Getting Started with Orbjson: Short Version

by James Britt, March 14, 2005

Overview

This document walks through the creation of a simple Orbjson application that lets a user search for Ruby libraries in a manner similar to Google Suggest.

Installation

The best way (and really, only, way, at the moment) to install Orbjson is via RubyGems:

```
% gem install orbjson
```

You should be prompted to also install two dependencies: *needle*, and *ruby-json*. They are required for Orbjson to work.

The Orbjson Architecture

An Orbjson application has three basic parts: a client (typically a Web browser executing JavaScript), one or more Ruby classes implementing some set of services, and an instance of Orbjson to act as the go-between. Here's how to works: The client sends a JSON-RPC message to the Orbjson server. JSON-RPC is a variation on XML-RPC: it's way to describe a remote method invocation using structured text as the wire format. JSON is the JavaScript Object Notation, and it allows you to describe native JavaScript data types as text.

All-in-all, JSON is really not all that complex, and you could hand-code your objects if you really needed to, but you don't. Typically, in any decent object marshaling system, both client and server have code to marshal and unmarshal JSON to and from objects automatically. There are a few GPL JavaScript libraries to handle this for the client, and one is included with the Orbjson distribution. On the server, Orbjson uses Florian Frank's Ruby-JSON library to turn JSON text into Ruby objects. Between these two you never really need to think about the low-level details; instead, you focus on the business logic.

Orbjson inspects the JSON request and dynamically invokes the desired method on the specific object. Now, the client can't just send arbitrary requests and hope something useful happens; Orbjson maintains a registry of client-side services using Jamis Buck's Needle library. You configure Orbjson by telling it what files to require and what classes to add to the registry; the current implementation then exposes all public methods on these objects as callable via JSON-RPC. There is a built-in method, listMethods, that returns to the client a list of available services.

When a request comes in, Orbjson tries to locate the desired object in the registry and dynamically invoke the specified method. If all has gone well, the results of that method

call are serialized to JSON and sent back down to the client as a JSON-RPC response message. The client then `eval`s the message to local, native JavaScript objects.

What's particularly handy about all this is that the business objects do not need to be concerned with JSON, object brokers, or anything of that sort. They need only expose a useful API. And the client-side business logic need know nothing of Ruby or object brokers either. As our example will show, the client code can be constructed so that it appears as if all events and request are occurring on local objects.

The Example

An example should help clarify things. The task is allow the user to search for Ruby libraries. Such a list is potentially huge; sending the entire list down to the browser would be overkill. Instead, the example will cop a page from Google Suggest and show a reduced selection of known libraries based on what the user types.

Since this is to be only an example, and not a real application, some things will be over-simplified. For example, the server code will load the names libraries from hard-code list, rather than pull them from database, as you might do in real life. Also, the client-side DHTML and layout is intended more for basic clarity than for any special aesthetic goodness.

The Server Code

When you install Orbjson you get a command-line application, `orbjson`, that will write out some skeletal code. Depending on what parameters you pass it will create either a bare-bones WEBrick application or some CGI files. This example will use the WEBrick version. You invoke this code-creator by typing `orbjson` along with the sort of application you want and the directory where the new application will be created:

```
% orbjson create-webrick ./lib-list
```

This will create a subdirectory, named `lib-list`, of off the current directory. You can also specify the full path for the new directory. (To see the options and basic help for the `orbjson` command-line script, call it with no arguments.)

There should now be `lib-list` subdirectory with a Ruby file named `server.rb`, a configuration file named `config.yml`, and a `/scripts` subdirectory with two JavaScript files: `jsonrpc.js` and `jsonrpc_async.js`.

Let's see what each of these files do, and what's needed to flesh out the application. To begin with, `server.rb` defines a very bare WEBrick server and mounts a few servlets. The server listens on port 2222; you may want to change that to suit your particular needs. Two trivial `Proc`-based servlets are attached to the relative URLs `/quit` and `/exit`. These simply shut down the server, and are handy when debugging your code. You should almost certainly delete these from any production application.

The main servlet is the `Orbjson::WEBrick_JSON_RPC` class, mounted on the relative URL `/json-rpc`. This defines the end point URL for the JSON-RPC client. Feel free to change this URL if you like, though this follows a fairly common naming convention, and is the end point used in this tutorial.

An Orbjson application must be told what classes to register and where to find them. This is done by giving Orbjson a hash that maps file paths to arrays of class names. There are three ways to get this hash to your Orbjson instance: Pass in the name of a YAML file; pass in a literal YAML string; or pass in an actual Hash object.

