Dynamic Languages World Europe 2008

Stephan Schmidt | 1&1 Internet AG

JSON-RPC-Proxy Generation with PHP 5

Connecting PHP and JavaScript

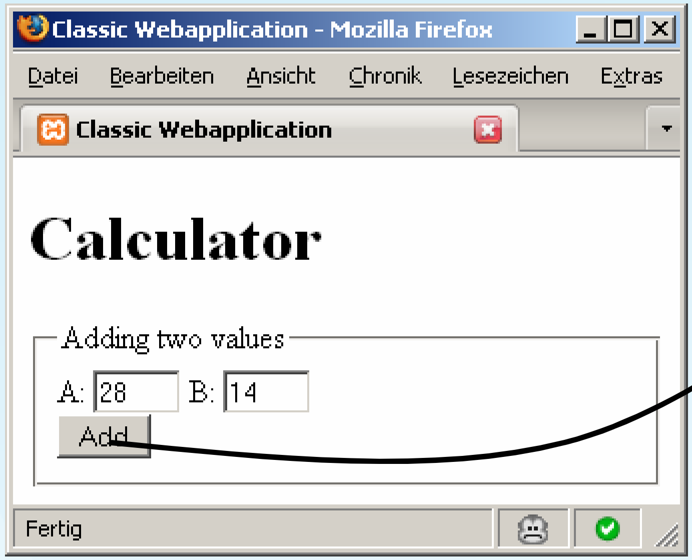
What will I talk about today?

- The Speaker
- Web 2.0 and the Proxy Pattern
- JavaScript-Object-Notation
- JSON in PHP
- The JSON-RPC Specification
- Server- and Clientside JSON-RPC
- Generating Remote Proxies
- SMD and the Dojo Toolkit

Who is Stephan Schmidt?

- Head of Web-Development at 1&1 Internet AG in Karlsruhe (Java, PHP, JavaScript, XSL)
- Developing PHP since 1999
- Contributor to various Open Source Projects since 2001 (*PEAR*, *pecl*, *pat*, *Stubbles*, …)
- Author of „*PHP Design Patterns*“ and co-author of several other PHP related books
- Speaker at international conferences since 2001

A classic web application


Classic Webapplication - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras
Classic Webapplication

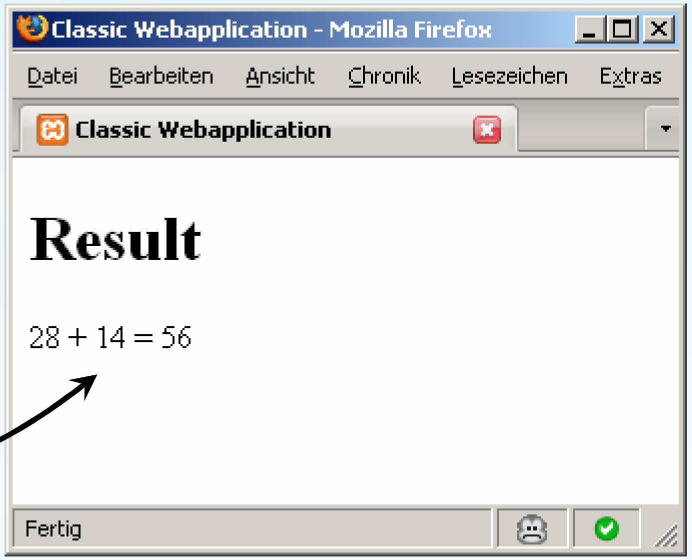
Calculator

Adding two values

A: 28 B: 14

Add

```
POST /projects/json-rpc/classic/result.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://localhost/projects/json-
rpc/classic/form.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
a=28&b=14
```


Classic Webapplication - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras
Classic Webapplication

Result

28 + 14 = 56

```
HTTP/1.x 200 OK
Date: Sat, 24 May 2008 16:28:32 GMT
Server: Apache/2.2.8 (Win32) DAV/2
X-Powered-By: PHP/5.2.5
Content-Length: 299
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<html>
<head>
  <title>Classic Webapplication</title>
</head>
…
```

Classic web applications

- Business logic and presentation layer are both on the server side
- Every action triggers an HTTP request, that reloads the complete page
- No functionality in the client
  - Except some small JavaScript components that are not business critical

