Web Services on Rails

By Kevin Marshall

.....

Publisher: **O'Reilly**
Pub Date: **May 2006**
Print ISBN-10: **0-596-52796-9**
Print ISBN-13: **978-0-59-652796-9**
Pages: **32**

Table of Contents

Overview

This PDF will show you how your small business or enterprise can publish its APIs (application programming interface) to a developer community just like the behemoths of the Internet--Google, Yahoo!, eBay, and Amazon.

These giants already offer their APIs to other programmers and allow for add-on services and mash-ups to develop from them--but until recently, the capability for most enterprises to do the same was limited by a myriad of competing standards and a lack of easy-to-use tools to accomplish the task.

Ruby on Rails levels the playing field for companies by simplifying the process of building web services and documenting APIs. Now with Rails, enterprise-quality tools are available for all developers.

In this document, we'll look at how Ruby on Rails makes building web service clients and servers simple and fun. Along the way, we'll give working examples and code details so you can see just how everything works.

Web Services on Rails

By Kevin Marshall

.....

Publisher: **O'Reilly**
Pub Date: **May 2006**
Print ISBN-10: **0-596-52796-9**
Print ISBN-13: **978-0-59-652796-9**
Pages: **32**

Table of Contents

Copyright
Web Services on Rails

Web Services with Ruby on Rails: Building Web Service Clients

The Basics of Web Services

Section 2.1.

Searching Yahoo using REST

Section 2.2.

Searching Google Using SOAP or SOAP with WSDL Files

Section 2.3.

Displaying Photos from Flickr Using XML-RPC:

Building Web Service Servers with Ruby on Rails

Section 3.1.

REST-Based Web Service Server Serving Proprietary XML Documents

Section 3.2.

SOAP and XML-RPC Web Service Servers

Section 3.3.

Putting It All Together: an Expanded Server Example

Section 3.4.

Summary

Section 3.5.

Additional Reading and Help

Copyright

Web Services on Rails

by Kevin Marshall

ISBN: 0-596-52796-9

Web Services on Rails

By Kevin Marshall

This PDF will show you how your small business or enterprise can publish its APIs (application programming interface) to a developer community just like the behemoths of the InternetGoogle, Yahoo!, eBay, and Amazon.

These giants already offer their APIs to other programmers and allow for addon services and mash-ups to develop from thembut until recently, the capability for most enterprises to do the same was limited by a myriad of competing standards and a lack of easy-to-use tools to accomplish the task.

Ruby on Rails levels the playing field for companies by simplifying the process of building web services and documenting APIs. Now with Rails, enterprise-quality tools are available for all developers.

In this document, we'll look at how Ruby on Rails makes building web service clients and servers simple and fun. Along the way, we'll give working examples and code details so you can see just how everything works.

Contents

Building Web Service Clients	**2**
The Basics of Web Services	**3**

Web Services with Ruby on Rails: Building Web Service Clients

From a high-level view, web service implementations can be broken down into two categories: servers and clients. I'll show how to create both clients and servers throughout this document. But since you're likely to build more clients throughout your career, I'll start with clients.

Most web services are based on one of three architectures: Representational State Transfer (REST), Simple Object Access Protocol (SOAP), or Extensible Markup Language Remote Procedural Calls (XML-RPC). Web services often offer access via two or more of these architectures. For example, we'll be offering both SOAP and XML-RPC access to clients in the ActionWebService server examples later in this book, so clients can implement the architecture that's easiest for their specific application.

If you're building your web service clients in Ruby and implementing them as part of a Rails application, you are in luck. Building web service clients with Ruby on Rails requires only a few simple steps and involves just a few Ruby libraries. Even better news is that the majority of libraries you use for building clientsCGI, NET, REXML, Soap4r, XSD, and XML-RPCare automatically loaded by your Rails environment. All you have to worry about is knowing when, how, and where to use each library, and that's just what I'll help you figure out throughout the rest of this document.

If you're wondering why I'm talking about Rails (a server platform) and web service clients in the same breath, take a step back. Most web service clients aren't desktop programs or command-line programs. Instead, most web service clients are servers: servers

that gather data from other web services and then repackage it for some other purpose. (That other purpose is frequently a human-readable web page, but it could just as well be yet another web service.) That's why the clients we develop here will be part of a Rails application.

I'll cover all three of the most popular types of web services: REST, SOAP, and XML-RPC. And to make things as practical as possible, I'll use three of the most popular, free, and useful web services out there for examples: Yahoo! Search, Google Search, and Flickr. Since I'm a big believer in learning by doing, I'll walk you through the details of building web service clients for Rails by developing real-world, working clients for each of these services.

All this document's examples will focus on building working clients to demonstrate web services concepts, so I won't discuss any one service in detail. My goal isn't to teach you everything you might need to know about a specific web service's interface, or even everything you might need to know about SOAP or REST concepts. Rather, my intention is to show you how to apply web service concepts to your Rails applications. My hope is that you will then take these examples as starting points and go on to build the next great client we've all been waiting for! But before I get too deep into specific examples, I'll cover a few quick basics about Ruby on Rails and web services.

The Basics of Web Services

As with any programming language, architecture, or standard, you must become familiar with a lot of terminology before you can start implementing anything. So I'll start by introducing some of the basic terminology and technologies you need to understand before coding your own web service clients and servers. Don't worry: I'm not going to go back to ground zero and explain XML. Nor am I going to give you a blow-by-blow description of what's in a WSDL document.

Probably the most important thing to cover is the idea of *web service signatures*. Web service signatures are really just data types that the service either expects or returns. Web services are not language-specific: you can use your programming language of choice to implement a client or a server. A client written in C# or Java must be able to communicate with a client written in Ruby, and vice versa. One consequence of this is that, to be compatible with strongly typed languages like Java, web services must also be strongly typed. For the most part, the object types you are likely to come across in web services are things you would expect: integers, strings, booleans, floats, and datetime types. However, web services often deal with more complex types as well, such as arrays and structures. From a Ruby point of view, dealing with all these different data types is really quite simple. All the standard data types basically map to their Ruby counterparts, arrays map to Ruby Arrays, and structures map to Ruby Hash types. You'll see these standard types in action through the examples.

Another important piece of understanding the web service puzzle is the *WSDL file*. WSDL stands for Web Service Description Language; a WSDL file is an XML document that defines the interface to a SOAP service. WSDL files provide details about the methods that the service exposes, the methods' arguments and return values, and the encodings used for data traveling between

the client and the server. Everything you would ever need to know about a SOAP service can be described in a WSDL file; WSDL files serve both as a form of documentation for SOAP services and as a key to automating many of the steps of building SOAP clients. Thus it's possible to read a WSDL file and find out everything you need to know about the API it describes. WSDL is documented at http://www.w3.org/TR/wsdl, and in many books about web services. Reading a WSDL file isn't fun, but, fortunately, it's hardly necessary these days (though still a good skill to have). The primary use of WSDL is to automate the client code so that you don't have to explicitly write code to handle different data encodings, different mappings between object types, and so on. (I'll talk a little bit more about using a WSDL file to automate some of your client code in the Google example.)

Where does this complex, magical WSDL file come from? In the bad old days, it had to be written by hand. But most modern web service platforms automatically generate WSDL files and make them available for clients to download. Rails is no exception. (The SOAP server example in this document includes a WSDL file that is automatically generated.) Given that the WSDL file is automatically generated and automatically consumed, you should never need to touch it.

Finally, I need to say a little about *web service architectures*. SOAP grew out of XML-RPC, so SOAP services and XML-RPC services are fundamentally similar. Both try to mimic "normal" programming operations: function calls (for XMLRPC) and remote method invocations (SOAP).

REST services represent a significantly different (and arguably much simpler) paradigm. REST stands for Representational State Transfer; the most authoritative description of this architecture is in Roy Fielding's dissertation, located at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. The basic idea behind REST is that you don't need to make web services

"look like" regular method calls or function calls. The basic HTTP operationsGET, PUT, POST, and DELETEcorrespond nicely to the four basic SQL operations: SELECT, UPDATE, INSERT, and DELETE. Therefore, it's possible to build complex applications by doing nothing more than using HTTP requests to move XML documents around.

