

Automate data entry with Web services and Ajax

Deploy Web 2.0 technologies to save time and ensure data accuracy

Skill Level: Advanced

Norbert (Norb) R. Ryan, III (nryan@us.ibm.com)
Software Engineer
IBM

08 Feb 2008

Let's cut through the chatter and find out how a Web service and Asynchronous JavaScript + XML (Ajax) can improve an application, in this case a Ruby on Rails (RoR) application. This article shows you how to spruce up a common Web activity—entering a street address—with Ajax and a call to a Web service. Learn a few tricks to combining these fundamental Web 2.0 components.

A little background on the idea

The United States Postal Service (USPS) has made several Web services available (see the [USPS Web tools sidebar](#)). One of those Web services accepts a ZIP code and returns the corresponding city and state. In this example application, you make use of this `CityStateLookupRequest` to save the user some typing. This feature also gives you better address data in your database, because you reduce the chances of typing errors.

Prerequisites and assumptions

David Heinemeier Hansson, the ideological spirit and creator of Ruby on Rails, is a smart guy! In RoR, he implemented a number of great ideas, ideas that make developing Web applications much easier and, as a friend of mine remarked, "It makes programming fun again!" There's little doubt in my mind that other

frameworks and programming paradigms will espouse these ideas. However, this isn't a tutorial on how to create an RoR application. (See the [Resources](#) section at the end of this article for links to good tutorials and reference information.)

Check out the [Ajax Resource Center](#), your one-stop shop for information on the Ajax programming model, including articles and tutorials, discussion forums, blogs, wikis, events, and news. If it's happening, it's covered here.

The assumption here is that you've created an RoR application that has an HTML input form for an address (for example, 590 Madison Ave, New York, NY 10022). This Rails application also has a model named *address* and a corresponding database table. Furthermore, let's assume that you:

- Understand basic design principles of Web application development.
- Have created an RoR application.
- Understand the basic parts of an RoR application: ActiveSupport, ActiveRecord, ActionView, ActionController, Migrations, and so on.
- Have a database (like IBM® DB2® or MySQL) configured to work with the RoR application.
- Connect with users by anticipating their needs and know the importance of saving them time.

Table 1. Assuming you have an RoR application with these objects

Ruby on Rails file	Directory	Description
edit.rhtml	../app/views/addressadmin	View for editing an address
_form.rhtml	../app/views/addressadmin	Partial used by edit.rhtml
addressadmin_controller.rb	../app/controllers	Controller called by the HTML input form
address.rb	../app/models	ActiveRecord object
001_create_addresses.rb	../db/migrate	Script for creating the <i>Addresses</i> database table