The business logic layer

```
class Calculator
{
    /**
     * Add two numbers
     *
     * @param   int     $a
     * @param   int     $b
     * @return  int
     */
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

The presentation layer (server-side)

```
<?php
require_once '../Calculator.php';
$calc = new Calculator();
?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de"
   lang="de">
<head>
  <title>Classic Webapplication</title>
</head>
<body>
<h1>Result</h1>
<?php echo $_POST['a']; ?> + <?php echo $_POST['b']; ?> =
<?php echo $calc->add($_POST['a'], $_POST['a']); ?>
</body>
</html>
```

What has changed with Web 2.0?

Presentation logic has been moved to the client

- DOM manipulations to show/hide parts of the page or change content
- Actions <u>do not</u> trigger a reload of the complete page

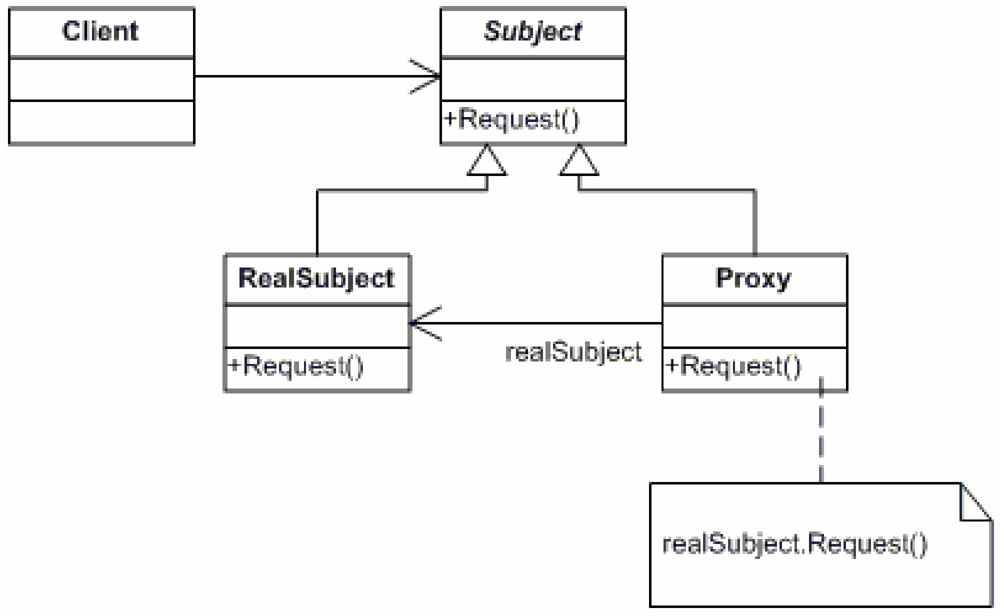
BUT: The business logic still resides on the server and must be accessed by the presentation layer

How can this dilemma be solved?

- Business logic has to stay in PHP, as you cannot trust the client
- We need a JavaScript object as a stand-in for the business logic implemented in PHP
- The presentation layer should make calls on this object and the calls should transparently be routed to the business logic

The Proxy Pattern

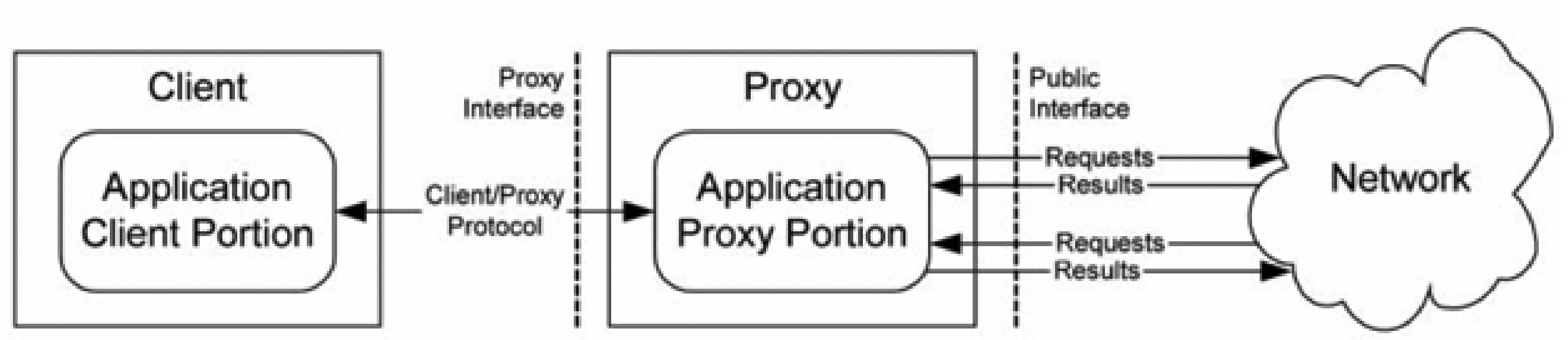
Provide a surrogate or placeholder for another object to control access to it.



Source: http://en.wikipedia.org/wiki/Proxy_pattern

The Remote Proxy Pattern

The Remote Proxy pattern uses a proxy to hide the fact that a service object is located on a different machine than the client objects that want to use it.



Source: http://www.titu.jyu.fi/modpa/Patterns/pattern-RemoteProxy.html

How does this pattern help us?

1. The proxy is implemented in JavaScript and provides the same public methods as the business logic class implemented in PHP