With REST services, there is a lot of debate between purists (who have a fairly strict interpretation of what each operation should be used for) and others, who've basically said, "Let's build applications by pushing XML around; who cares whether it corresponds to a nice architectural model?" The debate between the people representing these extremes can become heated; most REST sites in the real world fall somewhere in the middle. Since my goal isn't to teach you everything there is to know about web services, I sidestep most of this debate and simply show you the easiest way I know to make each type of server and client work with Rails. I leave it up to you to decide where and when to adhere to various web service practices.

We've covered enough of the "getting started" material, so let's dive into the fun stuff and look at the basics of each type of client and the steps you need to get them working with your Rails application.

2.1. Searching Yahoo using REST

REST is often considered the simplest web service architecture. For the most part, REST operations work just like standard web page requests, with no fancy extras. A REST application simply makes a query via a URL, just like a normal web page request. The HTTP server returns a document with the results of the request. This returning document is usually in a proprietary XML format (though XML isn't a requirement; the service can return any data structure). In fact, if your browser is capable of displaying XML, you can enter the URL generated by our first client into the URL bar and see the raw XML results that Yahoo! generates.

Note: The world wide web itself can be considered a REST system.

A Rails-based REST client needs to:

1. Connect to the web service with a standard GET or POST request (depending on the service requirements), using the NET library
2. Store the results as an in-memory REXML document
3. Parse the results with the REXML library for use in the Rails application

To demonstrate, let's build a simple controller that looks for the top three results of a Yahoo! search, using Yahoo!'s REST interface. Yahoo offers a web service API for many of their services, including the popular Yahoo! search engine. The Yahoo! search service API is currently free, but it does limit the number of requests you can make

(currently to 5,000 requests per IP address per day). To use the Yahoo API (and to test the sample code), you'll need a free Yahoo! Developer's Key. Get your key directly from Yahoo at http://api.search.yahoo.com/webservices/register_application.

Once you have your developer key, we're ready to build a simple controller (saved as *code_controller.rb* in the *app/controllers* folder):

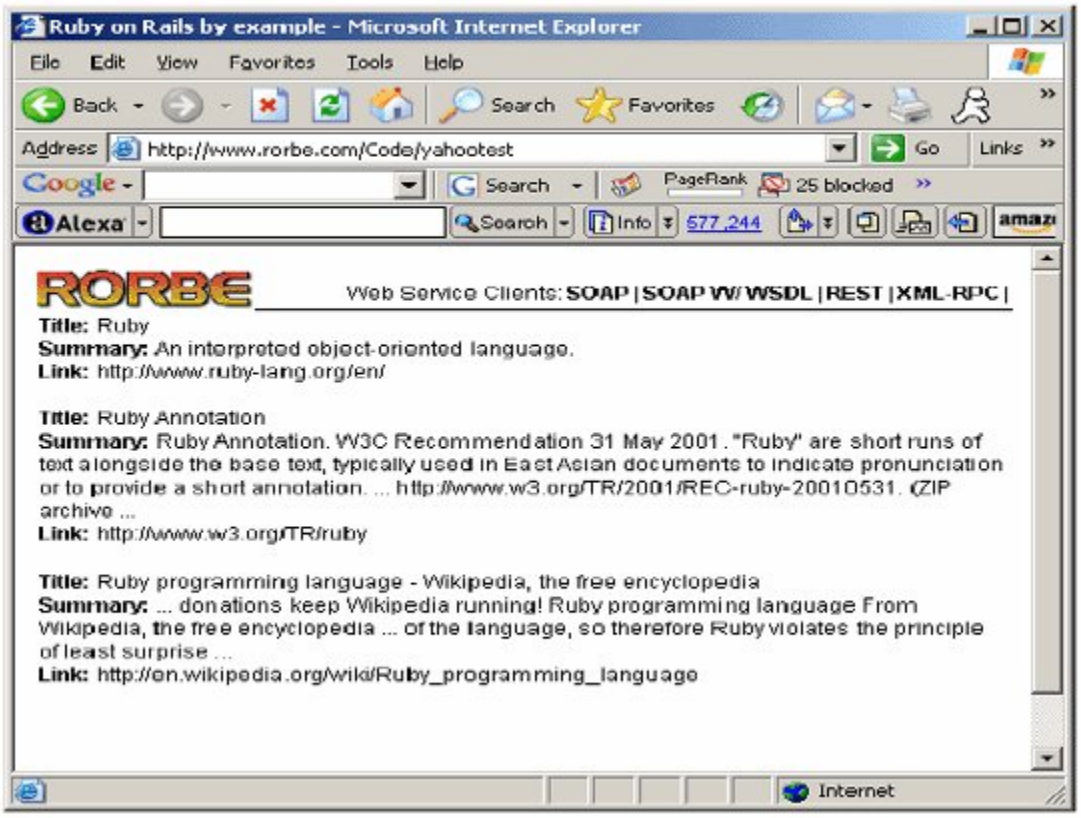
```
class CodeController < ApplicationController
  def yahootest
    query = CGI.escape("SEARCH TEXT")
# URL-encoded search value
    yahookey = "YOUR YAHOO DEVELOPER KEYs
# Your Yahoo! dev key
    url = "http://api.search.yahoo.com/" +
# The URL to the Yahoo!
      "WebSearchService/V1/webSearch?" +
# Search service
      "appid=#{yahookey}&query=#{query}" +
        "&results=3&start=1"
    result = Net::HTTP.get(URI(url))
# make the actual HTTP request
    @doc = REXML::Document.new result
# turn the results into a

# REXML document
 end
end
```

Then we build a view that displays the search results in a simple view. Save the following code in a file called *yahootest.rhtml* in the *app/code/views folder*:

```
<% @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 %>
```

Believe it or not, we're done. In just a few short lines of code, we've built a complete web service client for your Rails application. You should be able to test your application with your local version of Webrick at the URL *http://localhost:3000/code/yahootest*.

Yahoo! search results for our REST web service request



Now that we have the client working, let's look at what is going on with the code. First, even though there are no "require" statements, the client is using several important libraries: NET, CGI and REXML. All three libraries come standard with the Ruby 1.8.4 distribution and should be automatically loaded and ready for use by your Rails application. You don't have to do anything special to use them in your code. You just need to know that they are available.

I'm guessing you've heard of the NET and CGI libraries before, since Rails uses these libraries to accomplish a lot of its web-friendly magic, but you may not be familiar with the REXML library. REXML is a pure Ruby XML Processor with lots of nice features,

including full support for Xpath 1.0. For more information on REXML, including full documentation, visit http://www.germanesoftware.com/software/rexml/.

We first use the CGI library to escape our search term, ensuring that the search term is safe for use in our HTTP GET request:

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

Next, we save our Yahoo! Developer Key in a variable and build the URL with parameters specified in the documentation for the Yahoo! API. For additional parameters and options, see the online Yahoo! documentation at http://developer.yahoo.net/:

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

Note: You cna see the XML documents that Yahoo! returns by using this URL in any browser that can render XML.

Understanding the URL isn't too difficult: we're calling version 1 (V1) of the `webSearch` service, passing it four parameters: the `appid` (the developer key), the `query` (the search string), the number of `results` we want (3), and `start`, which tells the Yahoo! web service the number of the first result we want it to return. In a larger example, we might use this parameter to help page through a long set of

results; right now, we just want the first three results, so we start with result 1. Our next step is to use the NET library to make the HTTP GET request and store the returning document in a variable:

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

After we have our results stored in a local variable, we use the REXML library to convert the results into a REXML document:

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

Finally, our view uses the REXML library to parse the XML document and display the results. REXML offers a number of ways to access tag and attribute values. Here we use array-like methods:

```
<% @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 %>
```

We use the `each_element` method on the root element of the document (`ResultSet`) to access each result (`Result`). Since Element is a subclass of Parent, we can then use Array-like methods to access the children that we intend to display (`Title`, `Summary`, and `Url`). See the online documentation for additional information about how to access elements and attributes with REXML. Because it is so direct and conceptually simple, many developers feel that

RESTas we implemented itis the easiest type of web service to use. While that's probably true of many other languages, it could be said that building REST clients in Ruby actually involves more work than building clients with SOAP or XML-RPC. That's because the SOAP and XML-RPC Ruby libraries handle all the low-level details for you: you don't need to manage communication to and from the server, build the XML request, or in many cases even parse the XML response. These all happen automatically, leaving you free to work with your results and focus on your business logic.