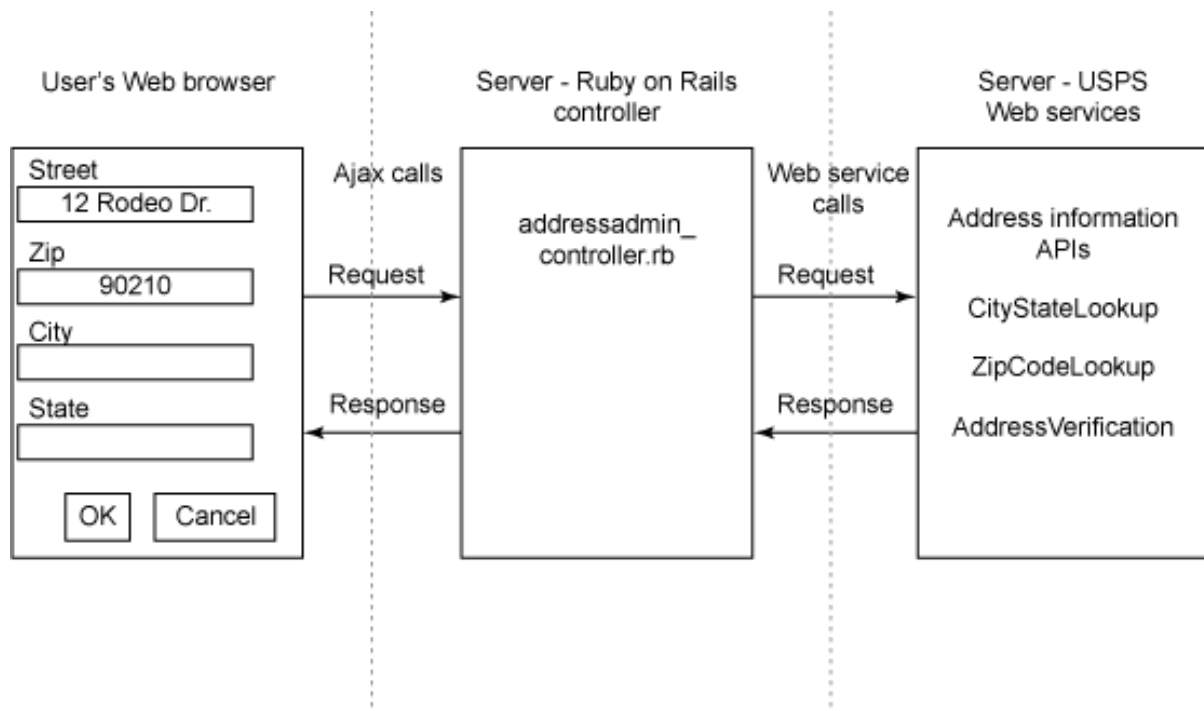
Solution overview

The list below shows the steps involved in completing this solution. (Don't worry; the remainder of this article walks you through these steps one by one.) Note that a *partial* is a Ruby on Rails term. It's a reusable piece of code related to what gets displayed in the Web browser. Most modern frameworks include some kind of template and partial functionality that dynamically assembles pieces of the template to produce a Web page. Partials are a real convenience to application developers

and drastically help reduce the burden of development. The RoR naming convention is to prefix a partial with an underscore (for example, `_addressForm.rhtml`).

1. Modify the `_form.rhtml` partial to display the ZIP code before the city and state.
2. Add a partial (`_cityState.rhtml`) to display the input fields for city and state.
3. Modify the `_form.rhtml` partial to "listen" for changes to the ZIP code field and make an Ajax call to the server.
4. Modify the controller to validate the ZIP code (5 numeric digits). If not valid, return a blank Ajax response to the client.
5. Modify the controller to create a valid XML request to send to the USPS Web service.
6. Modify the controller to receive and parse a XML response from the USPS Web service.
7. Modify the Ajax response to populate the `_cityState` partial with the Web services' values.
8. Figure out some ways to improve the solution, and e-mail the author with your suggestions.

Figure 1. Solution overview



Facts and trivia about postal codes around the world:

- Eleven countries use alphanumeric postal code systems, including the UK (mind those data types!).
- Ireland and Hong Kong don't use postal codes.
- India calls them Postal Index Numbers, or PINs.
- In Paris, Lyon, and Marseilles, the last two digits of the *code postal* indicate the *arrondissement*.
- Surprisingly, the small island of Singapore uses six digits in their postal codes, but the postal code is usually sufficient to identify a building or a condominium.
- Bancroft Hall, the midshipman dormitory at the United States Naval Academy, is one of the few buildings with its own ZIP code—21412.
- I was born in 78418 (you can't be made a Texan, you can only be born a Texan!) and currently live in 10028.
- The World Trade Center towers' exclusive ZIP code was 10048.

Step 1: Modify the view

The usability of this address form may be called into question. In the U.S., it's normal for the address input fields to be listed in the order of:

1. Street
2. City
3. State
4. ZIP

But the form here lists the fields in this order:

1. Street
2. **ZIP**
3. City
4. State

Let's assume that this usability issue can be overcome by user training, instructional text, or common sense. It might also make sense to darken the city and state fields or to prevent entry in them all together. Work with your usability experts to craft an acceptable solution.

