XML-RPC for PHP User Manual

Gaetano Giunta, Edd Dumbill

4.12.0

Table of Contents

A foreword

You might be surprised by some API design choices made by this library. In order to understand that, please keep in mind that this started out as a bare framework around the time of PHP 3, when Exceptions and the DateTime class did not exist, and PHP best practices were remarkably different from current ones. While many "nice bits" have been put in over time, *backwards compatibility has always taken precedence* over API cleanups.

In no particular order, this is partial list of things some developers might find perplexing.

Extensions to the XML-RPC protocol, such as support for the `<NIL>` tag, have to be manually enabled before usage.

Little HTTP response checking is performed (e.g. HTTP redirects are not followed by default and the Content-Length HTTP header, mandated by the xml-rpc spec, is not validated); cookie support still involves quite a bit of coding on the part of the user.

Very little type validation or coercion has been put in. PHP being a loosely-typed language, this is going to have to be done explicitly (in other words: you can call a lot of library functions passing them arguments of the wrong type and receive an error message only much further down the code, where it will be difficult to understand).

dateTime.iso8601 is supported opaquely, largely for historical reasons. Datetime conversion can't be done transparently as the XML-RPC specification explicitly forbids passing of timezone specifiers in ISO8601 format dates. You can, however, use the `PhpXmlRpc\Helper\Date` class to do the encoding and decoding for you, or force usage of DateTime objects via the `PhpXmlRpc\PhpXmlRpc::$xmlrpc_return_datetimes` variable.

There are a lot of static class variables which are meant be treated as if they were constants.

Usage of Exceptions is almost non-existent.

A mix of snake_case and CamelCase naming is used.

Requirements and installation instructions

See https://github.com/gggeek/phpxmlrpc/blob/master/INSTALL.md

API usage

Type conversion

A big part the job of this library is to convert between the data types supported by PHP (`null`, `bool`, `int`, `float`, `string`, `array`, `object`, `callable`, `resource`), and the value types supported by XML-RPC (`int`, `boolean`, `string`, `double`, `dateTime.iso8601`, `base64`, `struct`, `array`).

The conversion process can be mostly automated or fully manual. It is up to the single developer to

decide the best approach to take for his/her application.

Manual type conversion: PHP to XML-RPC

The `PhpXmlRpc\Value` class is used to encapsulate PHP primitive types into XML-RPC values.

The constructor is the normal way to create a Value. The constructor can take these forms:

```
Value new Value
Value new Value(string $stringVal)
Value new Value(mixed $scalarVal, string $scalarTyp)
Value new Value(Value[] $arrayVal, string $arrayTyp)
```

The first constructor creates an empty value, which must be altered using the methods `addScalar()`, `addArray()` or `addStruct()` before it can be used further.

The second constructor creates a string scalar value.

The third constructor is used to create a scalar value of any type. The second parameter must be a name of an XML-RPC type. Valid types are: "int", "i4", "i8", "boolean", "double", "string", "dateTime.iso8601", "base64" or "null". For ease of use, and to avoid compatibility issues with future revisions of the library, they are also available as static class variables:

```
Value::$xmlrpcI4 = "i4";
Value::$xmlrpcI8 = "i8";
Value::$xmlrpcInt = "int";
Value::$xmlrpcBoolean = "boolean";
Value::$xmlrpcDouble = "double";
Value::$xmlrpcString = "string";
Value::$xmlrpcDateTime = "dateTime.iso8601";
Value::$xmlrpcBase64 = "base64";
Value::$xmlrpcArray = "array";
Value::$xmlrpcStruct = "struct";
Value::$xmlrpcValue = "undefined";
Value::$xmlrpcNull = "null";
```

Examples:

```
use PhpXmlRpc\Value;

$myString = new Value("Hello, World!");
$myInt = new Value(1267, "int");
$myBool = new Value(1, Value::$xmlrpcBoolean);
// note: this will serialize a php float value as xml-rpc string
$myString2 = new Value(1.24, Value::$xmlrpcString);
// the lib will take care of base64 encoding
$myBase64 = new Value(file_get_contents('my.gif'), Value::$xmlrpcBase64);
$myDate1 = new Value(new DateTime(), Value::$xmlrpcDateTime);
```

```
// when passing in an int, it is assumed to be a UNIX timestamp
$myDate2 = new Value(time(), Value::$xmlrpcDateTime);
// when passing in a string, you have to take care of the formatting
$myDate3 = new Value(date("Ymd\TH:i:s", time()), Value::$xmlrpcDateTime);
```

The fourth constructor form can be used to compose complex XML-RPC values. The first argument is either a simple array in the case of an XML-RPC array or an associative array in the case of a struct. *The elements of the array must be Value objects themselves*. The second parameter must be either "array" or "struct".

Examples:

```
use PhpXmlRpc\Value;

$myArray = new Value(
    array(
        new Value("Tom"),
        new Value("Dick"),
        new Value("Harry")
    ),
    "array"
);

// nested struct
$myStruct = new Value(
    array(
        "name" => new Value("Tom", Value::$xmlrpcString),
        "age" => new Value(34, Value::$xmlrpcInt),
        "address" => new Value(
            array(
                "street" => new Value("Fifht Ave", Value::$xmlrpcString),
                "city" => new Value("NY", Value::$xmlrpcString)
            ),
            Value::$xmlrpcStruct
        )
    ),
    Value::$xmlrpcStruct
);
```

Manual type conversion: XML-RPC to PHP

For Value objects of scalar type, the php primitive value can be obtained via the `scalarVal()` method. For base64 values, the returned value will be decoded transparently. *NB: for dateTime values the php value will be the string representation by default.*

Value objects of type struct and array support the `Countable`, `IteratorAggregate` and `ArrayAccess` interfaces, meaning that they can be manipulated as if they were arrays:

```
if (count($structValue)) {
    foreach($structValue as $elementName => $elementValue) {
        // do not forget html-escaping $elementName in real life!
        echo "Struct member '$elementName' is of type " . $elementValue->scalarTyp() .
"\n";
    }
} else {
    echo "Struct has no members\n";
}
```

As you can see, the elements of the array are Value objects themselves, i.e. there is no recursive decoding happening.

Automatic type conversion: PHP to XML-RPC

Manually converting the data from PHP to Value objects can become quickly tedious, especially for large, nested data structures such as arrays and structs. A simpler alternative is to take advantage of the `PhpXmlRpc\Encoder` class to carry out automatic conversion of arbitrarily deeply nested structures. The same structure of the example above can be obtained via:

```
use PhpXmlRpc\Encoder;

$myStruct = new Encoder()->encode([
    "name" => "Tom",
    "age" => 34,
    "address" => [
        "street" => "Fifht Ave",
        "city" => "NY"
    ],
]);
```

Encoding works recursively on arrays and objects, encoding numerically indexed php arrays into array-type Value objects and non numerically indexed php arrays into struct-type Value objects. PHP objects are encoded into struct-type Value by iterating over their public properties, excepted for those that are already instances of the Value class or descendants thereof, which will not be further encoded. Optionally, encoding of date-times is carried-on on php strings with the corresponding format, as well as encoding of NULL values. Note that there's no support for encoding php values into base64 values - base64 Value objects have to be created manually (but they can be part of a php array passed to `encode`). Another example, showcasing some of those features:

```
use PhpXmlRpc\Encoder;
use PhpXmlRpc\Value;

$value = new Encoder()->encode(
    array(
        'first struct_element: a null' => null,
```

```
        '2nd: a base64 element' => new Value('hello world', 'base64'),
        '3rd: a datetime' => '20060107T01:53:00'
    ),
    array('auto_dates', 'null_extension')
);
```

See the phpdoc documentation for `PhpXmlRpc\Encoder::encode` for more details on the encoding process and available options.

Automatic type conversion: XML-RPC to PHP

In the same vein, it is possible to automatically convert arbitrarily nested Value objects into native PHP data by using the `PhpXmlRpc\Encoder::decode` method.

A similar example to the manual decoding above would look like:

```
use PhpXmlRpc\Encoder;

$data = new Encoder()->decode($structValue);
if (count($data)) {
    foreach($data as $elementName => $element) {
        // do not forget html-escaping $elementName in real life!
        echo "Struct member '$elementName' is of type " . gettype($element) . "\n";
    }
} else {
    echo "Struct has no members\n";
}
```

Note that when using automatic conversion this way, all information about the original xml-rpc type is lost: it will be impossible to tell apart an `i4` from an `i8` value, or to know if a php string had been encoded as xml-rpc string or as base64.

See the phpdoc documentation for `PhpXmlRpc\Encoder::decode` for the full details of the decoding process.

Notes on types

strict parsing vs. lenient parsing

When Value objects are created by the library by parsing some received XML text, the parsing code is lenient with invalid data. This means that if the other party is sending some junk, the library will, by default: - log error messages pinpointing the exact source of the problem, and - feed to your application "error values", which include `false` for bad base64 data, the string 'ERROR_NON_NUMERIC_FOUND' for integers and doubles, `false` for invalid booleans, and the received string for invalid datetimes. This behaviour can be changed to make the parsing code more strict and produce an error instead of letting through invalid data, by setting `PhpXmlRpc\PhpXmlRpc::$xmlrpc_return_datetimes = true`.

base64

Base 64 encoding is performed transparently to the caller when using this type. Decoding is also transparent. Therefore, you ought to consider it as a "binary" data type, for use when you want to pass data that is not XML-safe.

boolean

All php values which would be converted to a boolean TRUE via typecasting are mapped to an xml-rpc `true`. All other values (including the empty string) are converted to an xml-rpc `false`.

dateTime.iso8601

When manually creating Value objects representing an xml-rpc dateTime.iso8601, php DateTimes, integers (unix timestamps) and strings (representing dates in the specific xml-rpc ISO-8601 format) can be used as source values. For those, the original value will be returned when calling `$value->scalarVal();`.

When Value objects are created by the library by parsing some received XML text, all Value objects representing an xml-rpc dateTime.iso8601 value will by default return the string representation of the date when calling `$value->scalarVal();`.