So let's move on to the next web service type, SOAP, and see just how easy it is to write SOAP clients for Rails.

2.2. Searching Google Using SOAP or SOAP with WSDL Files

Over the past few years, the SOAP standard has gathered a lot of support, in part because it is now officially documented by the World Wide Web Consortium (W3C). Standardization makes working with most SOAP services dependable, at least as far as the information you need to know and the information you can reasonably expect to get back. Because of the W3C backing, it could be said that SOAP is now the preferred architecture for most web servicesthough, despite W3C support, there's significant evidence that REST-ful services are more widely used. Nevertheless, if you're going to be working with web services, you need to be conversant with all three architectures: REST, XML-RPC, and SOAP.

SOAP's biggest drawback is its complexity, but Rails hides most of that complexity from you. Creating a SOAP client takes four simple steps:

1. Create an instance of a SOAP driver
2. Define the SOAP methods you want to call
3. Call the SOAP methods
4. Use the results in your Rails application

To demonstrate, we'll build a Google search using SOAP. Like Yahoo!, Google offers a free web service API for many of their services, including their search engine. To use the Google search API (and to test the sample code), you'll need a free Google Developer's Key, which you can get directly from Google at https://www.google.com/accounts/NewAccount?continue=http://api.google.com/createkey&followup=http://api.google.com/createkey.

Here's a controller that uses SOAP to find the first three results from a Google search. Update the *code_controller.rb* file from the previous example to contain this new `googletest` method:

```
class CodeController < ApplicationController
  def googletest
     yourkey = 'YOUR GOOGLE DEVELOPER KEY'
# Your Google dev key
     @yourquery = 'SEARCH TEXT'
# Search value
     XSD::Charset.encoding = 'UTF8'
```

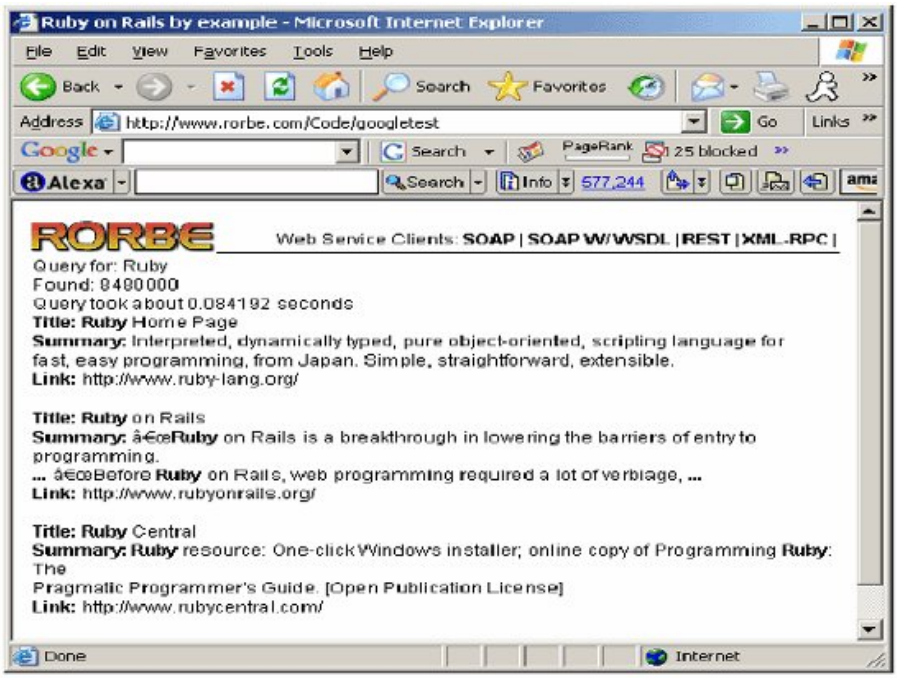
```
# Set encoding for response
     googleurl = "http://api.google.com/search/beta2"
     urn = "urn:GoogleSearch"
     driver = SOAP::RPC::Driver.new(googleurl, urn)
# Create our driver
     driver.add_method('doGoogleSearch', 'key', 'q',
# Set up methods we'll call
        'start', 'maxResults', 'filter', 'restrict',
        'safeSearch', 'lr', 'ie', 'oe')
     @result = driver.doGoogleSearch(yourkey,
# make our SOAP request
        @yourquery, 0, 3, false, '', false, '', '', '')
  end
end
```

Here's the code for the view. Save it as *googletest.rhtml* in your *app/views/code/* directory:

```
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 %>
```

And once again, we're done. We've built a complete SOAP client that you should be able to test at the URL *http://localhost:3000/code/googletest*.

Google Search results from our SOAP-based web service client



This client uses two additional Ruby libraries: Soap4r and XSD4R. Soap4r is a pure Ruby implementation of the SOAP 1.1 standard. For full documentation and more information, to report bugs, or to obtain the latest version of Soap4r, visit http://dev.ctor.org/soap4r. The XSD4R library is an XML support library that's used by Soap4r. It provides conversions between certain common data types. For complete documentation on the XSD library, see http://rubydoc.org/stdlib/libdoc/xsd/rdoc/index.html. Both Soap4r and XSD4R are part of the Ruby 1.8.4 distribution and should be loaded automatically; your code doesn't need to "require" them.

We start our Google example by creating an instance of a SOAP driver using the Google SOAP service URI and Namespace:

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

Next we define the methods we intend to invoke. The methods provided by the Google SOAP service are listed in Google's online API documentation:

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

According to the documentation, the Google web service returns all results as UTF-8 encoded values. UTF-8 can contain special characters that Ruby can't handle in a string, causing our driver to throw an `XSD::ValueSpaceError` error upon invocation. To avoid this problem, we manually set our encoding to UTF-8 using the XSD library.

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

Now our results will be properly encoded for use as native Ruby strings. The Soap4r library and the WSDL Factory library mentioned later also rely on the XSD library to handle the data-type conversions from web service results to native Ruby types.

We finish off the controller by invoking the `doGoogleSearch` remote method. `doGoogleSearch` is a method of the driver we created and configured previously:

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

The view displays the results, which are returned as SOAP Mapping objects. Mapping objects allow us to access the results as either methods (for example, `@result.estimatedTotalResultsCount`) or as hash values (`@result["estimatedTotalResultsCount"]`). Because Ruby assumes that identifiers that begin with an uppercase letter are class names or constants, the method approach automatically converts the first character of each method name to lowercase. This conversion means that the method names in the Ruby code don't necessarily match the method names from the web service. Sometimes the mismatch can be awkward; for example, you could access the URL returned in the result as `result.uRL`, but that's very unnatural. It's therefore a good idea to use the hash approach for any methods that start with an uppercase letterfor example, `rec["URL"]`. In this case, using the hash is a lot more natural. Our view uses both approaches to render the results:

```
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 %>
```

Though this example is simple, it's also a bit ugly. After creating the SOAP driver, we had to call `add_method` to tell it all the methods we were going to call. This requirement leads to code that is inflexible, bloated, and bug-prone: will you remember to update the call to `add_method` if, a year from now, you add a feature that calls a new method of the API? We can solve this problem by using a WSDL file, which is an XML description of the web service's API, and the *soap/wsdlfactory* library. The WSDLDriverFactory downloads the WSDL file, processes it, and creates a SOAP driver that understands the service's API. However, there is one catch: Rails doesn't automatically load *soap/wsdlfactory* when it starts. So we've got to make sure we configure our environment to include it first. Here's how to make a SOAP client for your Rails application that uses a WSDL file:

1. Update the Rails *environment.rb* file to load the WSDLDriver library
2. Create a SOAP instance from a WSDL file
3. Call the web service's methods
4. Use the results in your Rails application

Let's refactor our Google example to use a WSDL file, instead of manually defining the methods we want to call. Start by updating *environment.rb*:

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

Then restart your Webrick server, so that it picks up this change to the environment. Now you're ready for the WSDL-enabled version of the controller:

```
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
```

The only real change is that we create the driver using a WSDL file that's provided by the Google web service, rather by specifying an endpoint URI and namespace:

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

We've removed the call to `driver.add_method`, since we no longer need to specify which methods we intend to invoke; that information now comes from the WSDL file. Of course, this doesn't save us from reading the web service's documentation; we still need to know the required parameters and the returned values of any methods we call.