Listing 1 shows the input form named `_form.rhtml` and the input text fields. Notice that the input text field `zip5` has been moved above city and state. The last line, `debug(params)`, is optional. During the development and test phases, I usually include this debug data in my RoR views.

Listing 1. Partial with order of input fields changed (`_form.rhtml`)

```
<%= error_messages_for 'address' %>

<p><label for="address_street">Street</label><br/>
<%= text_field 'address', 'street' %></p>

<p><label for="address_zip5">Zip5</label><br/>
<%= text_field 'address', 'zip5', :size => "9",
:maxLength => "5" %></p>

<p><label for="address_city">City</label><br/>
<%= text_field 'address', 'city' %></p>

<p><label for="address_state">State</label><br/>
<%= text_field 'address', 'state' %></p>

<%= debug(params) %>
```

Step 2: Add a Rails partial

The second step in the solution is to break up the `_form.rhtml` input form by separating the city and state input fields into a new partial. The RoR naming convention is to prefix an underscore to partials. Therefore, the name of the new partial is `_cityState.rhtml`. The new file resides in the same directory as `_form.rhtml`. Listing 2 shows the code for the new file, `_cityState.rhtml`.

Listing 2. New RoR partial (`_cityState.rhtml`)

```
<p><label for="address_city">City</label><br/>
<%= text_field 'address', 'city' %></p>

<p><label for="address_state">State</label><br/>
<%= text_field 'address', 'state' %></p>
```

It would be nice to be able to leave the city and state in the same file as the other address fields. I tried to get it to work but was only able to make it work this way. Why? The difficulty has to do with updating multiple form fields with the response from the Ajax call. Either my experience level with RoR is insufficient, or the generated JavaScript code can't handle it. Most likely the former. Listing 3 shows the code for the `_form.rhtml` partial after removing the city and state input fields. Note that the new code is given an `id = "ajaxLookup"`; this is explained in the [next step](#).

Listing 3. Including the new partial (`_form.rhtml`)

```
<%= error_messages_for 'address' %>

<p><label for="address_street">Street</label><br/>
<%= text_field 'address', 'street' %></p>

<p><label for="address_zip5">Zip5</label><br/>
<%= text_field 'address', 'zip5', :size => "9",
:maxLength => "5" %></p>

<div id = "ajaxLookup">
  <%= render :partial => "cityStateFields" %>
</div>
```

The naming convention is an important idea in RoR. RoR assumes that partials are named with a leading underscore. And RoR assumes that references to that partial won't contain an underscore. So in Listing 3, it's expected that the line won't have an underscore: `<%= render :partial => "cityStateFields" %>` is correct. RoR sees this line and looks in the same directory for a file named `_cityStateFields.rhtml`.

Step 3: Listen for changes to the ZIP code

Rails has built-in support for Ajax. This is one of the areas where Rails really shines. Simply add the lines of code shown in Listing 4 to listen for changes to the ZIP code field.

Listing 4. Add an Ajax Listener to ZIP code (_form.rhtml)

```

01 <%= javascript_include_tag :defaults %>
02
03 <p><label for="address_street">Street</label><br/>
04 <%= text_field 'address', 'street' %></p>
05
06 <p><label for="address_zip5">Zip5</label><br/>
07 <%= text_field 'address', 'zip5', :size => "9",
08 :maxlength => "5" %></p>
09 <div id = "ajaxLookup">
10 <%= render :partial => "cityStateFields" %>
11 </div>
12
13 <%= observe_field :address_zip5,
14       :frequency => 2.00,
15       :update     => "ajaxLookup",
16       :url        => { :action =>
17 :cityStateSearch, :id => @address},
18       :with       => "'zip5=' +
19 encodeURIComponent(value)"
20 %>
21 <%= debug(params) %>

```

Note: The two-digit line numbers on the very left edge of the listing are for explanation purposes; they don't appear in the code.