Datetime conversion can't be safely done in a transparent manner as the XML-RPC specification explicitly forbids passing of timezone specifiers in ISO8601 format dates. You can, however, use multiple techniques to transform the date string into another representation of a timestamp, and take care of timezones by yourself: * use the `PhpXmlRpc\Helper\Date` class to convert the date string into a unix timestamp; * set `PhpXmlRpc\PhpXmlRpc::$xmlrpc_return_datetimes = true` to always get back a php DateTime from received xml (in which case the conversion is done using the timezone set in php.ini) * use the `PhpXmlRpc\Encoder::decode` method with the 'dates_as_objects' option to get back a php DateTime from a single or nested Value object (in which case the conversion is done using the `strtotime` function, which uses the timezone set in php.ini). Note that, when using `$xmlrpc_return_datetimes` or 'dates_as_objects', you might still get back a php `false` or `null` instead of a DateTime if the data received via xml does not represent a valid date and time. You can configure the library to outright reject such cases and avoid having to explicitly check for them in your own code, by setting `PhpXmlRpc\PhpXmlRpc::$xmlrpc_return_datetimes = true`.

double

The xml-rpc spec explicitly forbids using exponential notation for doubles. The phpxmlrpc toolkit serializes php float values using a fixed precision (number of decimal digits), which can be set using the variable `PhpXmlRpc::$xmlpc_double_precision`.

int

The xml parsing code will always convert "i4" to "int": int is regarded by this implementation as the canonical name for this type.

The type i8 on the other hand is considered as a separate type. Note that the library will never output integers as 'i8' on its own, even when php is compiled in 64-bit mode - you will have to create i8 Value objects manually if required.

string

When serializing strings, characters '<', '>', '"', "'", '&', are encoded using their entity reference as '<', '>', ''', '"' and '&'. All other characters outside the ASCII range are encoded using their unicode character reference representation (e.g. 'È' for 'é'). The XML-RPC spec recommends only encoding '<' and '&', but this implementation goes further, for reasons explained by the XML 1.0 recommendation. In particular, using character reference representation has the advantage of producing XML that is valid independently of the charset encoding assumed.

Note that the library assumes that your application will be using data in UTF-8. This applies both to string values sent and to string values received (i.e. the data fed to your application will be transparently transcoded if the remote client/server uses a different character set encoding in its requests/responses). If this is not the case, and you have the php mbstring extension enabled, you can set the desired character set to `PhpXmlRpc::$xmlrpc_internalencoding`, and the library will go out of its way to make character set encoding a non-issue (*).

In case the string data you are using is mostly outside the ASCII range, such as f.e. when communicating information in chinese, japanese, or korean, you might want to avoid the automatic encoding of all non-ascii characters to references, as it has performance implications, both in cpu usage and in the size of the generated messages. For such scenarios, it is recommended to set both `PhpXmlRpc::$xmlrpc_internalencoding` and `$client->setOption('request_charset_encoding', ...)` / `$server->setOption('response_charset_encoding', ...)` to 'UTF-8'.

The demo file *demo/client/windowscharset.php* showcases client-side usage of `$xmlrpc_internalencoding`.

Note that, despite what the specification states, string values should not be used to encode binary data, as control characters (such as f.e. characters nr. 0 to 8) are never allowed in XML, even when encoded as character references.

* = at the time of writing, fault strings and xml-rpc method names are still expected to be UTF-8

null

There is no support for encoding `null` values in the XML-RPC spec, but at least a couple of extensions (and many toolkits) do support it. Before using `null` values in your messages, make sure that the remote party accepts them, and uses the same encoding convention.

To allow reception of messages containing `<NIL/>` or `<EX:NIL/>` elements, set

```
PhpXmlRpc\PhpXmlRpc::$xmlrpc_null_extension = true;
```

somewhere in your code before the messages are received.

To allow sending of messages containing `<NIL/>` elements, simply create Value objects using the string 'null' as the 2nd argument in the constructor. If you'd rather have those null Values be serialized as `<EX:NIL/>` instead of `<NIL/>`, please set

```
PhpXmlRpc\PhpXmlRpc::$xmlrpc_null_apache_encoding = true;
```

somewhere in your code before the values are serialized.

Client

Client creation

The constructor accepts one of two possible syntax forms:

```
Client new Client(string $server_url)
Client new Client(string $server_path, string $server_hostname, int $server_port = 80,
string $transport = 'http')
```

Note that the second syntax does not allow to express a username and password to be used for basic HTTP authorization.

Here are a couple of usage examples of the first form:

```
use PhpXmlRpc\Client;

$client = new Client("https://phpxmlrpc.sourceforge.io/server.php");
$another_client = new Client
("https://james:bond@secret.service.com:443/xmlrpcserver?agent=007");
```

Note that 'http11', 'http10', 'h2' (for HTTP2) and 'h2c' can be used as valid alternatives to 'http' and 'https' in the provided url.

Here's another example client set up to query Userland's XML-RPC server at *betty.userland.com*:

```
use PhpXmlRpc\Client;

$client = new Client("/RPC2", "betty.userland.com", 80);
```

The `$server_port` parameter is optional, and if omitted will default to '80' when using HTTP and '443' when using HTTPS or HTTP2.

The `$transport` parameter is optional, and if omitted will default to 'http'. Allowed values are either 'http', 'https', 'http11', 'http10', 'h2' or 'h2c'. See the phpdoc documentation for the send method for more details about the meaning of the different values.

Sending requests

The Client's `send` method takes a `PhpmlRpc\Request` object as first argument, and always returns a `PhpmlRpc\Response` one, even in case of errors communicating with the server.

```
use PhpXmlRpc\Client;
use PhpXmlRpc\Request;
use PhpXmlRpc\Value;

$stateNo = (int)$_POST["stateno"];
$req = new Request(
    'examples.getStateName',
    array(new Value($stateNo, Value::$xmlrpcInt))
);
$client = new Client("https://phpxmlrpc.sourceforge.io/server.php");
$resp = $client->send($req);
if (!$resp->faultCode()) {
    $v = $resp->value();
    print "State number $stateNo is " . htmlentities($v->scalarval()) . "<BR>";
    print "<HR>I got this xml back<BR><PRE>" . htmlentities($resp->serialize()) .
        "</PRE><HR>\n";
} else {
    print "Fault <BR>";
    print "Code: " . htmlentities($resp->faultCode()) . "<BR>" . "Reason: '" .
        htmlentities($resp->faultString()) . "'<BR>";
}
```

Automatic decoding of the response's value

By default, the Response object's `value()` method will return a Value object, leaving it to the developer to unbox it further into php primitive types. In the spirit of making the conversion between the xml-rpc types and php native types as simple as possible, it is possible to make the Client object return directly the decoded data by setting a value to the Client's 'return_type' option:

```
use PhpXmlRpc\Client;
use PhpXmlRpc\Helper\XMLParser;
use PhpXmlRpc\Request;
use PhpXmlRpc\Value;

$stateNo = (int)$_POST["stateno"];
$req = new Request(
    'examples.getStateName',
    array(new Value($stateNo, Value::$xmlrpcInt))
);
$client = new Client("https://phpxmlrpc.sourceforge.io/server.php");
$client->setOption(Client::OPT_RETURN_TYPE, XMLParser::RETURN_PHP);
$resp = $client->send($req);
if (!$resp->faultCode()) {
    $v = $resp->value();
    // no need to call `scalarval` here
    print "State number $stateNo is " . htmlentities($v) . "<BR>";
    print "<HR>I got this xml back<BR><PRE>" . htmlentities($resp->serialize()) .
        "</PRE><HR>\n";
} else {
```

```
    print "Fault <BR>";
    print "Code: " . htmlentities($resp->faultCode()) . "<BR>" . "Reason: '" .
        htmlentities($resp->faultString()) . "'<BR>";
}
```

This style of making calls will result in reduced memory and cpu usage, and be slightly faster. It is recommended for scenarios where the expected responses are huge, or every little bit of optimization is required.

Please note that, just as with the `PhpXmlRpc\Encoder::decode` method, this will make it impossible to tell apart values which were sent over the wire as strings from values which were base64. On the other hand, unlike that method, at the moment it is not possible to make use of any options to tweak the decoding process.

Troubleshooting failed calls

To ease troubleshooting problems related to the underlying communication layer, such as authentication failures, character set encoding snafus, compression problems, invalid xml, etc..., the Client class can dump to the screen a detailed log of the HTTP request sent and response received. It can be enabled by calling the `setDebug` method with values `1` or `2`.

It is also possible to analyze the different parts of the HTTP response received by making use of the `PhpXmlRpc\Response::httpResponse` method.

Modifying the client's behaviour

A wide range of options can be set to the client to manage the details of the HTTP communication layer, including authentication (Basic, Digest, NTLM), SSL certificates, proxies, cookies, compression of the requests, usage of keepalives for consecutive calls, the accepted response compression, charset encoding used for the requests and the user-agent string.

The complete list of options available for the `setOption` call, and their current value, can be obtained via a call to `getOptions`. See the phpdoc documentation for details on the valid values of all the options.

cURL vs socket calls

Please note that, depending on the HTTP protocol version used and the options set to the client, the client will transparently switch between using a socket-based HTTP implementation and a CURL based implementation. If needed, you can make use of a `setOption(Client::OPT_USE_CURL, …)` method call to force or disable usage of the cURL based implementation.

When using cURL as the underlying transport, it is possible to set directly into the client any of the cURL options available in your php installation, via a `setOption(Client::OPT_EXTRA_CURL_OPTS, …)` method call.

When using sockets as the underlying transport, it is possible to set directly into the client any of the stream context options available for ssl and tcp in your php installation, via a `setOption(Client::OPT_EXTRA_SOCKET_OPTS, …)` method call.

Sending multiple requests

Both the Client and Server classes provided by the library support the "multicall" xmlrpc extension, which allows to execute multiple xml-rpc requests with a single http call, by wrapping them up in a call to the `system.multiCall` method.