The auto-generated WEBrick code is set up to get configuration details from an external YAML file. The boilerplate CGI code uses an in-line YAML string. Since the WEBrick application only loads the external file once, at start up, the overhead in largely insignificant. The CGI application, though, will have to configure a new Orbjson object on each call; using in-line YAML or a literal hash saves the overhead of reading an an external file. Both WEBrick and CGI may use any of the three approaches, though, and if you are adapting your code for FastCGI then you may prefer to use an external file.

Fleshing Things Out

The default configuration file is useful only to show you the basic structure; you'll have to change it to point to the actual files and classes to use. We can do that now, even though the files haven't been created yet. Edit `config.yml` so that it looks like this:

```
services/lib-list:
   - Library
```

(If you are not familiar with writing YAML files, be advised that it uses significant indentation to indicate where things begin and end, so be sure that the list of class names are indented from the first column, and are indented by the same number of spaces.)

This configuration tells Orbjson to require 'services/lib-list', and to register the `Library` class. This is the business object we'll use to implement our library search service.

Of course, we need to create this file and this class; let's do that now. Create a `/lib-list/service` subdirectory, and in that directory create a file named `lib-list.rb`. This file needs to define our business object.

`Library` is responsible for returning a list of Ruby libraries that match on a given string. It also returns a description given a library name. The base list is hard-coded in a class variable; in real life this is the sort of data that one might grab from a database.

Here's the code:

```
class Library
  @@list =  {
     'Orbjson' => 'JSON-RPC object request broker',
```

```
    'Catapult' => 'Lightweight Web-services toolkit',
    'OOo4R' => 'OpenOffice.org document manipulation using pure
Ruby',
    'CounterWeight' => 'Web-request throttling library',
    'Blogtari' => "World's greatest eternally-beta blogging
software"
   }

  def match( str )
    @@list.keys.select { |lib| lib.downcase =~ /^#
{str.downcase}/ }.sort
  end

  def details( key )
    @@list[ key ] || ''
  end
end
```

Yes, it's quite simple. It may not quite be how you might do it in real-life, but it serves the needs of our example.

The Client

Since we'll be employing the magic of JSON-RPC messaging we need just one Web page. Let's call it `index.html`. Orbjson provides the core JavaScript need to exchange messages with the server; you just need to invoke it and work with the resulting objects. This example will use asynchronous messaging; if you prefer to use the synchronous messaging API, then there is a a JavaScript library for that asa well, but you need to be aware of the differences when using asynchronous versus synchronous requests.

	Synchronous	***Asynchronous***
JavaScript file	jsonrpc.js	jsonrpc_async.js
Method format	`o.m( arg1, arg2, ...)`	`o.m( func, arg1, arg2, ...)`

The synchronous request format is the simplest. After creating a client-side instance of the JSONRpcClient object, defined in jsonrpc.js, you have access to client side version of server-side objects. So, for example, if the server is offering a service from the Library class, then the client has access to an object named `library`:

```
var library = jsonrpc.library
```

Code may then invoke library methods as defined by the server object:

```
var matches = library.match( some_.value )
```

A nice thing about using the synchronous calling is that client code is fairly oblivious to the particulars of its environment. The code is written as if everything is local. The

downside, though, is that many things are *not* local, and network issues may introduce noticeable delays. And when there are delays, the browser may be completely unresponsive, leading to a poor user experience.

The client example code uses asynchronous calling; it is much the same as the above examples, but there is one essential difference: methods invoked on client proxy objects (e.g, `library`), must pass a callback function as the first parameter. All the remaining parameters are the same as in the synchronous version.