That's it! In about 10 lines of Ruby code, this view has been embellished with Ajax functionality. Behind the scenes, RoR and the prototype library handles all the JavaScript. Let's go through these 20 lines of code.

Line 01 instructs RoR to include the Prototype and Scriptaculous JavaScript libraries. Lines 03-12 are the same as before. Line 13 uses the `observe_field` method from the Prototype library. `observe_field` is a helper method in the `PrototypeHelper` class. In plain language, lines 13-17 say to check the zip5 input field every two seconds. If the zip5 input field has user input, then call the action `cityStateSearch` in the current controller, which is `addressadmin_controller.rb`. This logic is being run by JavaScript within the user's browser. When the zip5 input field is changed, an Ajax call is made from the user's browser to the server. Notice the correlation between line 09 and line 15: Line 15 identifies what to do with the response from the action `cityStateSearch`. The response of the action, if any, updates the `<div>` tag named `ajaxLookup`. Line 09 has the `div` tag with the ID equal to `ajaxLookup`. So the response from the action `cityStateSearch` is passed into line 10. Line 17 explains what name and value pair to send to the action. So in this example, the string `zip5=90210` is passed to `addressadmin_controller`'s action named `cityStateSearch`.

Next, you begin work on the controller functionality. The last step specified that the user's browser would make an asynchronous call to the action named `cityStateSearch` when it detected a change in the ZIP code (that is, zip5) input field. Here are the main pieces of functionality that you have to code on the server

side:

- Validate the ZIP code.
- Build the XML to call the Web service.
- Call the Web service.
- Parse the response from the Web service.
- Send the response back to the user's Web browser.

Step 4: Validate the ZIP code

There's no sense calling the USPS Web service with a invalid ZIP code. With a little effort you can eliminate most invalid ZIP codes. In the U.S., the ZIP code is five numbers. Listing 5 shows some code that checks to see if the zip5 parameter consists of five numbers.

Listing 5. Validate the ZIP code (addressadmin_controller.rb)

```
01 def cityStateSearch
02
03   if params[:zip5].nil?
04     logger.debug("zip5 is null")
05   elsif !(params[:zip5] =~ /\d{5}/)
06     logger.debug("We have a bad ZIP code -- not 5
digits.")
07     logger.debug("zip5 = #{params[:zip5]}")
08   else
09     logger.debug("We have a good 5-digit ZIP
code.")
10     logger.debug("zip5 = #{params[:zip5]}")
11
12     if params[:address].nil?
13       @address = Address.new
14     else
15       @address = Address.find(params[:id])
16       if
@address.update_attributes(params[:address])
17         flash[:notice] = 'Address was successfully
updated.'
18       end
19     end
20   end
21
22 end #cityStateSearch
```

USPS Web Tools

The United States Postal Service offers five different types of Web services:

1. Address Information
2. Delivery Information

3. Rate Calculators
4. Shipping Labels
5. Carrier Pickup

This article focuses on the first one, Address Information APIs. Within this API there are a few different functions:

- **Address Standardization:** Eliminates addressing errors and helps ensure accurate and timely delivery. This Web Tool corrects errors in street addresses, including abbreviations and missing information. It also supplies a ZIP+4 code.
- **ZIP Code Lookup:** Finds matching ZIP codes or ZIP+4 codes for any given address, city, and state in the U.S.
- **City/State Lookup:** Provides accurate city and state information when you only have the ZIP code. (**Note:** Above information taken from USPS Web Tools documentation.)

Note: The USPS does not allow the use of these Web tools in batch or database cleansing activities.

Line 01 starts the definition for the new action `cityStateSearch`. Unlike other actions in this controller, the `cityStateSearch` action is called asynchronously by the JavaScript—the client-side Ajax code. Line 03 checks to see if the parameter value is null (or nil, as Ruby calls it). Line 05 is a regular expression that compares the parameter string value against `/\d{5}/`, which we all know and love as the regular expression for five digits. The exclamation point before the expression negates the `elsif` expression. (Yes, that's correct RoR syntax for what is otherwise known as *else if*.)