The expected advantage is a nice improvements in performances, especially when there are many small requests at play, but, as always, the devil is in the details: the multicall specification does not mandate for the server to execute the single requests within the multicall method in a specific order, nor how to handle execution errors happening halfway through the list.

The phpxmlrpc server will execute all the requests sequentially, in the same order in which they appear in the xml payload, and will try its best to execute them all, even if one of them fails, but there is no guarantee on the latter point.

In order to take advantage of multicall, either use the Client's `multicall` method, or just pass an array of Request to the `send` method:

```
$m1 = new PhpXmlRpc\Request('system.methodHelp');
$m2 = new PhpXmlRpc\Request('system.methodSignature');
$val = new PhpXmlRpc\Value('an-xmlrpc-method', "string");
$m1->addParam($val);
$m2->addParam($val);
$ms = array($m1, $m2);
$rs = $client->multicall($ms);
foreach($rs as $resp) {
    var_dump($rs->faultCode());
    var_dump($rs->value());
}
```

Please note that, in case of faults during execution of a multicall call, the Client will automatically fail back to sending every request separately, one at a time. If you are sure that the server supports the multicall protocol, you might want to optimize and avoid this second attempt by passing `false` as 2nd argument to `multicall()`.

If, on the other hand, after writing code which uses the `multicall` method, you are forced to migrate to a server which does not support the `system.multiCall` method, you can simply call `$client->setOption(Client::OPT_NO_MULTICALL, true)`.

In case you are not using multicall, but have to send many requests in a row to the same server, the best performances are generally obtained by forcing the Client to use the cURL HTTP transport, which enables usage of http keepalive, and possibly of HTTP2.

The demo file *_demo/client/parallel.php* is a good starting point if you want to compare the performances of a single multicall request vs. sending multiple requests in a row. It even shows a non-multicall implementation which uses cURL to achieve sending of multiple requests in parallel.

Server

The implementation of this class has been kept as simple to use as possible. The constructor for the server basically does all the work. Here's a minimal example:

```
use PhpXmlRpc\Request;
use PhpXmlRpc\Response;
use PhpXmlRpc\Server;

function foo(Request $xmlrpc_request) {
    ...
    return new Response($some_xmlrpc_val);
}

class Bar {
    public static function fooBar(Request $xmlrpc_request) {
        ...
        return new Response($some_xmlrpc_val);
    }
}

$s = new Server(
    array(
        "examples.myFunc1" => array("function" => "foo"),
        "examples.myFunc2" => array("function" => "Bar::fooBar"),
    )
);
```

This performs everything you need to do with a server. The single constructor argument is an associative array from xml-rpc method names to php callables.

The dispatch map

The first argument to the Server constructor is an array, called the *dispatch map*. In this array is the information the server needs to service the XML-RPC methods you define.

The dispatch map takes the form of an associative array of associative arrays: the outer array has one entry for each method, the key being the method name. The corresponding value is another associative array, which can have the following members:

- `function` - this entry is mandatory. It must be a callable: either a name of a function in the global scope which services the XML-RPC method, an array containing an instance of an object and a method name, an array containing a class name and a static method name (for static class methods the '$class::$method' syntax is also supported), or an inline anonymous function.
- `signature` - this entry is an array containing the possible signatures (see Method signatures) for the method. If this entry is present then the server will check that the correct number and type of parameters have been sent for this method before dispatching it.
- `docstring` - this entry is a string containing documentation for the method. The documentation

may contain HTML markup.

- `signature_docs` - this entry can be used to provide documentation for the single parameters. It must match in structure the 'signature' member. By default, only the `documenting_xmlrpc_server` class in the extras package will take advantage of this, since the `system.methodHelp` protocol does not support documenting method parameters individually.
- `parameters_type` - this entry can be used when the server is working in 'xmlrpcvals' mode (see Method handler functions) to define one or more entries in the dispatch map as being functions that follow the 'phpvals' calling convention. The only useful value is currently the string 'phpvals'. *NB: this is known to be broken atm*
- `exception_handling` - this entry can be used to let the server handle in different ways the exceptions/errors thrown during execution of specific method handlers. By default any such error results in an xml-rpc response with a fixed fault code and fault string, which is a good security practice, but this behaviour can be changed to help debugging or in scenarios where the code is known to only ever throw "meaningful" exceptions with no sensitive information

Methods `system.listMethods`, `system.methodHelp`, `system.methodSignature` and `system.multicall` are already defined by the server, and should not be reimplemented (see Reserved methods below).

Method signatures

A signature is a description of a method's return type and its parameter types. A method may have more than one signature.

Within a server's dispatch map, each method has an array of possible signatures. Each signature is an array, with the first element being the return type, and the others being the types of the parameters. For instance, the method

```
string examples.getStateName(int)
```

has the signature

```
use PhpXmlRpc\Value;

array(Value::$xmlrpcString, Value::$xmlrpcInt)
```

and, assuming that it is the only possible signature for the method, it might be used like this in server creation:

```
use PhpXmlRpc\Server;
use PhpXmlRpc\Value;

$findstate_sig = array(array(Value::$xmlrpcString, Value::$xmlrpcInt));

$findstate_doc = 'When passed an integer between 1 and 51 returns the name of a US ' .
    'state, where the integer is the index of that state name in an alphabetic
order.';
```

```
$srv = new Server(array(
    "examples.getStateName" => array(
        "function" => "...",
        "signature" => $findstate_sig,
        "docstring" => $findstate_doc
    )
));
```

Note that method signatures do not allow to check nested parameters, e.g. the number, names and types of the members of a struct param cannot be validated.

If a method that you want to expose has a definite number of parameters, but each of those parameters could reasonably be of multiple types, the list of acceptable signatures will easily grow into a combinatorial explosion. To avoid such a situation, the lib defines the class property `Value::$xmlrpcValue`, which can be used in method signatures as a placeholder for 'any xml-rpc type':

```
use PhpXmlRpc\Server;
use PhpXmlRpc\Value;

$echoback_sig = array(array(Value::$xmlrpcValue, Value::$xmlrpcValue));

$findstate_doc = 'Echoes back to the client the received value, regardless of its
type';

$srv = new Server(array(
    "echoBack" => array(
        "function" => "...",
        // this sig guarantees that the method handler will be called with one and
only one parameter
        "signature" => $echoback_sig,
        "docstring" => $echoback_doc
    )
));
```

In case a php method that you want to expose has `void` return type, given the fact that there is no such thing as "no return value" in the XML-RPC specification, the best approximation is to also use the class property `Value::$xmlrpcValue` to indicate the return type in the method signature.

Method handler functions

The same php function can be registered as handler of multiple xml-rpc methods.

No text should be echoed 'to screen' by the handler function, or it will break the xml response sent back to the client. This applies also to error and warning messages that PHP prints to screen unless the appropriate settings have been set in `php.ini`, namely `display_errors`. Another way to prevent echoing of errors inside the response and facilitate debugging is to use the server's `SetDebug` method

with debug level 3 (see setDebug() / Server::OPT_DEBUG).

Exceptions thrown during execution of handler functions are caught by default and an XML-RPC error response is generated instead. This behaviour can be fine-tuned by setting the 'exception_handling' server option (see Server::OPT_EXCEPTION_HANDLING); please be aware that allowing publicly accessible servers to return the information from php exceptions as part of the xml-rpc response is a sure way to get hacked.

Manual type conversion

In this mode of operation, the incoming request is parsed into a `Request` object and dispatched to the relevant php function, which is responsible for returning a `Response` object, that will be serialized back to the caller. The synopsis of a method handler function is thus:

```
Response $resp = function(Request $req)
```

Note that if you implement a method with a name prefixed by `system.` the handler function will be invoked by the server with two parameters, the first being the server itself and the second being the Request object.

Here is a more detailed example of what a handler function "foo" might do:

```
use PhpXmlRpc\PhpXmlRpc;
use PhpXmlRpc\Request;
use PhpXmlRpc\Response;
use PhpXmlRpc\Value;

/**
 * @param Request $xmlrpcreq
 * @return Response
 */
function foo ($xmlrpcreq)
{
    // retrieve method name
    $meth = $xmlrpcreq->method();
    // retrieve value of first parameter - assumes at least one param received
    $par = $xmlrpcreq->getParam(0);
    // decode value of first parameter - assumes it is a scalar value
    $val = $par->scalarVal();

    // note that we could also have achieved the same this way:
    //$val = new PhpXmlRpc\Encoder()->decode($xmlrpcreq)[0];

    ...

    if ($err) {
        // this is an error condition
        return new Response(
            null,
```

```
            PhpXmlRpc::$xmlrpcerruser + 1, // user error 1
            "There's a problem, Captain"
        );
    } else {
        // this is a successful value being returned
        return new Response(new Value("All's fine!"));
    }
}
```

Automatic type conversion

In the same spirit of simplification that inspired the Client's 'return_type' option, a similar option is available within the server class: 'functions_parameters_type'. When set to the string 'phpvals', the functions registered in the server dispatch map will be called with plain php values as parameters, instead of a single Request instance parameter. The return value of those functions is expected to be a plain php value, too. An example is worth a thousand words:

```
use PhpXmlRpc\PhpXmlRpc;
use PhpXmlRpc\Server;
use PhpXmlRpc\Value;

function foo($usr_id, $out_lang='en')
{
    ...

    if ($someErrorCondition)
        throw new \Exception('DOH!', PhpXmlRpc::$xmlrpcerruser+1);
    else
        return array(
            'name' => 'Joe',
            'age' => 27,
            // it is possible to mix php values and Value objects!
            'picture' => new Value(file_get_contents($picOfTheGuy), 'base64'),
        );
}

$srv = new Server(
    array(
        "examples.myFunc" => array(
            "function" => "foo",
            "signature" => array(
                array(Value::$xmlrpcStruct, Value::$xmlrpcInt),
                array(Value::$xmlrpcStruct, Value::$xmlrpcInt, $xmlrpcString)
            ),
            "parameters_type" => "phpvals"
        )
    ),
    false
);
```

```
// The line below is an alternative to specifying `parameters_type` for every method
in the dispatch map. It applies
// to all php functions registered with the server.
// If both the server option and the `parameters_type` are set, the latter takes
precedence.
//$srv->setOption(Server::OPT_FUNCTIONS_PARAMETERS_TYPE, 'phpvals');

$srv->service();
```

There are a few things to keep in mind when using this calling convention:

- to return an xml-rpc error, the method handler function must return an instance of Response. The only other way for the server to know when an error response should be served to the client is to throw an exception and set the server's `exception_handling` member var to 1 (but please note that this is generally a *very bad idea* for servers with public access);
- to return a base64 value, the method handler function must encode it on its own, creating an instance of a Value object;
- to fine-tune the encoding to xml-rpc types of the method handler's result, you can use the Server's 'phpvals_encoding_options' option
- the method handler function cannot determine the name of the xml-rpc method it is serving, unlike manual-conversion handler functions that can retrieve it from the Request object;
- when receiving nested parameters, the method handler function has no way to distinguish a php string that was sent as base64 value from one that was sent as a string value;
- this has a direct consequence on the support of `system.multicall`: a method whose signature contains datetime or base64 values will not be available to multicall calls;
- last but not least, the direct parsing of xml to php values is faster than using xmlrpcvals, and allows the library to handle bigger messages without allocating all available server memory or smashing PHP recursive call stack.