Nothing we've done requires any changes to the view, so you can test this new application at the URL *http://localhost:3000/code/googletest*. Complete documentation of the search features and the WSDL file used in our example can be found at *http://www.google.com/apis/*.

We have just one more type of web service client to cover, so let's shift our focus to XML-RPC and build a client for the very popular Flickr service.

2.3. Displaying Photos from Flickr Using XML-RPC:

Flickr is primarily a photo-sharing service provided by Yahoo!, though it has recently branched out to offer other features. Basic accounts are free, and Flickr has a web service API that provides access via REST, SOAP, and XML-RPC. Because we've already covered SOAP and REST, we'll build our client using the XML-RPC architecture.

As you probably expected, you need a Flickr key before you can access their API. Since Flickr is a part of Yahoo!, you must set up a Yahoo! account first, then associate that account with the Flickr service. Start at http://www.flickr.com/signup, which walks you through the sign-up process. Once you've got your Flickr key, you're ready to start building your XML-RPC client.

To make an XML-RPC client for your Rails application you need to:

1. Create an RPC driver
2. Call the methods
3. Use the results in your Rails application

We'll demonstrate by getting the latest three images from Flickr using their `interestingness.getList` method. Update the *code_controller.rb* file from the previous examples to contain this new `flickrtest` method. Here is the code for our controller:

```
class CodeController < ApplicationController
  def flickrtest
     flickruri =
"http://www.flickr.com/services/xmlrpc/"
```

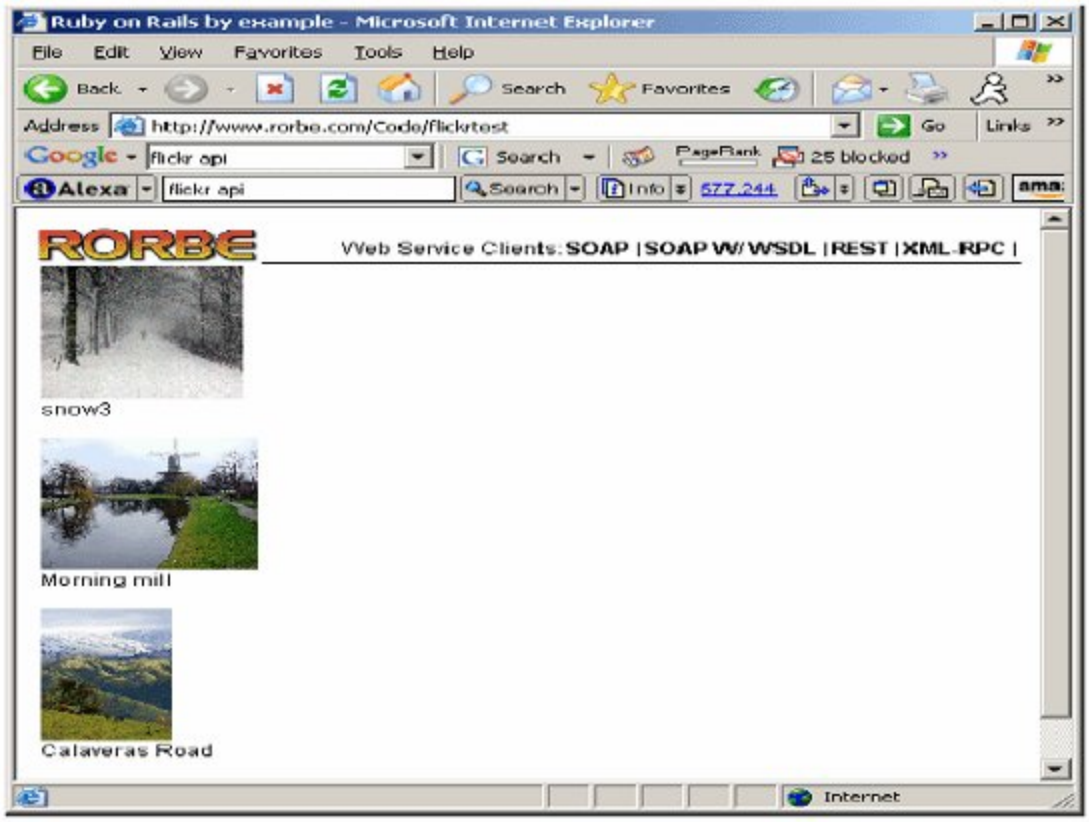
```
      server = XMLRPC::Client.new2(flickruri)
      flickrkey = "YOUR FLICKR KEY"
      details = {:api_key => flickrkey, :per_page
=> "3"}
      result =
server.call("flickr.interestingness.getList",
details)
      @doc = REXML::Document.new result
    end
end
```

And here's the view that displays the images and their titles. Save the following code as *flickrtest.rhtml* in the *app/views/code/* directory):

```
<%
@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 %>
```

Once more, that's all there is to it! You should now be able to test your application at *http://localhost:3000/code/flickrtest*.

Flickr results from our XML-RPC web service client



`flickertest` uses one extra Ruby library: the XMLRPC library. XMLRPC is a pure Ruby implementation of the XML-RPC specification. It comes standard with the Ruby 1.8.4 distribution. XMLRPC is also automatically loaded and ready for use by your Rails application. For full documentation and more information, to

report bugs, or to obtain the latest version of XMLRPC, see http://www.ntecs.de/projects/xmlrpc4r/.

Making XML-RPC calls is simple and straightforward. We start by creating an instance of our driver:

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

Then we make our actual method calls:

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

We set up a Hash to pass the Flickr key and to limit results to the top three items, because the Flickr documentation states the service expects a structure data type. The XMLRPC library, like the SOAP4R library, automatically maps a Ruby Hash to a web service struct data type (also known as a complex data type) when it creates our outgoing request. We don't have to do any additional mapping between the service and native Ruby types on either outgoing or incoming requests.

Note: Web service structures, also referred to as complex data types, map to the Ruby Hash type.

Finally, as in the REST example, we convert our results into a REXML document that we can then parse:

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

We need to convert the results into a REXML document because the Flickr `interestingness.getList` method returns results as a simple string containing the representation of an XML document. In general, though, complex XML-RPC results are accessed just like the SOAP results in the previous example were: either as methods or as a hash.

Once we have molded our results into a REXML document, we can then parse the document. Flickr's online documentation tells us how to build the URL to display images from our results and how to display the title for each image:

```
<%
@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 %>
```

In just a few short pages and with very little code, we've built working REST, SOAP, and XML-RPC clients for three of the most

popular public web services available today! But we're just getting started. The next section shifts the focus from clients to servers. This is where Rails kicks it up yet another notch, making things even easierand in my opinion more funfor developers.

Building Web Service Servers with Ruby on Rails

In the first section, I covered the basics of building clients for the three most poplar web service types (REST, XML-RPC, and SOAP). Now it's time to get down to the really interesting stuff: building web service servers! I'll once again cover the three most popular formats for web services: REST, SOAP, and XML-RPC. I'll also provide working examples to demonstrate each concept and give you a head start on building your own services.

3.1. REST-Based Web Service Server Serving Proprietary XML Documents

RESTfor our purposesis really the same thing as a simple web GET request: it requires no special server setup or request encoding and decoding. You simply use your favorite web server and serve your results in whatever format you choose to designusually a custom XML format. Regardless of the implementation language, most developers agree that REST services are simple to build and flexible enough to adapt to your needs. Additionally, if you closely adhere to the true intent of REST, you can take advantage of any HTTP-based filters and security measures your server has in place. Because of these types of benefits, and REST's simplicity, most of the web services out in the "real world" are REST services, rather than SOAP or XML-RPC services.