Lines 12-19 handle creating or updating the `@address` object. There's a little subtlety in lines 5-7. When the logic falls into these lines, it drops out of the action and returns to the browser. The end user is none the wiser. This action skips the rest of the logic and returns right away when the `zip5` is nil or is not 5 digits. As long as the user's cursor is in the `zip5` input text field on the view, then this action gets reinvoked every two seconds. That functionality was configured earlier in the `observe_field` method in the `frequency` parameter.

Line 21 is where you add the next section of code. Because this is development code, I use lots of debug statements. After the code has been polished and refined several times, I'll remove the `logger.debug` statements. Also, I'm an RoR newbie, so lots of debug statements give comfort to my style of programming, which has been accurately described elsewhere as "beat it into shape."

Step 5: Create a valid XML request to send to the USPS Web service

At this point in the process, you're in the server side of your RoR application, and you have a ZIP code with five digits. Because you have a reasonable expectation that the ZIP code is legitimate, it's now worth the effort to call the USPS Web service. To do this, you need to create a valid request. Jumping ahead a little, Listing 6 shows an example of a valid XML request.

Listing 6. Valid XML request

```
http://testing.shippingapis.com/ShippingAPITest.dll?API=CityStateLookup
&XML=<CityStateLookupRequest%20USERID="XXXXXXXXXXXX"><ZipCode
ID=
"0"><Zip5>90210</Zip5></ZipCode></CityStateLookupRequest>
```

To use the USPS Web Tools, you have to register with them. Registration is easy and free (see the [Resources](#) section for more information). After I registered for the USPS Web Tools, they sent me the name of the test server and a user ID. (In Listing 6 I did my best impersonation of a top-secret government agent by crossing out my user ID with XXXXXXXXXXXX.) Let's parse this request a little more. The Web service endpoint is specified by `API=CityStateLookup`. In the HTML form, you can now see why I called the input field `zip5`. That's the name that the USPS request expects. This `CityStateLookup` Web service accepts up to five ZIP code values in one request. To keep things simple, this code only passes one ZIP code, which has the XML tag `<ZipCode ID= "0">`. So the functionality becomes quite obvious: You need to get the five-digit value entered by the user and put it into this XML tag named `<Zip5>`.

So what kind of Web service is this?

It turns out that this seemingly simple question isn't always easy to answer. Web services have become like Baskin Robbins' 31 flavors of ice cream or the Starbucks coffee menu. It seems like ordering a cup of coffee should be easy to do until they hit you with something called a Grande Chai Latte with Soy. First, what it's not: This isn't a XML-RPC-style Web service, document-style Web service, SOAP Web service, or Representational State Transfer (REST) (noun-based) request Web service. The designers at the USPS have decided to implement this as a plain vanilla XML Web service. The USPS Web servers accept either GET or POST HTTP requests. The requests are stateless with no cookies or URL rewrites. Requests and responses are case sensitive. Once again, it's easy to register with USPS Web Tools, and there's plenty of documentation. (I should note that I have no affiliation with the USPS.)

To create the XML, you use the `Builder::XmlMarkup` library, which is included with

RoR. At the beginning of the controller class file `addressadmin_controller.rb`, you need to add the code shown in Listing 7.

Listing 7. Code to be added to `addressadmin_controller.rb`

```
require 'open-uri'
require 'uri'
require 'rubygems'
require_gem 'builder'
require "rexml/document"
```

Listing 8. Create the XML portion of the request

```
01 def cityStateSearch
02
03   if params[:zip5].nil?
04     logger.debug("zip5 is null")
05   elsif !(params[:zip5] =~ /\d{5}/)
06     logger.debug("We have a bad ZIP code -- not 5
07 digits.")
07     logger.debug("zip5 = #{params[:zip5]}")
08   else
09     logger.debug("We have a good 5-digit ZIP
10 code.")
10     logger.debug("zip5 = #{params[:zip5]}")
11     # Build the XML to call the web service
12     xm = Builder::XmlMarkup.new
13     xmlstuff =
14 xm.CityStateLookupRequest("USERID"=>"XXXXXXXXXXXXX") {
15     xm.ZipCode("ID"=>"0") {
16     xm.Zip5(params[:zip5]) }}
16
17   end
18 end #cityStateSearch
```