An example of a Server using automatic type conversion is found in demo file *demo/server/discuss.php*

Delaying the server response

You may want to construct the server, but for some reason not fulfill the request immediately (security verification, for instance). If you omit to pass to the constructor the dispatch map or pass it a second argument of `0` this will have the desired effect. You can then use the `service` method of the server instance to service the request. For example:

```
use PhpXmlRpc\Server;

// second parameter = 0 prevents automatic servicing of request
$s = new Server($myDispMap, 0);

// ... some code that does other stuff here
```

```
$s->service();
```

Note that the `service` method will print the complete result payload to screen and send appropriate HTTP headers back to the client, but also return the response object. This permits further manipulation of the response, possibly in combination with output buffering.

To prevent the server from sending HTTP headers back to the client, you can pass a second parameter with a value of `true` to the `service` method (the first parameter being the payload of the incoming request; it can be left empty to use automatically the HTTP POST body). In this case, the response payload will be returned instead of the response object.

Xml-rpc requests retrieved by other means than HTTP POST bodies can also be processed. For example:

```
use PhpXmlRpc\Server;

$srv = new Server(); // not passing a dispatch map prevents automatic servicing of
request

// ... some code that does other stuff here, including setting dispatch map into
server object

// parse a variable instead of POST body, retrieve response payload
$resp = $srv->service($xmlrpc_request_body, true);

// ... some code that does other stuff with xml response $resp here
```

See the file *demo/server/symfony/ServerController.php* for a complete example of such use.

Modifying the server's behaviour

A few options are available to modify the behaviour of the server. The only way to take advantage of their existence is by usage of a delayed server response (see above) and a `setOption` call.

setDebug() / Server::OPT_DEBUG

This function controls weather the server is going to echo debugging messages back to the client as comments in response body. Valid values: 0,1,2,3, with 1 being the default. At level 0, no debug info is returned to the client. At level 2, the complete client request is added to the response, as part of the xml comments. At level 3, a new PHP error handler is set when executing user functions exposed as server methods, and all non-fatal errors are trapped and added as comments into the response.

Note: the `Client` class also sports a `setDebug` method, but its valid values are -1 to 2.

Server::OPT_ALLOW_SYSTEM_FUNCS

Default value: `true`. When set to `false`, disables support for `System.xxx` functions in the server. It might be useful e.g. if you do not wish the server to respond to requests to `System.ListMethods`.

Server::OPT_COMPRESS_RESPONSE

When set to `true`, enables the server to take advantage of HTTP compression, otherwise disables it. Responses will be transparently compressed, but only when an xml-rpc client declares its support for compression in the HTTP headers of the request.

Note that the ZLIB php extension must be installed for this to work. If it is, 'compress_response' will default to TRUE.

Server::OPT_EXCEPTION_HANDLING

This option controls the behaviour of the server when an exception is thrown by a method handler php function. Valid values: 0,1,2, with 0 being the default. At level 0, the server catches the exception and returns an 'internal error' xml-rpc response; at 1 it catches the exception and returns an xml-rpc response with the error code and error message corresponding to the exception that was thrown - never enable it for publicly accessible servers!; at 2, the exception is floated to the upper layers in the code - which hopefully do not display it to end users.

Server::OPT_RESPONSE_CHARSET_ENCODING

Charset encoding to be used for responses (only affects string values).

If it can, the server will convert the generated response from internal_encoding to the intended one.

Valid values are: a supported xml encoding (only `UTF-8` and `ISO-8859-1` at present, unless mbstring is enabled), `null` (leave charset unspecified in response and convert output stream to US_ASCII), or `auto` (use client-specified charset encoding or same as request if request headers do not specify it (unless request is US-ASCII: then use library default anyway).

Troubleshooting server's method handlers

A tried-and-true way to debug a piece of php code is to add a `var_dump()` call, followed by `die()`, at the exact place where one thinks things are going wrong. However, doing so in functions registered as xml-rpc method handlers is not as handy as it is for web pages: for a start a valid xml-rpc request is required to trigger execution of the code, which forces usage of an xml-rpc client instead of a plain browser; then, the xml-rpc client in use might lack the capability of displaying the received payload if it is not valid xml-rpc xml.

In order to overcome this issue, two helper methods are available in the Server class: `error_occurred($message)` and `debugmsg($message)`. The given messages will be added as xml comments, using base64 encoding to avoid breaking xml, into the server's responses, provided the server's debug level has been set to at least 1 for debug messages and 2 for error messages. The xml-rpc client provided with this library can handle the specific format used by those xml comments, and will display their decoded value when it also has been set to use an appropriate

debug level.

Fault reporting

In order to avoid conflict with error codes used by the library, fault codes used by your servers' method handlers should start at the value indicated by the variable `PhpXmlRpc::$xmlrpcerruser` + 1.

Standard errors returned by the library include:

1 Unknown method

Returned if the server was asked to dispatch a method it didn't know about

2 Invalid return payload

This error is actually generated by the client, not server, code, and signifies that a server returned something it couldn't understand. A more detailed error message is sometimes added at the end of the phrase above.

3 Incorrect parameters

This error is generated when the server has signature(s) defined for a method, and the parameters passed by the client do not match any of signatures.

4 Can't introspect: method unknown

This error is generated by the server's builtin system.* methods when any kind of introspection is attempted on an undefined method.

5 Didn't receive 200 OK from remote server

This error is generated by the client when a remote server doesn't return HTTP/1.1 200 OK in response to a request. A more detailed error report is added onto the end of the phrase above.

6 No data received from server

This error is generated by the client when a remote server returns HTTP/1.1 200 OK in response to a request, but no response body follows the HTTP headers.

7 No SSL support compiled in

This error is generated by the client when trying to send a request with HTTPS and the cURL extension is not available to PHP.

8 CURL error

This error is generated by the client when trying to send a request with HTTPS and the HTTPS communication fails.

9-14, 18 multicall errors

These errors are generated by the server when something fails inside a system.multicall request.

15 Invalid request payload

...

`16` **No CURL support compiled in**

…

`17` **Internal server error**

…

`19` **No HTTP/2 support compiled in**

…

`20` **Unsupported client option**

…

`100-` **XML parse errors**

Returns 100 plus the XML parser error code for the fault that occurred. The faultString returned explains where the parse error was in the incoming XML stream.

It is possible to change the numeric value of the above errors to support an xml-rpc "standard" (possibly not widely known) for error codes, by making use of the call `PhpXmlrpc\PhpXmlRpc::useInteropFaults()`.

Reserved methods

In order to extend the functionality offered by XML-RPC servers without impacting on the protocol, reserved methods are supported.

All methods starting with *system.* are considered reserved by the server. PHPXMLRPC itself provides four special methods, detailed in this chapter.

Note that all Server objects will automatically respond to clients querying these methods, unless the option 'allow_system_funcs' has been set to false before calling the `service()` method. This might pose a security risk if the server is exposed to public access, e.g. on the internet.

system.getCapabilities

This method lists all the capabilities that the XML-RPC server has: the (more or less standard) extensions to the xml-rpc spec that it implements. It takes no parameters.

system.listMethods

This method may be used to enumerate the methods implemented by the XML-RPC server.

The system.listMethods method requires no parameters. It returns an array of strings, each of which is the name of a method implemented by the server.

system.methodSignature

This method takes one parameter, the name of a method implemented by the XML-RPC server.

It returns an array of possible signatures for this method. A signature is an array of types. The first

of these types is the return type of the method, the rest are parameters.

Multiple signatures (i.e. overloading) are permitted: this is the reason that an array of signatures is returned by this method.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers, its signature is "string, int, int, int".

For parameters that can be of more than one type, the 'undefined' string is supported.

If no signature is defined for the method, a not-array value is returned. Therefore, this is the way to test for a non-signature, if $resp below is the response object from a method call to system.methodSignature:

```
$v = $resp->value();
if ($v->kindOf() != "array") {
    // then the method did not have a signature defined
}
```

See the *demo/client/introspect.php* demo included in this distribution for an example of using this method.

system.methodHelp

This method takes one parameter, the name of a method implemented by the XML-RPC server.

It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned.

The documentation string may contain HTML markup.

system.multicall

This method takes one parameter, an array of 'request' struct types. Each request struct must contain a `methodName` member of type string and a `params` member of type array, and corresponds to the invocation of the corresponding method.

It returns a response of type array, with each value of the array being either an error struct (containing the `faultCode` and `faultString` members) or the successful response value of the corresponding single method call.

