JSON-RPC server & client

Contents

Introduction

This document describes

- a PHP based JSON-RPC server class
  - How to call it
  - How to add new methods
  - How to add new services
- a PHP based JSON-RPC client class and a
- Javascript AJAX JSON-RPC library

The PHP server class responds to messages formatted using the JSSON-RPCv2.0 specification. It supports the single request, batch request and notification modes of the specification. It is tested with PHP v7. It uses a pluggable framework for adding methods so that the methods served are independent of the server code itself.

The PHP client class uses CURL to format and send a request to the server and to retrieve the response, so your server needs to have the PHP CURL module included. It operates synchronously. It is tested with PHP v7

The Javascript AJAX library sends requests asynchronously using AJAX and processes the response with a callback function. It is written for ECMAScript 6 and tested with Chromium v64

The JSON-RPC v2.0 specification can be found at http://www.jsonrpc.org/specification#overview

Set up

Copy all the files and directories into a directory that is accessible to the server. The files are as follows:

examples/exampleViaAJAX.html	An example script which shows the Javascript AJAX client at work and how to use it
examples/exampleViaPHP.php	An example script which shows the jsonrpcClient class at work and how to use it
index.php	This file contains the HTTP Transport wrapper for the jsonrpcServer class
examples/jsonrpcClient.class.php	Use this class when you want to fire off JSON-RPC requests from a server side (PHP) script.
examples/jsonrpcClient.js	Use this library when you want to fire off JSON-RPC requests from an HTML page running in a modern browser.
jsonrpcServer.class.php	This class implements the server processes. It is invoked by index.php for HTTP transported requests, but could be used for requests arriving via other methods (e.g. email)
methods/test/test.php	This file implements a method (“test”) which is used in the example scripts. It simply reflects back the parameters that are sent to it. It is an example of a pluggable service module.
methods/	This directory contains a sub-directory for each declared method
services/	This directory contains a sub-directory for each declared service. Services follow the same standard as methods but cannot be invoked directly by a client.

services/dbconnect.php	A service to provide connection to a database. The database handle is cached for subsequent calls.
docs/	Documentation for RPC-JSON server
MaintenanceMode	This file is empty. Its presence signals the server to operate in maintenace mode. In this mode it is more verbose and allows operations that would otherwise not be allowed. Method directories may have their own MaintenanceMode file
methods/<various directories>	There may be additional method directories implementing further methods. These are documented separately.

JSON-RPC Server for PHP

The code for the server class is in the file ‘jsonrpcServer.class.php’. The file ‘index.php’ is an example of how to invoke the server to provide for requests sent by HTTP.

In addition there is an example service module ‘test.php’ which implements the method ‘test’.

Although the server can be addressed directly it is more likely to be called by using one of the client service modules that are described on the following pages.

This section describes the use of the server via HTTP POST.

Typically, the server will be used either from PHP scripts running on other servers or AJAX requests from browser clients. In either case, the request should be formatted as an HTTP POST request with the payload held in a ‘data’ variable. (Assuming you are using the supplied ‘index.php’ file)

The PHP and Javascript clients which can handle the request process for you, are described later in this document.

A typical request would look like this:

```
{"jsonrpc": "2.0", "id": 1, "method": "test", "params": {"p1": "one", "p2": "two"}}
```

which would result in the response:

```
{"jsonrpc": "2.0", "id": 1, "result": {"p1": "one", "p2": "two"}}
```

(“test” just reflects back the parameters you send it)

or, if there was an error in the request, e.g an unknown method, an error response would be returned:

```
{"jsonrpc": "2.0", "id": 1, "error": {"code": -31001, "message": "File wrong.php not found",
"data": "" }}
```

You can also send a batch of requests together by wrapping them in an array :

```
[{"jsonrpc":"2.0", "id":1, "method":"test", "params":{"p1": "one", "p2": "two"}},
{"jsonrpc":"2.0", "id":2, "method":"test", "params":{"p3": "three", "p4": "four"}}]
```

and the response would be returned similarly in an array:

```
[{"jsonrpc": "2.0", "id": 1, "result": {"p1": "one", "p2": "two"}}, {"jsonrpc": "2.0", "id":
2, "result": {"p3": "three", "p4": "four"}}]
```

Errors are handled request by request:

```
[{"jsonrpc":"2.0", "id":1, "method":"wrong", "params":{"p1": "one", "p2": "two"}},
{"jsonrpc":"2.0", "id":2, "method":"test", "params":{"p3": "three", "p4": "four"}}]
```

yields:

```
[{"jsonrpc": "2.0","id": 1, "error": {"code": -31001,"message": "File wrong.php not
found","data": "" } }, {"jsonrpc": "2.0", "id": 2, "result": {"p3": "three", "p4": "four"}}]
```

However, if the error is in the JSON formatting (in this case the closing square bracket is missing):

```
[{"jsonrpc":"2.0", "id":1, "method":"test", "params":{"p1": "one", "p2": "two"}},
{"jsonrpc":"2.0", "id":2, "method":"test", "params":{"p3": "three", "p4": "four"}}
```

You would not receive an array, even if your data contained an array (with faulty formatting the server would not know). Instead you would receive a single object like this:

```
{"jsonrpc": "2.0","id": null,"error": { "code": -32600, "message": "Not a JSON-RPC v2.0
request","data": "" }}
```

Methods and how to add them

The server is designed to invoke methods that are stored in (PHP) files separate to itself. In order to write a successful method, you need to understand how the server looks for the methods it has been asked to invoke.

