image_tag(image_url, :border => "0", :height => "100") %><br>
   <%= res.attributes["title"] %>
   <br><br>
<% end %>
```

I znowu działa.

flickrtest używa jednej dodatkowej biblioteki Ruby : biblioteki XMLRPC. XMLRPV jest czystą implementacją Ruby w specyfikacji XML-RPC. Jest również automatycznie ładowana i gotowa do użycia przez apliakcje Rails.

Wywołanie XML-RPC jest proste. Zaczniemy od stworzenia instancji naszego sterownika:

```
    flickruri = "http://www.flickr.com/services/xmlrpc/"
server = XMLRPC::Client.new2(flickruri)
```

Potem wywołujemy naszą aktualną metodę:

```
details = {:api_key => flickrkey, :per_page => "3"}
result = server.call("flickr.interestingness.getList", details)
```

Ustawiamy Hash dla przekazania klucza Flickr i ograniczenia wyników do trzech górnych pozycji, ponieważ dokumentacja Flickr mówi ,że usługa oczekuje strukturalnego typu danych. Biblioteka XMLRPC, podbnie jak biblioteka SOAP4R, automatycznie odwzorowuje Ruby Hash dla strukturalnyuch typów danych usługi sieciowej (znanje również jako złożona typ danych) kiedy tworzy nasze wyjściowe żądanie. Nie musimy robić doatkowego odwzorowania między usługą a natywnymi typami Ruby na wejściowych i wyjściowych żądań. W końcu, jak w przykładzie REST, konwertujemy nasze wyniki na dokument REXML, któy parsujemy:

```
@doc = REXML::Document.new result
```

Musimy sknwertować wyniki do dokumentu REXML ponieważ metoda Flickr interestingness.getList zwraca wyniki jako proste łańcuchy zaawierające reprezentację dokumentu XML. Kiedy mamy wyniki w dokumencie REXML, możemy potem parsować dokument. Dokumentacja Flickr mówi nam jak zbudować URL dla wyświetlenia obrazków jako naszych obrazków i jak wyświetlić tytuł dla każdego obrazka:

```
<%
@doc.root.each_element do |res|
  image_url = "http://static.flickr.com/" \
   "#{res.attributes["server"]}/" \
   "#{res.attributes["id"]}_#{res.attributes["secret"]}.jpg"
%>
   <%= image_tag(image_url, :border => "0", :height => "100") %><br>
   <%= res.attributes["title"] %>
   <br><br>
<% end %>
```