Global configuration

Many static variables are defined in the `PhpxmlRpc\PhpXmlRpc` class and other classes. Some of those are meant to be used as constants (and modifying their value might cause unpredictable behaviour), while some others can be modified in your php scripts to alter the behaviour of either the xml-rpc client and server.

$xmlrpc_defencoding

```
PhpxmlRpc\PhpXmlRpc::$xmlrpc_defencoding = "UTF8"
```

This variable defines the character set encoding that will be used by the xml-rpc client and server to decode the received messages, when a specific charset declaration is not found (in the messages sent non-ascii chars are always encoded using character references, so that the produced xml is valid regardless of the charset encoding assumed).

Allowed values: 'UTF8', 'ISO-8859-1', 'ASCII'.

Note that the appropriate RFC actually mandates that XML received over HTTP without indication of charset encoding be treated as US-ASCII, but many servers and clients 'in the wild' violate the standard, and assume the default encoding is UTF-8.

$xmlrpc_internalencoding

```
PhpxmlRpc\PhpXmlRpc::$xmlrpc_internalencoding = "UTF-8"
```

This variable defines the character set encoding that the library uses to transparently encode into valid XML the xml-rpc values created by the user and to re-encode the received xml-rpc values when it passes them to the PHP application. It only affects xml-rpc values of string type. It is a separate value from `$xmlrpc_defencoding`, allowing e.g. to send/receive xml messages encoded on-the-wire in US-ASCII and process them as UTF-8. It defaults to the character set used internally by PHP (unless you are running an MBString-enabled installation), so you should change it only in special situations, if e.g. the string values exchanged in the xml-rpc messages are directly inserted into / fetched from a database configured to return non-UTF8 encoded strings to PHP. Example usage:

```
use PhpXmlRpc\Value;

// This is quite contrived. It is done because the asciidoc manual is saved in UTF-
8...
$latin1String = utf8_decode('Hélène');
$v = new Value($latin1String);
// Feel free to set this as early as possible
PhpxmlRpc\PhpXmlRpc::$xmlrpc_internalencoding = 'ISO-8859-1';
// The xml-rpc value will be correctly serialized as the french name
$xmlSnippet = $v->serialize();
```

$xmlpc_double_precision

The number of decimal digits used to serialize Double values. This is a requirement stemming from

$xmlrpcName

```
PhpxmlRpc\PhpXmlRpc::$xmlrpcName = "XML-RPC for PHP"
```

The string representation of the name of the PHPXMLRPC library. It is used by the Client for building the User-Agent HTTP header that is sent with every request to the server. You can change its value if you need to customize the User-Agent string.

$xmlrpcVersion

```
PhpXmlRpc\PhpXmlRpc::$xmlrpcVersion = "4.11.5"
```

The string representation of the version number of the PHPXMLRPC library in use. It is used by the Client for building the User-Agent HTTP header that is sent with every request to the server. You can change its value if you need to customize the User-Agent string.

$xmlrpc_null_extension

```
PhpxmlRpc\PhpXmlRpc::$xmlrpc_null_extension = FALSE
```

When set to `TRUE`, the lib will enable support for the `<NIL/>` (and `<EX:NIL/>`) xml-rpc value, as per the extension to the standard proposed here. This means that `<NIL>` and `<EX:NIL/>` tags received will be parsed as valid xml-rpc, and the corresponding xmlrpcvals will return "null" for scalarTyp().

$xmlrpc_null_apache_encoding

```
PhpxmlRpc\PhpXmlRpc::$$xmlrpc_null_apache_encoding = FALSE
```

When set to `TRUE`, php NULL values encoded into Value objects will get serialized using the `<EX:NIL/>` tag instead of `<NIL/>`. Please note that both forms are always accepted as input regardless of the value of this variable.

Errors and Logging

Many of the classes in this library by default use the php error logging facilities to log errors in case there is some unexpected but non-fatal event happening, such as f.e. when an invalid xml-rpc request or response are received. Going straight to the log instead of triggering a php warning or error has the advantage of not breaking the xml-rpc output when the issue is happening within the context of an xmlrpc-server and `display_errors` is enabled.

In case things are not going as you expect, please check the error log first for the presence of any messages from PHPXMLRPC which could be useful in troubleshooting what is going on under the hood.

You can customize the way error messages are traced via the static method `setLogger` available for the classes `Charset`, `Client`, `Encoder`, `HTTP`, `Request`, `Server`, `Value` and `XMLParser`. To inject a custom logger to all classes supporting it, in a simplified manner, method `PhpXmlRpc::setLogger` is available. Last but not least, be aware that the same Logger is also responsible for echoing to screen the debug messages produced by the Client when its debug level has been set, and for logging deprecation messages when they have been activated; this allows to customize the debugging process in the same way.

Helper classes and functions

PHPXMLRPC contains some helper classes which you can use to make processing of XML-RPC requests easier.

Date handling

The XML-RPC specification has this to say on dates:

> Don't assume a timezone. It should be specified by the server in its documentation what assumptions it makes about timezones.

Unfortunately, this means that date processing isn't straightforward. Although XML-RPC uses ISO 8601 format dates, it doesn't use the timezone specifier.

We strongly recommend that in every case where you pass dates in XML-RPC calls, you use UTC (GMT) as your timezone. Most computer languages include routines for handling GMT times natively, and you won't have to translate between timezones.

For more information about dates, see ISO 8601: The Right Format for Dates, which has a handy link to a PDF of the ISO 8601 specification. Note that XML-RPC uses exactly one of the available representations: `CCYYMMDDTHH:MM:SS`.

iso8601_encode

```
string iso8601_encode(string $time_t, int $utc = 0)
```

Returns an ISO 8601 formatted date generated from the UNIX timestamp $time_t, as returned by the PHP function `time()`.

The argument $utc can be omitted, in which case it defaults to `0`. If it is set to `1`, then the function corrects the time passed in for UTC. Example: if you're in the GMT-6:00 timezone and set $utc, you will receive a date representation six hours ahead of your local time.

The included demo program *demo/client/vardemo.php* includes a demonstration of this function.

iso8601_decode

```
int iso8601_decode(string $isoString, int $utc = 0)
```

Returns a UNIX timestamp from an ISO 8601 encoded time and date string passed in. If $utc is 1 then $isoString is assumed to be in the UTC timezone, and thus the result is also UTC: otherwise, the timezone is assumed to be your local timezone and you receive a local timestamp.

Decoding xml

```
Value | Request | Response Encoder::decodeXml(string $xml, array $options)
```

Decodes the xml representation of either an xml-rpc request, response or single value, returning the corresponding phpxmlrpc object, or `false` in case of an error.

The options parameter is optional. If specified, it must consist of an array of options to be enabled in the decoding process. At the moment, no option is supported.

Example:

```
$text = '<value><array><data><value>Hello world</value></data></array></value>';
$val = $encoder::decodeXml($text);
if ($val)
    echo 'Found a value of type ' . $val->kindOf();
else
    echo 'Found invalid xml';
```

Transferring PHP objects over XML-RPC

In case there is a (real) need to transfer php object instances over XML-RPC, the "usual" way would be to use a `serialize` call on the sender side, then transfer the serialized string using a base64 xml-rpc value, and call `unserialize` on the receiving side.

The phpxmlrpc library does offer an alternative method, which might offer marginally better performances and ease of use, by usage of `PhpXmlRpc\Encoder::encode` and `PhpXmlRpc\Encoder::decode`:

1. on the sender side, encode the desired object using option 'encode_php_objs'. This will lead to the creation of an xml-rpc struct value with an extra xml attribute: "php_class"
2. on the receiver side, decode the received Value using option 'decode_php_objs'. The xml-rpc struct with the extra attribute will be converted back into an object of the desired class instead of an array

*WARNING:* please take extreme care before enabling the 'decode_php_objs' option: when php objects are rebuilt from the received xml, their constructor function will be silently invoked. This means that you are allowing the remote end to trigger execution of uncontrolled PHP code on your server, opening the door to code injection exploits. Only enable this option when you trust

completely the remote server/client. DO NOT USE THIS WITH UNTRUSTED USER INPUT

Note also that there are multiple limitations to this: the same PHP class definition must be available on both ends of the communication; the class constructor will be called but with no parameters at all, and methods such as `__unserialize` or `__wakeup` will not be called. Also, if a different toolkit than the phpxmlrpc library is used on the receiving side, it might reject the generated xml as invalid.

Code generation, Proxy objects & co.

For the extremely lazy coder, helper functions have been added that allow to expose any pre-existing php functions (or all the public methods of a Class) as xml-rpc method handlers, and convert a remotely exposed xml-rpc method into a local php function - or a set of xml-rpc methods into a php class. This allows to use the library in a "transparent" fashion, ie. without having to deal with the Value, Client, Request and Response classes - but it comes with many gotchas and limitations.

wrapXmlrpcMethod

```
Closure|string|false PhpXmlRpc\Wrapper::wrapXmlrpcMethod(Client $client, string
$methodName, array $extraOptions = [])
```

Given a pre-built client pointing to a given xml-rpc server and a method name, creates a php "wrapper" function that will call the remote method and return results using native php types for both params and results. The generated php function will return a Response object by default for failed xml-rpc calls.

The server must support the `system.methodSignature` xml-rpc method call for this function to work.

The client param must be a valid Client object, previously created with the address of the target xml-rpc server, and to which the preferred options for http communication have been set.

The optional parameters can be passed as key,value pairs in the `$extra_options` argument.

The `signum` option has the purpose of indicating which method signature to use, if the given xml-rpc method has multiple signatures (defaults to 0).

The `timeout` and `protocol` options are the same as the arguments with same name of the `Client::send()` method.

If set, the `new_function_name` option indicates which name should be used for the generated function. In case it is not set the function name will be auto-generated.

If the `return_source` option is set, the function will return the php source code of the wrapper function, instead of evaluating it. This useful to save the code and use it later as stand-alone xml-rpc client with no performance hit and no dependency on `system.methodSignature`.