How the server locates method definitions

All method definitions are held in a ‘methods’ sub-directory of the server (i.e. the directory where index.php is stored)

Each method has its own sub-directory. The name of the sub-directory must be the same as the name of the method.

Within the sub-directory there must be a file [methodname].php where [methodname] is the name of the method.

How to write a method/function

The file [methodname].php (and any other scripts in the directory) MUST declare a namespace which is the same as the method name or a sub-namespace of it.

The file MUST at least contain a function whose name is the same as the method. This function will be invoked by the server when it receives a client request specifying that method name.

The [methodname] function will be passed the content of the ‘params’ property of the jsonrpc request in the form of a PHP object.

The function must return a PHP object. This will form the content of the ‘result’ property of the message sent back to the server by the client.

Any errors should be thrown using **`throw new Exception(‘message’, $code)`**, where **`$code`** must be an integer

The ‘test’ method that is supplied as an example is fully documented. You will find it in **`methods/test/test.php function test($params) {…}`**

The ‘test’ method simply responds with a result containing the ‘params’ that were sent to it;

Here is another example:

File ‘**`methods/myFunc/myFunc.php`**’:

```
namespace myFunc;

function myFunc($params) {
    if ($params->question ==”is this my life?”) { //this would yield true
        $result = ‘yes it is.’;
    } else {
        $result = ‘what do you mean?’;
    }
    return (object) [“answer”=>$result];
}
```

(note you must return the result as an object, not an array) and called it with :

```
{"jsonrpc":"2.0", "id":1, "method":"myFunc", "params":{"question":"is this my life?"}
```

You would get the response:

```
{"jsonrpc": "2.0", "id": 2, "result": {"answer": "yes it is."}}
```

Adding Services

Services operate the same way as methods, but they can only be invoked from scripts not by client requests. The purpose of services is to provide services that are commonly needs.

There is an example service '**`services/dbconnect/dbconnect/php`**' which implements a database connection and returns it in the '**`db`**' property of the returned object.

Services are invoked like this:

```
$return_value = call_service('dbconnect', $params);
```

(You can actually also invoke methods similarly:

```
$return_method = call_service('dbconnect', $params);)
```

Services are written to the same rules as methods except that they are stored in a sub-directory of the **`services`** directory.

You SHOULD always return your result in the form of an object even if it only has a single property. This allows for additional parameters to be passed in the future without breaking existing code

JSON-RPC Client for Javascript

The JSON-RPC Client for Javascript is in the file **‘jsonrpc.js’.** Add this file to your web page like this:

```
<script src=jsonrpc .js></script>
```

and use it like this:

```
//First get the promise:
let p = jsonrpcClient("/pathTo/index.php", "test", {"p1":"one", "p2":"two"});

//specify what happens if the promise is successfully resolved
p.then(function(response) { alert("ALL OK:" + JSON.stringify(response, undefined,4); });

//Specify what happens if the promise fails to deliver
p.catch(function(response) { { alert("OH NO!:" + JSON.stringify(response, undefined,4);});
```

There is an example file "**examples/exampleViaAJAX.html**" that lets you send requests and receive responses. It appears as shown below. Type the RPC code in the box on the left (there is a sample supplied) and see the result in an alert box after you click the Send! button.

(Note: ‘test’ is an example method that just replies with the parameters you send to it.)

Note that this is an asynchronous request. The function in .then() will not be actioned in line but later, when the server responds.

JSON-RPC Client for PHP

The JSON-RPC Client for PHP is in the file **`'jsonrpc.class.php'.`** Add this file to your web page PHP script like this:

```
require_once('jsonrpc.class.php');
```

and use it like this:

$cli = new jsonrpcClient();

Then, assuming you have already derived values for

$server_url	the fully qualified name of the server e.g. 'http://localhost/test/JSON_RPC/index.php'
$method	e.g. 'test'
$id	a number or a text string that does not start with a number. Note that if you do not supply an id, the request will be treated as a notification and there will be no reply.
$params	an object that contains the parameters expected by the method