The Essential Stuff

The Web page starts off with this markup and code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
8" />
  <title>Library List</title>
  <script type="text/javascript"
src="/script/jsonrpc_async.js"></script>
```

Note that the file must include a reference to the proper jsonrpc JavaScript file. For asynchronous calls, that would be `jsonrpc_async.js`. If you prefer synchronous code, use `jsonrpc.js`.

The page also needs to instantiate a `JSONRpcAsyncClient` object. This is what handles the requests. The object needs to be created with the URL of the server. It is important that this URL use the same domain name as the Web page hosting the object or security features will block the requests. The example assumes you are trying this out on a local machine and uses a literal IP address. Some global variable are defined, and an `init` function constructed to handle some basic initialization. This function will be called when the page loads.

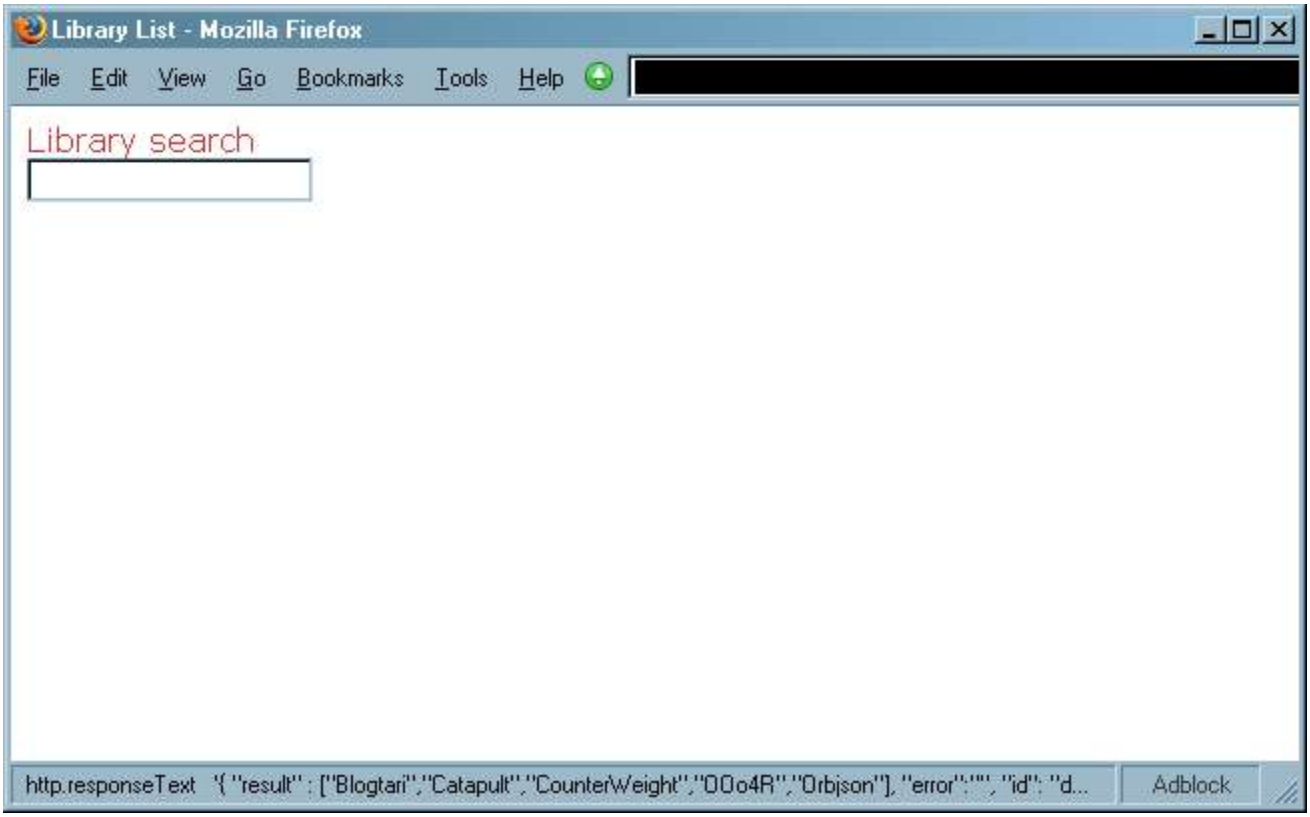
```
<script language="javascript" type="text/javascript">
    var jsonurl = "http://127.0.0.1:2222/json-rpc";
    var jsonrpc = null;
    var library  = null;
   //--------------------------------------------------
    function init() {
      jsonrpc = new JSONRpcAsyncClient( jsonurl );
      hide_list();
      hide_details_block();
      document.getElementById( 'lib_list' ).focus();
    }
```