If the `throw_on_fault` option is set, the unction will throw an exception instead of returning a Response object in all cases of communication errors or xml-rpc faults.

If the `encode_php_objs` option is set, instances of php objects later passed as parameters to the newly created function will receive a 'special' treatment that allows the server to rebuild them as php objects instead of simple arrays. Note that this entails using a "slightly augmented" version of the xml-rpc protocol (i.e. using element attributes), which might not be understood by xml-rpc servers implemented using other libraries; it works well when the server is built on top of phpxmlrpc.

If the `encode_nulls` option is set, php `null` values passed as arguments to the generated function will be encoded as xml-rpc '<NIL/>' values instead of being encoded as empty string values.

If the `decode_php_objs` option is set, instances of php objects that have been appropriately encoded by the server using a corresponding option will be deserialized as php objects instead of simple arrays (the same class definition should be present server side and client side).

Note that this might pose a security risk, since in order to rebuild the object instances their constructor method has to be invoked, and this means that the remote server can trigger execution of unforeseen php code on the client: not really a code injection, but almost. Please enable this option only when you absolutely trust the remote server.

In case of an error during generation of the wrapper function, FALSE is returned.

Known limitations: the server must support system.methodsignature for the desired xml-rpc method; for methods that expose multiple signatures, only one can be picked; for remote calls with nested xml-rpc params, the caller of the generated php function has to encode on its own the params passed to the php function if these are structs or arrays whose (sub)members include values of type base64.

Note: calling the generated php function 'might' be slow: a new xml-rpc client is created on every invocation and an xmlrpc-connection opened+closed.

An extra 'debug' argument is appended to the argument list of the generated php function, useful for debugging purposes.

Example usage:

```
use PhpXmlRpc\Client;
use PhpXmlRpc\Wrapper;

$c = new Client('https://phpxmlrpc.sourceforge.io/server.php');

$function = new Wrapper()->wrapXmlrpcMethod($client, 'examples.getStateName');

if (!$function)
  die('Failed introspecting remote method');
else {
  $stateNo = 15;
  $stateName = $function($stateNo);
  // NB: in real life, you should make sure you escape the received data with
  // `htmlspecialchars` when echoing it as html
  if (is_a($stateName, 'Response')) { // call failed
    echo 'Call failed: '.$stateName->faultCode().'. Calling again with debug on...';
```

```
    $function($stateNo, true);
  }
  else
    echo "OK, state nr. $stateNo is $stateName";
}
```

wrapXmlrpcServer

```
string|array|false PhpXmlRpc\Wrapper::wrapXmlrpcServer(Client $client, array
$extraOptions = [])
```

Similar to wrapXmlrpcMethod, but instead of creating a single php function this creates a php class, whose methods match all the xml-rpc methods available on the remote server.

Note that a simpler alternative to this, doing no type-checks on the arguments of the invoked methods, and providing no support for IDE auto-completion, can be found in the *demo/client/proxy.php* demo file.

wrapPhpFunction

```
array|false PhpXmlRpc\Wrapper::wrapPhpFunction(Callable $callable, string $newFuncName
= '', array $extraOptions = [])
```

Given a user-defined PHP function, create a PHP 'wrapper' function that can be exposed as xml-rpc method from a Server object and called from remote clients, and return the appropriate definition to be added to a server's dispatch map.

The optional `$newFuncName` specifies the name that will be used for the auto-generated function.

Since php is a typeless language, to infer types of input and output parameters, it relies on parsing the phpdoc-style comment block associated with the given function. Usage of xml-rpc native types (such as datetime.dateTime.iso8601 and base64) in the docblock @param tag is also allowed, if you need the php function to receive/send data in that particular format (note that base64 encoding/decoding is transparently carried out by the lib, while datetime values are passed around as strings).

Known limitations: only works for user-defined functions, not for PHP internal functions (reflection does not support retrieving number/type of params for those); the wrapped php function will not be able to programmatically return an xml-rpc error response.

If the `return_source` option parameter is set, the function will return the php source code to build the wrapper function, instead of evaluating it (useful to save the code and use it later in a stand-alone xml-rpc server). It will b e in the stored in the `source` member of the returned array.

If the `suppress_warnings` optional parameter is set, any runtime warning generated while processing the user-defined php function will be caught and not be printed in the generated xml response.

If the `encode_nulls` option is set, php `null` values returned by the php function will be encoded as xml-rpc '<NIL/>' values instead of being encoded as empty string values.

If the extra_options array contains the `encode_php_objs` value, wrapped functions returning php objects will generate "special" xml-rpc responses: when the decoding of those responses is carried out by this same lib, using the appropriate param in php_xmlrpc_decode(), the objects will be rebuilt.

In short: php objects can be serialized, too (except for their resource members), using this function. Other libs might choke on the very same xml that will be generated in this case (i.e. it has a nonstandard attribute on struct element tags)

If the `decode_php_objs` optional parameter is set, instances of php objects that have been appropriately encoded by the client using a coordinate option will be deserialized and passed to the user function as php objects instead of simple arrays (the same class definition should be present server side and client side).

Note that this might pose a security risk, since in order to rebuild the object instances their constructor method has to be invoked, and this means that the remote client can trigger execution of unforeseen php code on the server: not really a code injection, but almost. Please enable this option only when you trust the remote clients.

Example usage:

```
use PhpXmlRpc\Server;
use PhpXmlRpc\Wrapper;

/**
* State name from state number decoder. NB: do NOT remove this comment block.
* @param integer $stateno the state number
* @return string the name of the state (or an error description)
*/
function findstate($stateno)
{
    $stateNames = array(...);
    if (isset($stateNames[$stateno-1]))
    {
        return $stateNames[$stateno-1];
    }
    else
    {
        return "I don't have a state for the index '" . $stateno . "'";
    }
}

// wrap php function, build xml-rpc server
$methods = array();
$findstate_sig = new Wrapper()->wrapPhpFunction('findstate');
if ($findstate_sig)
    $methods['examples.getStateName'] = $findstate_sig;
```

```
$srv = new Server($methods);
```

Please note that similar results to the above, i.e. adding to the server's dispatch map an existing php function which is not aware of xml-rpc, can be obtained without the Wrapper class and the need for introspection, simply by calling `$server->setOption('functions_parameters_type', phpvals')` (see chapter Automatic type conversion). The main difference is that, using the Wrapper class, you get for free the documentation for the xml-rpc method.

wrapPhpClass

```
array|false PhpXmlRpc\Wrapper::wrapPhpClass(string|object $className, array
$extraOptions = [])
```

Similar to `wrapPhpFunction`, it works on all public methods of a given object/class. The server must support both the `system.methodList` and `system.methodSignature` xml-rpc method calls for this function to work.

Code generation

Using the Wrapper class to create some code and execute it directly inline has the worst possible performances, as it relies on either using introspection of existing php code or making extra calls to the xml-rpc introspection methods of the server. It also does not provide the benefit of allowing IDEs to inspect the generated code and provide auto-completion for it, nor for security-minded developers to be able to examine it before executing it. It is thus recommended to always use the `return_source` option when using the Wrapper methods, and save to disk the generated code instead of executing it directly.

Other tools

Other tools exist which share the same goal of generating php code implementing xml-rpc clients or server, starting from either an Interface Definition Language or existing php code.

One such project, not affiliated with this library, can be found at: https://github.com/mumitr0ll/xrdl

Performances

Although the library is not designed to be the most memory-efficient nor the most fast possible implementation of the xml-rpc protocol, care is taken not to introduce unnecessary bloat.

The *extras/benchmark.php* file is used to assess the changes to performance for each new release, and to compare the results obtained by executing the same operation using different options, such as f.e. manual vs. automatic encoding of php values to Value objects. You will have to install the library with the Compose option `--prefer-install=source` in order to have it available locally.

Performance optimization tips

- avoid spending time converting the received xml into Value objects, instead have the library pass primitive php values directly to your application by calling `$client->setOption('return_type', XMLParser::RETURN_PHP)` and `$server->setOption('functions_parameters_type', XMLParser::RETURN_PHP)`
- reduce the encoding of non-ascii characters to character entity references both by setting `PhpXmlRpc::$xmlrpc_internalencoding = 'UTF-8'` and calling `$client->setOption('request_charset_encoding', 'UTF-8')` / `$server->setOption('response_charset_encoding', 'UTF-8')`
- if the server you are communicating with does support it, and the requests you are sending are big, or the network slow, you should enable compression of the requests, via setting `$client->setOption('request_compression', true)`
- add a call `$server->setDebug(0)`
- add a call `$client->setDebug(-1)` - unless you need to access the response's http headers or cookies
- boxcar multiple xml-rpc calls into a single http request by making usage of the `system.multicall` capability. Just passing in an array of Request objects to `$client->send()` is usually enough. If the server you are talking to does not support `system.multicall`, see the *demo/client/parallel.php* example instead for how to send multiple requests in parallel using cURL
- use http2

Upgrading

our Backward Compatibility promise

The PHPXMLRPC library adheres to Semantic Versioning principles, as outlined in https://semver.org/.

In short, Semantic Versioning means that only major releases (such as 3.0, 4.0 etc.) are allowed to break backward compatibility. Minor releases (such as 4.1, 4.2 etc.) may introduce new features, but must do so without breaking the existing API of the given release branch (4.x in the previous example).

However, *backward compatibility* comes in many different flavors. In fact, almost every change made to the library can potentially break an application. For example: * if we add a new argument to an existing class method, this will break an application which extended this class and reimplemented the same method, as it will have to add the new argument as well - even if we give it a default value * if we add a new method to a class, this will break an application which extended this class and added the same method, but with a different method signature * if we add a new protected property to a class, this will break an application which extended this class and added the same method property, and uses it for a different purpose * etc...

The guiding principle used to inform the development process and the version number applied to new releases is: * we adhere to strict Semantic Versioning Backward Compatibility rules for

consumers of the library. * we *strive* to maintain Semantic Versioning Backward Compatibility for developers *extending* the library, for as long as humanly possible, but do not guarantee it 100% The distinction between *consumers* and *extenders* is defined as: an extender is anyone implementing a subclass of any of the PHPXMLRPC classes, or replacing it wholesale with his/her own implementation.