When it comes to Rails applications, RXML templates help make it simple to build REST-based servers with custom XML documents. In fact, to build REST-based web servers with Ruby on Rails, you really only need to:

1. Make sure you turn off layouts for the methods you'll be using in your service

2. Associate RXML templates with the methods in your controllers, instead of RHTML templates

Let's build a simple service to demonstrate. Here's a controller with some dummy data. Update your existing *code_controller.rb* file from the previous examples to contain this new `restserver` method and layout command:

```
class CodeController < ApplicationController
 layout "application", :except => [:restserver]
```

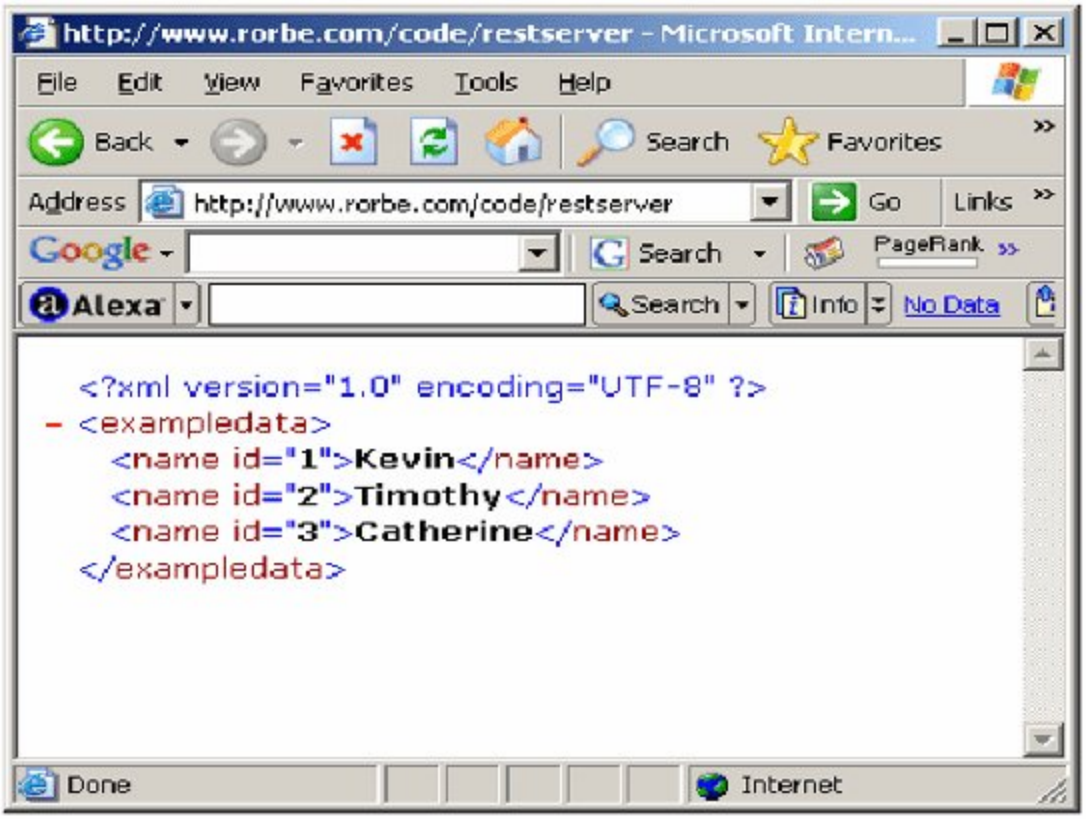
```
 def restserver
   @sampledata = ["Kevin","Timothy","Catherine"]
 end
end
```

Here's the view associated with the controller; save it as *restserver.rxml* in the *app/views/code* directory):

```
xml.instruct! :xml, :version=>"1.0",
:encoding=>"UTF-8"
xml.exampledata do
  counter = 0
  @sampledata.each do |rec|
     counter = counter + 1
     xml.name(rec, :id => counter)
  end
end
```

And just like that, your simple REST-based server is done! REST clients can access the service at the URL *http://localhost:3000/code/restserver*.

Custom XML generated by our REST-based web service



This example used one additional Ruby library: the Builder library. According to its documentation, the Builder library "provides a simple way to create XML markup and data structures." It gets installed with Rails by default, but if you want to make sure you have the latest version, you can use the `gem` command:

```
gem install builder
```

Complete documentation on the Builder library can be found at http://builder.rubyforge.org.

The controller for our application is very simplein fact, too simple. In real life, the controller would include other actions associated with a "regular" web site. So let's assume those methods exist and that they use standard RHTML templates. We need to disable the layout, but only for those actions that are part of the REST service (the `restserver` action). So we call the `layout` method with the `except` parameter:

```
layout "application", :except => [:restserver]
```

We've disabled the layout, but what about any action-specific templates? RHTML templates take precedence over RXML templates. If you have both *restserver.rhtml* and *restserver.rxml* in your *views/code* directory, *restserver.rhtml* will be used for rendering. So it's important to be careful about your file extensions when building REST-based services.

That's about all there is to the controller; the `restserver` method just packs an instance variable with data that the view will put into the XML document.

The view (in this case, an RXML template) uses the Builder library to build an XML document. Rails' ActionPack library knows to use the Builder library to render this view because of the RXML extension. Using Builder to generate your XML is straightforward. Since we want our document to be a true XML document, we start by calling the `xml.instruct` method, where `xml` is an XML document object that's available in any RXML template:

```
xml.instruct! :xml, :version=>"1.0",
:encoding=>"UTF-8"
```

To add tags to the document, we call methods on the `xml` object. The method name is the name of the tag. The tag's content (the text between the beginning and ending tags) is the first parameter to this method. Pass any attributes for this tag as hash values. Here's a very simple example:

```
xml.name("Kevin", :id => 1, "age" => "31")
# creates <name id="1" age="31">Kevin</name>
```

In a real document, of course, you need to deal with nested tags. You can nest tags by defining the children tags inside the block of the parent tag:

```
xml.exampledata do
     xml.name("Kevin")
 end
 # creates <exampledata><name>Kevin</name>
</exampledata>
```

Additional details and complete documentation on the Builder library can be found online at http://builder.rubyforge.org/.

As you can see, building REST web services really is straightforward. But you still need to do a lot of grunt work: you need to create an XML document, then tell your clients how to parse your proprietary documents and how to invoke your service. This is one of the main reasons SOAP and XML-RPC were created: to give web services a standard for communication between the server and client. SOAP and XML-RPC allow developers to build clients without spending a lot of time learning how to communicate with the server for each new service. So let's shift focus to building SOAP and XML-RPC servers.

3.2. SOAP and XML-RPC Web Service Servers

Rails comes with a component called ActionWebService (often referred to as AWS) that makes hosting SOAP and XML-RPC web services simple and efficient. ActionWebService allows you to make SOAP and XML-RPC methods available by binding your web service to controllers in your Rails application. It takes care of almost all the technical details for you: parsing the XML request, creating the XML response, and even creating the appropriate WSDL file for your SOAP service. This all means you're free to focus on your business logic, without having to worry about all the protocol specifics.