Just four lines, lines 12 through 15, create the properly formatted XML for the request. The string variable `xmlstuff` contains this XML:

```
<CityStateLookupRequest%20USERID="XXXXXXXXXXXXX"><ZipCode ID=
"0"><Zip5>90210</Zip5></ZipCode></CityStateLookupRequest>
```

There are a couple of more important steps to getting this request properly formatted. You need to escape the special characters of the request with the two lines of code in Listing 9.

Listing 9. Escape the request's special characters

```
uri_enc =
URI.escape('http://testing.shippingapis.com/ShippingAPITest.dll
?API=CityStateLookup&XML=' + xmlstuff)
uri = URI.parse(uri_enc)
```

These lines take care of the conversion of all the special characters into proper encoding as an HTTP request. I'll leave it up to you to improve this code by setting up variables or property files for the server name, API name, and so on. Your goal is just to get the call to the Web service working; you can polish and refine later. Now you have a properly formatted HTTP/XML request and are ready to invoke the

USPS Web service.

Step 6: Call the Web service and receive a response

In this step, you call the USPS Web service and receive a response. The code must parse the XML response. Listing 10 shows a sample of a USPS `CityStateLookup` response.

Listing 10. USPS `CityStateLookup` response

```
<?xml version="1.0"?>
<CityStateLookupResponse><ZipCode
ID="0"><Zip5>90210</Zip5>
<City>BEVERLY HILLS</City><State>CA</State></ZipCode>
</CityStateLookupResponse>
```

Again, I took this example directly from the USPS Web Tools documentation. Your goal in this step is to parse the city and state information and place it in your `@address` object's variables. Eventually those variables are returned as a part of the Ajax response.

Keep this in mind: the Builder library allows for creation of the XML, while the module REXML enables parsing the XML data.

Listing 11. Call the Web Service and parse the response (`addressadmin_controller.rb`)

```
    # The call to the Web service -- response is in
    var 'doc'
    doc = REXML::Document.new open(uri)
    logger.debug("doc = " + doc.to_s)
    doc.elements.each("CityStateLookupResponse/ZipCode") {
    |element|
        logger.debug(element)
        logger.debug("element[0] = " +
        element[0].to_s)
        logger.debug("element[0].text = " +
        element[0].text)
        logger.debug("element[1] = " +
        element[1].to_s)
        logger.debug("element[1].text = " +
        element[1].text)
        logger.debug("element[2] = " +
        element[2].to_s)
        logger.debug("element[2].text = " +
        element[2].text)
        # Set the model field values to the response
        from the Web service
        @address.city = element[1].text
        @address.state = element[2].text
    }
}
```

The code in Listing 11 iterates through the XML response to find the city

`element[1]`) and the state (`element[2]`). `element[0]` is the zip5 value. As mentioned earlier, the Web service processes up to five ZIP code lookups in one request. A future refinement of this code should consider looping through those values. But in this simple example, you're always only passing in one ZIP code value per request. This code could also do a better job of handling no city/state response values. The point here is to get minimal function working so you can refine it per the constraints or demands of your particular project. If you have questions about the use of the Builder library or the REXML module, see the RoR API documentation., which has plenty of examples.

If you're familiar with the way other languages create or parse XML, you'll immediately recognize how easy it is in RoR. One of the messages that RoR is quietly shouting is, *It doesn't have to be a huge pain in the butt!* XML creation—no sweat. Ajax—with one hand behind my back. XML parsing—easy peasy. RoR frequently reminds me that the objective is to get real work done, real quick, for real end users. It's a nice feeling.