2. The proxy serializes the method call and sends the request to the server
3. On the server, the request is deserialized and the requested method is invoked

How does this pattern help us?

4. The return value of the PHP business logic call again is serialized and sent as a response
5. On the client, the proxy deserializes the response and returns the result.

This is completely transparent to the client!

Serializing the method call

Complex data structures need to be serialized

- anything that could be a parameter of a method call
- Strings, integers, booleans, arrays, objects, …

Serializing the method call

Different formats for serializing data are available

- XML
  - XML is very verbose, the requests should be as small as possible ☹
- PHP‘s serialize format
  - Restricted to PHP ☹
- Custom formats (binary or plaintext)

Enter JSON

JSON = **J**ava**S**cript **O**bject **N**otation

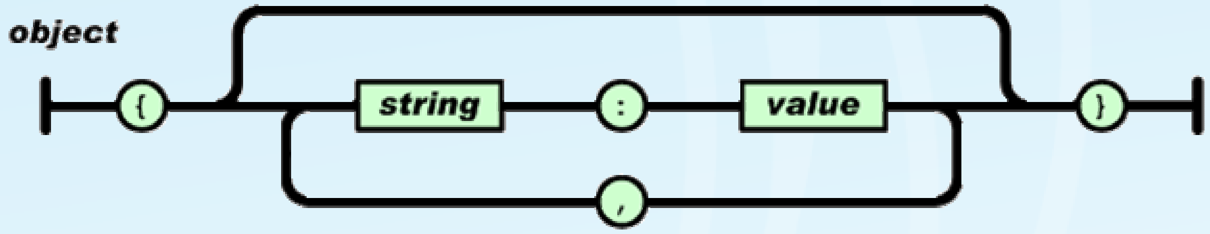
- ☺ lightweight data-interchange format
- ☺ Easy to read/parse for humans and machines
- ☺ completely language independent
- ☺ uses conventions that are familiar to programmers of the C-family of languages

Enter JSON

JSON is built on two structures:

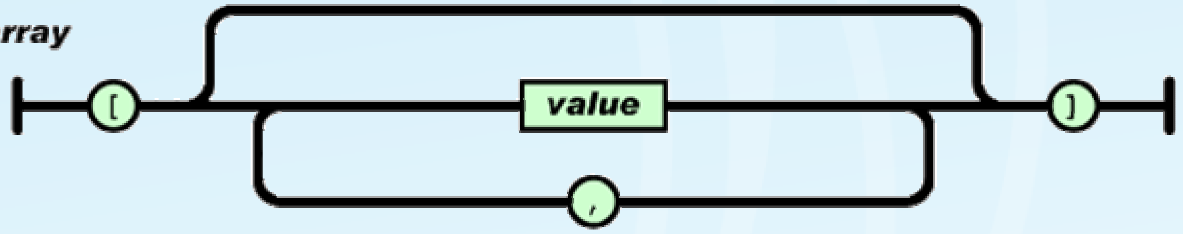
- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

Objects in JSON



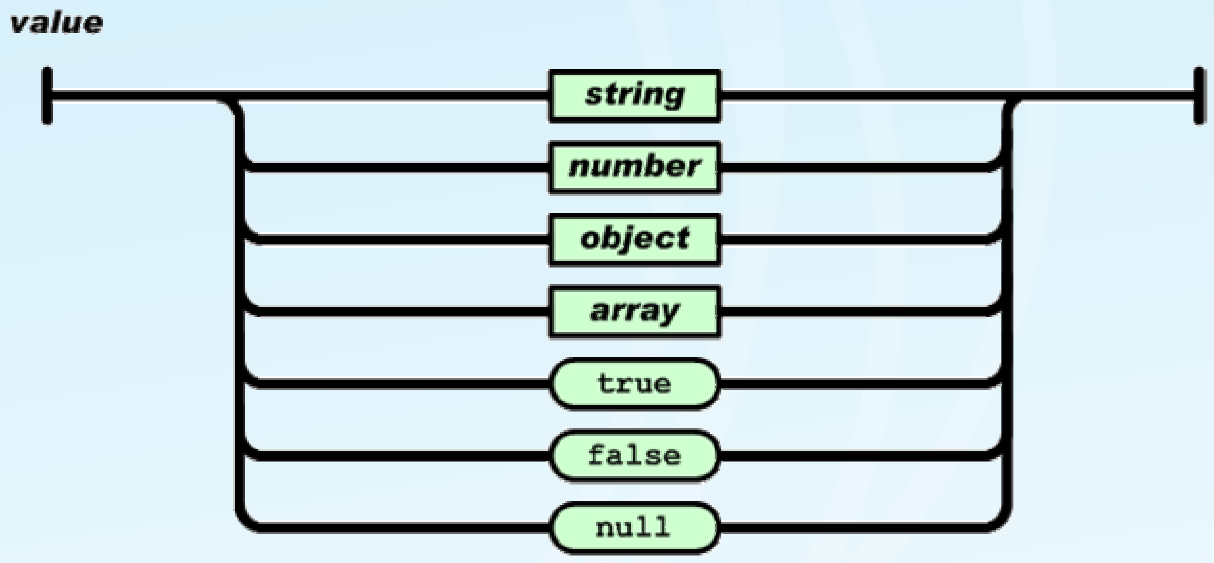
- Set of unordered key/value pairs
- Starts and ends with curly braces
- Key and value are separated by a colon
- Key/value pairs are separated by commas

Arrays in JSON



- Ordered collection of values
- Starts and ends with square brackets
- Values are separated by commas

Values in JSON



- Can be any of the types listed above
- Objects and arrays are also values, which leads to nesting

JSON examples

PHP

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)

stdClass Object
(
    [id] => schst
    [mail] => schst@php.net
)
```