To update your version of ActionWebService, or to install it independently of Rails, use the `gem` command:

```
gem install actionwebservice
```

AWS supports many of the common Rails tools to speed up the development cycle. It includes a `web_service_scaffold` method and a script to generate the base files, code, and some functional tests:

```
script/generate web_service YOURSERVICE
YOURMETHOD1 YOURMETHOD2
```

All of this makes building web services with AWS very easythough we will skip the use of these convenience tools in order to give a more detailed explanation of

the steps involved in creating your servers. There are only three simple steps you need to follow:

1. Determine which dispatching mode fits your needs (Direct, Delegated, or Layered).

2. Create your Application Programming Interface (API), providing the details about the methods you'll make available.

3. Create your methods (either in controllers for direct dispatching or in models for layered or delegated modes).

Before we talk too much about the details and options of each step, let's build a very basic service so we have a point of reference. We'll create a web service with a single method called `dogreeting`. This method expects a username of type `string` as a parameter and returns a greeting in the form of a string. We start by defining our API in the file *app/apis/greeting_api.rb*. This file lists the methods our services makes available:

```
class GreetingApi < ActionWebService::API::Base
   api_method :dogreeting,
     :expects => [{:username => :string}],
     :returns => [{:greeting => :string}]
end
```

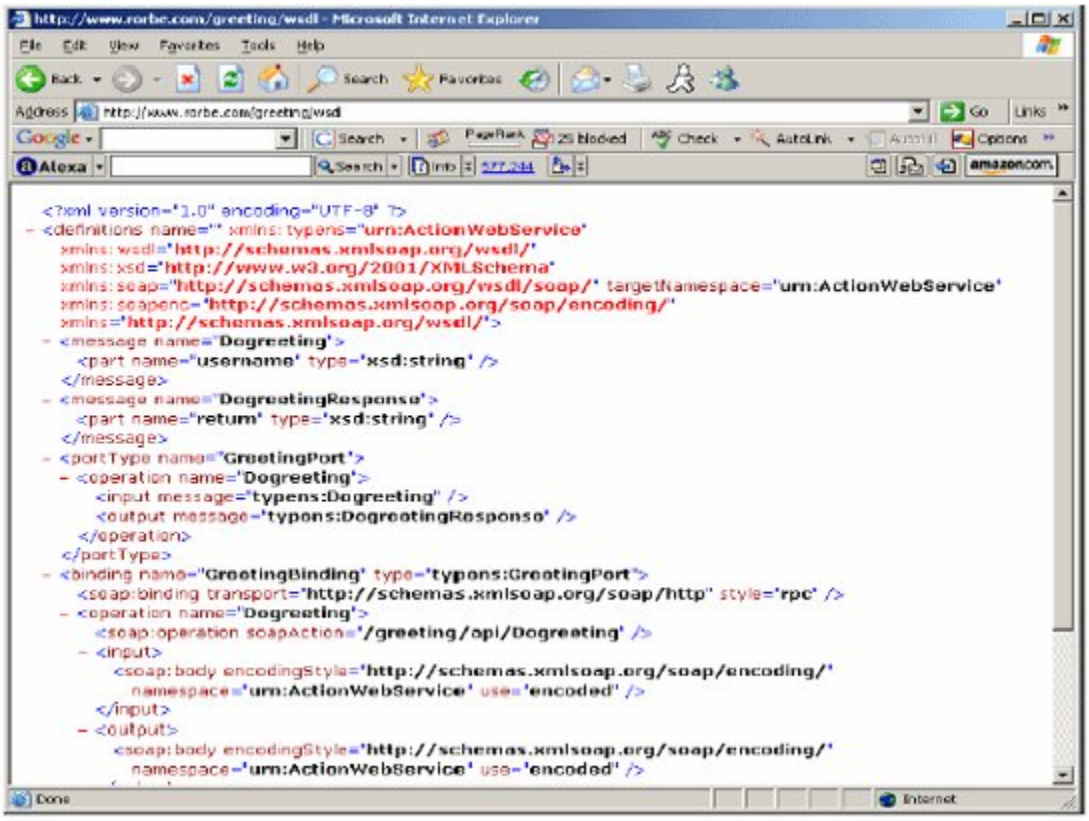
Note: You could also use the command line (`ruby script/generate webservice greeting dogreeting`) to generate the base files and then edit them to match our example.

Next we write a controller that implements the methods defined in the API. Rails automatically looks for a `GreetingApi` implemented by the GreetingController. So we'll save the following file as *app/controllers/greeting_controller.rb*:

```
class GreetingController < ApplicationController
  def dogreeting(username)
      "Hello #{username}"
  end
end
```

That's all there is to it! We've now built a very simple, direct-dispatching web service with Rails! ActionWebService generates a WSDL file for the service automatically; SOAP clients can download this WSDL file from *http://localhost:3000/greeting/wsdl*.

Our dynamically generated WSDL file



SOAP clients can call methods using *http://localhost:3000/greeting/api* as the endpoint URI and *urn:ActionWebService* as the namespace. XML-RPC clients can access the service with the URI *http://localhost:3000/greeting/api*.

Now it's time to test our service to make sure it's really working as expected. Save the following code as *test/functional/greeting_api_test.rb*:

```
require File.dirname(__FILE__) + '/../test_helper'
require 'greeting_controller'
```

```
class GreetingController; def rescue_action(e)
raise e end; end

class GreetingControllerApiTest <
Test::Unit::TestCase
     def setup
         @controller = GreetingController.new
         @request =
ActionController::TestRequest.new
         @response =
ActionController::TestResponse.new
     end

     def test_dogreeting
         result = invoke :dogreeting, "Kevin"
# this is the real test call
         assert_equal "Hello Kevin", result
     end
end
```

Make sure that you have Webrick running in one command window, and then execute the above test in a separate command window with the command `ruby test\functional\gretting_api_test.rb`. If everything goes as planned, you should get results similar to:

```
Loaded suite greeting_api_test
Started
.
Finished in 0.75 seconds.
1 tests, 1 assertions, 0 failures, 0 errors.
```

Note: The following code can also be used to test your service. Save the XML-RPC and SOAP clients as local Ruby programs and run them from the command line:

This simple example takes advantage of a number of "magical" things Rails does, such as setting the dispatching mode and automatically associating the API to the controller. So before starting another example, let's talk about the some of these magical things, and what our other options for them are.

The first step in building a web service with Railsafter going through the design phase, of courseis to determine your dispatching mode. AWS offers three dispatching options, `:direct`, `:delegated`, or `:layered`. The dispatching mode controls the routing of your web service method invocations (from clients) to your Ruby methods. The options differ in where you implement your methods, and in the address your clients use to access your web service. To specify the dispatching mode, the controller for your service calls the `web_service_dispatching_mode` method with the argument `:direct`, `:delegated`, or `:layered`.

The default mode is `:direct` (`web_service_dispatching_mode:direct`). With direct dispatching, you just have one controller and one API. You code everything like any other Rails application, except that your web service methods won't have RHTML views. Direct mode's disadvantage is that you can associate only one API file with a controller. This limitation becomes a problem if you want multiple points of entry to your web service or if you want to combine existing services into one larger web service.

For example, let's say you develop a `:direct` web service that lets users search the data at *yourdomain.com/data/getuserdata* and *yourdomain.com/data/getproductdata*. Now let's say that you decide to break these out into two additional access points:

yourdomain.com/user/getdata and *yourdomain.com/product/getdata*. Direct dispatching would require two controllers (one for *user* and one for *product*). These two controllers would have essentially the same code. While this might be acceptable in certain situations, it's definitely not the Rails way and goes against the "don't repeat yourself" (DRY) philosophy.

The other two dispatching options are `:delegated` and `:layered`. The code for these two modes is identical. You define all your web service methods in a model, associate the model with your API, and then reference the models from any controllers you like. To solve the *user* and *product* problem, you would create separate product and user models. Each model would then contain its own `getdata` method and would be associated with an API file defining its related `geTData` web service input and output parameters. Finally, each controller would use the `web_service` method to declare which methods are available to the outside worldfor example `web_service :getdata, Product.new`.

When you use the `:delegated` or `:layered` dispatching modes, your code is slightly harder to follow: you define the methods for each API in a separate model, rather than directly in the controller. However, `:delegated` or `:layered` is the best way to go for any large or complex web service; they are more flexible in the long run. `:direct` looks good as long as the service remains simple and has relatively few features, but when have real-world applications stayed simple or gone according to design?

Since our Rails code is the same for both `:layered` and `:delegated` dispatching, you are probably wondering why there are two options at all. The answer is in the endpoint URL that clients use to access the web service. With `:delegated`, clients use a distinct URL for each method in the APIfor example, *yourdomain.com/product/getdata* and *yourdomain.com/product/getprice*. With `:layered`, clients use the same URL for all attached methods and rely on AWS to route the request based on information passed in its headerfor example,

yourdomain.com/product/api. If a SOAP client uses a WSDL file to define their method calls, there really is no difference between the `:delegated` and `:layered` dispatching modes.