Recapping the action, you validated the ZIP code, created the XML request to call the Web service, parsed the XML response, and put the city and state fields in the `@address` object. From the point of view of the user, all of this processing has been done asynchronously and in about one second. The user's cursor is still in the ZIP code field on the html form of the Web browser. Before they know it, this action is returning the `@address` object with the correct city and state values. And the information is sent to the partial `_cityStateFields.rhtml` and populated in the user's browser. Hopefully it's a good surprise to the user—and you've given the feeling that you're on their side and are here to help them out. You've anticipated their needs and done your best to make their lives a little easier.

Listing 12 shows the full code for the `cityStateLookup` action in the file `addressadmin_controller.rb`.

Listing 12. Full code for action `cityStateLookup` (`addressadmin_controller.rb`)

```
require 'open-uri'
require 'uri'
require 'rubygems'
require_gem 'builder'
require "rexml/document"

class AddressAdminController < ApplicationController

  <!-- other methods/actions -->

  def cityStateSearch

    if params[:zip5].nil?
      logger.debug("zip5 is null")
    elsif !(params[:zip5] =~ /\d{5}/)
      logger.debug("We have a bad ZIP code -- not 5
digits.")
      logger.debug("zip5 = #{params[:zip5]}")
    else
      logger.debug("We have a good 5-digit ZIP code.")
    end
  end
end
```

```

    logger.debug("zip5 = #{params[:zip5]}")

    if params[:address].nil?
      @address = Address.new
    else
      @address = Address.find(params[:id])
    end
    if
      @address.update_attributes(params[:address])
      flash[:notice] = 'Address was successfully
updated.'
    end
  end

  # Build the XML to call the Web service
  xm = Builder::XmlMarkup.new
  xmlstuff =
xm.CityStateLookupRequest("USERID"=>"XXXXXXXXXXXX") {
  xm.ZipCode("ID"=>"0") {
  xm.Zip5(params[:zip5]) }}

  webservice =
'http://testing.shippingapis.com/ShippingAPITest.dll?'
  uri_enc = URI.escape(webservice +
'API=CityStateLookup&XML=' + xmlstuff)
  uri = URI.parse(uri_enc)

  # The call to the Web service -- response is in
var 'doc'
  doc = REXML::Document.new open(uri)
  logger.debug("doc = " + doc.to_s)
doc.elements.each("CityStateLookupResponse/ZipCode") {
|element|
  #logger.debug(element.attributes["name"])
  logger.debug(element)
  logger.debug("element[0] = " +
element[0].to_s)
  logger.debug("element[0].text = " +
element[0].text)
  logger.debug("element[1] = " +
element[1].to_s)
  logger.debug("element[1].text = " +
element[1].text)
  logger.debug("element[2] = " +
element[2].to_s)
  logger.debug("element[2].text = " +
element[2].text)
  # Set the model field values to the response
from the web service
  @address.city = element[1].text
  @address.state = element[2].text
}
end # valid ZIP code if-statement-checkers
render :partial => "cityStateFields"
end

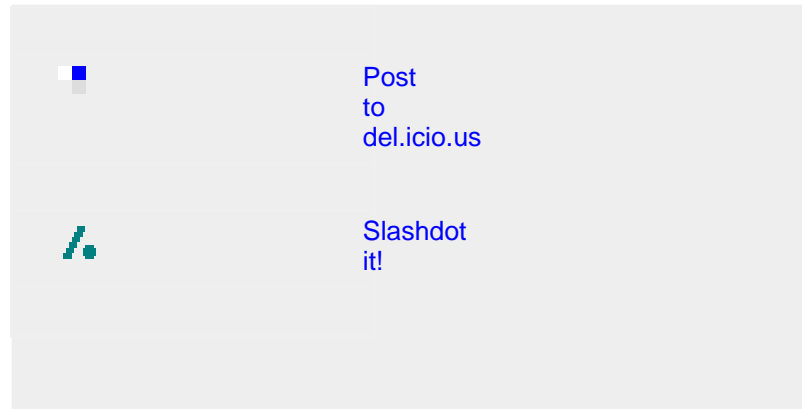
```

Miscellaneous thoughts

Share this...