JSON

```
[1,2,3]







{"id":"schst",
 "mail":"schst@php.net"}
```

JSON in JavaScript

- JSON is a subset of the literal notation of JavaScript
- Deserialization can be done via **`eval()`**
  - Leads to security issues
- json.org provides a JSON parser/stringifier at http://www.json.org/js.html
  - Smaller than 2k, if minified

JSON in PHP

- Various parsers / stringifiers are available
  - PHP based: Zend_JSON, Solar_JSON, …
  - C based: pecl/json
- Since PHP 5.2, the PECL extension is part of PHP core
  - Use it, if possible
- Benchmark available at http://gggeek.altervista.org/sw/article_20070425.html

Encoding data with PHP

- Extension provides **`json_encode()`** function
- Encodes any value, except resources, in JSON format
- Requires UTF-8 data

```
$data = array(1,2,3);
$json = json_encode($data);
```

Decoding data with PHP

- Extension provides **`json_decode()`** function
- Decodes any JSON-string to PHP data structures
- Objects can be decoded to **`stdClass`** instances or arrays

```
$json = "[1,2,3]";
$data = json_decode($json);
```

Invoking a remote method

Method of an object that is located on a different machine is invoked via the network.

Already used in different areas:

- RMI
- SOAP
- XML-RPC

The JSON-RPC protocol

JSON-RPC is a stateless and light-weight remote procedure call (RPC) protocol for inter-networking applications over HTTP. It uses JSON as the data format for all facets of a remote procedure call, including all application data carried in parameters.