In detail: * to avoid breaking compatibility with extenders, the library will not implement strict type declarations. This might include weird practices such as declaring interfaces but not checking or enforcing them at the point of use * new classes, interfaces, traits, exceptions, class methods, constants and properties are allowed to be added in minor versions. NB: this includes adding magic methods `__get`, `\__set`, `__call`, etc... to classes which did not have them * new arguments to functions and methods are allowed to be added in minor versions, provided they have a default value * types of function and method arguments are only allowed to be changed in major versions * return types of functions and methods are only allowed to be changed in major versions * existing function and method arguments, class methods and properties, interfaces, traits and exceptions might be deprecated in minor versions * existing class methods and properties might be removed or renamed in minor versions, provided that their previous incarnations are kept working via usage of php magic functions `\__get`, `__set`, `\__call` and friends * exceptions thrown by the library are allowed to be moved to a subclass of their previous class in minor versions * exception messages and error messages are not guaranteed to be stable. They mostly get more detailed over time, but might occasionally see complete rewording * any class, method, property, interface, exception and constant documented using the `@internal` tag is considered not to be part of the library's API and might not follow the rules above * using deprecated methods and properties will eventually lead to deprecation warnings being generated - but that is not yet enabled by default (it can be enabled by the forward-looking developer, though)

Finally, *always read carefully the NEWS file before upgrading*. It is the single most important source of truth regarding changes to the API (after the code itself, of course)

Upgrading to version 4

If you are upgrading to version 4 from version 3 or earlier you have two options:

1. adapt your code to the new API (all changes needed are described in https://github.com/gggeek/phpxmlrpc/blob/master/doc/api_changes_v4.md)
2. use instead the **compatibility layer** which is provided. Instructions and pitfalls described at https://github.com/gggeek/phpxmlrpc/blob/master/doc/api_changes_v4.md#enabling-compatibility-with-legacy-code

In any case, read carefully the docs available online and report back any undocumented issue using GitHub.

Upgrading to version 2

The following two functions have been deprecated in version 1.1 of the library, and removed in version 2, in order to avoid conflicts with the PHP xml-rpc extension, which also defines two functions with the same names.

The following documentation is kept for historical reference:

xmlrpc_decode

```
mixed mlrpc_decode(Value $xmlrpc_val)
```

Alias for `php_xmlrpc_decode`.

xmlrpc_encode

```
Value xmlrpc_encode(mixed $phpval)
```

Alias for `php_xmlrpc_encode`.

Bundled debugger

A webservice debugger is included in the library to help during development and testing.

Debugger setup

NB to avoid turning your webserver into an open relay for http calls, please keep the debugger outside your webserver's document root by default / in production deployments!

In order to make usage of the debugger, you will need to have a webserver configured to run php code, and make it serve the `/debugger` folder within the library.

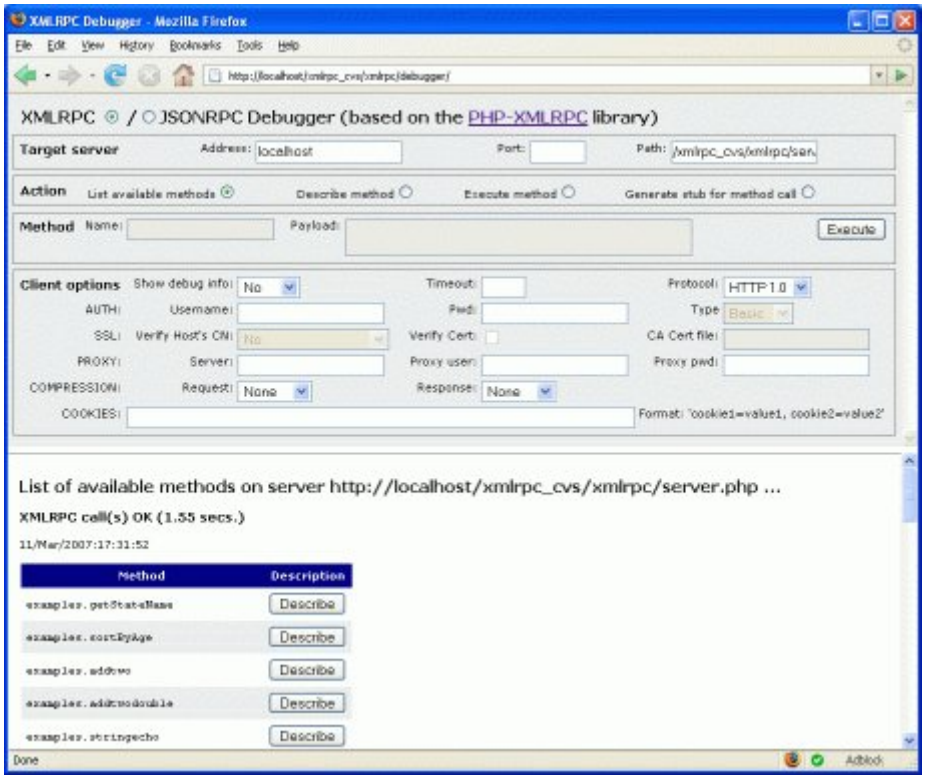
The simplest method is to start the php command-line webserver, but if you do so, you should make use of the experimental multi-process setup. Ex:

```
cd debugger; PHP_CLI_SERVER_WORKERS=2 php -S 127.0.0.1:8081
```

then access the debugger by pointing your browser at *http://127.0.0.1:8081*

Debugger usage

The interface should be self-explicative enough to need little documentation.



To make sure that the debugger is working properly, you can use it make f.e. a "list available methods" call against the public demo server available at: Address: *tanoconsulting.com*, Path: */sw/xmlrpc/demo/server/server.php*

The most useful feature of the debugger is without doubt the "Show debug info" option. It allows to have a screen dump of the complete http communication between client and server, including the http headers as well as the request and response payloads, and is invaluable when troubleshooting problems with charset encoding, authentication or http compression.

Using the debugger against a local server

If the webserver used to run the debugger is prevented from making http calls to the internet at large for security or connectivity reasons, one way to make sure that it is working as expected and get acquainted with the library's workings is to test against the "demo" server which comes bundled with the library:

- install the library using the Composer option `--prefer-install=source`, to make sure the demo files are also downloaded
- make sure both the `/debugger` and the `/demo` folders are within your webserver's root folder, e.g. run `PHP_CLI_SERVER_WORKERS=2 php -S 127.0.0.1:8081` from the root of the phpxmlrpc library
- access the debugger at *http://127.0.0.1:8081/debugger* and use it with Address: *127.0.0.1*, Path: */demo/server/server.php*

Debugger extension

The debugger can take advantage of the JSXMLRPC library's visual editor component to allow easy mouse-driven construction of the payload for remote methods. To enable the extra functionality, it has have to be downloaded separately and copied to the debugger directory. The easiest way to

achieve that is to run the command

```
./taskfile setup_debugger_visualeditor
```

If that command does not work on your installation (it has not been widely tested on MacOS, and it does not support Windows) and you have NodeJS installed, you can achieve the same by executing

```
cd debugger && npm install @jsxmlrpc/jsxmlrpc
```

Alternative debugger setup

Since November 2022, the same interactive xml-rpc debugger which is bundled with this library is also available as a Docker Container image, making it easy to use it as a standalone tool in any environment, without the need for having PHP or Composer installed.

Installation and usage instructions can be found at https://github.com/gggeek/phpxmlrpc-debugger

Running tests

The recommended way to run the library's test suite is via the provided Docker containers. A handy shell script is available that simplifies usage of Docker.

The full sequence of operations is:

```
./tests/ci/vm.sh build
./tests/ci/vm.sh start
./tests/ci/vm.sh runtests
./tests/ci/vm.sh stop
```

```
# and, once you have finished all testing related work:
./tests/ci/vm.sh cleanup
```

By default, tests are run using php 8.1 in a Container based on Ubuntu 22 Jammy. You can change the version of PHP and Ubuntu in use by setting the environment variables PHP_VERSION and UBUNTU_VERSION before building the Container.

To generate the code-coverage report, run `./tests/ci/vm.sh runcoverage`

Note: to reduce the size of the download, the test suite is not part of the default package installed with Composer. In order to have it onboard, install the library using Composer option `--prefer-install=source`.

Frequently Asked Questions

My client returns "XML-RPC Fault #2: Invalid return payload: enable debugging to examine incoming payload": what should I do?

The response you are seeing is a default error response that the client object returns to the php application when the server did not respond to the call with a valid xml-rpc response.

The most likely cause is that you are not using the correct URL when creating the client object, or you do not have appropriate access rights to the web page you are requesting, or some other common http misconfiguration.

To find out what the server is really returning to your client, you have to enable the debug mode of the client, using `$client->setDebug(1)`. You can also inspect the http connection information in `$response->httpResponse()` - see below

How can I save to a file the xml of the xml-rpc responses received from servers?

If what you need is to save the responses received from the server as xml, you have multiple options:

1 - use the Response's `httpResponse` method

```
$resp = $client->send($msg);
if (!$resp->faultCode())
    $data_to_be_saved = $resp->httpResponse()['raw_data'];
```

Note that, while the data saved this way is an accurate copy of what is received from the server, it might not match what gets parsed into the response's value, as there is some filtering involved, such as stripping of comments junk from the end of the message, character set conversion, etc...

Note also that, in the future, this might need some debug mode to be enabled in order to work.

2 - use the `serialize` method on the Response object.

```
$resp = $client->send($msg);
if (!$resp->faultCode())
  $data_to_be_saved = $resp->serialize();
```

Note that this will not be 100% accurate, since the xml generated by the response object can be different from the xml received, especially if there is some character set conversion involved, or such (e.g. if you receive an empty string tag as "<string/>", `serialize()` will output

"<string></string>"), or if the server sent back as response something invalid (in which case the xml generated client side using serialize() will correspond to the error response generated internally by the lib).