Digg
this
story



Some miscellaneous comments on this solution and Web services:

- Web services work regardless of the programming language. This example used Ruby on Rails, but it also works with other languages and frameworks.
- This solution lacks a plan for what to do when the USPS Web service isn't available. Maybe a local cache can help minimize the impact of an outage.
- The parsing of the XML response from the Web service is probably the least elegant code written. Maybe a template can help this area of the solution.
- Why bother the user with entering a city and state when it's possible to look them up with just a ZIP code?
- How effective are these various *working groups* that bequeath Web service standards into the marketplace?
- Has anybody ever thought for one second about implementing the Universal Description Discovery Integration (UDDI) standard?
- Are Web service standards too complex? Can developers wade through all the acronyms?
- Rarely does the marketplace declare only one winner. Many competitors advance to the next round. But the marketplace seems quite efficient at weeding out the losers.
- How long until the naysayers question RoR's performance, security, or production worthiness?

The marketing muscle of many big companies is clouding the Web service waters, and this article addressed a common problem with a simple and easily understood Web service. The people at the United States Postal Service have implemented a

solid and very useful Web Service. Jakarta Struts was a vast improvement over previous efforts to address Web application frameworks—to address the whole model view controller (MVC) stack. Ruby on Rails is, at least, a significant improvement over Struts.

Downloads

Description	Name	Size	Download method
Sample application for this article	ajaxsoademo.zip	176KB	HTTP

[Information about download methods](#)

Resources

Learn

- Visit [developerWorks' Architecture area](#) for the resources you need to advance your skills in the architecture arena.
- Check out these wikipedia entries to learn more about [ZIP codes](#) and [postal codes](#).
- Get more information on the services offered by the [United States Postal Services Web Tools](#). The USPS APIs can only be used in conjunction with USPS shipping services.
- Check out the official [Ruby on Rails](#) site and the [API](#).
- The two books that got me started with Rails are *[Agile Web Development with Rails: Second Edition](#)* and *[Programming Ruby](#)*. My third book purchase, and a worthy addition to my Rails library, was *[Rails Recipes](#)*.
- The [SOA and Web services zone](#) on IBM developerWorks hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- Play in the [IBM SOA Sandbox!](#) Increase your SOA skills through practical, hands-on experience with the IBM SOA entry points.
- The [IBM SOA Web site](#) offers an overview of SOA and how IBM can help you get there.
- Stay current with [developerWorks technical events and webcasts](#).
- Browse for books on these and other technical topics at the [Safari bookstore](#).
- Check out a quick [Web services on demand demo](#).

Get products and technologies

- Innovate your next development project with [IBM trial software](#), available for download or on DVD.

Discuss

- [Participate in the discussion forum for this content](#).
- Get involved in the developerWorks community by participating in [developerWorks blogs](#), including the following SOA and Web services-related blogs:
 - [Service Oriented Architecture -- Off the Record](#) with Sandy Carter
 - [Best Practices in Service-Oriented Architecture](#) with Ali Arsanjani

- [WebSphere® SOA and J2EE in Practice](#) with Bobby Woolf
- [Building SOA applications with patterns](#) with Dr. Eoin Lane
- [Client Insights, Concerns and Perspectives on SOA](#) with Kerrie Holley
- [Service-Oriented Architecture and Business-Level Tooling](#) with Simon Johnston
- [SOA, ESB and Beyond](#) with Sanjay Bose

About the author

Norbert (Norb) R. Ryan, III

Since receiving a Commodore 64 in 1983, Norb has been fascinated with computers and software engineering. Since 1990, he has been implementing solutions for IBM.

Trademarks

DB2, IBM, the IBM logo, and WebSphere are registered trademarks of IBM in the United States, other countries or both.