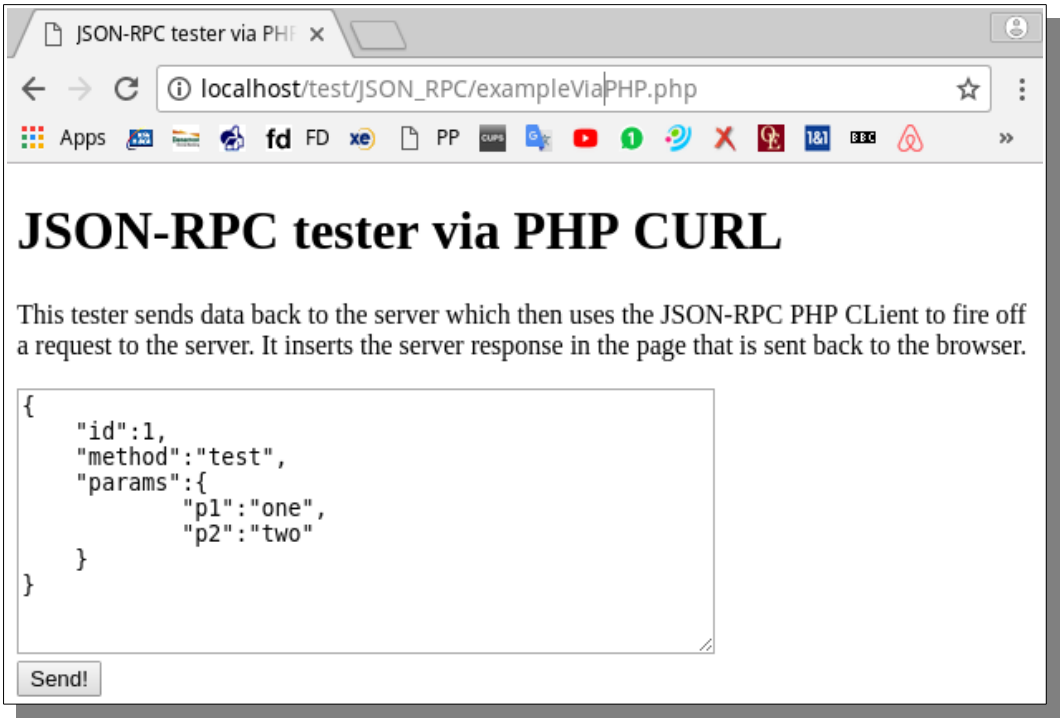
This will retrieve a result from the server:

```
$result = $cli->request($server_url, $method, $id, $params);
```

There is an example file **`'examples/exampleViaPHPCURL.php'`** that lets you send requests and receive responses. It appears as shown below. Type the RPC code in the box on the left (there is a sample supplied) and see the result in an alert box after you click the Send! button.

(Note: 'test' is an example method that just replies with the parameters you send to it.)

This is a round trip transaction. The form is sent to the web server which carries out the request on behalf of the client and then sends back the page with the results.



Other methods and services

The following additional methods are supplied in separate submissions:

getdata

getdata – this will execute an SQL command on the database that is connected by the **`dbconnect`** service. It accepts the following parameters:

parameter **`query`** *string* – this is the name of a query in a library of pre-written queries, or, if in MaintenanceMode, an actual string of SQL code (dangerous, do not allow maintenance mode recklessly!!!)

parameter **`querydata`** *object* – this object contains a property whose name matches a placeholder in the (stored) SQL query and whose value is the value to be used to replace that placeholder when the query is executed.

parameter **`pageno`** *integer* – Used when paging through a request that returns a large number of rows and you only want a few at a time. (optional - default - show the first page).

parameter **`pagelength`** *integer* - Used when paging through a request that returns a large number of rows and you only want a few at a time. (optional - default - return all rows).

The result of the query will contain at least two fields in the 'result' property of the message returned to the client. These are:

param **`rows_affected`** *integer* - the number of rows changed or retrieved by the query.

param **`resultData`** *object/array* – for SELECT, contains an array element for each row. Each element is an object with one property per column where the name is the column name and the value is the value for that column in the current row.

In addition, if the query is an INSERT to a row with an auto-increment column,

param **`last_insert`** *integer* - will contain the id of the inserted row.

getmeta

`getmeta` has exactly the same parameters as **`getdata`**, but instead of returning the data from a query it returns as much data ***about*** each column as it can. It uses a variety of techniques to achieve this.

If the query is from a library and that library contains custom metadata for a column, it will return that.

If the query is an SQL statement or there is no custom metadata, it will query the database using various techniques to get as much information as possible.

Two properties are returned

parameter **`resultfields`** *object* - contains properties where the name is a column name and the value is an object containing all of the metadata properties for that field.

parameter **`requestfields`** *object* - contains properties where the name is a placeholder name and the value is an object containing all of the metadata properties for that placeholder (provided the placeholder name is also a valid column name.). These are the values you would have to supply if making a **`getdata`** request.

NOTE: It is not unusual to batch getdata with a getmetadata request. This saves a server round trip (HTTP is baggage heavy) and a connection to the database server in the JSON-RPC server.

listsql

`listsql` returns a list of all the query definitions in the library. It takes no parameters. The result is an array of objects with one library query per array element. The objects contain just one property `queryName` whose value is the name of a query.