Source: http://www.json-rpc.org/

The JSON-RPC protocol

- Inspired by the XML-RPC protocol
- Request and response are serialized in JSON
- Data is transported via HTTP
- Very often used asynchronously, to avoid that the application is frozen while the request is being processed

JSON-RPC Request

The request is a single object with the following properties:

- **`method`**
  A string containing the name.
- **`params`**
  An array of objects
- **`id`**
  A unique identifier of the request

JSON-RPC Response

The response is a single object with the following properties:

- **`result`**
  The result of the method call
- **`error`**
  An error object, if an error occurred
- **`id`**
  The same identifier used in the request

JSON-RPC examples

Request

```
{ "method": "echo",
  "params": ["Hello JSON-RPC"],
  "id": 1}
```

Response

```
{ "result": "Hello JSON-RPC",
  "error": null,
  "id": 1}
```

Let’s get practical

- Implement a JavaScript class that provides a method to call arbitrary JSON-RPC services
- Implement a PHP script, that acts as a server for this class
- Use PHP to generate JavaScript proxies for existing PHP classes that make use of this JavaScript JSON-RPC client

JSON-RPC Client (simplified)

```
var JsonRpcClient = function(clientObj) {
  // map ids to callbacks
  var reqRespMapping  = [];

  var callback = {
    // callback functions for asynchronous calls
  }

  this.createId = function() {
    // create a unique id
  };
};
```

JSON-RPC Client (simplified)

```
var JsonRpcClient = function(clientObj) {
  this.doCall = function(classAndMethod, args) {
    var id = this.createId();
    var jsonRpcReq = {
            method: classAndMethod,
            params: arr,
            id: id
        };

    YAHOO.util.Connect.asyncRequest('POST', finalServiceUrl,
                                    callback, jsonRpcReq.toJSONString());
    reqRespMapping.push(jsonRpcReq);
    return id;
  };
};
```

Using the JSON-RPC Client (simplified)

```
 // Callback object, as calls are asynchronous
var CalculatorCallback = {
  callback__add: function(id, result, error) {
    alert(result);
  }
}

function add() {
  var a = document.getElementById('a').value;
  var b = document.getElementById('b').value;

  var client = new JsonRpcClient(CalculatorCallback);
  client.doCall('Calculator.add', [a,b]);
}
```

JSON-RPC Client

- Name of the PHP class and method has been separated using a dot (“.”)
- Callback method is called “**`callback__`*`$METHOD`***”
- YUI is used for XmlHttpRequest abstraction
- The complete code can be downloaded from http://www.stubbles.net

JSON-RPC Server

- Decode the JSON-RPC request
- Extract the class and the method name from the “**`method`**” property of the request object
- Dynamically load the class
- Create a **`stdClass`** object to contain the response and copy the **`id`** property from the request

JSON-RPC Server (simplified)

```
// Get the request
$requestRaw = file_get_contents('php://input');
$request    = json_decode($requestRaw);
// Prepare the response
$response = new stdClass();
$response->id = $request->id;

// Get the class and method
$methodRaw = $request->method;
list($class, $method) = explode('.', $methodRaw);
require_once "../{$class}.php";
```

JSON-RPC Server

- Use the Reflection API to create a new instance of the class
- Use the Reflection API to dynamically invoke the method on this instance, passing the parameters from the JSON-RPC request

JSON-RPC Server (simplified)

```
try {
  $clazz = new ReflectionClass($class);
  $service = $clazz->newInstanceArgs();
  // Invoke the method
  $methodObj = $clazz->getMethod($method);
  $result = $methodObj->invokeArgs($service,
                                    $request->params);
  $response->error  = null;
  $response->result = $result;
} catch (Exception $e) {
}
```