Regardless of the dispatching mode, you always develop Rails web services by defining an API that lists the methods and data types the web service will provide. AWS uses this API to route requests and generate errors if parameters are missing or a connection problem arises. The `api_method` method defines the methods that the services makes available. It can have the optional `:expects` and `:returns` parameters to define web service signatures. If you omit `:expects`, the web service will generate errors for any clients that attempt to pass parameters during the method call. If you omit `:returns`, any calls to the method will not receive a result.

Being that Ruby is a loosely typed language, `:expects` and `:returns` may feel very strange, but they're necessary because web services are strongly typed creatures. Specifying data types for parameters and return values makes sense when you consider that your web service will communicate with clients written in many languages, including statically typed languages, such as Java and C#. Without defining types, you could potentially send these clients an object they weren't expecting or couldn't handle, causing the client to fail or deliver inconsistent and unreliable results.

In the call to `api_method`, the `:expects` and `:returns` symbols are used as Hash keys that point to an array of parameter types or an array of return-value types. (Like Ruby methods, web service methods support multiple return values.) The elements of these arrays are symbols representing standard Ruby types `(:string`, `:int`, `:bool`, `:float`, `:time`, `:datetime`, `:date`, and `:base64`), or the names of `ActiveRecord::Base` or `ActionWebService::Struct` classes (for example, `Greeting` or `Account`), or a single element array to represent arrays of objects (for example, `[:string]` to represent an array of `String`s or `[Account]` to represent an array of `ActionWebService::Struct Account` objects). Here's how to specify the signature of a method named

`dogreeting` that expects a string as a parameter and returns a complex data structure:

```
api_method :dogreeting, :expects => [:string],
:returns => [Greeting]
```

If you want to provide more documentation, you can provide a one-item Hash with the name of the parameter as the key and one of the previously mentioned types for the valuefor example, `:expects => [{:username => :string}]`. Providing a parameter name doesn't change the implementation of your web services, but it makes it possible for ActionWebService to generate a more descriptive WSDL file. And a more descriptive WSDL file may make it easier for other developers (or you) to write clients for the web service.

Although `api_method` can describe some very complex signatures, the underlying idea isn't that difficult. However, the `ActiveRecord::Base` and `ActionWebService::Struct types` may throw you for a little bit of a loop, so I'll expand a little bit on those.

If you use the `ActiveRecord::Base` type, AWS returns a record in the table as a complex web service type. Ruby's implementation of this type is a Hash in which the column names are the keys. So, if you have a table where the columns are `first_name`, `last_name`, `age`, and `birth_date`, `ActiveRecord::Base` would treat rows in this table as if you had listed them explicitly. For example:

```
[{:first_name => :string},
 {:last_name => :string},
 {:age => :int},
 {:birth_date => :date}].
```

You can use the `ActionWebService::Struct` type in much the same way; it's also an array of one-item Hash data types in which the member names are the keys. You would use it in situations in which you don't have a database in back of the web service but want the service to return an arbitrary object from your application much like a database would. The `member` method defines members of the `Struct`:

```
# in our API, we can now say :expects =>
[Greeting]
class Greeting < ActionWebService::Struct
 member :username, :string
 member :password, :string
 member :logtime, :datetime
end
```

The first parameter of the `member` method serves two purposes: it defines a read/write attribute of the `Greeting` class, which you use to store or retrieve data from a `Greeting` object (just like any other attribute), and it is used to build a more descriptive WSDL file for clients of your web service. The second argument is the Web Services data type used to transfer the attribute from client to server, or vice versa. Again, the `Struct` type is useful when you have a large or complex set of data, much like you would find in a database table but without the database.

`ActiveRecord::Base` and `ActionWebService::Struct` are simplified ways to define complex data types for your web service (there's no reason you can't do everything yourself, by populating an array of Hash types with your data). They save you from having to deal with the details of defining the Hash and then mapping your data into it. In the next example, we'll use an `ActionWebService::Struct` type to show how easy they really are to use in code.

3.3. Putting It All Together: an Expanded Server Example

Now that we've covered the various dispatching modes and discussed how to define a web service API w appropriate signatures, we're ready to dive into a more realistic example. We'll develop the basics of a foo statistics web service, starting with two methods: `Listgames` and `Getgamestats`. The first method requires an integer parameter that represents a year; it returns a list of games in that year for which statistics are available. The second method requires two parameters, both strings: a `username` and `gamename`. If the usern is valid, this method returns the actual statistics for a specific game.

We'll start by defining the web service API in *app/apis/stats_api.rb*:

```
class StatsApi < ActionWebService::API::Base
  api_method :listgames,
     :expects => [{:year => :int}],
     :returns => [[:string]]

  api_method :getgamestats,
     :expects => [{:username => :string}, {:gamename => :string}],
     :returns => [[Footballstats]]
end
```

The `StatsApi` class reflects the service's design requirements. It defines two methods. The first, `listgames`, requires an integer argument; it returns an array of strings to our clients. The second requires two string arguments and returns an array of `Footballstats`, which we will define as an `ActionWebService::Struct` type

The next step is to set up a controller in the file *app/controllers/stats_controller.rb*:

```
class StatsController < ApplicationController
   wsdl_service_name 'Stats'
   wsdl_namespace 'urn:sportsxml'
   web_service_dispatching_mode :delegated

 web_service :footballstats, Getstats.new

 web_service_scaffold :test

 def about
    # information about our service
    # accessed through reg. web browser
    # http://localhost:3000/stats/about
  end
end
```

The controller sets up a name for the web service by calling `wsdl_service_name 'Stats'`, and defines a namespace by calling `wsdl_namespace 'urn:sportsxml'`, so that the automatically generated WSDL file has useful and unique values that our clients can quickly understand. We also set the dispatching mode to `:delegated` so we can use distinct endpoint URLs for each method and can have the flexibility to mix in oth web services or APIs in the future.

Because we are not using the `:direct` dispatching mode, we are required to use the `web_service` method t specify the web service models that this controller exposes to the clients. The first parameter of the `web_service` method, `:footballstats`, is a symbol used to reference the web service. This symbol is used a part of the endpoint URI for XML-RPC and some SOAP calls. The second parameter is an instance of the model itself: `Getstats.new`, in this case. (We'll define the `Getstats` class shortly). Rather than providing an instance of the model, you can reference the model in a block and pass that block to the `web_service` meth `web_service :footballstats {Getstats.new }`. If you use the block form, the Model is instantiated at runtime has access to the instance variables and methodsincluding helper methodsof the controller.

There are two additional things we do in our controller. First, we define a basic method `about` that we inter use to provide documentation and other information to users via our web site. Second, we include a call t `web_service_scaffold:test`, which allows us to do some quick additional testing of our service via a web browser at the address *http://localhost:3000/stats/test*. Using the built-in scaffold feature gives us some q feedback about our services, but it should not be considered a complete test. It's still highly recommende that you take advantage of the functional tests Rails supports, as well as the additional client code tests mentioned at the end of this section.

Next, we define the `Getstats` class, saved as *app/models/getstats.rb*:

```
class Getstats < ActionWebService::Base
     web_service_api StatsApi
     before_invocation :checkusername, :except => [:listgames]

 def getgamestats(username, gamename)
     # get the stats for a specific game
     stats = []
     statdetails = Footballgames.find_by_sql(
         ["select statfor, as stattype, statvalue, statlogged from gametimesta
         where gamename = ?",gamename])
     statdetails.each do |rec|
         stats << Footballstats.new(:statfor => rec.statfor, :stattype =>
rec.stattype,
             :statvalue => rec.statvalue, :statlogged => rec.statlogged)
      end
     return stats
  end

   def listgames(year)
       # get a list of football games stats are available for
       Footballgames.find_by_sql(["select distinct gamename from gametimestat
        where gametimestats_year = ?", year]).map {|rec| rec.gamename}
       end

    protected
      def checkusername(name, args)
        if (args[0] != "kevin")
            raise "Access denied!"
         end
      end
   end
```

The model implements the web service's business logic. It's where you'll spend most of your developmen effort, so let's take a little extra time to explain everything that's going on here.