This simply initializes the `jsonrpc` variable, calls a helper functions that hides some page content, and sets the focus to the input element used to search for libraries.

The HTML uses various div elements to hold search results and a description for the top matching library. When there are no matches, these elements are hidden from view; this is the default when the page loads. There are also corresponding functions to reveal these elements. See the source code for this article for the details.

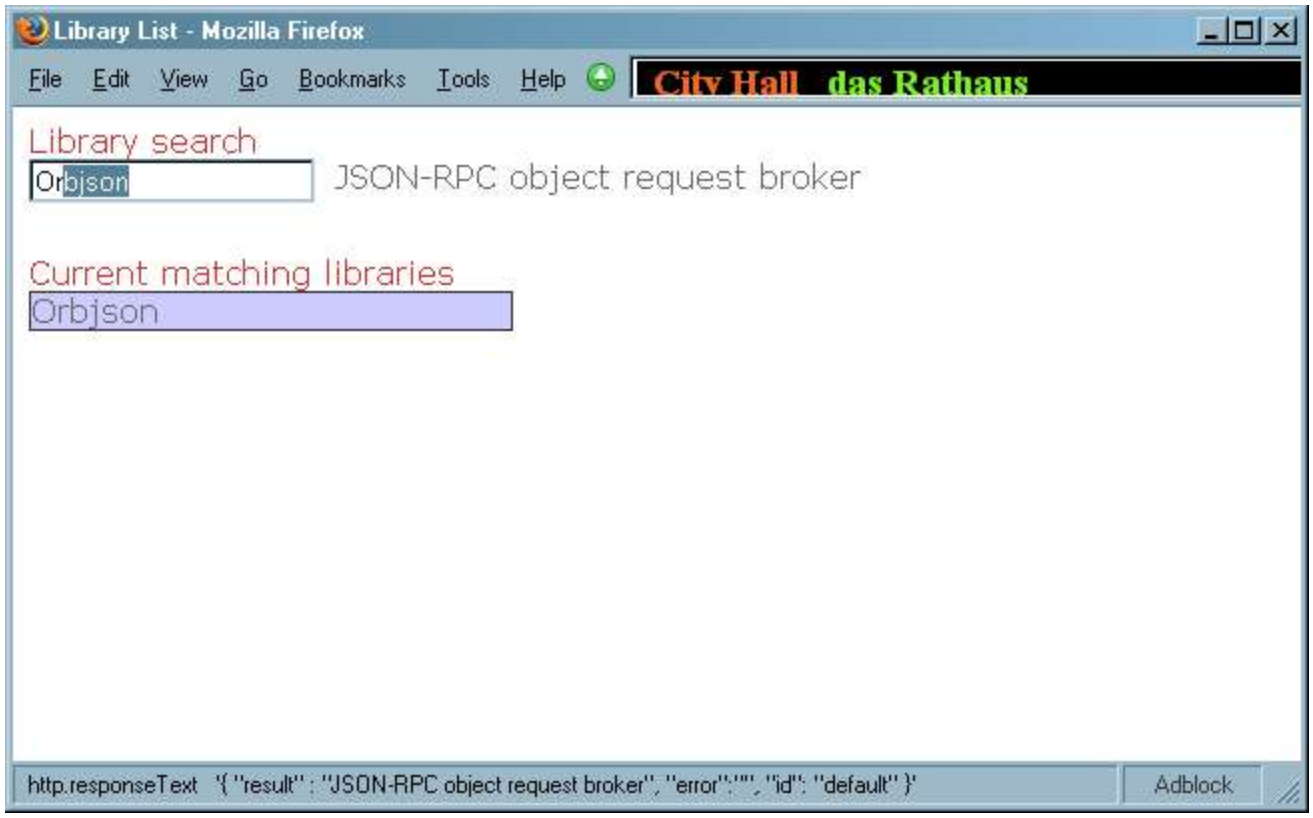
The Page

Let's jump ahead and see what the page looks like, then go back and examine the mechanics. When the page is load, there is a simple input field



As the user types in the Library Search field, a list of matches, based on the partial library name so far typed, appears. The first match is presented in the search field, with the guessed completion text preselected. A description for this presumed match also appears. For example, if the user types the letter "o", all libraries beginning with that letter appear, with the first match presented as auto-completed text, and the description to the right.

The search list shows all matches so far, so the user knows what the viable choices are. If the next letter typed is "r", then the list is reduced, and the auto-complete text and library description change:



The user may also use the backspace key to go back to previous matches.

The main event happens when the user enters text in the search field. The `body` HTML looks like this:

```
 <body onload='init()' >
  <label>Library search</label>
  <input type='text'  id='lib_list'
         onkeyup='show_match( event )' />
  <div id='details'></div>
   <div id='match_area'>
    <label>Current matching libraries</label>
   <div id='match_list'></div>
  </div>
 </body>
</html>
```

The `init` function is called when the page is loaded, initializing the required objects. The `onkeyup` event of the `lib_list` input field is attached to the `show_match` function. It passes in the key event so that the code can determine if this is new text or a deletion. Here's `show_match`:

```
function show_match( e ) {
  var lib_list =  document.getElementById( 'lib_list' )
  var kc
```

```
  if ( window.event )
    kc = window.event.keyCode
  else if ( e )
    kc =  e.which

  if (!library )  library = jsonrpc.library

  if ( is_erase_key( kc ) )  {
    lib_list.value =   lib_list.value.slice( 0,
                  lib_list.value.length-1)
  }

  var  fmatch = function( res) {
    display_matching_items( res , lib_list)
  }
  library.match( fmatch, lib_list.value )

}
```

The function gets a reference to the `lib_list` input field element so that it can read, and later manipulate, its value. It then does a bit of browser-dependent finagling to get the key code for the entered text.

Next there's some lazy initialization; if the global `library` object has not yet been initialized, then it happens now.

If the user was deleting text, then the function updates the field value before initiating a library search.

A callback function is then defined, assigned to to the local variable `fmatch`. The function needs to take a single argument, which will be the results of the JSON-RPC request. This particular function takes that value and passes it off to another function, `display_matching_items`, which handle the actual page updating.

Finally, `library.match` is invoked to get a list of libraries matching on the text in the search field.

The magic

Ideally, what happens now is this: the call to `library.match` is, thanks to the `jsonrpc_async.js` code, converted into an asynchronous JSON-RPC call, using the `XmlHttpRequest` object, back to the server. On the server, Orbjson will grab the JSON-RPC message, locate a corresponding service object, invoke the named method, and return the results as a JSON-RPC response.

Right after the request is sent by the client, process control returns to the browser. However, the callback function passed when invoking match has now been attached to an `XmlHttpRequest` state-change event. The `XmlHttpRequest` object waits for the response and invokes this callback upon receipt. Most of the time the response is so fast the user is unaware that this background request has been made (other then the fact that the page looks different). It is possible, though, that network or server issues could

impede a speedy response. The user can still go on merrily editing the search field, though the updates will not be seen.

When the function `display_matching_items` is called it takes the resulting list of matching items and updates the page. The code checks if the list is empty; if so, it hides the elements showing the matches and description. Otherwise, it iterates over the list and updates the page. The first item is assigned to the search field itself. any trailing text that was not part of what the user entered is auto-selected. The remaining items are listed in the div element below the search field.

The method then looks at that first match item and makes another call to the server to fetch the library description. Note that, as JSON-RPC allows the transfer of reasonably complex JavaScript objects, this description could very well have been sent back right along with the list of matches. The inefficiency has been introduced here as an excuse to show another remote call.

As `display_matching_items` goes along, it makes a call to the `get_details` function. This function takes the current matching library name and uses it to retrieve a description.

```
function get_details( current_match ){
  var  fdetails = function( res) {
    show_details( res )
  }
  library.details( fdetails, current_match  )
}
```

As before, callback function is defined, and passed in as the first argument to the proxied request. The function `show_details` just takes the returned description and updates a particular `div` element with the new text.

And that's it. The source code for this article should give you enough to get going on your own application.

Some Parting Observations

The current method for configuring Orbjson services does not allow for passing arguments to class constructors. There are basically two ways to fix this: invent some textual means for defining parameters to pass to `new`, or ignore it and leave it to the user.

The argument for the latter approach is that when selecting what services to implement you need to consider what methods are being exposed. Recall that all public methods are available. Ideally, your business classes should not be written for use in any particular specialized environment. Instead, you may be better served by using wrapper classes that handle the initialization of the actual business object, and expose only those methods you explicitly want available through Orbjson.

Here's an example. Suppose we want to pull the list of libraries from a database. Suppose also that the `Library` class was written for a variety of purposes, but not all public methods are suitable for Web client invocation.

Our new library service class might then look like this:

```
require 'path/to/actual/lib-list'

class LibraryWrapper

  def initialize(  )
     @library  = Library.new(){
        :user => "dbuser",
        :password => "dbpassword",
        :db_url => "foo.server.url:5000"
      }
  end

  def match( str )
    @library.match( str )
  end

  def details( key )
    @library.details( key )
  end
end
```

The configuration details would also have to be changed to refer to this wrapper class rather than the actual `Library` class.

It might also be preferable to have all JavaScript JSON-RPC code in a single file, and have the choice of synchronous or asynchronous calls available whenever a method need be invoked. At them moment, though, the focus has been on writing Ruby rather than JavaScript.

Resources

Orbjson

http://orbjson.rubyforge.org/

Code for this article

http://orbjson.rubyforge.org/tutorial/tutorial_code.tgz

JSON-RPC

http://json-rpc.org/

Needle

http://needle.rubyforge.org/

Ruby JSON

http://www.ping.de/~flori/flott/exe/session/project/ruby/json

Google Suggest

http://www.google.com/webhp?complete=1&hl=en

Page 13 of 13