JSON-RPC Server

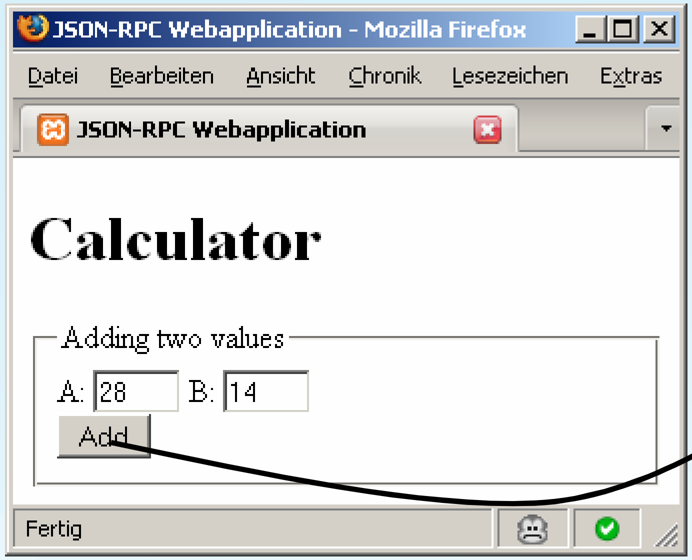
- Set the "**`result`**" property, if the method call was successful.
- Catch any exception during processing and set the "**`error`**" property
- Encode the result in JSON and send it to the browser

JSON-RPC Server (simplified)

```
try {
  …
} catch (Exception $e) {
  $response->error  = $e->getMessage();
  $response->result = null;
}

// Send the response
echo json_encode($response);
```

A JSON-RPC web application


JSON-RPC Webapplication - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras
JSON-RPC Webapplication
Add
Fertig

Calculator

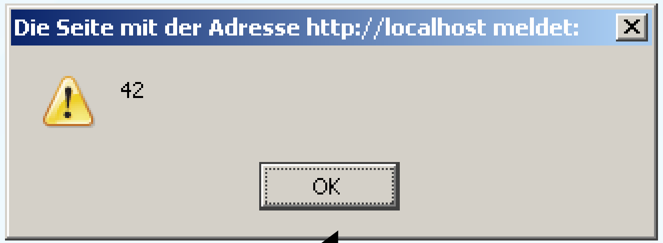
Adding two values

A: 28 B: 14

```
POST /projects/json-rpc/json-rpc/server.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0
Accept-Encoding: gzip,deflate
Keep-Alive: 300
Connection: keep-alive
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Referer: http://localhost/projects/json-rpc/json-rpc/form.php
Content-Length: 63
Pragma: no-cache
Cache-Control: no-cache
{"method":"Calculator.add","params":["28",
"14"],"id":"1248167"}
```

```
HTTP/1.x 200 OK
Date: Sun, 25 May 2008 10:48:44 GMT
Server: Apache/2.2.8 (Win32) DAV/2 mod_ssl/2.2.8 OpenSSL/0.9.8g
mod_autoindex_color PHP/5.2.5
X-Powered-By: PHP/5.2.5
Content-Length: 41
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

{"id":"1246838","error":null,"result":42}
```



Die Seite mit der Adresse http://localhost meldet:

42

JSON-RPC Proxy generation

- Dynamically create a JavaScript class, that encapsulates the **`doCall()`** calls on the **`JsonRpcClient`** object
- Use PHP’s reflection API to extract all public methods from any PHP class
- Provide this functionality over HTTP to generate the clients on-the-fly.

JSON-RPC Proxy Generator

```php
// dynamically load the class
$className = $_GET['class'];
require_once "../{$className}.php";
$clazz = new ReflectionClass($className);

// Create a JavaScript class for this class
echo "var {$className} = function(clientObj) {\n";
echo "  this.dispatcher =
                 new JsonRpcClient(clientObj);\n";
echo "};\n";
```

JSON-RPC Proxy Generator

```
// Iterate over all methods
foreach ($clazz->getMethods() as $method) {
  // only export public methods to the proxy
  if (!$method->isPublic()) {
    continue;
  }
  $methodName = $method->getName();
  // JS generation on next slide
}
```

JSON-RPC Proxy Generator

```
// Iterate over all methods
foreach ($clazz->getMethods() as $method) {
  $methodName = $method->getName();
  echo "{$className}.prototype.{$methodName} =
                                    function() {\n";
  // route this call to the server
  echo “  return this.dispatcher.doCall(
                      '{$className}.{$methodName}',
                      arguments);\n";
  echo "};\n";
}
```

Using the JSON-RPC Proxy generator

- Generated JavaScript code can be loaded the same way as static JavaScript code, using the **`src`**-attribute of the **`<script/>`** tag
- Generated JavaScript provides a new JavaScript class that resembles the PHP class and is used as a proxy