The first line, `web_service_api StatsApi`, starts by associating the model with the `StatsApi`. Remember that one API can be associated with a class, so every method in the model that we plan to expose through the web service must have a matching `api_method` definition in the `StatsApi` class.

Next, apply some access control by calling `before_invocation:checkusername, :except => [:listgames`. This method arranges for the `checkusername` method to be called before any other method in this class is called (except for the `listgames` method). If the call to `checkusername` returns anything other than the value `"true"` (which is the default return value for any Ruby method that has no return value), the web service is not invoked, and an error message is returned to the client. Thus, we're using `before_invocation` to restrict acc to all our methods except `listgames`.

Like the Rails ActionPack, AWS supports both a `before_invocation` and `after_invocation` method. Each of these methods can call a symbol referring to a method in the controller (`before_invocation :checkusername`) block of code (`before_invocation {|obj, meth, args| false}`), or an object referring to a model (`before_invocation Checkdata`). If you pass an object as the parameter, the object is expected to have an `intercept()` method that is automatically called upon instantiation. `before_invocation` provides you with the name of the method that was called, as well as the parameters that were passed. Instance methods calle `before_invocation` should expect two parameters: the name of the method that's being called and an array consisting of the method call's parameters. Blocks and objects are passed three parameters: the object containing the web service method, the method name, and an array of parameters.

Similarly, `after_invocation` provides you with the method that was called, the parameters sent in, and the results from the web service call. Instance methods being called via the `after_invocation` method should expect three parameters: the method name, an array of the parameters sent with the method request, an method return value. Blocks and objects are passed four parameters: the object containing the web servi method, the method name, an array of parameters sent with the method request, and the method's returr value.

Our model uses `before_invocation` to insert a call to `checkusername` before method invocations. Our implementation of `checkusername` is extremely simple: it tests against a hardcoded value. If your name is `"kevin"`, it lets you in. In a real implementation, you would check against a database and possibly use son encryption procedures. Remember that `before_invocation` calls are passed the method name and an array consisting of the parameters that were sent with the method call. Since we know the first parameter of `getgamestats` method is the username, we check against the value of `args[0]`. There's a consequence to th design. In the future, we may add methods to the web service, and those methods may require validation `checkusername`. We'll therefore have to make sure that any future method that needs validation has a userr as its first parameter.

Finally, we're ready to discuss the actual web service methods. Our API specified two methods, `getgamest` and `listgames`. `getgamestats` is supposed to return an array of `ActionWebService::Struct` types, with data fro actual game. So it performs a simple database query and uses the results to build an array of `Footballsta` which we will define as an `ActionWebService::Struct` subclass. (We'll see that definition shortly.) Our secor method, `listgames`, searches for a list of game names that were played in a given year. We use the `find_by_sql` method of `ActiveRecord` to query the database and then collect the results of that query into ar array of strings, which is what our API expects `listgames` to return.

Note: In a real web service, it's more likely that you would implement `getgamestats` using the ActiveRecord model with the `gamename` parameter to get the statistics from a given game. To

show `ActionWebService::Struct` in action, I chose to `find_by_sql` to query the database, a
then store the results into an array of Struct values.

We're almost done building our second web service. But before we're done, we have to define a couple o
data models. The `Footballgames` class is an ActiveRecord model we'll use to query the database for the ga
for which we have statistics available. In our examples, we used the `find_by_sql` method to query the
database, so we are really just treating `Footballgames` as a generic ActiveRecord class to establish a
connection to our database. `Footballstats.rb` is an `ActionWebService::Struct` model that we'll use to hold th
actual statistics for a game. Remember that `ActionWebService::Struct` is really just another way of defining
Hash that we intend to use a lot; since we could potentially have hundreds or even thousands of stats per
game, using a Struct saves us a lot of typing and hopefully makes our code easier to follow. We save both
these files in the models directory:

```
class Footballgames < ActiveRecord::Base
end
class Footballstats < ActionWebService::Struct
  # our football stats struct model
  member :statfor, :string
  member :stattype, :string
  member :statvalue, :string
  member :statlogged, :datetime
  member :statnote, :string
end
```

Finally, our web service is complete and ready for clients. SOAP clients can use the automatically genera
WSDL at *http://localhost:3000/stats/wsdl*, or the URI of *http://localhost:3000/stats/footballstats* and the
namespace of *urn:sportsxml*. XML-RPC clients can use the URI *http://localhost:3000/stats/footballstats*.

As with any code you write, it's important to test before you release. If you've used the Rails generator sc
to create the files above, you already have the outline of some functional tests available. If not, you can
manually create the file using our previous functional test example as an outline. Keep in mind that you u
different `invoke` command based on your dispatching mode: `invoke` for `:direct`, `invoke_delegated` for `:deleg`
and `invoke_layered` for `:layered`. Additionally, the following code snippets can be used to test your service
real clients. Simply save each as a local Ruby program and run them at the command line:

```
# XML-RPC client
require 'xmlrpc/client'
server = XMLRPC::Client.new2("http://localhost:3000/stats/footballstats")
result = server.call("Listgames", 2004)
puts result

# SOAP client using WSDL
require 'soap/wsdlDriver'
driver =
SOAP::WSDLDriverFactory.new("http://localhost:3000/stats/wsdl").create_rpc_dr
results = driver.getgamestats("kevin","GB@DET")
results.each do |rec|
  puts rec["statfor"]
  puts rec["stattype"]
  puts rec["statvalue"]
  puts rec["statlogged"]
```

```
  puts rec["statnote"]
end
```

This is a good point to look at the difference between `:delegated` and `:layered` dispatching. As I said earlie
there's no difference in the code, aside from the symbol passed to the `web_service_dispatching_mode` metho
(and the invoke command you'll use in your functional tests). Had we chosen `:layered` dispatching, XML-R
clients would use the URI *http:///localhost:3000/stats/api*, and would reference our methods as
`footballstats.Listyears` and `footballstats.Getgamestats`.

SOAP clients not wishing to use WSDL would also use *http:///localhost:3000/stats/api* as the endpoint UR

3.4. Summary

If you can believe it, in just a few short pages and with very little code, we've now built working REST, SOAP, and XML-RPC servers! I think the examples showed that building web service servers with Ruby on Rails is a pretty straightforward process. Most of the work building your servers with Ruby on Rails will be in designing the features you need to meet your specific needs and requirements. Implementing the code should be a pretty easy task as long as you follow and understand the simple steps I outlined: pick a delegation mode, create an API, and code your business logic in models and controllers.

I moved pretty fast through the examples, which were all simple, with very limited design specifications. Real-world web services would involve a lot more in terms of security checks and application features, which I will leave up to you to determine and implement. Still, we covered everything you need to know to build full-featured, real-world clients and servers with Ruby on Rails. Now all you have to do is go out and build them!

3.5. Additional Reading and Help

If you're looking for more help with Ruby on Rails, I recommend the following web sites, all of which were at least partially used in compiling information for this book:

http://www.ruby-forum.com

A great source to get quick answers from real programmers

http://www.rubyonrails.org

The official Rails site, full of useful HOWTOs

http://www.ruby-lang.org

The official Ruby site; useful for keeping up-to-date on Ruby events

http://www.rubycentral.com/ref

Online reference documents for Ruby

http://www.rorbe.com

The companion site to this document, with additional examples and updates

There are also many good Ruby and Ruby on Rails books in print today that cover the basics of the Ruby syntax and Rails framework, both essential to building quality web services with Ruby on Rails. Additionally, there are a number of good books covering web service concepts and technologies, including REST, XML RPC, and SOAP. All of them are worth at least browsing through if you're just getting started with web services.

The code examples from this book are also available online at http://www.rorbe.com. I will also put additional examples, updates, and any other information that comes about from the readers of this book on http://www.rorbe.com.

If you build a publicly available web service with Ruby on Rails or want to share your Rails web service client code, with others please submit the details to http://www.rorbe.com. Additionally, if you have questions or comments about anything you've read here or need help getting past a specific problem with your Rails web service, please feel free to email me directly at kevinm@rorbe.com.