3 - set the client object to return the raw xml received instead of the decoded objects:

```
$client = new Client($url);
$client->return_type = 'xml';
$resp = $client->send($msg);
if (!$resp->faultCode())
    $data_to_be_saved = $resp->value();
```

Note that using this method the xml response will not be parsed at all by the library, only the http communication protocol will be checked. This means that xml-rpc responses sent by the server that would have generated an error response on the client (e.g. malformed xml, responses that have faultCode set, etc...) now will not be flagged as invalid, and you might end up saving not valid xml but random junk...

How can I save to a file the xml of the xml-rpc requests/responses generated from the library?

Classes `Request`, `Response` and `Value` all have a method `serialize()` which can be used to obtain the xml representation of their value.

Note that, if what you want is to check with absolute certainty what is being sent over the wire, you are better off using the `setDebug` method in both the client and the server.

Is there any limitation on the size of the requests / responses that can be successfully sent?

Yes. But there is no hard figure to be given; it most likely will depend on the version of PHP in usage and its configuration.

Keep in mind that this library is not optimized for speed nor for memory usage. Better alternatives exist when there are strict requirements on throughput or resource usage, such as the php native xmlrpc extension (see the PHP manual for more information).

Keep in mind also that HTTP is probably not the best choice in such a situation, and XML is a deadly enemy. CSV formatted data over socket would be much more efficient. Or rpc protocols Googles' ProtoBuffer.

If you really need to move a massive amount of data around, and you are crazy enough to do it using phpxmlrpc, your best bet is to bypass usage of the Value objects, at least in the decoding phase, and have the server (or client) object return to the calling function directly php values (see Client option `return_type` and Server option `functions_parameters_types` for more details, and the tips in the Performances section).

How to send custom XML as payload of a method call

Unfortunately, at the time the XML-RPC spec was designed, support for namespaces in XML was not as ubiquitous as it became later. As a consequence, no support was provided in the protocol for embedding XML elements from other namespaces into an xml-rpc request.

To send an XML "chunk" as payload of a method call or response, two options are available: either send the complete XML block as a string xml-rpc value, or as a base64 value. Since the '<' character in string values is encoded as '<' in the xml payload of the method call, the XML string will not break the surrounding xml-rpc, unless characters outside the assumed character set are used. The second method has the added benefits of working independently of the charset encoding used for the xml to be transmitted, and preserving exactly whitespace, whilst incurring in some extra message length and cpu load (for carrying out the base64 encoding/decoding).

See the example given in *demo/client/which.php* for the possibility of sending "standard xml-rpc" xml which was generated outside the phpxmlrpc library.

My server (client) returns an error whenever the client (server) sends accented characters

To be documented...

Can I use the MS Windows character set?

If the data your application is using comes from a Microsoft application, there are some chances that the character set used to encode it is CP1252 (the same might apply to data received from an external xml-rpc server/client, but it is quite rare to find xml-rpc toolkits that encode to CP1252 instead of UTF8). It is a character set which is "almost" compatible with ISO 8859-1, but for a few extra characters.

PHPXMLRPC always supports the ISO-8859-1 and UTF-8 character sets, plus any character sets which are available via the mbstring php extension.

To properly encode outgoing data that is natively in CP1252, you will have to make sure that mbstring is enabled, then set

```
PhpXmlRpc\PhpXmlRpc::$xmlrpc_internalencoding = 'Windows-1252';
```

somewhere in your code, before any outgoing data is serialized.

The same setting will also ensure that the data which is fed back to your application will also be transcoded by the library into the same character set, regardless by the character set used over the wire.

This feature is available since release 4.10, and can be seen in action in file *demo/client/windowscharset.php*

Does the library support using cookies / http sessions?

In short: yes, but a little coding is needed to make it happen.

The code below uses sessions to e.g. let the client store a value on the server and retrieve it later.

```
use PhpXmlRpc/Request;
use PhpXmlRpc/Value;

$resp = $client->send(new Request(
    'registervalue',
    array(new Value('foo'), new Value('bar')))
);
if (!$resp->faultCode())
{
    $cookies = $resp->cookies();
    // nb: make sure to use the correct session cookie name
    if (array_key_exists('PHPSESSID', $cookies))
    {
        $session_id = $cookies['PHPSESSID']['value'];

        // do some other stuff here...

        // ...also, we should check for the cookie validity by looking at things such
as its expiration, domain, etc...
        $client->setcookie('PHPSESSID', $session_id);
        $val = $client->send(new Request('doStuff', array(new Value('foo'))));
    }
}
```

Server-side sessions are handled normally like in any other php application. Please see the php manual for more information about sessions.

NB: unlike web browsers, not all xml-rpc clients support usage of http cookies. If you have troubles with sessions and control only the server side of the communication, please check with the makers of the xml-rpc client in use.

Does the library support following http redirects?

Yes, but only when using cURL for transport.

```
$client->setOption(Client::OPT_USE_CURL, \PhpXmlRpc\Client::USE_CURL_ALWAYS);
$client->setOption(Client::OPT_EXTRA_CURL_OPTS, [CURLOPT_FOLLOWLOCATION => true,
CURLOPT_POSTREDIR => 3]);
```

Does the library support setting custom cURL options?

Yes. Use `$client->setOption(Client::OPT_USE_CURL, \PhpXmlRpc\Client::USE_CURL_ALWAYS)` then use the Client method `$client->setOption(Client::OPT_EXTRA_CURL_OPTS, array(...))`

Does the server support cross-domain xml-rpc calls?

It is trivial to make phpxmlrpc servers support CORS preflight requests, allowing them to receive xml-rpc requests sent from browsers visiting different domains. However, this feature is not enabled out of the box, for obvious security concerns. See at the top of the file *demo/server/server.php* for an example of enabling that.

How to enable long-lasting method calls

To be documented...

Integration with the PHP xmlrpc extension

In short: for the fastest execution possible, you can enable the php native xmlrpc extension, and use it in conjunction with phpxmlrpc. The following code snippet gives an example of such integration:

```
/*** client side ***/
$c = new Client('https://phpxmlrpc.sourceforge.io/server.php');

// tell the client to return raw xml as response value
$c->setOption('return_type', 'xml');

// let the native xmlrpc extension take care of encoding request parameters
$r = $c->send(xmlrpc_encode_request('examples.getStateName', (int)$_POST['stateno']));

if ($r->faultCode()) {
    // HTTP transport error
    echo 'Got error ' . $r->faultCode();
} else {
    // HTTP request OK, but XML returned from server not parsed yet
    $v = xmlrpc_decode($r->value());
    // check if we got a valid xml-rpc response from server
    if ($v === NULL)
        echo 'Got invalid response';
    else
    // check if server sent a fault response
    if (xmlrpc_is_fault($v))
        echo 'Got xml-rpc fault '.$v['faultCode'];
    else
        echo'Got response: '.htmlentities($v);
}
```

NB: Please note that, as of PHP 8.2, the native xmlrpc extension has been moved to Pecl, and it is not bundled in the stock PHP builds anymore. Moreover, its development has all but ceased, and its usage is discouraged.

Substitution of the PHP xmlrpc extension

Yet another interesting situation is when you are using a ready-made php application, that provides support for the XML-RPC protocol via the native php xmlrpc extension, but the extension is not available on your php install (e.g. because of shared hosting constraints, or because you are using php 8.2 or later).

Since version 2.1, the PHPXMLRPC library provides a compatibility layer that aims to be 100% compliant with the xmlrpc extension API. This means that any code written to run on the extension should obtain the exact same results, albeit using more resources and a longer processing time, using the PHPXMLRPC library and the extension compatibility module.

The module was originally part of the EXTRAS package, available as a separate download from the sourceforge.net website; it has since become available as Packagist package `phpxmlrpc/polyfill-xmlrpc` and can be found on GitHub at https://github.com/gggeek/polyfill-xmlrpc

Appendix A: Files in the distribution

debugger/*

a graphical debugger which can be used to test calls to xml-rpc servers

demo/*

example code for implementing both client and server functionality. Only included when installing with `--prefer-install=source`

doc/*

the documentation, including this manual, and the list of API changes between versions 3 and 4

extras/*

php utility scripts, such as a benchmark suite and an environment compatibility checker. Only included when installing with `--prefer-install=source`

lib/*

a compatibility layer for applications which still rely on version 3 of the API

src/*

the XML-RPC library classes. You can autoload these via Composer, or via a dedicated Autoloader class

tests/*

the test suite for the library, written using PhpUnit, and the configuration to run it in a local Docker container. Only included when installing with `--prefer-install=source`

Note the standard procedure to download locally the demo and test files is to use Composer with the option `--prefer-install=source` on the command line. That requires to have `git` installed. If that is not the case on your server, you might be able to download the complete source code from GitHub with other tools, such as f.e. TortoiseSVN. Starting with release 4.9.4, the demo files are also available for download as a separate tarball from the releases page on GitHub.

Note when downloading the demo files, make sure that the demo folder is not directly accessible from the internet, i.e. it is not within the webserver root directory.

Appendix B: Exception hierarchy

```
Exception
`-PhpXmlRpc/Exception
  |-PhpXmlRpc/Exception/FaultResponseException.php
  |-PhpXmlRpc/Exception/ParsingException.php
  | |-PhpXmlRpc/Exception/XmlException.php
  | `-PhpXmlRpc/Exception/XmlRpcException.php
  |-PhpXmlRpc/Exception/TransportException.php
  | `-PhpXmlRpc/Exception/HttpException.php
  |-PhpXmlRpc/Exception/ServerException.php
  | `-PhpXmlRpc/Exception/NoSuchMethodException.php
  |-PhpXmlRpc/Exception/StateErrorException.php
  |-PhpXmlRpc/Exception/TypeErrorException.php
  `-PhpXmlRpc/Exception/ValueErrorException.php
```

Note not all of the above exceptions are in use at the moment, but they might be in the future