Using the JSON-RPC Proxy generator

```
<script type="text/javascript"
  src="./proxygen.php?class=Calculator"></script>
```

Resulting JavaScript

```
var Calculator = function(clientObj) {
  this.dispatcher = new JsonRpcClient(clientObj);
};
Calculator.prototype.add = function() {
  return this.dispatcher.doCall(
                      'Calculator.add', arguments);
};
```

Using the JSON-RPC Proxy

```
var CalculatorCallback = {
  callback__add: function(id, result, error) {
   alert(result);
  }
}
function add() {
  var a = document.getElementById('a').value;
  var b = document.getElementById('b').value;
  var calc = new Calculator(CalculatorCallback);
  calc.add(a, b);
}
```

Dynamic Languages World Europe 2008

Problems with this approach

- Server-side code generates client side code, which leads to a tight coupling between PHP code and JavaScript code
- Client framework cannot easily be replaced
- We need something like WSDL for JSON-RPC

Enter SMD

SMD = **S**imple **M**ethod **D**escription

- Very easy data structure, that describes all methods and their parameters, that a service provides
- Encoded in JSON format
- Invented by the Dojo Toolkit
- Might be replaced by “**S**ervice **M**apping **D**escription”

Format of an SMD

SMD always contains:

- The SMD version
- Name of the object or class it represents
- Service type (e.g. JSON-RPC)
- Service URL
- Array of all available methods and their parameters

Example SMD

```
{"SMDVersion":".1",
 "objectName":"Calculator",
 "serviceType":"JSON-RPC",
 "serviceURL":"./server.php",
 "methods":[
   {"name":"add",
    "parameters":[
      {"name":"a", "type":"INTEGER"},
      {"name":"b", "type":"INTEGER"}
    ]}
]}
```

Generating SMD with PHP

```
$className = $_GET['class'];
require_once "../{$className}.php";
$clazz = new ReflectionClass($className);

$smdData = new stdClass();
$smdData->SMDVersion  = 1;
$smdData->serviceType = 'JSON-RPC';
$smdData->serviceURL  = './server-smd.php?class=' .
                                          $className;
$smdData->objectName  = $className;
$smdData->methods     = array();
```

Generating SMD with PHP

```
foreach ($clazz->getMethods() as $method) {
  if (!$method->isPublic()) {
    continue;
  }
  $methodDef = new stdClass();
  $methodDef->name = $method->getName();
  $methodDef->parameters = array();
  $smdData->methods[] = $methodDef;
  foreach ($method->getParameters() as $parameter) {
    $paramDef = new stdClass();
    $paramDef->name = $parameter->getName();
    $methodDef->parameters[] = $paramDef;
  }
}
echo json_encode($smdData);
```

Using SMD with the Dojo Toolkit

- First framework that makes use of SMD
- Dynamically generates proxies at run-time based on an SMD structure, JSON-String or URL
- Extremely easy to use
- Similar to ext/soap in PHP 5

Using SMD with the Dojo Toolkit

```
<script type="text/javascript"
  src="http://...dojo/dojo.xd.js"></script>
```

```
djConfig.usePlainJson = true;
dojo.require('dojo.rpc.JsonService');
var smdURL = './smdgen.php?class=Calculator';
proxy = new dojo.rpc.JsonService(smdURL);
```

Server returns this SMD:

```
{"SMDVersion":1,"serviceType":"JSON-RPC","serviceURL":".\/server-
    smd.php?class=Calculator","methods":[{"name":"add","parameters":[{"name":"a"},
    {"name":"b"}]}],"objectName":"Calculator"}
```

Using the proxy

```
// Dojo only needs a callback function, no object
function doAddCallback(result) {
  alert(result);
}

function add() {
  var a = document.getElementById('a').value;
  var b = document.getElementById('b').value;
  // Execute the call and add the callback
  proxy.add(a, b).addCallback(doCalcCallback);
}
```

Existing Frameworks

Several frameworks already provide the generation of proxies:

- pecl/SCA_SDO
- PEAR::HTML_AJAX (no real JSON-RPC)
- Sajax (no real JSON-RPC)
- Stubbles
  - The only framework that implements the JSON-RPC and SMD specs

Thank you

Any Questions?

schst@stubbles.net

http://www.stubbles.net

http://www.stubbles.org